

Graph Projections for Fast Graph Algorithms

A dissertation submitted towards the degree
Doctor of Natural Sciences
of the Faculty of Mathematics and Computer Science
of Saarland University

by

Yonggang Jiang

Day of Colloquium: 2026/09/25

Chair of the Committee: Raimund Seidel

Reporters:

Sanjeev Khanna

Rasmus Kyng

Danupon Nanongkai

Academic Assistant: Rex Lei



Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form in einem Verfahren zur Erlangung eines akademischen Grades vorgelegt.

Declaration of original authorship

I hereby declare that this dissertation is my own original work except where otherwise indicated. All data or concepts drawn directly or indirectly from other sources have been correctly acknowledged. This dissertation has not been submitted in its present or similar form to any other academic institution either in Germany or abroad for the award of any other degree.

Saarbrücken, 2026

gez. / signed

Yonggang Jiang

Abstract

This thesis develops *graph projection techniques* for designing fast graph algorithms. Many classical graph-algorithmic techniques simplify a problem by replacing the input graph with a structurally simpler graph on essentially the same vertex set. Examples include tree embeddings, decompositions, spanners, sparsifiers, and shortcut constructions. This thesis develops a different viewpoint: expanding the graph. We construct a larger auxiliary graph $G' = (V', E')$ together with a many-to-one projection map $\pi : V' \rightarrow V \cup \{\perp\}$. The key insight is that, although G' is larger, it can have a much simpler structure than the original graph. This allows us to solve problems faster after lifting them to G' , and then project the solution back to G . Using this perspective, we obtain new techniques that break long-standing bounds in graph algorithms.

First, we develop **DAG projection**, which projects a DAG G' to an input directed graph G while preserving distance, flow, and cut information. This allows many distance-, flow-, and cut-related problems on general directed graphs to be reduced to corresponding problems on DAGs. In particular, it gives a black-box reduction from minimum-cost flow on general graphs to minimum-cost flow on DAGs, with only a subpolynomial loss in running time. As a consequence, we obtain the first almost-linear-time combinatorial algorithm for minimum-cost flow on dense graphs, giving the first combinatorial improvement over the Goldberg–Tarjan algorithm from 1987.

Second, we develop **path cover**, which projects from a directed graph G' whose strongly connected components have small diameter to an input directed graph G while covering all short paths. This serves as a deterministic counterpart to the powerful low-diameter decomposition paradigm and leads to the first deterministic near-linear-time algorithm for negative-weight shortest paths, giving the first deterministic combinatorial improvement since Goldberg’s algorithm in 1995.

Third, we propose **common-neighborhood clustering**, which projects a union of graphs in which pairs of vertices share many common neighbors to an input graph G . This leads to the first deterministic algorithm for global minimum vertex cut that closes the gap between deterministic and randomized time complexities, answering an open question posed by Gabow in 2006, and yields new results for vertex connectivity in parallel and distributed computing.

Finally, we study **Steiner shortcuts**, which project a shallow graph G' to the input graph G while preserving distance, flow, and cut information. We show how the viewpoint of “expanding the graph” has the potential to circumvent the classical barrier for good-quality shortcuts on general directed graphs, and discuss its broader consequences.

Altogether, these results show that expanding a graph into a larger but more structured object can be a powerful algorithmic principle.

Zusammenfassung

Diese Dissertation entwickelt *Graphprojektions-Techniken* zum Entwurf schneller Graphalgorithmen. Viele klassische graphalgorithmische Techniken vereinfachen ein Problem, indem sie den Eingabegraphen durch einen strukturell einfacheren Graphen auf im Wesentlichen derselben Knotenmenge ersetzen. Beispiele hierfür sind Baumeinbettungen, Zerlegungen, Spanner, Sparsifizierer und Shortcut-Konstruktionen. Diese Dissertation entwickelt eine andere Sichtweise: die Erweiterung des Graphen. Wir konstruieren einen größeren Hilfsgraphen $G' = (V', E')$ zusammen mit einer viele-zu-eins-Projektionsabbildung $\pi : V' \rightarrow V \cup \perp$. Die zentrale Einsicht ist, dass G' , obwohl er größer ist, eine deutlich einfachere Struktur als der ursprüngliche Graph haben kann. Dies erlaubt es uns, Probleme schneller zu lösen, nachdem wir sie auf G' gehoben haben, und die Lösung anschließend zurück auf G zu projizieren. Aus dieser Perspektive erhalten wir neue Techniken, die langjährige Schranken in Graphalgorithmen durchbrechen.

Zunächst entwickeln wir die **DAG-Projektion**, die einen DAG G' auf einen Eingabegraphen G projiziert und dabei Distanz-, Fluss- und Schnittinformationen erhält. Dadurch können viele distanz-, fluss- und schnittbezogene Probleme auf allgemeinen gerichteten Graphen auf entsprechende Probleme auf DAGs reduziert werden. Insbesondere ergibt sich daraus eine Black-Box-Reduktion von Min-Cost-Flow auf allgemeinen Graphen auf Min-Cost-Flow auf DAGs, mit nur einem subpolynomiellen Verlust in der Laufzeit. Als Konsequenz erhalten wir den ersten kombinatorischen Algorithmus in nahezu linearer Zeit für Min-Cost-Flow auf dichten Graphen und damit die erste kombinatorische Verbesserung gegenüber dem Goldberg–Tarjan-Algorithmus von 1987.

Zweitens entwickeln wir **Path Cover**, das von einem gerichteten Graphen G' , dessen stark zusammenhängende Komponenten kleinen Durchmesser haben, auf einen Eingabegraphen G projiziert und dabei alle kurzen Pfade abdeckt. Dies dient als deterministisches Gegenstück zum mächtigen Paradigma der Low-Diameter Decomposition und führt zum ersten deterministischen Algorithmus in nahezu linearer Zeit für kürzeste Wege mit negativen Kantengewichten, was die erste deterministische kombinatorische Verbesserung seit Goldbergs Algorithmus von 1995 darstellt.

Drittens schlagen wir **Common-Neighborhood Clustering** vor, das eine Vereinigung von Graphen, in denen Knotenpaare viele gemeinsame Nachbarn haben, auf einen Eingabegraphen G projiziert. Dies führt zum ersten deterministischen Algorithmus für den globalen minimalen Knotenschnitt, der die Lücke zwischen deterministischen und randomisierten Zeitkomplexitäten schließt, beantwortet eine von Gabow 2006 gestellte offene Frage und liefert neue Ergebnisse für Knotenzusammenhang in parallelem und verteiltem Rechnen.

Schließlich untersuchen wir **Steiner-Shortcuts**, die einen flachen Graphen G' auf den Eingabegraphen G projizieren und dabei Distanz-, Fluss- und Schnittinformationen erhalten. Wir zeigen, wie die Sichtweise der „Erweiterung des Graphen“ das Potenzial hat, die klassische Barriere für Shortcuts guter Qualität auf allgemeinen gerichteten Graphen zu umgehen, und diskutieren ihre weiterreichen-

den Konsequenzen.

Zusammen zeigen diese Ergebnisse, dass die Erweiterung eines Graphen zu einem größeren, aber stärker strukturierten Objekt ein wirkungsvolles algorithmisches Prinzip sein kann.

Publication List

During my doctoral studies, I coauthored 20 research works, including 15 papers accepted at peer-reviewed conferences and 5 manuscripts currently under submission. In all of these works, the authors are listed in alphabetical order, following the standard convention in theoretical computer science.

This section classifies the papers produced during my doctoral studies according to their role in the dissertation. The papers are divided into three categories: (1) papers that are fully included in the thesis; (2) papers that are partly included in the thesis; and (3) papers that were produced during my doctoral studies but are not included in the thesis.

For the papers in category (1), I was the main contributor to the results included in this dissertation. My contributions include the development of the main technical ideas, the proofs, and the writing in the technical parts. The coauthors contributed through discussions, feedback, refinement of the arguments, and revision of the manuscripts.

For the papers in category (2), I contributed to parts of the theorem-proving and writing. The material from these papers that is included in this dissertation corresponds to parts to which I made substantial contributions.

The papers in category (3) are listed for completeness. Although they were produced during my doctoral studies, they are not included in this dissertation because they are less directly connected to its central theme. This classification concerns only their role in the present dissertation: it should not be understood as a judgment of their scientific significance. Some of these works address substantial problems and are, in my view, important research contributions.

List of Papers Fully Included in the Thesis

- **Paper A** [HJS26a]
DAG Projections: Reducing Distance and Flow Problems to DAGs.
 Bernhard Haeupler, Yonggang Jiang, Thatchaphol Saranurak.
 STOC 2026.
- **Paper B**
Combinatorial Minimum-Cost Flow in Almost-Linear Time on Dense Graphs.
 Bernhard Haeupler, Yonggang Jiang, Thatchaphol Saranurak.
 Manuscript.
- **Paper C** [HJS25]
Deterministic Negative-Weight Shortest Paths in Nearly Linear Time via Path Covers.
 Bernhard Haeupler, Yonggang Jiang, Thatchaphol Saranurak.
 STOC 2026.

- **Paper D** [Bli+25]
Global vs. s-t Vertex Connectivity Beyond Sequential: Almost-Perfect Reductions & Near-Optimal Separations.
 Joakim Blikstad, Yonggang Jiang, Sagnik Mukhopadhyay, Sorrachai Yingchareonthawornchai.
 STOC 2025.
- **Paper E** [Jia+25b]
Deterministic Vertex Connectivity via Common-Neighborhood Clustering and Pseudorandomness.
 Yonggang Jiang, Chaitanya Nalam, Thatchaphol Saranurak, Sorrachai Yingchareonthawornchai.
 STOC 2025.

List of Papers Partly Included in the Thesis

- **Paper F** [Ber+25b]
Reviving Thorup’s Shortcut Conjecture.
 Aaron Bernstein, Henry Fleischmann, George Z. Li, Bernhard Haeupler, Gary Hoppenworth, Yonggang Jiang, Maximilian Probst Gutenberg, Seth Pettie, Thatchaphol Saranurak, Leon Schiller.
 STOC 2026.
- **Paper G** [Hae+25a]
Parallel $(1 + \epsilon)$ -Approximate Multi-Commodity Mincost Flow in Almost Optimal Depth and Work.
 Bernhard Haeupler, Yonggang Jiang, Yaowei Long, Thatchaphol Saranurak, Shengzhe Wang.
 FOCS 2025.

List of Omitted Papers

- **Paper H** [HJS26b]
Reducing Shortcut and Hopset Constructions to Shallow Graphs.
 Bernhard Haeupler, Yonggang Jiang, Thatchaphol Saranurak.
 SODA 2026.
- **Paper I** [Fis+26]
Directed and Undirected Vertex Connectivity Problems are Equivalent for Dense Graphs.
 Olivier Fischer, Yonggang Jiang, Sagnik Mukhopadhyay, Sorrachai Yingchareonthawornchai.
 SODA 2026.
- **Paper J** [JPP26]
New Oracles and Labeling Schemes for Vertex Cut Queries.
 Yonggang Jiang, Merav Parter, Asaf Petruschka.
 SODA 2026.
- **Paper K** [JNS26]
Minimum s-t Cuts with Fewer Cut Queries.
 Danupon Nanongkai, Yonggang Jiang, Pachara Sawettamalya.
 SODA 2026.
- **Paper L** [Cha+26a]
Shortcuts and Transitive-Closure Spanners Approximation.

Parinya Chalermsook, Yonggang Jiang, Sagnik Mukhopadhyay, Danupon Nanongkai.
SODA 2026.

- **Paper M** [FJY26]
Perfect Simulation of Las Vegas Algorithms via Local Computation.
Xinyu Fu, Yonggang Jiang, Yitong Yin.
ITCS 2026.
- **Paper N** [Bra+25]
Parallel Minimum Cost Flow in Near-Linear Work and Square Root Depth for Dense Instances.
Jan van den Brand, Hossein Gholizadeh, Yonggang Jiang, Tijn de Vos.
SPAA 2025 (Distinguished Paper Award).
- **Paper O** [Ash+24a]
Parallel and Distributed Exact Single-Source Shortest Paths with Negative Edge Weights.
Vikrant Ashvinkumar, Aaron Bernstein, Nairen Cao, Christoph Grunau, Bernhard Haeupler, Yonggang Jiang, Danupon Nanongkai, Hsin Hao Su.
ESA 2024.
- **Paper P** [JM23]
Finding a Small Vertex Cut on Distributed Networks.
Yonggang Jiang, Sagnik Mukhopadhyay.
STOC 2023.
- **Paper Q** [GJ25]
A Subquadratic Two-Party Communication Protocol for Minimum Cost Flow.
Hossein Gholizadeh, Yonggang Jiang.
Manuscript.
- **Paper R** [JY25]
Parallel Small Vertex Connectivity in Near Linear Work and Polylogarithmic Depth.
Yonggang Jiang, Changki Yun.
Manuscript.
- **Paper S** [Cha+26b]
The Complexity of Distributed Minimum Weight Cycle Approximation.
Yi-Jun Chang, Yanyu Chen, Dipan Dey, Yonggang Jiang, Gopinath Mishra, Hung Thuan Nguyen, Mingyang Yang.
Manuscript.
- **Paper T** [Jia+25a]
Approximating Directed Minimum Cut and Arborescence Packing via Directed Expander Hierarchies.
Yonggang Jiang, Yaowei Long, Thatchaphol Saranurak, Benyu Wang.
Manuscript.

Contents

Abstract	vii
Zusammenfassung	ix
Publication List	xi
I Synopsis	1
1 Introduction	3
2 DAG Projection	9
2.1 DAG Projection	9
2.2 Applications: Reducing Problems to DAGs.	10
2.3 Related Work on Removing Cycles	12
2.4 Existential Proofs	13
2.4.1 Distance DAG Projections	14
2.4.2 Flow DAG Projections	16
3 Combinatorial Minimum-Cost Flow	19
3.1 Our Results	19
3.2 Overview of the Algorithm	20
3.2.1 Exact-to-Approximation Reduction	20
3.2.2 Flow-Cost-Preserving DAG Projection	21
3.2.3 Weighted Push-Relabel Algorithm for MCF on DAGs	25
4 Path Cover	27
4.1 Path Cover	27
4.2 Application: Deterministic Negative-Weight SSSP	28
4.3 Related Work on LDD and Neighborhood Covers	29
4.4 Existential Proof	29
4.5 A Barrier for Covering Paths via Subgraphs	32
5 Common-Neighborhood Clustering	35
5.1 Common-Neighborhood Clustering	35
5.2 Applications to Minimum Vertex Cut	36
5.3 Construction	38

6	Steiner Shortcut	39
6.1	Introduction to Steiner Shortcut	39
6.2	Generalized Steiner Shortcuts	40
6.2.1	Exact Shortest Paths via Distance Shortcuts	42
6.2.2	Parallel Exact Max Flow via Congestion Shortcuts	43
II	Included Papers	47
A	DAG Projections: Reducing Distance and Flow Problems to DAGs	49
A.1	Introduction	51
A.1.1	Our Structural Results	52
A.1.2	New Landscape of Parallel Shortest Paths and Maximum Flow	53
A.1.3	Transferring Results from DAGs to Directed Graphs	54
A.1.4	Related Work	55
A.1.5	Organization	55
A.2	Overview	55
A.2.1	Distance DAG Projections	55
A.2.2	Congestion DAG Projections	58
A.3	Preliminaries	59
A.3.1	Distances and Shortest Paths	60
A.3.2	Flows and Cuts	61
A.4	DAG Projections and Their Applications	63
A.4.1	Distance DAG projections	63
A.4.2	Useful Tools for SSSP	64
A.4.3	Applications of Distance DAG Projections	65
A.4.4	Congestion DAG Projections	67
A.4.5	Applications of Congestion DAG Projections	68
A.5	Distance DAG Projection Construction	69
A.5.1	SSSP via DAG Projections for Smaller Length Constraints	70
A.5.2	DAG Projections via SSSP	72
A.5.3	Analysis: Approximation	73
A.5.4	Analysis: Size and Complexity	75
A.6	Congestion DAG Projection Construction	77
A.6.1	Expander Decomposition and Hierarchy	78
A.6.2	Congestion DAG Projections via Expander Hierarchy	79
A.6.3	Expander Hierarchy via Expander Decomposition	85
A.6.4	Expander Decomposition via MFMC Oracle	86
A.6.5	MFMC on General Graphs via MFMC on DAGs and DAG Projections . . .	92
A.6.6	Completing the Spiral	93
B	Combinatorial Minimum Cost Flow in Almost-Linear Time on Dense Graphs	95
B.1	Introduction	97
B.2	Overview	98
B.2.1	Exact-to-Approximation Reduction	99
B.2.2	Flow-Cost-Preserving DAG Projection	101
B.2.3	Weighted Push-Relabel Algorithm for MCF on DAGs	105
B.3	Preliminaries	106

B.3.1	Minimum cost flow.	107
B.3.2	Projection map.	108
B.4	Boosting Approximate Minimum Cost Flow	108
B.4.1	Exact Flow with Approximate Cost	109
B.4.2	Flow Boosting	112
B.4.3	Converting MCF solution: From MBCF to MCF	114
B.5	Flow-Cost-Preserving DAG Projections	116
B.5.1	Reducing PD-MCF on General Graphs to DAGs	118
B.6	Length Constrained Expander Hierarchy Construction	120
B.6.1	FatLDD for Round-Trip Distance	122
B.6.2	LC-Expander Decomposition	123
B.6.3	LC-Expander Hierarchy	130
B.7	DAG Projection Construction via LC Expander Hierarchy	134
B.7.1	An Algorithm for Constructing the DAG Projection	135
B.7.2	Size and Complexity of the Construction	138
B.7.3	Projection Algorithm for Flows	139
B.7.4	Projection Algorithm for Moving Cuts	145
B.7.5	Combine Everything	152
B.8	Minimum Cost Flow Algorithm on DAGs	155
B.8.1	Algorithm	156
B.8.2	Correctness	157
B.8.3	Complexity (Implementation Details)	161
B.9	Missing Proofs in Section B.6	164
B.9.1	Union of Sparse Moving Cuts	164
B.9.2	FatLDD on Directed Graphs	167
B.9.3	The Cutmatch Algorithm	171
B.10	Path Cover with Bounded Width	176
B.10.1	The Algorithm	176
B.10.2	The Global Potential	177
B.10.3	Correctness: Path Covering	180
B.10.4	Width, Size and Running time	181
B.10.5	Amplifying the covered length by concatenation	181
C	Deterministic Negative-Weight Shortest Paths in Nearly Linear Time via Path	
	Covers	183
C.1	Introduction	185
C.2	Overview	187
C.3	Preliminaries	189
C.4	Path Cover via a Clustered Graph	190
C.4.1	Definitions and the Main Theorem	190
C.4.2	The Algorithm	191
C.4.3	Correctness	193
C.4.4	Size and Time Complexity	195
C.5	Deterministic Negative Weight SSSP	198
C.5.1	An Algorithm for Restricted SSSP	198
C.5.2	Correctness	200
C.5.3	Time Complexity	202

C.6	Randomized or Undirected Solutions for Problem C.2	202
C.7	A Barrier for Covering Paths via Subgraphs	203
D	Global vs. s-t Vertex Connectivity Beyond Sequential: Almost-Perfect Reductions & Near-Optimal Separations	205
D.1	Introduction	207
D.1.1	Our Results: Reductions and Separations	208
D.1.2	Our Results: Implications	209
D.2	Technical Overview	210
D.2.1	Building the Framework: Common-neighborhood Cluster	211
D.2.2	A Decomposition Algorithm for Common-neighborhood Clustering	213
D.2.3	Distributed Algorithms	215
D.2.4	Parallel Algorithm in Matrix Multiplication Work	217
D.2.5	Communication Complexity	218
D.3	Preliminaries	219
D.3.1	Terminologies	219
D.3.2	Computational Models	220
D.3.3	Problem Definitions	220
D.3.4	Sketching	221
D.4	Parallel and Distributed Common-Neighborhood Clustering	222
D.5	A Schematic Reduction for Vertex Connectivity	226
D.5.1	The Reduction	226
D.5.2	Implementation in CONGEST and PRAM models	227
D.5.3	Cluster Sparsification	229
D.6	Isolating Cuts Lemma for Parallel and Distributed Algorithms (Proof of Lemma D.43)	231
D.7	Parallel and Distributed Algorithms for MinNNCC (Proof of Lemma D.44)	234
D.7.1	PRAM Algorithm	234
D.7.2	CONGEST Algorithm	235
D.8	Parallel Implementation in Matrix Multiplication Work	239
D.8.1	Proof of Lemma D.68	240
D.9	Communication and Streaming	243
D.9.1	A Simple Communication and Streaming algorithm	243
D.9.2	Lower Bounds	245
D.10	Reductions Between Problems and Proofs of Main Theorems	247
D.11	Bipartite Matching in CONGEST and PRAM	250
D.11.1	Parallel BMM	250
D.11.2	A CONGEST Algorithm	251
D.11.3	Generalizing a Semi-Streaming Algorithm	253
D.11.4	Solving Bipartite Matching on a Subgraph in CONGEST	256
D.12	Graph Sparsification	256
E	Deterministic Vertex Connectivity via Common-Neighborhood Clustering and Pseudorandomness	259
E.1	Introduction	261
E.1.1	Our Results	262
E.1.2	Our Tools	263
E.2	Preliminaries	266

E.3	An $\hat{O}(mn)$ -Time Algorithm for Weighted Directed Graphs	267
E.3.1	Overview	267
E.3.2	Lopsided Case: Proof of Lemma E.12	268
E.3.3	Symmetric Case: Proof of Lemma E.13	273
E.4	An $\hat{O}(m\kappa)$ -Time Algorithm for Unweighted Undirected Graphs	274
E.4.1	Organization and Overview	276
E.4.2	Terminal-Balanced Case: Proof of Lemma E.26	281
E.4.3	Faster Common-Neighborhood Clustering Algorithm	283
E.4.4	Oracle for the Unbalanced Case	284
E.4.5	Unbalanced Case: Proof of Lemma E.23	288
E.4.6	Terminal Reduction: Proof of Lemma E.25	289
E.5	Construction of Pseudorandom Objects	294
E.5.1	Crossing Family	294
E.5.2	Selectors	296
E.6	Exposition of Gabow's Algorithm	299
E.6.1	Ramanujan Expander	299
E.6.2	Overview of Gabow's algorithm	300
E.6.3	Exploiting a Large Gap	301
E.6.4	Gap Enlargement	304
E.7	Omitted Proofs from Section E.5.1	306
E.7.1	Proof of Lemma E.61	306
	Bibliography	309

Part I

Synopsis

CHAPTER 1

Introduction

Mathematics often advances by replacing a complicated object with a simpler one that preserves the structure of interest. A matrix may be understood through its spectrum or a low-rank approximation; a high-dimensional dataset through a lower-dimensional representation; a graph through an embedding or sparsifier. Such simplifications, often viewed as forms of *compression*, have driven major breakthroughs across theory and practice.

This thesis explores a complementary direction. Instead of compressing an object into a smaller representation, we may lift it to a larger but more structured auxiliary object, solve the problem there, and then project the solution back. This lift-and-project perspective appears in several forms across mathematics and theoretical computer science; see, e.g., resolution of singularities [Hir64a; Hir64b], graph lifts and covering graphs [ALM02], and extended formulations [KSZ20]. However, it has remained underexplored in algorithm design.

This thesis explores the lift-and-project perspective in fast graph algorithms. We develop new structural tools for graphs that are difficult to obtain from the classical compression viewpoint alone. These tools lead to new algorithms breaking several barriers for fundamental graph problems, including minimum-cost flow, negative-weight shortest paths, and minimum vertex cuts. More broadly, this thesis suggests that the lift-and-project perspective deserves greater attention in algorithm design and that it has the potential to become as influential as the compression-based viewpoint that has shaped algorithms over the past decades.

Below, we explain the precise meaning of this perspective in graphs and then briefly discuss the specific tools and their applications.

Lift-and-project perspective in graphs. Given an input graph $G = (V, E)$, we construct a structurally simpler graph $G' = (V', E')$ together with a projection map $\pi : V' \rightarrow V \cup \{\perp\}$. Equivalently, each vertex $v \in V$ may have multiple representatives in G' , namely the vertices in $\pi^{-1}(v)$. We call these vertices the *lifts* of v . Nodes in V' that are mapped to $\perp$ are called *Steiner vertices*; they do not correspond to any node in V . See Figure 1.1 and Figure 1.2 for illustrations.

Many classical graph techniques can be interpreted as special cases of this viewpoint in which the projection is *one-to-one*, and hence $|V'| = |V|$. For example, a *tree embedding* constructs a random tree G' such that the distance between every pair of vertices u, v in G' approximately preserves the distance between the corresponding vertices $\pi(u), \pi(v)$ in the input graph G (details in Section 2.3). Another example is *low-diameter decomposition*, where G' is a disjoint union of small diameter components on the same vertex set as G that preserves every short paths (details in Section 4.3). Other examples include spanners [Alt+93], sparsifiers [ST11], and expander

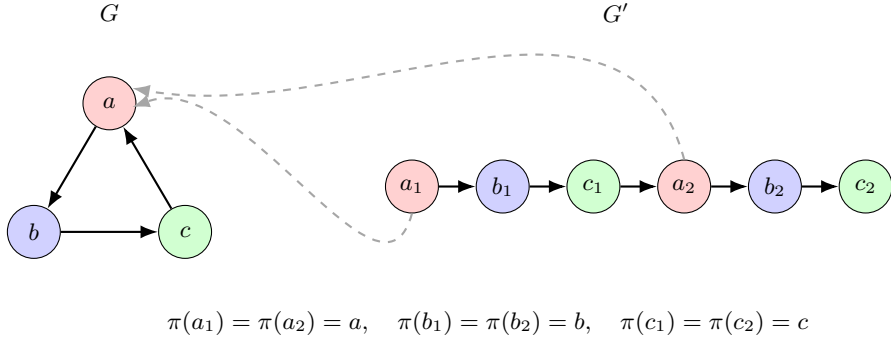


Figure 1.1: A graph projection where the directed cycle G is lifted to a directed path G' . The path unfolds the cycle while preserving all directed paths in the cycle. For example, the path $c \rightarrow a \rightarrow b$ in G is preserved as $c_1 \rightarrow a_2 \rightarrow b_2$ in G' . One can verify that every directed path in G is preserved in a similar way.

decompositions [SW19].

In contrast to the one-to-one mapping, this thesis explores the power of expanding the vertex set: a vertex of G may correspond to multiple vertices in G' , and G' may also contain additional Steiner vertices. We highlight the following techniques and applications developed from this viewpoint.

DAG projection (Chapter 2). In *DAG projection*, the graph G' is a directed acyclic graph (DAG) that approximately preserves distances, cuts, and flow information in the original graph G . DAG projections immediately transfer results on DAGs, usually simpler and more efficient, to directed graphs.

As one highlighted application, DAG projection yields a black-box reduction from exact minimum-cost flow on general graphs to its approximate version on DAGs, with only an $n^{o(1)}$ loss on the running time. This reduction allows future work on combinatorial minimum-cost flow to focus on DAGs, which are typically much easier to handle than general graphs. By combining this reduction with a simple augmenting-path-based approximation algorithm on DAGs, we give a combinatorial algorithm that computes an exact minimum-cost flow in $n^{2+o(1)}$ time. This is the first combinatorial improvement since Goldberg–Tarjan [GT87], and is almost optimal in dense graphs. All previous faster algorithms, including the almost-linear-time algorithm [Che+25], rely on interior-point methods. We discuss more details in Chapter 3.

DAG projections also reduce several major open problems on almost-optimal parallel and combinatorial algorithms for *exact* single-source shortest paths (SSSP) and maximum flow to easier settings:

1. exact directed SSSP to $(1 + 1/\text{polylog}(n))$ -approximate SSSP on DAGs; and
2. exact directed maximum flow to $n^{o(1)}$ -approximate maximum flow on DAGs.

Importantly, by combining the first reduction with a reduction from SSSP on DAGs to undirected graphs [HXX25], we show for the first time that parallel exact SSSP on directed and undirected graphs is *equivalent* up to subpolynomial factors.

Beyond SSSP and maximum flow, DAG projection is versatile enough to yield a broader range of applications: We improve the state of the art for $(1 + \epsilon)$ -approximate distance preservers [HXX25] and single-source minimum cut [CLL13], and obtain simpler constructions of $(n^{1/3}, \epsilon)$ -hopsets [KP22b; BW23] and combinatorial maximum-flow algorithms [Ber+24; Ber+25a].

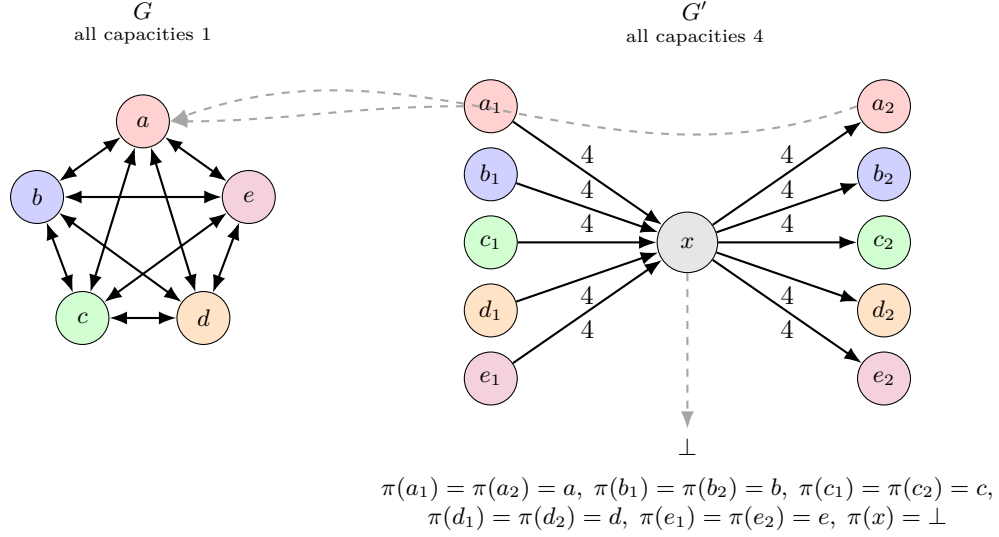


Figure 1.2: A graph projection with a Steiner node. The bidirected clique G has unit capacities, while every edge in G' has capacity 4. There is a Steiner node x , with $\pi(x) = \perp$. The flow values are preserved: For example, the maximum-flow value from a to b in G is 4, and this is exactly preserved by the maximum-flow value from a_1 to b_2 in G' . More generally, every two vertices in G have maximum-flow value 4, and these values are preserved exactly in G' .

Path Cover (Chapter 4). In *path covers*, the graph G' is a directed graph whose strongly connected components have small diameter, and G' covers all short paths of the input directed graph G .¹ Path covers play a role analogous to *neighborhood covers* in undirected graphs [Awe+98], which have long been central to derandomizing algorithms based on low-diameter decompositions in the undirected setting. We believe that path covers will serve as a useful tool for the design of future deterministic algorithms on directed graphs.

As one highlighted application, path cover leads to the first deterministic nearly-linear time algorithm for negative weight single-source shortest paths. All known near-linear time algorithms for this problem have been inherently randomized [BNW22; BCF23; Fis+25; LM24], as they crucially rely on low-diameter decompositions.

Path covers also have applications beyond derandomization. For example, they serve as an important subroutine in our combinatorial minimum-cost flow algorithm; see Section 3.2.2 for more details.

Common-neighborhood clustering (Chapter 5). In *common-neighborhood clustering*, the graph G' is a disjoint union of graphs in which every two vertices share many common neighbors, while preserving the unbalanced vertex cuts² of the original graph G . The motivation for this tool is twofold: first, the main difficulty in vertex-connectivity problems often lies in handling unbalanced vertex cuts; second, finding a minimum unbalanced vertex cut becomes easier in graphs where every two vertices have many common neighbors, as observed in [Li+21].

We show that this tool has cross-paradigm applications to minimum vertex-cut problems, with consequences for derandomization, parallel computing, and distributed computing.

- In parallel and distributed models, we obtain almost-tight reductions from global vertex connec-

¹The notion of “cover” is defined formally in Chapter 4. Informally, it means that every short path in G has a corresponding lift in G' , as illustrated in Figure 1.1.

²A vertex cut (L, S, R) is a partition of the vertex sets into disjoint sets L, S, R where L has no edges connected to R . An unbalanced vertex cut (L, S, R) is one in which L is much smaller than S .

tivity to its s - t version. This leads to an $n^{\omega+o(1)}$ -work³ and $n^{o(1)}$ -depth parallel algorithm for vertex connectivity, improving over the 35-year-old $\tilde{O}(n^{\omega+1})$ -work⁴, $O(\log^2 n)$ -depth algorithm of [LLW88]. In the distributed CONGEST model, the reduction implies the first sublinear-round algorithm for vertex connectivity when the diameter is moderately small.

- In the sequential model, we present a deterministic $O(m\kappa)$ -time algorithm for vertex connectivity, where $\kappa \leq n$ is the vertex connectivity. For moderate values of κ , this strictly improves over all previous deterministic algorithms for unweighted graphs, whose running times were $\hat{O}(m(n + \kappa^2))$ [Eve75]⁵, $\hat{O}(m(n + \kappa\sqrt{n}))$ [Gab06], and $\hat{O}(m2^{O(\kappa^2)})$ [SY22].
- In the sequential model, we give a deterministic algorithm for computing a global minimum vertex cut in a vertex-weighted graph in $mn^{1+o(1)}$ time. In dense graphs, this breaks the long-standing $\Omega(n^4)$ -time barrier achieved by the trivial approach of computing all-pairs maximum flows. Up to subpolynomial factors, our algorithm matches the fastest randomized $\tilde{O}(mn)$ -time algorithm of [HRG00], and affirmatively answers the question of [Gab06] of whether deterministic $O(mn)$ -time algorithms exist even for unweighted graphs. The algorithm also works for directed graphs.

Steiner shortcut (Chapter 6). Shortcutting is not new but a classical technique. Given a directed graph $G = (V, E)$, a reachability shortcut adds edges $E_+ \subseteq E^*$, where E^* is the transitive closure⁶ of G , producing a graph $G_+ = (V, E \cup E_+)$ that preserves reachability exactly but has much smaller reachability diameter.⁷ Equivalently, G_+ has the same transitive closure as G , but every reachable pair is connected by a short path in G_+ . From the viewpoint of graph projection, this is the special case where the projection map is one-to-one: the shortcut graph and the input graph have essentially the same vertex set.

This definition faces strong barriers. Thorup [Tho92] conjectured that every directed graph with m edges admits a reachability shortcut of near-linear size and polylogarithmic reachability diameter. More precisely, he asked whether one can add $\tilde{O}(m)$ shortcut edges so that every reachable pair has distance at most $\text{polylog}(n)$. This conjecture was refuted by Hesse [Hes03], and a long line of recent work has sharpened the corresponding lower bounds [HP21; Lu+22; BH23; VXX24; HSW25].

We revisit this barrier through the *lift-and-project* perspective. If shortcuts are viewed as projections from a shallow graph $G' = (V', E')$ that is *not* required to have the same vertex set as $G = (V, E)$, then the shortcut graph should be allowed to contain not only new shortcut edges, but also new *Steiner vertices*. This leads to the notion of *Steiner shortcuts*: shallow auxiliary graphs G' that project back to the input graph G while preserving reachability information.⁸

This change of viewpoint substantially alters the landscape. We give explicit Steiner-shortcut constructions that bypass all known shortcut lower-bound instances [Hes03; HP21; Lu+22; BH23; VXX24; HSW25]. We also propose a candidate hard instance as a next step toward refuting the Steiner version of Thorup’s conjecture. See **Paper F** for more details.

Beyond reachability, we also develop shortcuts that preserve flow and cost information. On undirected graphs, these shortcuts lead to almost-optimal parallel algorithms for $(1 + \epsilon)$ -approximate minimum-cost flow with edge and vertex capacities and costs; see **Paper G** for more details. Whether

³Here n^ω denotes the work of matrix multiplication, which can be achieved with $n^{o(1)}$ depth. Matrix-multiplication algorithms are typically highly parallelizable.

⁴We use $\tilde{O}(\cdot)$ to hide polylogarithmic factors throughout this thesis.

⁵We use $\hat{O}(\cdot)$ to hide subpolynomial factors throughout the thesis.

⁶A transitive closure E^* has an edge $(u, v) \in E^*$ iff. there is a path from u to v in G , i.e., u can reach v in G .

⁷Here the reachability diameter means $\max_{(u,v) \in E^*} \text{dist}_{G_+}(u, v)$, where distance assume edges have unit lengths.

⁸We do not emphasize “many-to-one” mapping from G' to G here because for this shortcut problem, lifting each vertex in G to multiple vertices in G' is equivalent to lifting to one vertex. The emphasis is on expanding the graph G by Steiner vertices, which gives the name “Steiner shortcut”. We will discuss more in **Chapter 6**.

similar results can be obtained for directed graphs using Steiner shortcuts remains open, and we discuss this further in [Chapter 6](#).

CHAPTER 2

DAG Projection

Cycles are a central obstacle in directed graph algorithms. In an acyclic graph, algorithms can be designed much more easily: trees can be processed bottom-up, and DAGs admit topological orderings that enable dynamic programming. Thus, one of the most appealing ways to simplify a graph is to remove its cycles. In this chapter, we will show how to represent an arbitrary directed graph by a DAG while preserving distance, flow, and cut information.

[Section 2.1](#) formally defines DAG projection. [Section 2.2](#) discusses its applications. [Section 2.3](#) surveys related work on removing cycles from graphs, both directed and undirected. Finally, [Section 2.4](#) gives an overview of the existential construction of DAG projections. This chapter is based on [Paper A](#).

2.1 DAG Projection

We show that, given any directed graph $G = (V, E)$, we can construct an almost-linear-size DAG $G' = (V', E')$, together with a projection map $\pi : V' \rightarrow V \cup \{\perp\}$, which approximately preserves both distance and flow information in G in the following sense.

For $u, v \in V$, we use $\text{dist}_G(u, v)$ and $\text{maxflow}_G(u, v)$ to denote the distance and maximum-flow value, respectively, from u to v in G . For disjoint sets $S, T \subseteq V'$, let $\text{dist}_{G'}(S, T)$ be the distance from S to T after contracting S and T into single vertices. Similarly, let $\text{maxflow}_{G'}(S, T)$ denote the maximum-flow value, equivalently the minimum-cut value, from S to T after contracting S and T into single vertices. The DAG G' has the following guarantees: for every pair of vertices $u, v \in V$, the distance and maximum-flow value from $\pi^{-1}(u)$ to $\pi^{-1}(v)$ in G' approximate the corresponding quantities from u to v in G .

Theorem 2.1. *For every directed graph $G = (V, E)$ with edge lengths and capacities, there exists a DAG $G' = (V', E')$ together with $\pi : V' \rightarrow V \cup \{\perp\}$ such that $|E'| = |E|^{1+o(1)}$, $|V'| = |V|^{1+o(1)}$, and*

$(1 + \varepsilon)$ -distance-preserving: *For every $u, v \in V$, we have*

$$\text{dist}_G(u, v) \leq \text{dist}_{G'}(\pi^{-1}(u), \pi^{-1}(v)) \leq (1 + \varepsilon) \cdot \text{dist}_G(u, v). \quad (2.1)$$

$n^{o(1)}$ -flow-preserving: *For every $u, v \in V$, we have*

$$\text{maxflow}_G(u, v) \leq \text{maxflow}_{G'}(\pi^{-1}(u), \pi^{-1}(v)) \leq n^{o(1)} \cdot \text{maxflow}_G(u, v). \quad (2.2)$$

Distance	From $\rightarrow$ To	Approximation	Total size
[MN07]	undirected $\rightarrow$ tree	$O(k)$	$O(kn^{1+1/k})$
[BFN19]	undirected $\rightarrow$ tree	$O(n^{1/k} \log^{1-1/k} n)$	nk
[AHW25]	directed $\rightarrow$ DAG	$O(\log^3 n \log \log n)$	$O(m \log^3 n)$
[Fil26]	directed $\rightarrow$ DAG	$O(\log n \log \log n)$	$O(m \log^3 n)$
Ours (Theorem 2.1)	directed $\rightarrow$ DAG	$(1 + 1/\text{polylog}(n))$	$m^{1+o(1)}$

Maximum flow	From $\rightarrow$ To	Approximation	Total size
[Rac02]	undirected $\rightarrow$ tree	$O(\log^3 n)$	$2n$
[HHR03]	undirected $\rightarrow$ tree	$O(\log^2 n \log \log n)$	$2n$
[Rac08]	undirected $\rightarrow$ tree	$O(\log n)$	$O(mn)$
[RS14]	undirected $\rightarrow$ tree	$O(\log n \log \log n)$	$2n$
Ours (Theorem 2.1)	directed $\rightarrow$ DAG	$n^{o(1)}$	$m^{1+o(1)}$

Table 2.1: Summary of results on approximating undirected and directed graphs by trees and DAGs, respectively. For distances, we trade $O(m \log^3 n)$ size and $\Omega(\log n)$ approximation for $m^{1+o(1)}$ size and $(1 + o(1))$ approximation. For maximum flow, we give the first such DAG.

Moreover, if we only want $(1 + \varepsilon)$ -distance-preserving, the projection can be made to be a strict projection: for every $(u', v') \in E'$, we have $(\pi(u'), \pi(v')) \in E$.

The existential statement is already non-trivial. In fact, many works studied compact DAG representations of arbitrary directed graphs, but they were limited to much larger approximation factors, mainly because the projections considered there were one-to-one; see, e.g., [AHW25]. Our result shows that the lift-and-project viewpoint provides a way to overcome this barrier. See Table 2.1 for a summary of previous results on removing cycles of undirected or directed graphs.

As two concrete examples, observe that Figure 1.1 gives a DAG projection that preserves distances exactly, when all edges in the figure have length 1. Similarly, Figure 1.2 gives a DAG projection that preserves flow: if all edges in G have capacity 1 and all edges in G' have capacity 4, then every two vertices in G have maximum-flow value 4, and this value is preserved exactly in G' .

2.2 Applications: Reducing Problems to DAGs.

Besides the existential results, we also give fast algorithms for constructing G' and π , both in the sequential and parallel models. To discuss parallel computing, we recall that parallel algorithms are measured by two parameters: (1) *work*, which is the total number of operations performed by the algorithm, and (2) *depth*, which is the length of the longest dependency chain in the computation.

Theorem 2.2. *If $(1 + \varepsilon)$ -approximate SSSP, respectively $n^{o(1)}$ -approximate max flow, can be solved in $W(m, n)$ work and $D(m, n)$ depth on DAGs, then the DAG projection in Theorem 2.1 can be constructed in $n^{o(1)} \cdot W(m, n)$ work and $n^{o(1)} \cdot D(m, n)$ depth on general directed graphs.*

In particular, the DAG projection in Theorem 2.1 can be constructed in almost-linear work and $n^{0.5+o(1)}$ depth.

Theorem 2.2 leads to many algorithmic applications, which we discuss below.

New Landscape of Parallel Shortest Paths and Maximum Flow. We give the following reductions for single-source shortest paths (SSSP) and maximum flow. These results show that, when the goal is to obtain polynomial improvements, it is enough to focus on DAGs.

Theorem 2.3. *If $(1 + \varepsilon)$ -approximate SSSP (or $n^{o(1)}$ -approximate max flow) can be solved in $W(m, n)$ work and $D(m, n)$ depth on DAGs, then exact SSSP (or exact max flow), can be solved in $n^{o(1)} \cdot W(m, n)$ work and $n^{o(1)} \cdot D(m, n)$ depth on general directed graphs.*

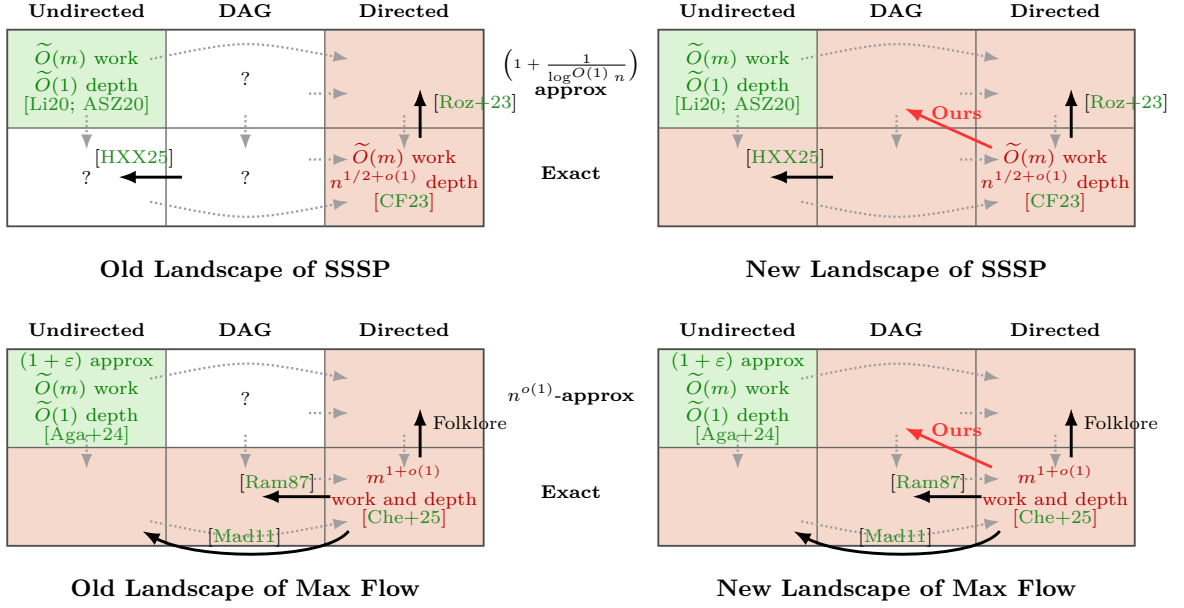


Figure 2.1: Old and new landscapes of parallel SSSP and maximum flow algorithms. The green area highlights the settings where near-optimal parallel algorithms are known. The red area highlights the settings as hard as the exact directed setting. The solid arrows represent non-trivial efficient parallel reductions, while the dotted arrows represent trivial ones.

Theorem 2.3 can be proved by the following reduction chains.

1. Reduce exact SSSP (or exact max flow) on general digraphs to $(1 + \varepsilon)$ -approximate SSSP (or $n^{o(1)}$ -approximate max flow) on general digraphs using the existing exact-to-approximate reduction framework [Roz+23];
2. Reduce $(1 + \varepsilon)$ -approximate SSSP (or $n^{o(1)}$ -approximate max flow) on general digraphs to $(1 + \varepsilon)$ -approximate SSSP (or $n^{o(1)}$ -approximate max flow) on DAGs, using our DAG projection **Theorem 2.1**.

Thus, we successfully reduce major open problems of finding parallel algorithms for *exact* single-source shortest paths (SSSP) and maximum flow to easier settings, leading to a clean landscape of both problems: Consider algorithms in the six settings based on the following combinations: (1) exact or approximate, and (2) on directed graphs, DAGs, or undirected graphs. Our reductions categorize these six settings for both SSSP and maximum flow into only two regimes. See **Figure 2.1**. The *easy* regime consists of only the approximate undirected setting, and the *hard* regime contains the remaining five settings. These five settings are equivalent in the sense they all admit algorithms with the same work and depth up to subpolynomial factor.

This significantly cleans up the landscape of both problems. With this new landscape, to improve the state-of-the-art of either exact directed SSSP or maximum flow, it suffices to improve either (1) approximation algorithms on DAGs, or (2) exact algorithms on undirected graphs.

Other applications. Beyond SSSP and maximum flow, DAG projection also reduces other fundamental problems on general directed graphs to their counterparts on DAGs, including hopsets, distance preservers, and bounded single-source maximum flow.

Distance preservers: improved. Given a directed graph $G = (V, E)$ with edge lengths and a set of demand pairs $P \subseteq V \times V$, a $(1 + \varepsilon)$ -approximate distance preserver $((1 + \varepsilon)$ -DP) is a

subgraph $H \subseteq G$ such that for every $(s, t) \in P$, we have $\text{dist}_H(s, t) \leq (1 + \varepsilon) \cdot \text{dist}_G(s, t)$. A long line of work (e.g. [CE06; Bod17; HXX25]) have been trying to find $(1 + \varepsilon)$ -DPs of smallest size $|E(H)|$.

Currently, the best bounds for DAGs are strictly better than the ones for general directed graphs. To simplify the discussion, we focus on the $O(n)$ -size regime. There exists a $(1 + \varepsilon)$ -DP for DAGs of size $O(n + p\sqrt{n})$, which gives $O(n)$ as long as $p = O(\sqrt{n})$ [CE06; HXX25]. However, the best bound for general directed graphs achieves $O(n)$ size only when $p = O(n^{1/3})$ [Bod17]. Our DAG projection **Theorem 2.1** closes this gap up to $n^{o(1)}$ factor.

Corollary 2.4. *For $\varepsilon \geq 1/\text{polylog}(n)$, there exists a $(1 + \varepsilon)$ -approximate distance preserver for any n -node directed graph with polynomial edge lengths and p demand pairs of size $(n + p\sqrt{n}) \cdot n^{o(1)}$, which is $n^{1+o(1)}$ for $p = O(\sqrt{n})$.*

Hop-set: simplified. Given a directed graph $G = (V, E)$ with edge lengths, a (β, ε) -hop-set of G is a set of additional edges $H \subseteq V \times V$ added to the graph so that for every $s, t \in V$ we have (i) $\text{dist}_G(s, t) \leq \text{dist}_{G \cup H}(s, t)$, and (2) $\text{dist}_{G \cup H}^{(\beta)}(s, t) \leq (1 + \varepsilon) \cdot \text{dist}_G(s, t)$ where $\text{dist}_{G \cup H}^{(\beta)}(s, t)$ denotes the lengths of the shortest path from s to t using at most β edges.

The breakthrough result of Kogan and Parter [KP22b] showed the existence of linear-sized $(O(n^{1/3}), \varepsilon)$ -hop-sets for DAGs, but only linear-sized $(O(n^{2/5}), \varepsilon)$ -hop-sets for general graphs. With significant effort, Bernstein and Wein [BW23] closed this gap by showing linear-sized $(O(n^{1/3}), \varepsilon)$ -hop-sets for any graph.

Theorem 2.1 can bypass this significant effort and close the gap up to $n^{o(1)}$ factor in a black-box way. Indeed, by applying the construction of [KP22b] for DAGs on top of our distance-preserving DAG projection, we immediately obtain $n^{1+o(1)}$ -sized $(O(n^{1/3}), \varepsilon)$ -hop-sets.

Bounded Single-Source Max flow: improved. Let G be a graph with positive integral edge capacities and source $s \in V(G)$. Let $\text{maxflow}_G^k(s, t) = \min\{k, \text{maxflow}_G(s, t)\}$ denote the k -bounded maximum flow value from s to t . If G is a DAG, the algorithm based on network coding by [CLL13] can compute $\text{maxflow}_G^k(s, t)$ for all $t \in V(G)$ in $O(k^{\omega-1}m)$ total time, while the best known algorithm on general graphs is to trivially compute $\text{maxflow}_G^k(s, t)$ for every t using $\Omega(mn)$ time.

Using **Theorem 2.1**, we obtain a non-trivial algorithm that beats the mn bound when $k \in (n^{\Omega(1)}, n^{1/\omega-\Omega(1)})$.

Theorem 2.5. *There is a randomized algorithm that, given a directed graph with positive integral edge capacities, a source s , and an integer k , computes $n^{o(1)}$ -approximation of $\text{maxflow}_G^k(s, t)$ for all vertices t in $k^\omega m^{1+o(1)}$ time.*

These results illustrate the broad impact of DAG projection, and we believe that the applications listed here are far from exhaustive.

2.3 Related Work on Removing Cycles

The question of removing cycles has a long history, starting from *tree embeddings* on undirected graphs.

Undirected graphs. On undirected graphs, this idea has been remarkably successful through the literature on *tree embeddings*. A *tree embedding* of a graph G is a collection $\mathcal{T}$ of trees on the same

vertex set such that, for every pair of vertices $s, t \in V(G)$,

$$\text{dist}_G(s, t) \leq \min_{T \in \mathcal{T}} \text{dist}_T(s, t) \leq \alpha \cdot \text{dist}_G(s, t),$$

where α is called the *length slack*. A remarkable result of Mendel and Naor [MN07] shows that length slack $O(\log n)$ is achievable using only $O(\log n)$ trees. A stronger slack of $(1 + \varepsilon)$ can be achieved by changing the definition to *clan embeddings*¹, introduced by Filtser and Le [FL21]. There is also a literature on *probabilistic tree embeddings*, which embed the shortest-path metric of a graph into a distribution over tree metrics with small expected distortion [Bar96b; FRT04].

Beyond distances, there is another line of work on preserving cut and flow structure by trees. For example, Mendel and Naor [MN07] show that every undirected graph $G = (V, E)$ admits a tree T such that the minimum cut, equivalently the maximum-flow value, between every two disjoint vertex sets is preserved up to an $O(\log n \log \log n)$ approximation factor.

Directed analogue. A natural question is whether an analogous form of cycle removal is possible for directed graphs. The direct analogue of replacing an undirected graph by trees is to replace an arbitrary directed graph by a small collection of DAGs.

The distance version of this problem has received increasing attention in recent years [AHW25; Fil26; Bho+26]. Formally, it asks for a collection of DAGs $D_1, \dots, D_g$ on the same vertex set as G such that, for every pair of vertices (s, t) ,

$$\text{dist}_G(s, t) \leq \min_i \text{dist}_{D_i}(s, t) \leq \lambda \cdot \text{dist}_G(s, t).$$

This line of work [AHW25; Fil26; Bho+26] shows that length slack $\lambda = \tilde{O}(\log n)$ is achievable using $O(\log n)$ DAGs, while $(1 + \varepsilon)$ slack is known only for certain restricted graph classes. Whether $(1 + \varepsilon)$ length slack is possible for general directed graphs remains open and was conjectured to be impossible in [AHW25].

Notice that this line of work follows the viewpoint of simplifying a general digraph by a collection of DAGs on the *same vertex set*, which is exactly why obtaining $(1 + \varepsilon)$ length slack is so hard. Our contribution on DAG projection suggests that the lift-and-project viewpoint is the right way to circumvent this barrier.

Besides DAGs that preserve distances, no analogous result was known for preserving flow or cut structure in directed graphs before our work.

2.4 Existential Proofs

In this section, we sketch the proof of [Theorem 2.1](#). In [Section 2.4.1](#), we present a construction with the $(1 + \varepsilon)$ -distance-preserving property. In [Section 2.4.2](#), we present a construction with the $n^{o(1)}$ -flow-preserving property.

Although [Theorem 2.1](#) is stated as preserving both distances and flow, the two guarantees can be obtained separately and then combined by taking the disjoint union of the two auxiliary graphs.

¹A clan embedding does not create a collection of trees, but instead creates a single tree T on a new vertex set V' together with a one-to-many map from V to V' . This is similar in spirit to lift-and-project, but was not previously extended to other settings.

2.4.1 Distance DAG Projections

Remember that a strict DAG projection from $G' = (V', E')$ to $G = (V, E)$ using $\pi : V' \rightarrow V$ satisfies that for every $(u', v') \in E'$, we have $(\pi(u'), \pi(v')) \in E$.

Given a directed graph $G = (V, E)$ with edge lengths $\ell_G : E \rightarrow \mathbb{N}_{\geq 1}$, the goal is to construct a $(1 + \varepsilon)$ -distance-preserving DAG projection D onto G . For convenience, when we say that D *preserves the (s, t) -distance* for $s, t \in V(G)$, we mean: there exist $s', t' \in V(D)$ with $\pi(s') = s$ and $\pi(t') = t$ such that

$$\text{dist}_G(s, t) \leq \text{dist}_D(\pi^{-1}(s), \pi^{-1}(t)) \leq (1 + \varepsilon) \cdot \text{dist}_G(s, t).$$

We will use the following directed low-diameter decomposition (LDD) as a subroutine.

Lemma 2.6 ([BCF23; BNW22]). *Let $G = (V, E)$ be a directed graph with edge lengths and let d be a positive integer. There exists a random set of edges E^{rem} (a low-diameter decomposition) satisfying:*

- *each SCC of $G - E^{\text{rem}}$ has diameter at most d ,*
- *for every $e \in E$, $\Pr[e \in E^{\text{rem}}] \leq s_{\text{LDD}} \cdot \ell_G(e)/d$,*

where $s_{\text{LDD}} = \tilde{O}(1)$.

Now, the construction is as follows.

Step 1 (LDD). Compute LDDs of G with diameters 2^i for all $0 \leq i \leq O(\log n)$. Fix one such LDD for a given i with parameter $d = 2^i$, yielding an edge set E^{rem} as in Lemma 2.6. We show how to build a DAG projection that preserves (s, t) -distances in G whenever $\text{dist}_G(s, t)$ is on the order of d/ε . The final projection D will be the union of the $O(\log n)$ projections over all i , thereby handling all pairs (we assume edge lengths are polynomial on n).

Let $\mathcal{C}$ be the family of SCCs of $G - E^{\text{rem}}$. By construction, each $C \in \mathcal{C}$ has $\text{diam}(G[C]) \leq d$. Let $\sigma = n^{o(1)}$ (to be fixed later), and classify clusters by size:

- *large* if $|C| \geq n/\sigma$,
- *small* if $|C| < n/\sigma$.

Step 2 (recursive construction for small clusters). For each small $C \in \mathcal{C}$, recursively construct a DAG projection D_C of $G[C]$ that $(1 + \varepsilon)$ -approximately preserves distances for *all* pairs of vertices in $G[C]$ (instead of just length bounded pairs).

Step 3 (shortest-path trees for large clusters). For each large $C \in \mathcal{C}$, pick an arbitrary root $r_C \in C$. In G , compute a shortest-path tree T_C^{out} rooted at r_C and a reversed shortest-path tree T_C^{in} rooted at r_C (i.e., every (s, r_C) -path in T_C^{in} is a shortest (s, r_C) -path). Define D_C to be the DAG projection that combines T_C^{out} and T_C^{in} (as *disjoint copies* of subgraphs of G) by merging the common root r_C .

Step 4 (combining everything). Let $\mathcal{C} = (C_1, \dots, C_z)$ be a topological ordering of the SCCs in $G - E^{\text{rem}}$. Form D' by concatenating the D_{C_i} using copies of edges in G : for every $1 \leq i < j \leq z$, every $u' \in V(D_{C_i})$, and every $v' \in V(D_{C_j})$, if $(\pi(u'), \pi(v')) \in E(G)$ then add the edge (u', v') to D' .

Finally, let $x = s_{\text{LDD}}/\varepsilon$. Create x copies $D'_1, \dots, D'_x$ of D' and concatenate them in the same manner: for every $1 \leq i < j \leq x$, $u' \in V(D'_i)$, and $v' \in V(D'_j)$, if $(\pi(u'), \pi(v')) \in E(G)$ then add (u', v') . The resulting DAG is D .

Distance preservation. It is easy to see that

$$\text{dist}_G(s, t) \leq \text{dist}_D(\pi^{-1}(s), \pi^{-1}(t))$$

since π is a (weight-preserving) graph homomorphism (the only edges we added to D are in Step 3 and 4, which are copies of edges in the original graph): Every path in D connecting a vertex in $\pi^{-1}(s)$ to a vertex in $\pi^{-1}(t)$ can be mapped to a path in G connecting s to t with the same length. Hence the distance in G is at most the distance in D .

In what remains, we will only prove the harder direction

$$\text{dist}_D(\pi^{-1}(s), \pi^{-1}(t)) \leq (1 + \varepsilon) \cdot \text{dist}_G(s, t).$$

Fix $s, t \in V(G)$ with $\text{dist}_G(s, t) = d/\varepsilon$. Let p be an $s \rightarrow t$ path in G of length $\ell_G(p) = d/\varepsilon$. By [Lemma 2.6](#), in expectation there are $s_{\text{LDD}}/\varepsilon$ edges of p lie in E^{rem} . Hence we can partition p into subpaths $p_1, p_2, \dots$, each entirely contained in $G - E^{\text{rem}}$. Notice that edges connecting p_i to p_{i+1} is preserved in D according to Step 4.

Let $a, b \in V$ be the endpoints of some subpath p_i . Let p' be a shortest $a \rightarrow b$ path in $G - E^{\text{rem}}$. Decompose p' into maximal subpaths $p'_1, \dots, p'_y$, so that each p'_j lies within a single SCC of $G - E^{\text{rem}}$. Notice that edges connecting p'_i to p'_{i+1} is preserved in D' according to Step 4. If that SCC is a small cluster, then the endpoints' distance is preserved by the recursive construction. Otherwise, it is a large cluster.

Let C_L be the first large cluster meeting p (not p' , we switch our attention to the whole path p) and, similarly, C_R be the last. Let $\tilde{p}$ be the subpath of p from its first vertex in C_L to its last vertex in C_R . Denote these endpoints by $\text{Start}(\tilde{p})$ and $\text{End}(\tilde{p})$. By the definition of D_C for a large cluster C , D_C contains T_C^{in} followed by T_C^{out} , both rooted at r_C . Thus a copy of $\text{Start}(\tilde{p})$ in $T_{C_L}^{\text{in}}$ reaches r_{C_L} within distance at most d (since $G[C_L]$ has diameter d and $T_{C_L}^{\text{in}}$ is a shortest path tree), and from r_{C_L} we can reach a copy of $\text{End}(\tilde{p})$ within distance $\ell_G(\tilde{p})$ plus at most d additional length via $T_{C_R}^{\text{out}}$ (this is because the original distance from C_L to $\text{End}(\tilde{p})$ is at most $d + \ell_G(\tilde{p})$: from C_L we can reach $\text{Start}(\tilde{p})$ within distance d , then we follow $\tilde{p}$ to reach $\text{End}(\tilde{p})$). Hence, the total distance is at most $\ell_G(\tilde{p}) + 2d$, which has an additive $O(d)$ overhead. Since $\text{dist}_G(s, t) = d/\varepsilon$, this additive term is negligible and absorbed in the $(1 + \Theta(\varepsilon))$ multiplicative guarantee.

Size of the projection. If we write $f(n)$ as the width (maximum number of copies) of the DAG projection returned by the above construction on an n -vertex graph, then $f(n)$ can be calculated as follows.

- (1) In Step 1 and Step 4, we know that D contains $x = \tilde{O}(1/\varepsilon)$ copies of D' .
- (2) In Step 2, each vertex is copied $f(n/\sigma)$ times in D' .
- (3) In Step 3, each vertex is copied $2 \cdot \sigma^2$ times in D' , where the factor 2 comes from the forward and reversed shortest path trees, and the factor σ^2 comes from the fact that there are at most σ large clusters (because each large cluster has at least n/σ vertices).

To summarize, we get the following recursive inequality:

$$f(n) \leq \tilde{O}(1/\varepsilon) \cdot (f(n/\sigma) + 2\sigma^2).$$

Notice that $\varepsilon = 1/\text{polylog}(n)$ according to our assumption. By choosing an appropriate parameter $\sigma = n^{o(1)}$, the desired bound $f(n) = n^{o(1)}$ follows.

Algorithmic Aspect. The algorithm above runs in $m^{1+o(1)}$ time; it takes near-linear time in the size of the output, which is $m^{1+o(1)}$, because we simply call LDD which can be computed using SSSP as a subroutine.

In the parallel setting, our goal is to reduce DAG projection construction to approximate SSSP on DAGs. But, currently, the algorithm needs to call as a subroutine the SSSP algorithm on general graphs, which does not admit efficient parallel algorithms yet. Our strategy is to reduce SSSP on general graphs to SSSP on DAGs using our distance DAG projection itself. This leads to the ‘chicken-and-egg’ situation: (When we are given an SSSP on DAGs oracle), we can reduce distance DAG projection to SSSP on general graphs, and we can reduce SSSP on general graphs to distance DAG projection.

Our strategy (detailed in [Paper A](#)) is to find a spiral recursion: We define h -length DAG projection that only preserves distances up to h , and we reduce it to $h/n^{o(1)}$ -length SSSP on general graphs which computes distance up to $h/n^{o(1)}$, reducing to $h/n^{o(1)}$ -length DAG projection. This spiral recursion will finally make h small enough so the construction is trivial.

2.4.2 Flow DAG Projections

Given a graph $G = (V, E)$ with edge capacities denoted by $U_G : E \rightarrow \mathbb{R}_{\geq 0}$, we will show how to construct a $n^{o(1)}$ -flow-preserving DAG projection.

We consider *expander decomposition* as the analogue of LDD in the flow setting. We first give some necessary definitions.

For a flow f , we represent the demand routed by f as a pair of vectors (Δ, ∇) , where $\Delta, \nabla : V \rightarrow \mathbb{R}_{\geq 0}$ denote the source demands and sink demands routed by f , respectively.

We say a demand (Δ, ∇) is **d-respecting** for some function $\mathbf{d} : V \rightarrow \mathbb{R}_{\geq 0}$ if $\Delta(v), \nabla(v) \leq \mathbf{d}(v)$ for every $v \in V$. For an edge set $E' \subseteq E$, define

$$\text{vol}_{E'}(v) = \sum \{ U_G(u, v) \mid (u, v) \in E' \text{ or } (v, u) \in E' \},$$

and for a vertex set $C \subseteq V$ write $\text{vol}_{E'}|_C$ for the restriction of function $\text{vol}_{E'}$ to C .

A flow has congestion κ if the flow value on each edge is at most κ times the edge’s capacity.

Definition 2.7 (Terminal Expanding). Given a directed graph $G = (V, E)$ with edge capacities and a function $\mathbf{d} : V \rightarrow \mathbb{R}_{\geq 0}$, we say $\mathbf{d}$ is ϕ -*expanding* on G if every $\mathbf{d}$ -respecting demand is routable via a flow f in G with congestion at most $1/\phi$.

We next formalize the hierarchical structure we will use.

Definition 2.8 (Expander Hierarchy). Given a directed graph $G = (V, E)$ with edge capacities, an ϕ -*expander hierarchy* with t layers consists of edge sets $E_1, \dots, E_t$ with $E_1 = E$ and $E_t = \emptyset$. Let $E_{>i} = \bigcup_{t \geq j > i} E_j$. For every $1 \leq i < t$ and every strongly connected component (SCC) C of $G - E_{>i}$, the function $\text{vol}_{E_i}|_C$ is ϕ -expanding in $G[C]$.

By [\[Ber+24\]](#), such a hierarchy exists with expansion $\phi = 2^{-O(\sqrt{\log n})}$ and $t = \sqrt{\log n}$.

Constructing the DAG projection D. Let $E_1, \dots, E_t$ be the expander hierarchy of G with expansion $\phi = 2^{-O(\sqrt{\log n})}$ and $t = \sqrt{\log n}$. The construction proceeds recursively from top to bottom using the hierarchy. We will only describe the top level.

Fix an SCC C of G . By definition, $\text{vol}_{E_{t-1}}|_C$ is ϕ -expanding in $G[C]$. Let D_C be a DAG projection of $G[C] - E_{t-1}$ obtained *recursively*. We build a DAG projection D'_C of $G[C]$ similar to [Figure 1.2](#) as follows:

- Create two disjoint copies $D_C^{(1)}$ and $D_C^{(2)}$ of D_C , and add a dummy vertex w_C .
- For every $v \in V(D_C^{(1)})$, add an edge (v, w_C) with capacity $\text{vol}_{E_{t-1}}(\pi(v))$.
- For every $v \in V(D_C^{(2)})$, add an edge (w_C, v) with capacity $\text{vol}_{E_{t-1}}(\pi(v))$.

Here π denotes the projection map from the vertices of D_C to G .

Let $C_1, \dots, C_z$ be the SCCs of G in a topological order. We obtain D by concatenating the graphs $D'_{C_1}, \dots, D'_{C_z}$: for every $1 \leq i < j \leq z$ and for every $u \in V(D'_{C_i}), v \in V(D'_{C_j})$, add an edge (u, v) in D whenever $(\pi(u), \pi(v)) \in E(G)$.

Size of D . There are $t = O(\sqrt{\log n})$ layers, and in each layer we make two copies of every vertex in the next layer. Hence, every vertex of G (and every dummy vertex w_C) is copied at most $2^t = n^{o(1)}$ times overall. For edges, we only add edges (u, v) in D if $(\pi(u), \pi(v)) \in E$ or one of u, v is a dummy vertex. Since the number of vertex copies is $n^{o(1)}$, it follows that $|E(D)| \leq m^{1+o(1)}$.

Correctness. We first show that $\text{maxflow}_G(s, t) \leq \text{maxflow}_{G'}(\pi^{-1}(s), \pi^{-1}(t))$. Given a flow f in G from s to t , we show how to route f in D from $\pi^{-1}(s)$ to $\pi^{-1}(t)$ with the same value. Because we concatenate $D'_{C_1}, \dots, D'_{C_z}$ using all edges of G respecting the topological order, it suffices to show that the restriction f_C of f to a fixed SCC C can be routed in D'_C . Consider a flow path p of f_C ; we decompose it into three segments:

1. The prefix of p before it uses any edge of E_{t-1} . This segment is routed inside $D_C^{(1)}$ by the recursive embedding.
2. The subsequence of p from its first edge $(u_1, v_1) \in E_{t-1}$ to its last edge $(u_2, v_2) \in E_{t-1}$. We replace this by the two-hop path (u'_1, w_C, v'_2) where $\pi(u'_1) = u_1$ and $\pi(v'_2) = v_2$. Feasibility holds because the capacity of (u'_1, w_C) is $\text{vol}_{E_{t-1}}(u_1)$, and analogously for (w_C, v'_2) so no congestion larger than 1 is introduced when considering all flow paths of f (a feasible flow in G) passing through E_{t-1} .
3. The suffix of p after its last edge in E_{t-1} , which is routed inside $D_C^{(2)}$ by recursion.

Then we show that $\text{maxflow}_{G'}(\pi^{-1}(s), \pi^{-1}(t)) \leq n^{o(1)} \text{maxflow}_G(s, t)$. Given a flow f^D in D , we map it back to G with congestion at most $n^{o(1)}$. For all portions of f^D that do not traverse dummy vertices, we use the projection π ; since each original vertex has at most $n^{o(1)}$ copies, this incurs at most an $n^{o(1)}$ congestion blow-up.

Next, consider every subpath of flow paths of f^D that are incident to the dummy vertex w_C . This induces a demand that is $n^{o(1)} \cdot \text{vol}_{E_{t-1}}$ -respecting in $G[C]$ (the $n^{o(1)}$ factor comes from the number of vertex copies), hence is routable in $G[C]$ with congestion at most $1/\phi = 2^{O(\sqrt{\log n})} = n^{o(1)}$. Although each w_C may itself be replicated $n^{o(1)}$ times across lower layers, the resulting multiplicative factors still yield total congestion $n^{o(1)}$ overall.

The congestion can be made 1 by scaling down the flow value by $n^{o(1)}$, which gives the correctness.

Algorithmic Aspect. Excluding the time for computing the expander hierarchy in [Definition 2.8](#), the construction of DAG projection takes near-linear time in the output size, which is $m^{1+o(1)}$ time.

In the sequential setting, by using almost-linear time maximum flow algorithm to compute expander decomposition, this gives $m^{1+o(1)}$ total time.

Unfortunately, to obtain efficient parallel reduction, this suffers from a similar bottleneck as in the distance DAG projection algorithm: Expander decomposition usually requires maximum flow on a general graph as a subroutine (which does not admit fast parallel algorithms), and we intend to reduce max flow on general graphs to max flow on DAGs.

The idea is again to design a spiral recursion: (Given a max flow on DAGs oracle) We define δ additive-error DAG projection to tolerate some δ additive error, and reduce it to $\delta \cdot n^{o(1)}$ additive-error maximum flow on general graphs, which reduces to $\delta \cdot n^{o(1)}$ additive-error DAG projection. The additive error will finally be large enough so the problem is trivial.

CHAPTER 3

Combinatorial Minimum-Cost Flow

Beyond the applications discussed in [Chapter 2](#), we highlight another application of DAG projection to minimum-cost flow (MCF). This application requires strengthening the DAG-projection guarantees developed in [Chapter 2](#).

In [Section 3.1](#), we review the history of MCF and state our main results. In [Section 3.2](#), we give a detailed overview of the algorithm. This section is based on [Paper B](#).

3.1 Our Results

MCF can be viewed as a combination of shortest paths and max flow: it minimizes the summation of all flow paths distances, or cost, while maximizing the flow value. Fast algorithms for MCF have been studied since the 1950s [[Dan51](#); [Ful61](#); [Kle67](#); [EK72](#); [Tar85](#); [Orl88](#)]. Classical combinatorial algorithms follow a residual-graph paradigm: repeatedly improve the current solution by augmenting along a negative cycle or a shortest augmenting path in the residual graph. This line culminated in push-relabel-based algorithms [[GT87](#); [GT90](#); [Ahu+92](#)] with running time $\tilde{O}(mn)$. Within the classical combinatorial framework, progress then appeared stuck at this bound for decades.

Minimum-cost flow is also a graph-structured linear program. After the long stagnation of combinatorial algorithms, faster algorithms increasingly relied on continuous optimization, especially interior-point methods, while exploiting the special structure of the MCF linear program [[DS08](#); [LS19](#); [AMV22](#)]. This line culminated in almost-linear-time algorithms [[Che+25](#); [Bra+23](#)].

Recent advances have given combinatorial algorithms for special cases of MCF, including bipartite matching and max flow [[CK24a](#); [CK24b](#); [Ber+24](#); [Ber+25a](#)], but the general MCF problem has remained stuck at the $\tilde{O}(mn)$ running time of Goldberg and Tarjan [[GT87](#)] from 1987. We give the first combinatorial improvement and achieve almost-linear time on dense graphs.

Theorem 3.1 (Combinatorial exact minimum-cost flow). *There is a randomized combinatorial algorithm that computes a minimum-cost flow in $n^{2+o(1)}$ time.*

The main significance of our result is not only the running time, but also the structural reduction behind it. We give a black-box reduction from minimum-cost flow on general directed graphs to approximate minimum-cost flow on DAGs. Notably, this implies that future work on faster combinatorial MCF can focus on the simpler structure of DAGs.

Theorem 3.2 (Reduction from exact MCF to approximate MCF on DAGs). *There is a combinatorial algorithm for computing an exact minimum-cost flow whose running time is proportional to the cost*

of oracle calls for computing approximate¹ MCF on DAGs, with cumulative total size $m^{1+o(1)}$ and cumulative total number of vertices $n^{1+o(1)}$.

From DAG projection to Theorem 3.2. We have shown in the last section that DAG projection allows us to reduce distance-, flow-, and cut-related problems to DAGs, with only a subpolynomial loss in complexity. Since MCF naturally combines shortest paths and maximum flow, it can also be reduced to DAGs using DAG projection, leading to Theorem 3.2. This step, however, is non-trivial: it requires strengthening the definition of DAG projection so that it preserves both flow values and costs. We defer the details to Section 3.2.2.

From Theorem 3.2 to Theorem 3.1. The main MCF result, Theorem 3.1, can then be derived by combining Theorem 3.2 with an $n^{2+o(1)}$ -time MCF algorithm on DAGs. The latter can be obtained by a simple almost-linear-time combinatorial algorithm for minimum-cost flow on dense DAGs, similar to [Ber+24]. More details can be found in Section 3.2.3.

3.2 Overview of the Algorithm

In this section, we sketch the three ingredients behind our $n^{2+o(1)}$ combinatorial algorithm Theorem 3.1 for minimum-cost flow:

1. in Section 3.2.1, we reduce exact MCF to the approximate primal-dual version of MCF;
2. in Section 3.2.2, we reduce this approximate problem on general directed graphs to the same problem on DAGs via a *flow-cost-preserving DAG projection*;
3. in Section 3.2.3, we give a simple augmenting-path-based algorithm for approximate PD-MCF on DAGs.

For a directed graph $G = (V, E)$ and an edge $e \in E$, we write $\ell_G(e)$ for its length and $U_G(e)$ for its capacity. For a vertex $v \in V$, we write $\deg_G(v) \stackrel{\text{def}}{=} \sum_{v \in e} U_G(e)$ for its capacity-weighted degree. When the graph is clear from context, we omit the subscript G . For a flow f , we write its value $\text{val}(f)$ as the amount of flow it sends, and $\text{cost}(f)$ as the cost of the flow defined as $\sum_{e \in E} f(e) \cdot \ell(e)$, i.e., we call $\ell(e)$ both the *length* of an edge and also the *cost* of the edge.²

The demands D in this section are all (single-commodity) demands, where each vertex gets a value: positive means flow should leave this node, and negative means flow should enter this node. A flow routing a demand is defined in the natural way. We write $|D| = \sum_{v \in V, D(v) > 0} D(v)$.

We say that an algorithm makes g -efficient oracle calls to an oracle $\mathcal{O}$ if:

- the total number of vertices over all instances given to $\mathcal{O}$ is at most $g \cdot n$,
- the total number of edges over all instances given to $\mathcal{O}$ is at most $g \cdot m$, and
- the additional running time outside the oracle calls is at most $g \cdot m$.

3.2.1 Exact-to-Approximation Reduction

Throughout this section, all MCF instances are assumed to have non-negative edge lengths. For *exact* MCF, this is without loss of generality: if an edge (u, v) has negative length, we may reverse it to (v, u) , negate its length, and modify the demands by adding $U(u, v)$ to v and $-U(u, v)$ to u . This

¹The definition of approximation here is involved, and we defer the precise definition to Section 3.2.1.

²Although mixing cost and length might be confusing, if one considers minimum cost flow as a combination of shortest path and maximum flow, then the notation of ‘length’ makes sense. It also combines many recent development of length bounded flows.

folklore reduction transforms the instance into one with non-negative lengths. For multiplicative approximation problems, non-negative lengths are the natural setting.

We first explain how to reduce exact MCF to the approximate primal-dual version of MCF.

Minimum cost flow has the following primal-dual LP.

$$\begin{aligned}
(\text{P}) \quad & \min \sum_{e \in E} \ell(e) \cdot f(e) & (\text{D}) \quad & \max \sum_{u \in V} (-D(u)) \cdot \phi(u) - \sum_{e \in E} U(e) \cdot \tilde{\ell}(e) \\
\text{s.t.} \quad & \sum_{e \in \delta^+(u)} f(e) - \sum_{e \in \delta^-(u)} f(e) = D(u) \quad \forall u \in V, & \text{s.t.} \quad & \phi(v) - \phi(u) \leq \ell(e) + \tilde{\ell}(e) \quad \forall e = (u, v) \in E, \\
& 0 \leq f(e) \leq U(e) & & \tilde{\ell}(e) \geq 0 \quad \forall e \in E. \\
& & & \forall e \in E.
\end{aligned}$$

We can view $\tilde{\ell} : E \rightarrow \mathbb{R}_{\geq 0}$ as a length increase. We can view ϕ as a distance function (from source to sink). The goal of the dual is to enlarge the distance from source to sink by applying a small length increase. We write

$$\text{Dual}(\phi, \tilde{\ell}) = \sum_{u \in V} (-D(u)) \cdot \phi(u) - \sum_{e \in E} U(e) \cdot \tilde{\ell}(e).$$

We say a primal-dual triple $(f, \phi, \tilde{\ell})$ is a (κ, λ) -approximate PD-MCF if (i) f has congestion at most κ , and (ii) $\text{cost}(f) \leq \lambda \cdot \text{Dual}(\phi, \tilde{\ell})$.

The problem of (κ, λ) -approximate PD-MCF is: given a graph G with positive, polynomially bounded integer capacities and lengths, find a (κ, λ) -approximate PD-MCF. We denote an oracle solving the (κ, λ) -approximate PD-MCF by $\mathcal{O}_{\kappa, \lambda}^{\text{PD-MCF}}$.

Theorem 3.3. *For $\varepsilon = o(1/\log^9 n)$, there is an algorithm solving exact minimum-cost flow using*

$$\tilde{O}\left(n^{O(1/\log(1/\varepsilon))} \cdot \kappa^2/\varepsilon^3\right)$$

-efficient calls to a $(\kappa, 1 + \varepsilon)$ -approximate PD-MCF oracle.

By setting $\kappa = n^{o(1)}$ and $\varepsilon = 1/n^{o(1)}$, **Theorem 3.3** implies that exact MCF can be solved using only $n^{o(1)}$ calls to a $(\kappa, 1 + \varepsilon)$ -approximate PD-MCF oracle.

Theorem 3.3 is interesting in its own right. However, since it is not directly related to the main theme of graph projection, we omit the details here and refer interested readers to **Paper B**.

3.2.2 Flow-Cost-Preserving DAG Projection

We next explain how to reduce approximate PD-MCF on a general directed graph to approximate PD-MCF on a DAG. This is our main structural contribution.

We first need to strengthen the definition of DAG projection defined in **Chapter 2** to preserve both flows and costs: A *flow-cost-preserving DAG projection* of a directed graph G is defined as follows.

Definition 3.4. A flow-cost-preserving DAG projection of $G = (V, E)$ is a DAG G' along with a projection map $\pi : V(G') \rightarrow V$. We call it (κ, λ) -flow-cost-preserving DAG projection, or say it has congestion slack κ and length slack λ if

1. For every demand D' routable in G' by a feasible flow with cost B , $\pi(D')$ is routable in G with congestion κ and cost B and,
2. For every demand D routable in G by a feasible flow with cost B , $\hat{\pi}^{-1}(D)$ is routable in G' by a feasible flow with cost at most $\lambda \cdot B$.

$\pi(D')$ is defined in the natural way: $\pi(D')(v) = \sum_{\pi(v')=v} D'(v')$. $\hat{\pi}^{-1}(D)$ is a bit trickier: we fix a topological order of G' , if $D(v) > 0$, then we choose the first vertex v' in the topological order with $\pi(v') = v$ and let $\hat{\pi}^{-1}(D)(v') = D(v)$; if $D(v) < 0$, then we choose the last vertex v' in the topological order with $\pi(v') = v$ and let $\hat{\pi}^{-1}(D)(v') = D(v)$; otherwise we let $\hat{\pi}^{-1}(D)(v') = 0$. i.e., we lift the demand to be a demand on the DAG from the first copy of sources to the last copy of sinks.

The following lemma shows the power of DAG projection.

Lemma 3.5. *Let G be a directed graph, and suppose G' is a (κ', λ') -flow-cost-preserving DAG projection of G . Then one call to $\mathcal{O}_{\kappa, \lambda}^{\text{PDMCF}}$ on G' can be transformed into a solution to $\mathcal{O}_{\kappa\kappa', \lambda\lambda'}^{\text{PDMCF}}$ on G .*

The formal statement additionally requires a projection algorithm that maps primal and dual objects between G and G' . For the overview, the key intuition is simple: if G' approximately preserves all relevant flows and their costs, then solving PD-MCF on G' is essentially as good as solving it on G .

We now sketch how such a DAG projection is constructed.

Existence

The construction relies on two primitives: a length-constrained expander hierarchy, and a bounded-width path cover.

Primitive 1: LC expander hierarchy. A *moving cut* C assigns a non-negative value to every edge. Each moving cut is associated with a length parameter denoted by $\text{leng}(C)$. When we view C as an h -length moving cut, we denote graph $G - C$ by the graph increasing the length of each edge e in G by $C(e) \cdot h$.

For such a cut C , define $\text{vol}_C(v) \stackrel{\text{def}}{=} \sum_{v \in e} C(e) \cdot U(e)$.

We say that a demand D is *vol_C-respecting* if $|D(v)| \leq \text{vol}_C(v)$ for every vertex v .

We say that a demand is *h-length symmetrically constrained* if it can be decomposed into pieces, each supported on a strongly connected subgraph of diameter at most h .

An (h, s) -length ϕ -expander with respect to vol_C is a graph in which every vol_C -respecting h -length symmetrically constrained demand can be routed with congestion at most $1/\phi$ and average cost at most hs . Here s is the *length slack*.

A ϕ -expander hierarchy with length slack s is a sequence of moving cuts $C_1, \dots, C_t$ such that for every scale h and every level i , if we interpret $C_{>i} \stackrel{\text{def}}{=} \sum_{j>i} C_j$ as an h -length moving cut, then $G - C_{>i}$ is an (h, s) -length ϕ -expander with respect to vol_{C_i} .

We require $C_1(e) = 1$ for all $e \in E$, and in our setting we have $s = n^{o(1)}$ and $\phi = 1/n^{o(1)}$. Such hierarchies are known to exist [Hae+25b].

Primitive 2: bounded-width path cover. We have discussed path covers in the intro (details in Chapter 4), and it has an important role here.

A d_{cov} -path cover of G with diameter d_{diam} is a graph G' , together with a projection map $\pi : V(G') \rightarrow V(G)$, such that:

1. every edge of G' projects to an edge of G (so π is essentially a graph homomorphism from G' to G);
2. every path p of length at most d_{cov} in G lifts to a path in G' , i.e., there is a path p' such that $\pi(p') = p$ (under the natural definition of $\pi(p')$); and

3. every SCC of G' projects to an induced subgraph of G and has strong diameter at most d_{diam} .

The *width* of the path cover is

$$\omega \stackrel{\text{def}}{=} \max_{v \in V(G)} |\pi^{-1}(v)|,$$

that is, the maximum number of copies of a vertex. Bounding the width is crucial for controlling congestion.

One-layer hierarchy. To illustrate the main idea, first assume that the input graph G itself is already an (h, s) -length ϕ -expander with respect to $\deg_G$, for every scale h .

For each dyadic scale h' , we construct a DAG $\widehat{G}^{(h')}$ that is responsible for routing flow paths of cost roughly $h' \cdot n^{o(1)}$. The final DAG projection is the disjoint union over all dyadic scales.

We start from a bounded-width path cover $G_{\text{cov}}^{(h')}$ with diameter parameter $h_{\text{diam}} = h'$, covering parameter $h_{\text{cov}} = h' \cdot n^{o(1)}$, and width $\omega = n^{o(1)}$. Similar to the idea of [Coh00], we look at its SCCs and classify them as follows.

Light SCCs. If $|S| < n/\sigma$, where $\sigma = n^{o(1)}$ is a parameter, we recurse on $G[S]$ and replace S by its recursively constructed DAG projection.

Heavy SCCs. If $|S| \geq n/\sigma$, we do not recurse. Instead, we use the expander property: since $G[S]$ has strong diameter at most h' , any sufficiently local demand inside S can be routed at low congestion and average cost $h's$. We therefore replace S by a simple *star gadget*: one copy of S for entering the gadget, one central hub vertex v_S , and one copy of S for exiting it. The edges incident to the hub have capacity $\deg_G(\cdot)$ and length $h's$.

After replacing every SCC in this way, the SCC structure disappears and we obtain a DAG $D^{(h')}$. The crossing SCC edges can also be defined naturally in $D^{(h')}$.

The final step is to add *bridges* between heavy SCCs. We duplicate $D^{(h')}$ into two copies and, for every pair of heavy SCCs (S_1, S_2) , we add $O(\log n)$ edges from the hub of S_1 to the hub of S_2 . The i -th bridge has capacity equal to the maximum flow that can be sent from S_1 to S_2 in G with average cost at most $r_i = (1 + \varepsilon)^i$.

Forward mapping. Given a feasible flow F in G , we lift each flow path P to the appropriate scale $\widehat{G}^{(h')}$ according to its cost. Because the path cover lifts all paths up to length h_{cov} , the path P lifts to a path in $G_{\text{cov}}^{(h')}$, which decomposes into pieces inside SCCs.

For a light SCC, we can simply recurse. So the case that all SCCs are light is easy to solve.

If the path passes through at least one heavy SCC, let the first one be S_1 and last one be S_2 . For each S_1, S_2 , we route through the corresponding star gadget. We connect the two hub vertices v_{S_1}, v_{S_2} using the appropriate bridge. The bridge length approximates the cost of the corresponding subpath from S_1 to S_2 up to a factor of $(1 + \varepsilon)$, while the additional star cost is only $h's$, which can be made negligible compared to the total cost of the path by parameter choice (since $h's = h' \cdot n^{o(1)}$ and the path length cost(P) = $h' \cdot n^{o(1)}$, we could make $hs = \varepsilon \cdot \text{cost}(P)$). Thus, the total loss is only a $(1 + O(\varepsilon))$ factor.

Feasibility follows because star capacities are chosen according to vertex degree, and bridge capacities are chosen according to the maximum amount of flow that can be routed at the corresponding average-cost scale.

Backward mapping. Conversely, a flow in the DAG can be mapped back to G :

- the light-SCC part is handled recursively,

- the star gadgets are implemented using the expander property inside the corresponding heavy SCC, and
- each bridge is realized by the definition of its capacity.

This gives the backward projection.

Multiple layers. For a full LC-expander hierarchy, the construction recurses through the hierarchy levels. At a high level, instead of building a star over each heavy cluster, we replace each heavy cluster by a lower layer DAG projection, then build the star and bridges. The analysis follows a similar flavour but is much more complicated.

Fast construction

The existence proof above is not yet algorithmic; the main bottleneck is that it seems to require exactly the approximate MCF oracle we are trying to obtain. To be precise, there are two places that require an MCF oracle:

1. constructing the LC-expander hierarchy usually requires a MCF oracle for the cut-matching game, and
2. computing the capacities of the bridges between heavy SCCs requires at least an approximate MCF.

This results in a 'chicken-and-egg' situation: we want to compute DAG projection to get a MCF algorithm, but it requires MCF as a subroutine.

A similar issue also appears in [HJS26a]. For their purpose, they wish to construct a DAG projection that only preserves distance so that it can help the algorithm for single-source shortest path (SSSP), but the construction requires (1) low-diameter decomposition (LDD), which requires an SSSP oracle, (2) computing bridges similar to our case, which requires an SSSP oracle. Their solution is to introduce an *h-length* version of the projection problem: instead of preserving all distances, the projection only needs to preserve distances up to scale h . They can also define the *h-length* version of LDD and SSSP. Then they try to reduce the *h-length* problem to a smaller-scale h/z -length problem for a suitable parameter $z = n^{o(1)}$, and repeat this $O(\log n)$ times to get the final algorithm.

We apply the same philosophy here. We define:

- an *h-length* DAG projection, which only needs to preserve flows of average cost at most h ;
- an *h-length* approximate MCF problem; and
- an *h-length* LC-expander hierarchy.

At a high level, we obtain the following cycle of reductions:

1. reduce *h-length* DAG projection to *h-length* LC-expander hierarchy plus *h-length* approximate MCF.
2. reduce *h-length* LC-expander hierarchy to *h-length* approximate MCF;
3. reduce *h-length* approximate MCF to *h-length* DAG projection with the help of the DAG MCF oracle;
4. reduce *h-length* DAG projection to (h/z) -length DAG projection using concatenation.

Taken together, these reductions decrease the length scale from h to h/z , and the recursion terminates once h becomes small enough to handle directly.

A natural question is why we do not simply apply the last reduction repeatedly, since it already decreases the length scale. The reason is that the concatenation enlarges the size of the DAG by a factor of z each time. Repeating this for $O(\log_z n)$ levels would lead to a polynomial blow-up. In contrast, the four-step recursion above avoids this: the parameter z enters the *running time* of the recursion, but not the final approximation guarantees or the final size of the projection.

3.2.3 Weighted Push-Relabel Algorithm for MCF on DAGs

Given the first two ingredients, it remains to solve approximate PD-MCF on DAGs.

Lemma 3.6. *There exists an **augmenting-path-based** algorithm on DAGs that returns a $(3, 1 + \varepsilon)$ -approximate PD-MCF solution.³*

The algorithm extends the weighted push-relabel method for maximum flow on DAGs from [Ber+24] to the minimum-cost flow setting.

As in classical push-relabel for MCF [GT90], we maintain a potential function $\phi : V \rightarrow \mathbb{R}$. For an edge (u, v) , its reduced length is

$$\ell_\phi(u, v) \stackrel{\text{def}}{=} \ell(u, v) - (\phi(u) - \phi(v)).$$

The algorithm alternates between:

- *relabel* steps, which decrease vertex potentials, and
- *push* steps along *admissible* edges, i.e., edges whose reduced length is sufficiently negative.

The classical Goldberg–Tarjan algorithm takes $\tilde{O}(mn)$ time essentially because an edge can become admissible up to $\tilde{O}(n)$ different times. Our key idea is to define admissibility using a carefully chosen weight function $w : E \rightarrow \mathbb{R}_{>0}$.

Following [Ber+24], on a DAG one can assign weights so that:

1. every flow f of interest satisfies

$$\frac{w(f)}{\text{val}(f)} = O(n), \quad \text{where } w(f) \stackrel{\text{def}}{=} \sum_{e \in E} f(e) w(e),$$

and

- 2.

$$\sum_{e \in E} \frac{1}{w(e)} = O(n).$$

We then declare an edge e admissible when its reduced length is roughly $-\Theta(w(e))$. Saturating such an edge changes the relevant potentials by at least $w(e)$. Since the total potential range is $O(n)$, edge e can be saturated only $O\left(\frac{n}{w(e)}\right)$ times. Summing over all edges gives

$$\sum_{e \in E} O\left(\frac{n}{w(e)}\right) = O\left(n \sum_{e \in E} \frac{1}{w(e)}\right) = O(n^2),$$

which is the desired bound.

³In fact, the weighted push-relabel algorithm on DAGs returns a $(3, 1 + \varepsilon)$ -strongly-approximate PD-MCF solution, which is stronger than what the rest of the framework requires.

CHAPTER 4

Path Cover

In many cases, we might want more than just the preservation of distances or flows between pairs of nodes. In this chapter, we introduce an object with a stronger property: it covers all bounded-length paths, of which there may be exponentially many. Naturally, because it has a stronger property, it is less structured: it is no longer a DAG, but it is close to a DAG in the sense that its strongly connected components (SCCs) have small diameter.

In [Section 4.1](#), we define path covers. In [Section 4.2](#), we discuss their application to deterministic negative-weight single-source shortest paths. In [Section 4.3](#), we discuss related work on low-diameter decompositions and neighborhood covers, as well as their connection to path covers. In [Section 4.4](#), we present the existential proof of path covers. In [Section 4.5](#), we prove a lower bound showing that the classical one-to-one viewpoint does not suffice for our purposes, and that graph projection provides the right perspective. This chapter is based on [Paper C](#).

4.1 Path Cover

We show that, given any directed graph $G = (V, E)$ and a parameter d , there exists a nearly-linear-size directed graph $G' = (V', E')$ that covers all paths of G and has small SCC diameter, in the following sense.

Given a path $p = (v_1, \dots, v_k)$, we say that a path $p' = (v'_1, \dots, v'_k)$ in G' is a *lift* of p if $\pi(v'_i) = v_i$ for all $i \in [k]$. We say that π is a *strict projection* if $\pi : V' \rightarrow V$ and for every $(u', v') \in E'$, we have $(\pi(u'), \pi(v')) \in E$, i.e., π is a graph homomorphism.

Theorem 4.1. *For any directed graph $G = (V, E)$ with edge lengths and parameter d , there is a graph $G' = (V', E')$ and a strict projection $\pi : V' \rightarrow V$ such that*

- $|E'| \leq (1 + 1/\log n)|E|$,
- every path of length at most d in G has a lift in G' ,
- every SCC of G' has diameter $O(d \log^6 n)$.

See [Figure 1.1](#) for an example: (1) G' is not too large compared to G , (2) every path in G has a lift in G' , and (3) G' is a path, and thus every SCC has diameter 0.

The name *path cover* comes from the literature on *neighborhood covers* in undirected graphs. In fact, both are closely connected to the influential concept of *low-diameter decomposition (LDD)*. We discuss more details in [Section 4.3](#).

4.2 Application: Deterministic Negative-Weight SSSP

Path cover is a versatile tool. Here, we discuss one highlighted application: the first near-linear-time deterministic algorithm for negative-weight SSSP. Given a directed graph $G = (V, E, w)$ with edge-weight function $w : E \rightarrow \mathbb{Z}^1$ and a source vertex $s \in V$, the goal is to compute the distance from s to every other vertex t , denoted by $\text{dist}_G(s, t)$.

History. The classical Bellman–Ford algorithm solves this problem deterministically in $O(mn)$ time, and a long line of work improved this bound to $\tilde{O}(m\sqrt{n})$ [Gab85; GT89; Gol95].

A major breakthrough occurred in 2022, when Bernstein, Nanongkai, and Wulff-Nilsen gave the first near-linear-time algorithm for Neg-SSSP [BNW22]; subsequent works further improved the polylogarithmic factors [BCF23; LM24]. These algorithms are randomized and may fail with small probability. Before the concept of path cover, it remained open whether one could obtain a *deterministic* near-linear-time algorithm.²

Our contribution. We give the first deterministic near-linear-time algorithm for negative-weight SSSP using path cover, as stated below.³

Theorem 4.2. *There is a deterministic algorithm that solves negative-weight SSSP on a directed graph with integer edge weights in*

$$O(m \log^8 n \log(nW))$$

time, where W bounds the absolute values of the edge weights.

Bottleneck: Derandomizing LDD. The main bottleneck in obtaining Theorem 4.2 is that all previous near-linear-time randomized algorithms relied on a highly randomized primitive: *directed low-diameter decomposition* (LDD). At a high level, LDD takes as input a parameter d and produces a randomized *subgraph* G' of the original graph G such that (1) each strongly connected component (SCC) of G' has diameter at most $\tilde{O}(d)$, and (2) every path of length at most d in G is covered by G' with good probability. By generating $O(\log n)$ independent samples of LDD, each path is covered in one of them with high probability.⁴ The randomized algorithm only needs polynomially many relevant paths to be covered by G' since we only care about shortest paths between pairs of vertices, so a union bound over the high-probability guarantee for each relevant path suffices.

The difficulty in derandomizing directed LDD is that a deterministic analogue must work for *all* paths simultaneously. The first question is whether such an object even exists. Unfortunately, if one follows the usual LDD intuition—namely, trying to find a collection of *subgraphs* with SCC diameter at most $\tilde{O}(d)$ that cover all length- d paths—then such an object provably cannot be efficient on directed graphs: Section 4.5 shows that $\Omega(m^{1.5})$ edges are needed in total to cover all length- d paths using subgraphs. This explains why derandomizing directed LDD, and hence derandomizing negative-weight shortest paths, had remained stuck.

¹We do not discuss the interesting case of real weights in this thesis. We assume that all weights are integers polynomially bounded in the size of the graph.

²There is also another line of work based on continuous optimization and interior-point methods for minimum-cost flow, which implies Neg-SSSP and achieves $m^{1+o(1)}$ time deterministically [Che+25; Bra+23]. Since these algorithms are highly non-combinatorial, solve a much more general problem at the cost of substantial additional complex machinery, and currently run in almost-linear rather than near-linear time, we do not emphasize them in this shortest-path narrative.

³Another concurrent work [Li25] also gives a deterministic near-linear-time algorithm for Neg-SSSP and was published at the same time as Paper C.

⁴We use *with high probability* to denote an event that happens with probability at least $1 - \frac{1}{n^c}$, where c can be an arbitrarily large constant.

Circumventing the bottleneck: Path Cover. One can see that [Theorem 4.1](#) gives exactly what we want: G' covers all length- d paths and has small SCC diameter. The crux of circumventing the lower bound is exactly the lift-and-project viewpoint. To see a concrete example, consider the cycle in [Figure 1.1](#): if we want to use subgraphs to cover *all* paths, then we need to take n subgraphs, each of which is a path obtained by deleting one edge from the cycle. However, if we follow [Figure 1.1](#) and construct G' by connecting two copies of the cycle with one edge deleted, then G' suddenly covers all paths in the cycle, while having only twice the size of G .

It also turns out that G' , together with a strict projection map, is enough to solve SSSP. The details are given in [Paper C](#).

4.3 Related Work on LDD and Neighborhood Covers

Undirected graphs. *Low-diameter decomposition* and *neighborhood covers* are two closely related concepts. Low-diameter decomposition was initially proposed as a powerful tool in distributed computing [[Awe85](#)], and later became highly influential and powerful in general algorithm design [[BNW22](#)]. On an undirected weighted graph $G = (V, E, w)$, it is defined as follows: given d , output a random set of edges E' such that (1) $G - E'$ has components of diameter at most d , and (2) each edge e is included in E' with probability $O(w(e) \log n)/d$. The first property follows the same idea of simplifying graph structures to make the problem easier, and the second property is usually used to argue that a certain path is covered after deleting E' : every path of length at most $O(d \log n)$ is completely preserved, or covered, in $G - E'$ with probability 0.5. This motivates the study of *neighborhood covers*.

In an undirected graph, a neighborhood cover with parameter d is defined as a sequence of subgraphs $E_1, \dots, E_{O(\log n)}$ such that (1) every connected component of E_i has diameter at most $O(d \log n)$ for every i , and (2) every d -neighborhood, defined as the set of all vertices at distance at most d from a vertex $u \in V$ and denoted by $\text{Ball}(u, d)$, is completely contained in one connected component of E_i for some i . One can see that (1) and (2) correspond to the properties guaranteed by LDD, and naturally, neighborhood covers are considered deterministic analogues of LDD. See [[Awe+98](#)] for more details.

Directed graphs. We can also define low-diameter decomposition on directed graphs in a way similar to undirected graphs: given d , output a random set of edges E' such that (1) $G - E'$ has SCC diameter at most d , and (2) each edge e is included in E' with probability $O(w(e) \log^2 n)/d$. An efficient algorithm for constructing such an LDD can be found in [[BNW22](#)].

Due to the success of neighborhood covers in undirected graphs, it is natural to expect an analogue for directed graphs. However, it is hard to define what a neighborhood should mean in a directed graph, since distance is not symmetric. Thus, we instead focus on paths. One could then try to define a path cover as a sequence of subgraphs that covers all paths and has small SCC diameter, in direct analogy with neighborhood covers in undirected graphs. As mentioned above, this definition is impossible to achieve efficiently, and this leads to our contribution of path cover. We view path cover as the right directed analogue of the relationship between LDD and neighborhood covers in undirected graphs.

4.4 Existential Proof

This section sketches the *existence* of the d -path cover G' of G in [Theorem 4.1](#).

We start with some standard notation: For every vertex $v \in V$ let $\deg_G^{\text{out}}(v)$ be the out-degree of v , i.e., the number of edges leaving v . For a vertex set $S \subseteq V$ let $\deg_G^{\text{out}}(S) = \sum_{v \in S} \deg_G^{\text{out}}(v)$ be the number of edges starting in S . Lastly, for any vertex $u \in V$, define

$$\text{Ball}_G^{\text{out}}(u, r) = \{v \in V \mid \text{dist}_G(u, v) \leq r\}$$

to be the ball centered at u with radius r .

Construction. Our algorithm is recursive.

If G is a strongly connected component with diameter $O(d \log^3 n)$, then one can trivially return the whole graph and terminate the recursion.

Otherwise, the algorithm finds a good partition by picking an arbitrary node u and growing a ball from it by computing $\text{Ball}_G^{\text{out}}(u, i \cdot d)$ for $i = 1, 2, 3, \dots$ until we find a “thin layer” i^* such that

$$\deg_G^{\text{out}}(\text{Ball}_G^{\text{out}}(u, i^* \cdot d)) \leq \left(1 + \frac{1}{\log^2 n}\right) \cdot \deg_G^{\text{out}}(\text{Ball}_G^{\text{out}}(u, (i^* - 1) \cdot d)). \quad (4.1)$$

Since the ball cannot grow multiplicatively too often it is clear that $i^* = O(\log^3 n)$. The algorithm works separately and recursively on both

$$B^{\text{out}} := \text{Ball}_G^{\text{out}}(u, i^* \cdot d) \quad \text{and} \quad \bar{B}^{\text{out}} := V \setminus \text{Ball}_G^{\text{out}}(u, (i^* - 1) \cdot d).$$

Note that B^{out} and $\bar{B}^{\text{out}}$ only *overlap* at the “ring” around u from radius $(i^* - 1) \cdot d$ to $i^* \cdot d$.

We focus here on the interesting case that the ball grown from u is not more than a constant fraction of the graph, i.e., $\deg_G^{\text{out}}(B^{\text{out}}) \leq 0.9m$, our proof shows that otherwise there is a way to carve out a large ball and still make progress. Note that by running Dijkstra’s algorithm locally from u the time needed to grow the ball B^{out} is $\tilde{O}(\deg_G^{\text{out}}(B^{\text{out}}))$ and therefore essentially proportional to its size and independent of the size of $\bar{B}^{\text{out}}$. This makes the recursion efficient. Let H^{out} and $\bar{H}^{\text{out}}$ be d -path covers of the induced subgraphs $G[B^{\text{out}}]$ and $G[\bar{B}^{\text{out}}]$, respectively, which are constructed *recursively*.

For technical reasons, we strengthen the property of H^{out} (and likewise for $\bar{H}^{\text{out}}$): not only do we require each d -path in $G[B^{\text{out}}]$ to have a lift p' in H^{out} , we require a *one-to-one* mapping between the start vertices of p and p' . In other words, each vertex v in $G[B^{\text{out}}]$ has a *representative* vertex v' in H^{out} such that if a d -path in G starts at v , then its lift in G' starts at v' .

We build the d -path cover of G (denoted G') by concatenating $\bar{H}^{\text{out}}$ and H^{out} as follows: for every $(u, v) \in E$ with $u \in \bar{B}^{\text{out}}$ and $v \in V \setminus \bar{B}^{\text{out}}$, we add edges from all copies of the vertex u in $\bar{H}^{\text{out}}$ to the *representative* of the vertex v in H^{out} . Having a representative for each vertex avoids having to add prohibitively many edges from all copies of u to all copies of v . Defining the representatives of vertices in G' throughout the recursion is straightforward once the path-covering property in the next paragraph is clear.

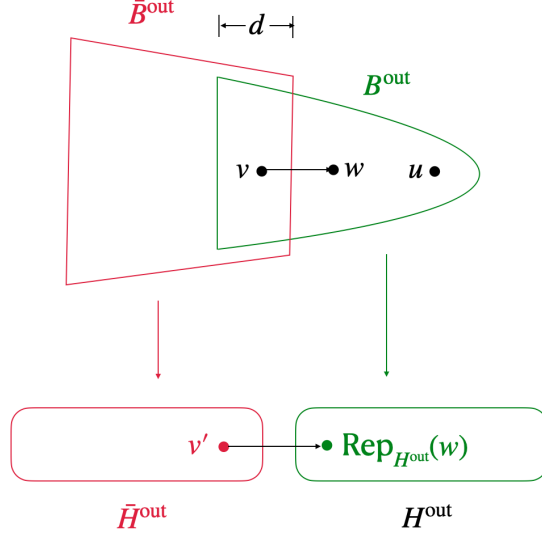


Figure 4.1: An illustration of the construction. A path entering $V - \bar{B}^{\text{out}}$ (for example a path starting at w) will not leave B^{out} again because of the length d layer in the middle, thus covered by H^{out} .

$\tilde{O}(d)$ -clustered graph. Notice that if $\bar{H}^{\text{out}}, H^{\text{out}}$ are $\tilde{O}(d)$ -clustered graphs, then G' is a $\tilde{O}(d)$ -clustered graph because edges can only be from $\bar{H}^{\text{out}}$ to H^{out} .

Path Covering. We show that G' indeed covers all d -paths in G .

We consider the case where a path p starts at $v_p \in \bar{B}^{\text{out}}$ and leaves $\bar{B}^{\text{out}}$ at some point (i.e., enters $V \setminus \bar{B}^{\text{out}}$). Other situations (completely inside $\bar{B}^{\text{out}}$ or starting in $V \setminus \bar{B}^{\text{out}}$) can be handled similarly. Let v'_p be the first vertex on p that is not in $\bar{B}^{\text{out}}$. Let p_1 be the subpath of p before v'_p (excluding v'_p) and let p_2 be the remainder of p (starting at v'_p).

Notice that

$$v'_p \in V \setminus \bar{B}^{\text{out}} = V \setminus (V \setminus \text{Ball}_G^{\text{out}}(u, (i^* - 1) \cdot d)) = \text{Ball}_G^{\text{out}}(u, (i^* - 1) \cdot d).$$

This means that every path starting at v'_p with length at most d (in particular, p_2) is contained in

$$\text{Ball}_G^{\text{out}}(u, (i^* - 1) \cdot d + d) = \text{Ball}_G^{\text{out}}(u, i^* \cdot d) = B^{\text{out}}.$$

Therefore p_1 lies completely inside $G[\bar{B}^{\text{out}}]$ and p_2 lies completely inside $G[B^{\text{out}}]$, so $\bar{H}^{\text{out}}$ covers p_1 with a lift p'_1 and H^{out} covers p_2 with a lift p'_2 .

We then concatenate p'_1 and p'_2 using the edge from the last vertex of p'_1 to the first vertex of p'_2 (which is the representative of v'_p in H^{out}). The concatenated path maps to p .

Size of the path cover. We first show that the size of G' satisfies

$$|E(G')| \leq \sum_{v' \in V(G')} \deg_G^{\text{out}}(\pi(v')).$$

Let (u', v') be an edge concatenating $\bar{H}^{\text{out}}$ and H^{out} , where $u' \in V(\bar{H}^{\text{out}})$ and $v' \in V(H^{\text{out}})$ are copies of $u \in \bar{B}^{\text{out}}$ and $v \in V \setminus \bar{B}^{\text{out}}$, respectively, so that $(u, v) \in E$ and v' is the representative of v in H^{out} . In this case, we say that (u, v) contributes to u' . Notice that (i) (u, v) can contribute to u'

at most once among all edges concatenating $\bar{H}^{\text{out}}$ and H^{out} , because we add only one edge from u' to the representative of v in H^{out} (which is why we define representatives); and (ii) (u, v) cannot contribute to u' again in the recursive construction of $\bar{H}^{\text{out}}$, because (u, v) is not an edge of $G[\bar{B}^{\text{out}}]$. Combining these two facts, every vertex u' is contributed to by at most $\deg_G^{\text{out}}(\pi(u'))$ edges, which establishes the claimed inequality.

It remains to bound $\sum_{v' \in V(G')} \deg_G^{\text{out}}(\pi(v'))$. Intuitively, Equation (4.1) guarantees that each recursion layer increases the total degree by at most a factor of $(1 + 1/\log^2 n)$ times the smaller side which is sufficient to yield an overall bound of $(1 + 1/\log n) \cdot m$ on the final total degree (we defer the calculation to the technical section). Since we have $|E(G')| \leq \sum_{v' \in V(G')} \deg_G^{\text{out}}(\pi(v'))$, we get the desired size bound

$$|E(G')| \leq (1 + 1/\log n) \cdot m$$

4.5 A Barrier for Covering Paths via Subgraphs

In this section, we prove the following lemma.

Lemma 4.3. *For every $m, \lambda \geq 1$, there exists an $O(m)$ -edge directed graph G and d such that every collection of $d\lambda$ -clustered graphs $G'_1, \dots, G'_z \subseteq G$ covering all d -paths in G must have total size $\sum_i |E(G'_i)| = \Omega(m\sqrt{m/\lambda})$.*

Fix $m, \lambda \geq 1$. We will build a directed graph G with $O(m)$ edges and a distance parameter d .

Step 1: choose parameters. Let

$$L := \lfloor \sqrt{m/\lambda} \rfloor \quad \text{and} \quad d := 2 + 3L.$$

So $L = \Theta(\sqrt{m/\lambda})$ and $d = \Theta(\sqrt{m/\lambda})$.

We will also use one more parameter

$$R := 2d\lambda.$$

Note that $R > d\lambda$ by construction.

Step 2: ladder gadget. Create vertices $a_1, a_2, \dots, a_{L+1}$. For each layer $j = 1, \dots, L$ do:

- create R new vertices $w_{j,1}, \dots, w_{j,R}$,
- add the directed cycle

$$w_{j,1} \rightarrow w_{j,2} \rightarrow \dots \rightarrow w_{j,R} \rightarrow w_{j,1},$$

- add edges $a_j \rightarrow w_{j,r}$ for every $r \in [R]$,
- add edges $w_{j,r} \rightarrow a_{j+1}$ for every $r \in [R]$.

Thus, to “go through” layer j you can do

$$a_j \rightarrow w_{j,r} \rightarrow w_{j,r+1} \rightarrow a_{j+1},$$

which is 3 edges.

The number of edges in one layer is

$$R \text{ (cycle)} + R \text{ (entries)} + R \text{ (exits)} = 3R = 3 \cdot 2d\lambda = 6d\lambda.$$

Since there are L layers, the total number of edges in the ladder is

$$6d\lambda \cdot L = 6(2 + 3L)\lambda L = O(\lambda L^2) = O(\lambda \cdot (m/\lambda)) = O(m).$$

Key property of the ladder. Let H be any subgraph of G whose SCCs all have diameter at most $d\lambda$. Consider a fixed layer j . If H contained all R edges of the directed cycle of layer j , then that SCC would have directed diameter $R - 1 \geq d\lambda$, contradicting that H is $d\lambda$ -clustered. Hence: For every $d\lambda$ -clustered subgraph H and every layer $j \in [L]$, H omits at least one edge of that layer's cycle.

Step 3: star in front of the ladder. Add a special vertex r (the center of the star) and add the edge $r \rightarrow a_1$. Then add

$$M := \lfloor m/20 \rfloor$$

new vertices $s_1, \dots, s_M$ and add star edges

$$s_t \rightarrow r \quad \text{for } t = 1, \dots, M.$$

This adds $M + 1 = O(m)$ edges. Since the ladder already cost $O(m)$ edges, the whole graph G has $O(m)$ edges.

Step 4: the d -paths we care about. Consider a path that:

1. starts with some star edge $s_t \rightarrow r$,
2. then takes $r \rightarrow a_1$,
3. then for each layer $j = 1, \dots, L$ takes a 3-edge traversal

$$a_j \rightarrow w_{j,r_j} \rightarrow w_{j,r_j+1} \rightarrow a_{j+1}.$$

This path has length

$$1 \text{ (star)} + 1 \text{ (to } a_1) + 3L = 2 + 3L = d.$$

So these are d -paths in G .

Step 5: diagonalization for one star edge. Let $\{G'_1, \dots, G'_z\}$ be any collection of $d\lambda$ -clustered subgraphs of G that covers all d -paths.

Fix one star edge $e_t := s_t \rightarrow r$. Define

$$I_t := \{i \in [z] : e_t \in E(G'_i)\}.$$

We claim that

$$|I_t| \geq L. \tag{4.2}$$

Suppose toward a contradiction that $|I_t| = q < L$. For each $i \in I_t$, for every layer j it omits at least one cycle edge in that layer. Since $q < L$, we can pick for every $i \in I_t$ a *distinct* layer $\ell(i) \in [L]$ and let f_i be one cycle edge in layer $\ell(i)$ that G'_i omits.

Now we build a d -path P as in Step 4, but with the following extra choice: in layer $\ell(i)$ we force the traversal to *use* exactly the edge f_i , and in all other layers we traverse arbitrarily. This is a valid d -path because every layer allows us to pick any two consecutive cycle vertices.

By construction:

- If $i \notin I_t$, then G'_i does not contain the first edge e_t , so $P \not\subseteq G'_i$.

- If $i \in I_t$, then G'_i does not contain f_i (by choice), and P uses f_i , so again $P \not\subseteq G'_i$.

Thus P is a d -path that is not contained in any G'_i , contradicting the assumption that the family covers all d -paths. Therefore (4.2) holds.

Step 6: counting. For each of the $M = \Theta(m)$ star edges e_t we have found at least L subgraphs that contain it. So the total number of *edge-subgraph incidences* is at least

$$\sum_{t=1}^M |I_t| \geq M \cdot L.$$

But

$$\sum_{i=1}^z |E(G'_i)|$$

counts all edge-subgraph incidences (each time an edge appears in some G'_i it is counted once). Hence

$$\sum_{i=1}^z |E(G'_i)| \geq M \cdot L.$$

Finally, $M = \Theta(m)$ and $L = \Theta(\sqrt{m/\lambda})$, so

$$\sum_{i=1}^z |E(G'_i)| \geq \Omega(m) \cdot \Omega(\sqrt{m/\lambda}) = \Omega(m\sqrt{m/\lambda}),$$

which is exactly the desired lower bound.

This completes the proof.

CHAPTER 5

Common-Neighborhood Clustering

In this section, we introduce another object called *common-neighborhood clustering*, which can be interpreted as graph projection. It is useful for vertex-cut-related problems.

This chapter is organized as follows. In [Section 5.1](#), we motivate the definition of common-neighborhood clustering through the problem of vertex connectivity. In [Section 5.2](#), we present its broad applications to vertex-cut problems. In [Section 5.3](#), we discuss its construction. This chapter is based on [Paper D](#) and [Paper E](#).

5.1 Common-Neighborhood Clustering

Consider the following problem called vertex connectivity: given a connected undirected graph $G = (V, E)$, how can we find the smallest number of vertices whose removal makes G disconnected? [\[Li+21\]](#) observed that this problem becomes easy if every two nodes share many neighbors. This motivates the following attempt to simplify the graph by lift-and-project: Given any undirected graph $G = (V, E)$, can we build a graph $G' = (V', E')$ and a function $\pi : V' \rightarrow V$ so that (1) solving vertex connectivity on G' gives the answer on G , and (2) every two vertices in G' share many neighbors?

Common-neighborhood clustering is designed exactly for this purpose. The goal is clear, although there are many technical details to handle in order to obtain a correct definition. Let us go further into the problem details in order to reach a rigorous definition.

Balanced vs. unbalanced vertex connectivity. If removing a vertex set $S \subseteq V$ makes $G = (V, E)$ disconnected, we say that S is a *vertex cut*. This also means that $G - S$ contains several connected components, at least two. Let $L \subseteq V$ be one of them with the smallest number of vertices. Then we can represent the vertex cut by (L, S, R) , where $R = V - L - S$.

The vertex cut (L, S, R) is called *balanced* if $|L| \geq \Omega(|S|)$. [\[Li+21\]](#) observed that if the minimum vertex cut is balanced, then it is relatively easy to find: This part requires the *isolating cut lemma* and we do not go into the details here. Intuitively, if $|L| \geq \Omega(|S|)$, then one can sample each vertex with probability $1/|L|$. With good probability, exactly one sampled vertex lies in L , no sampled vertex lies in S , and all other sampled vertices lie in R . Then an isolating cut lemma solves this case by a reduction to max-flow computation.

Thus, the hard case is an *unbalanced* vertex cut, where $|L| < o(|S|)$. Therefore, it suffices for common-neighborhood clustering to construct a graph G' that preserves unbalanced vertex cuts.

Common-neighborhood clustering. Now we can give the definition. For $u \in V$, we use $N_G(u)$ to denote the neighbors of u , i.e., $\{v \in V \mid (u, v) \in E\}$. We use $A \Delta B$ to denote the symmetric difference between the two sets A and B , i.e., $(A - B) \cup (B - A)$.

Theorem 5.1. *For any undirected graph $G = (V, E)$, there exist vertex sets $V_1, \dots, V_k \subseteq V$ such that*

- *there exists a minimum unbalanced vertex cut (L, S, R) of G such that $L \subseteq V_i$ for some $i \in [k]$,*
- *for all $i \in [k]$ and $u, v \in V_i$, we have $|N_G(u) \Delta N_G(v)| \leq \tilde{O}(|L|)$.*

Moreover, each vertex $u \in V$ is contained in at most polylogarithmically many different sets V_i .

The first property ensures that it suffices to find $L \subseteq V_i$ for each $i \in [k]$ in order to minimize $N_G(L)$. The second property ensures that finding such an L is relatively easy according to the observation of [Li+21].

Extension to directed weighted graphs. Theorem 5.1 is very flexible and can be extended to the more general setting where the graph is directed and each vertex is assigned a weight by a function $w : V \rightarrow \mathbb{N}$. We can define minimum vertex cut in such a graph naturally as follows: define $w(S) = \sum_{s \in S} w(s)$; (L, S, R) is called a vertex cut of size $w(S)$ if there are no edges from L to R ; and it is called unbalanced if $w(L) < o(w(S))$.

It turns out that the following theorem is true.

Theorem 5.2. *For any directed graph $G = (V, E)$ with vertex weights $w : V \rightarrow \mathbb{N}$, there exist vertex sets $V_1, \dots, V_k \subseteq V$ such that*

- *there exists a minimum unbalanced vertex cut (L, S, R) of G such that $L \subseteq V_i$ for some $i \in [k]$,*
- *for all $i \in [k]$ and $u, v \in V_i$, we have $w(N_G(u) \Delta N_G(v)) \leq \tilde{O}(w(L))$.*

Moreover, each vertex $u \in V$ is contained in at most polylogarithmically many different sets V_i .

Theorem 5.2 will be useful for solving weighted minimum vertex cut in directed graphs.

Connection to graph projection. Although Theorem 5.1 and Theorem 5.2 do not immediately look like graph projection, they can also be viewed as graph projection: the graph G is lifted to the disjoint union of the clusters $V_1, \dots, V_k$, where each vertex is lifted to at most polylogarithmically many vertices. This lifted graph, although larger, is more structured and makes the problem easier to solve.

5.2 Applications to Minimum Vertex Cut

Both Theorem 5.1 and Theorem 5.2 can be made algorithmically efficient depending on the setting of the problem. In this section, we discuss each problem and the result we obtain from common-neighborhood clustering.

Parallel and distributed vertex connectivity. Recall that vertex connectivity is given an undirected graph and asks to find the minimum number of vertices whose removal disconnects the graph. In a breakthrough result [Li+21], they show that vertex connectivity can be solved using polylogarithmically many calls to a max-flow subroutine in the sequential model, thus giving an $m^{1+o(1)}$ -time algorithm for vertex connectivity using the state-of-the-art max-flow algorithm. However, their algorithm is highly sequential, and it is not known whether this almost-perfect reduction can be carried out in other models as well.

Our contribution is to use *common-neighborhood clustering* as a tool to make this reduction work beyond the sequential model.

Theorem 5.3. *Vertex connectivity can be reduced to vertex-capacitated max flow in the parallel PRAM and distributed CONGEST models, with an $n^{o(1)}$ loss in complexity.*

The way we obtain [Theorem 5.3](#) is to use common-neighborhood clustering from the last section to break the problem into clusters, and then solve the problem inside each cluster. See [Paper D](#) for more details.

Deterministic small vertex connectivity. [\[Li+21\]](#) gives a randomized algorithm, and a fascinating question is how to close the gap between randomized and deterministic algorithms. There have been many attempts on this problem, dating back to 1975 [\[Eve75; Gab06; SY22\]](#). To simplify the literature, let κ be the vertex connectivity of the input graph, and focus on small κ . We mention that the current best known bound is $m^{1+o(1)} \cdot 2^{\kappa^2}$, where κ is the vertex-connectivity value. The dependence on κ is exponential, which is not ideal. There are algorithms that run better for larger κ [\[Eve75; Gab06\]](#), but they have complexity at least $O(mn)$ even for constant κ .

Our contribution is an exponential improvement in the dependence on κ : we obtain a deterministic vertex-connectivity algorithm running in $m^{1+o(1)} \cdot \kappa$ time, which subsumes and strictly improves all previous results [\[Eve75; Gab06; SY22\]](#) up to subpolynomial factors on this problem.

Theorem 5.4. *There is a deterministic algorithm solving vertex connectivity in $m^{1+o(1)} \cdot \kappa$ time, where κ is the vertex connectivity of the input graph.*

The way we obtain [Theorem 5.4](#) is to again apply *common-neighborhood clustering*, together with the terminal-reduction framework introduced by [\[LP20\]](#). The crucial point is that common-neighborhood clustering, as a general graph object, is inherently deterministic, and can therefore help with derandomization. More details can be found in [Paper E](#).

Deterministic directed weighted minimum vertex cut. A natural generalization of vertex connectivity is the following *directed minimum weighted vertex cut* problem: given a directed graph $G = (V, E)$ with vertex weights $w : V \rightarrow \mathbb{N}$, find a vertex set S whose removal makes G not strongly connected, while minimizing $w(S)$. This problem is very general: it subsumes vertex connectivity, directed edge min-cut, max flow, and many others.

A randomized algorithm was proposed for this problem in $\tilde{O}(mn)$ time [\[HRG00\]](#) 25 years ago. Although there have been many attempts at derandomization for the unweighted case [\[Gab06; SY22\]](#), no deterministic algorithm was known for the vertex-weighted case even to break the trivial algorithm that makes n^2 calls to max flow. Our contribution is to close this gap up to subpolynomial factors.¹

Theorem 5.5. *There is a deterministic algorithm solving the directed minimum weighted vertex cut problem in $m^{1+o(1)}n$ time.*

The way we obtain [Theorem 5.5](#) is to use the weighted version of common-neighborhood clustering, [Theorem 5.2](#), which breaks the problem into easier pieces and allows it to be solved there. More details can be found in [Paper E](#).

¹After we closed this gap, a subsequent work broke the randomized bound from mn to $mn^{0.99+o(1)}$ [\[CT25\]](#).

5.3 Construction

A proof of existence. We first give a simple, but inefficient, algorithm that shows the existence of common-neighborhood clustering. Define the neighborhood-difference graph $G' = (V, E')$ as follows: for any pair of vertices $(u, v) \in V \times V$, $(u, v) \in E'$ if and only if $|N_G(u) \Delta N_G(v)| \leq 2|L^*|$ where (L^*, S, R) is one of the minimum vertex cut of the input graph.

We also need the notion of the sparse neighborhood cover that is studied in the literature [Awe+98]. We define it as follows.

Definition 5.6 (Sparse neighborhood cover (SNC)). Given an undirected graph $G = (V, E)$, the *sparse neighborhood cover* of G is a set of vertex sets $\mathcal{C}$ (called clusters) such that

1. (sparse) each vertex $v \in V$ is contained in at most $\tilde{O}(1)$ clusters in $\mathcal{C}$,
2. (low diameter) the diameter of $G[C]$ for any $C \in \mathcal{C}$ is at most $O(\log n)$,
3. (cover) for every $v \in V$, there exists a cluster $C \in \mathcal{C}$ such that $N_G[v] \subseteq C$.

It is also proved in [Awe+98] that SNC can be constructed efficiently. We claim that applying SNC on G' gives us a common-neighborhood clustering. Let us verify each property of common-neighborhood clustering one by one.

1. Each vertex $u \in V'$ is contained in at most $\tilde{O}(1)$ clusters in $\mathcal{C}$ according to the (*sparse*) condition.
2. For any $C \in \mathcal{C}$ and any $u, v \in C$, there exists a path from u to v in G' with length $O(\log n)$ according to *low diameter*, denoted by $(u, v_1, v_2, \dots, v)$. Notice that an edge $(u, v_1) \in E'$ means we can delete and add $2|L^*|$ nodes from $N_G(u)$ to get $N_G(v_1)$. We can repeat this argument along the path, which shows that $|N_G(u) \Delta N_G(v)| = O(|L^*| \log n)$. This proves that C has neighborhood difference at most $O(|L^*| \log n)$, i.e., it is a common-neighborhood cluster.
3. L^* has neighborhood difference at most $2|L^*|$ as observed in [Li+21], L^* must be a clique in G' according to the definition of G' (actually here we do not require $G[L^*]$ to be connected, this additional property is to assist a faster algorithm which we will explain later). Thus, take an arbitrary $v \in L^*$, and note $L^* \subseteq N_{G'}[v]$, which means L^* is contained in some cluster due to property (*cover*).

This completes the existential proof. □

The construction is not efficient however. We showed in [Paper D](#) and [Paper E](#) how to build it efficiently in different settings.

CHAPTER 6

Steiner Shortcut

This chapter is organized as follows. In [Section 6.1](#), we motivate and define Steiner shortcuts. In [Section 6.2](#), we extend the definition of Steiner shortcuts to broader settings that preserve distance and flow information. This chapter is based on [Paper F](#) and [Paper G](#).

6.1 Introduction to Steiner Shortcut

Shortcuts were initially proposed by Thorup [[Tho92](#)]. In a graph, we say that u can reach v if there is a path from u to v . A shortcut is defined as follows.

Definition 6.1 (Shortcut). Given an arbitrary directed graph $G = (V, E)$, a *shortcut of depth d* or *d -shortcut* is a set of additional edges $E' \subseteq V \times V$ that gives a graph $G' = (V, E \cup E')$ such that, for every $a, b \in V$, (1) if a can reach b in G , then a can reach b in G' through a path with d edges, and (2) if a can reach b in G' , then a can reach b in G .

This can be viewed as a graph lift-and-project object as well: the reachability problem, which asks which vertices a given source vertex s can reach, is lifted from G to G' . The graph G' , although larger, is more structured and allows reachability to be solved faster. For example, running BFS from s up to depth d gives an algorithm with depth d and linear work in the parallel model.

Shortcutting is closely connected to the problem of parallel reachability: if one can compute a depth- d shortcut of linear size, meaning $O(m)$ size where m is the number of edges of the input graph, for any directed graph, then one can solve reachability in parallel with linear work and depth d . The first question to ask is therefore whether such an object really exists, namely

What is the smallest d such that every directed graph admits a d -shortcut of linear size?

Unfortunately, strong lower-bound results rule out subpolynomial d : a long line of work [[Hes03](#); [HP21](#); [BH23](#); [VXX24](#)] leads to a lower bound of $n^{1/5}$.

A new viewpoint: Steiner shortcut. Instead of giving up on the search for a subpolynomial-depth shortcut, we revisit the definition in [Definition 6.1](#): if the goal is to lift a problem to a larger but more structured graph $G' = (V', E')$, then the vertex set V' does not need to be the original vertex set V . This lift-and-project viewpoint leads to the following definition of a Steiner shortcut.

Definition 6.2 (Steiner Shortcut). Given an arbitrary directed graph $G = (V, E)$, a *Steiner shortcut of depth d* is a set of additional vertices V' and additional edges $E' \subseteq (V \cup V') \times (V \cup V')$ that gives

a graph $G' = (V \cup V', E \cup E')$ such that, for every $a, b \in V$, (1) if a can reach b in G , then a can reach b in G' through a path with d edges, and (2) if a can reach b in G' , then a can reach b in G .

Why one copy per original vertex suffices. At first glance, [Definition 6.2](#) may seem more restrictive than the graph-projection viewpoint from the introduction, where a vertex of G may correspond to multiple vertices in G' . For reachability shortcuts, however, the two viewpoints are essentially equivalent.

Suppose a more general projection maps a set of vertices $\pi^{-1}(v)$ in the auxiliary graph to a vertex $v \in V$. We can choose one representative $r_v \in \pi^{-1}(v)$ and identify it with the original vertex v . All remaining vertices in $\pi^{-1}(v) \setminus \{r_v\}$ are treated as Steiner vertices. Finally, we add bidirectional edges between v and each vertex in $\pi^{-1}(v) \setminus \{r_v\}$. This allows paths to enter and leave any copy of v through the representative v , while introducing only Steiner vertices besides the original vertex set. Thus, for the purpose of reachability shortcuts, the formulation in [Definition 6.2](#) is sufficient.

Circumventing the lower bound. In [Paper F](#), we show that this new definition circumvents all previous lower-bound attempts [[Hes03](#); [HP21](#); [BH23](#); [VXX24](#)]. Moreover, despite several attempts in our work, we cannot rule out the existence of an almost-linear-size Steiner shortcut with depth $n^{o(1)}$ for arbitrary directed graphs. This leaves a promising direction for future work: prove or disprove the following Steiner shortcut conjecture.

Is there an almost-linear-size Steiner shortcut of subpolynomial depth for every directed graph?

Applications of the Steiner shortcut conjecture. Proving the Steiner shortcut conjecture would have a huge impact on the parallel computing community: it could lead to the first almost-optimal reachability algorithm, both in work and depth, which currently seems out of reach.

Moreover, by extending the definition to preserve distances and flows, it could also lead to the first combinatorial almost-linear-time algorithms for bipartite matching, max flow, and minimum-cost flow, all with almost-linear work and subpolynomial depth. This would reshape our understanding of graph algorithms.

On undirected graphs, this has already been achieved in our work [Paper G](#): we use an almost-linear-size Steiner shortcut of subpolynomial depth that preserves properties beyond reachability—namely, flow and cost—to obtain the first almost-linear-work combinatorial algorithm for approximate MCF on undirected graphs, with subpolynomial depth. Whether a similar result can be obtained for directed graphs remains wide open. We discuss this formally in the next section.

6.2 Generalized Steiner Shortcuts

Let us refer to shortcuts in [Definition 6.1](#) as *reachability shortcuts*. In this section, we formalize the generalization of reachability shortcut that also preserves distances and flows. Then, we explain why the construction of these shortcuts would imply algorithmic breakthroughs in parallel algorithms for shortest paths and maximum flow in [Section 6.2.1](#). For convenience, we assume lengths and capacities are polynomially bounded integers in this section.

Here, we define a notion of *length-constrained flow shortcuts* (LC-flow shortcuts). This is a shortcut added to the graph so that every multi-commodity flow can be routed with approximately the same congestion and length, but all flow paths only require a small number of edges.

To formalize this, we need notation for multi-commodity flows. Let $G = (V, E)$ be a graph with edge capacity $c : E \rightarrow \mathbb{R}_{\geq 0}$ and edge length $\ell : E \rightarrow \mathbb{R}_{> 0}$. A *(multi-commodity) flow* F in

G assigns value $F(P) \geq 0$ for each directed path P in G . We say that P is a flow path of F if $F(P) > 0$. The *congestion* on edge e of F is $\text{cong}_F(e) = \frac{\sum_{P \ni e} F(P)}{c(e)}$ and the congestion of F is $\text{cong}_F = \max_{e \in E} \text{cong}_F(e)$. The *length* of F is the maximum total length overall flow paths of F , i.e., $\max_{P: F(P) > 0} \ell(P)$. The *hopbound* of F is the maximum number of edges overall flow paths of F , i.e., $\max_{P: F(P) > 0} |E(P)|$. For any vertex set U , a *demand* D on U assigns a value $D(u, v) \geq 0$ for each vertex pair $(u, v) \in U$. The demand D_F routed by F is defined as $D_F(u, v) := \sum_{(u, v)\text{-path } P} F(P)$. We say that D is routable with length ℓ , congestion κ , and hopbound h if there exists a flow F with length ℓ , congestion κ , and hopbound h that routes D . When D is routable in G with congestion 1, we sometimes just say that D is routable in G . With this, we are ready to define LC-flow shortcuts.

Definition 6.3 (Length-Constrained Flow Shortcuts). A shortcut $G' = (V \cup V', E \cup E')$ is a *length-constrained flow shortcut* of G with hopbound h ,¹ congestion slack κ , and length slack s if:

1. Every demand on V routable in G with length λ and congestion 1 is routable in G' with length λs , congestion 1, and hopbound h , and
2. Every demand on V routable in G' with length λ and congestion 1 is routable in G with length λ and congestion κ .

The concept of LC-flow shortcut above is also defined in undirected graphs in our paper [Paper G](#). We show that, for any n -vertex undirected graph G with capacity and length at most $\text{poly}(n)$ and a parameter k , there exists a flow shortcut G' with hopbound $h = O(k)$, length slack $s = O(k)$, congestion slack $\text{poly}(k \log n) n^{O(1/\sqrt{k})}$, and size $|E'| \leq n^{1+O(1/\sqrt{k})} \text{poly}(\log n)$. It is an interesting open problem if an LC-flow shortcut with non-trivial guarantee exists for directed graphs.

Reachability Shortcuts, Distance Shortcuts, and Congestion Shortcuts. Observe that a reachability shortcut is precisely an LC-flow shortcut when the constraints related to length and congestion in [Definition 6.3](#) are removed.

We will use the terms *distance shortcut* and *congestion shortcut* to refer to an LC-flow shortcut when the respective constraint related to congestion and length in the definition of [Definition 6.3](#) is removed.

More specifically, when focusing only on length, [Definition 6.3](#) simplifies to distance shortcuts as follows. Let $\text{dist}_G(s, t)$ denote the distance from s to t in G . Let $\text{dist}_G^h(s, t)$ denote the minimum length of paths with hopbound h from s to t in G .

Definition 6.4 (Distance Shortcuts). A shortcut $G' = (V \cup V', E \cup E')$ is a *distance shortcut* of G with hopbound h and length slack s if, for every vertex pair $(s, t) \in V \times V$,

1. If $\text{dist}_G(s, t) \leq \lambda$, then $\text{dist}_{G'}^h(s, t) \leq \lambda s$ and
2. If $\text{dist}_{G'}(s, t) \leq \lambda$, then $\text{dist}_G(s, t) \leq \lambda$.

When focusing only on flow (or equivalently flow *congestion*), [Definition 6.3](#) simplifies to congestion shortcuts as follows:

Definition 6.5 (Congestion Shortcuts). A shortcut $G' = (V \cup V', E \cup E')$ is a *congestion shortcut* of G with hop bound h and congestion slack κ if

1. Every demand on V routable in G is routable in G' with hopbound h , and
2. Every demand on V routable in G' is routable in G with congestion κ .

¹The hopbound in this section has the same purpose as the diameter in previous sections. We change our terminology to hopbound to avoid confusion, because the graph here has edge lengths.

6.2.1 Exact Shortest Paths via Distance Shortcuts

In this section, we observe that a near-optimal parallel algorithm for constructing a distance shortcut with length slack $(1+o(1/\log n))$ would imply a near-optimal parallel algorithm for *exact* single-source shortest paths with $\tilde{O}(m)$ work and $\text{polylog}(n)$ depth.

In the SSSP problem, given a graph G and a vertex s , we need to output the exact distance $d(u)$ from s to every other vertex u in G . For α -approximate SSSP, we are given a graph G and a vertex s , output an approximate distance $\tilde{d}(u)$ from s to every other vertex u in G , such that $d(u) \leq \tilde{d}(u) \leq \alpha d(u)$.

The reduction from parallel exact SSSP to distance shortcut construction follows from combining tools from [KS97] and [Roz+23].

Theorem 6.6. *For any n, m, m', h , suppose there is an algorithm $\mathcal{A}$ which takes an $O(m)$ -edge $O(n)$ -vertex directed graph and computes a distance shortcut of hopbound $O(h)$, length slack $(1+o(1/\log n))$ and size $O(m')$. Then, there is an algorithm computing exact SSSP of an m -edges n -vertex directed graph with $\tilde{O}(1)$ calls to $\mathcal{A}$, with additional $\tilde{O}(m + m')$ work and $\tilde{O}(h)$ depth.*

In undirected graphs, there exist distance shortcuts of hopbound $n^{o(1)}$, length slack $(1+1/\log^2 n)$ and size $n^{1+o(1)}$ (e.g. [Coh00; HP19]). Moreover, [EN19] showed how to construct them in $m^{1+o(1)}$ work and $n^{o(1)}$ depth.

If this can be generalized to directed graphs even when the shortcut size is $m^{1+o(1)}$, then Theorem 6.6 would imply an exact SSSP algorithm with $m^{1+o(1)}$ work and $n^{o(1)}$ depth.

Corollary 6.7. *Suppose there is an algorithm that, given $O(m)$ -edge $O(n)$ -vertex directed graph, computes a distance shortcut of hopbound $n^{o(1)}$, length slack $(1+o(1/\log n))$, and size $m^{1+o(1)}$. Then, there is an algorithm for computing exact SSSP in an m -edge n -vertex directed graph in $m^{1+o(1)}$ work and $n^{o(1)}$ depth.*

Indeed, in directed graphs, the state-of-the-art is still far from this goal, because the problem is even harder than reachability shortcut. There exist distance shortcuts of hopbound $\tilde{O}(n^{1/3})$, length slack $(1+\varepsilon)$ and size $\tilde{O}(n/\varepsilon^2)$ [BW23], but they require large polynomial time to construct. [CFR20] showed a parallel algorithm with $\tilde{O}(m/\varepsilon^2)$ work and $n^{0.5+o(1)}$ depth to construct a distance shortcut of hopbound $n^{1/2+o(1)}$, length slack $(1+\varepsilon)$ and size $\tilde{O}(n/\varepsilon^2)$.

Note that, since Theorem 6.6 is a *work-efficient* reduction, any improvement to the $n^{1/2+o(1)}$ hopbound of [CFR20] would immediately improve the state of the art for the parallel exact SSSP problem which currently requires $\tilde{O}(m)$ work and $n^{1/2+o(1)}$ depth [CFR20; Roz+23].

Proof. We first present an algorithm for computing $(1+o(1/\log n))$ -approximate SSSP in an $O(m)$ -edge, $O(n)$ -vertex directed graph with 1 call to $\mathcal{A}$ and additional $\tilde{O}(m + m')$ time and $\tilde{O}(h)$ depth. We then show how to solve exact SSSP of an m -edge n -vertex graph using $\tilde{O}(1)$ calls to the approximation algorithm with additional $\tilde{O}(m)$ work and $\tilde{O}(h)$ depth. Combining the two reductions finishes the proof.

Approximate SSSP. Given a directed G with $O(m)$ edges and $O(n)$ vertices, we first compute a distance shortcut of size $O(m')$, denoted by $G' = (V \cup V', E \cup E')$. We will apply [KS97, Lemma 3.2], to G' , which we state as follows.

Lemma 6.8 ([KS97, Lemma 3.2]). *Given a directed graph G , a source vertex u and parameters h, ε , there is an algorithm computing an h -limited $(1+\varepsilon)$ -approximate shortest path $\tilde{d}_h$ defined as*

$\forall v \in V, d_h(v) \leq \tilde{d}_h(v) \leq (1 + \varepsilon) \cdot d_h(v)$ where $d_h(v)$ is the shortest path distance restricted to only using h edges from the source u , in $\tilde{O}(m\varepsilon^{-1})$ work and $\tilde{O}(h\varepsilon^{-1})$ depth.

We can compute an h -limited $(1 + o(1/\log n))$ -approximate shortest path distances $\{\tilde{d}_h^{G'}(u)\}_{u \in V}$ in $\tilde{O}(m + m')$ work and $\tilde{O}(h)$ depth (by setting the parameter $p = m/k$ in their paper). We use $\{\tilde{d}_h^{G'}(u)\}_{u \in V}$ as the output. This finishes the algorithm description, which uses 1 call to $\mathcal{A}$ with additional $\tilde{O}(m)$ work and $\tilde{O}(1)$ depth.

In their definition, an h -limited $(1 + o(1/\log n))$ -approximate shortest path distances $\{\tilde{d}_h^{G'}(u)\}_{u \in V}$ satisfies that $d_h^{G'}(u) \leq \tilde{d}_h^{G'}(u) \leq (1 + o(1/\log n))d_h^{G'}(u)$, where $d_h^{G'}(u)$ is the shortest path distance restricted to only using h edges. According to [Definition 6.4](#), if we write $d^G(u)$ as the shortest path distance on G , then we have $d_h^{G'}(u) \leq (1 + o(1/\log n)) \cdot d^G(u)$ and $d^G(u) \leq d_h^{G'}(u)$. Thus, we have $d^G(u) \leq \tilde{d}_h^{G'}(u) \leq (1 + o(1/\log n))(1 + o(1/\log n))d^G(u)$, which means $\tilde{d}_h^{G'}(u)$ is a $(1 + o(1/\log n))$ -approximate distance.

Exact SSSP. The following lemma is a direct implication of Theorem 1.7 and Lemma 3.1 in [\[Roz+23\]](#),

Lemma 6.9 ([\[Roz+23\]](#), Theorem 1.7 and Lemma 3.1). *Given a directed graph G , a source $s \in V(G)$, there is an algorithm computing the exact distance from s to every vertex in G , using $\tilde{O}(1)$ calls to a $(1 + O(1/\log n))$ -approximate distance oracle $\mathcal{O}$ on directed graphs and an additional $\tilde{O}(m)$ work and $\tilde{O}(1)$ depth.²*

Given an m edges n vertices graph G , we apply [Lemma 6.9](#) to get an exact distance function by $O(\log n)$ calls to the $(1 + o(1/\log n))$ -approximate distance oracle presented in the last paragraph, and an additional $\tilde{O}(m)$ work and $\tilde{O}(1)$ depth. The lemma follows. $\square$

6.2.2 Parallel Exact Max Flow via Congestion Shortcuts

In this section, we observe that a near-optimal parallel algorithm for constructing a congestion shortcut with congestion slack $\text{polylog}(n)$ (together with an embedding algorithm defined below) would imply a near-optimal parallel algorithm for *exact* maximum flow with $\tilde{O}(m)$ work and $\text{polylog}(n)$ depth.

Definition 6.10 (Backward Mapping). For a shortcut algorithm $\mathcal{A}$ which outputs a shortcut G' of the input graph $G = (V, E)$, a *backward mapping algorithm* for $\mathcal{A}$ takes input G and G' generated by $\mathcal{A}$, along with a flow F' in G' with congestion 1, and outputs a flow F in G routing the same demand as F' with congestion κ .

The reduction from parallel exact max flow to congestion shortcut construction follows quite straightforwardly by repeatedly solving approximate max flow on the residual graph.

Theorem 6.11. *Given n, m, m', h, κ , suppose there is an algorithm $\mathcal{A}$ which takes an $O(m)$ -edge $O(n)$ -vertex directed graph and computes a congestion shortcut of hopbound $O(h)$, congestion slackness κ and size $O(m')$, along with a corresponding embedding algorithm $\mathcal{A}'$ for $\mathcal{A}$, then there is an algorithm computing exact max flow with $\tilde{O}(\kappa)$ calls to $\mathcal{A}$ and $\mathcal{A}'$ and additional $\tilde{O}(\kappa \cdot \text{poly}(h) \cdot (m + m'))$ work and $\tilde{O}(\kappa \cdot \text{poly}(h))$ depth.*

²One might notice that Lemma 4.1 in [\[KS97\]](#) is also about reducing exact shortcut path to approximate shortest path oracles. However, they require a stronger approximate shortest path oracle: The approximate distance must be an exact distance on the shortcut G' . Thus, it cannot be used here.

In undirected graphs, congestion shortcuts with near-optimal quality follow from *tree flow sparsifiers* defined as follows.

Definition 6.12. A *tree flow sparsifier* T of G with congestion slack κ is such that

1. The set of leaves of T corresponds to $V(G)$.
2. Every demand on $V(G)$ routable in G is routable in T
3. Every demand on $V(G)$ routable in T is routable in G with congestion κ .

Clearly from the definition, a tree flow sparsifier with depth d and congestion slack κ is a congestion shortcut with congestion slack κ , hopbound $2d$, and size $O(n)$. There exists a tree flow sparsifier with depth $O(\log n)$ and congestion slack $O(\log^2 n \log \log n)$ [HHR03]. Moreover, there is a parallel algorithm that constructs a tree flow sparsifier with depth $O(\log n)$ and congestion slack $O(\log^9 n)$ in $\tilde{O}(m)$ work and $\text{polylog}(n)$ depth [Aga+24].

If this can be generalized to directed graphs even when the shortcut size is $\tilde{O}(m)$, then Theorem 6.11 would imply an exact max flow algorithm with $\tilde{O}(m)$ work and $\text{polylog}(n)$ depth.

Corollary 6.13. Suppose there is an algorithm that, given $O(m)$ -edge $O(n)$ -vertex directed graph, computes a congestion shortcut of hopbound $\text{polylog}(n)$, congestion slack $\text{polylog}(n)$, and size $\tilde{O}(m)$. Then, there is an algorithm for computing exact max flow in an m -edge n -vertex directed graph in $\tilde{O}(m)$ work and $\text{polylog}(n)$ depth.

Unfortunately, there is no prior work on this, and we leave it an interesting open problem.

Proof of Theorem 6.11

Proof. We first give an algorithm computing an approximate max flow that routes a feasible flow whose value is at least $1/(2\kappa)$ times the optimal value on the input graph. Then show how to turn it into an exact max flow algorithm.

Approximate max flow. We first use $\mathcal{A}$ to compute a congestion shortcut on G , denoted by E' . Let $G' = G \cup E'$. We will use Theorem 3.1 (1) of [HHS23], stated as follows.

Lemma 6.14 (Theorem 3.1 (1) of [HHS23]). *Given a directed graph G with edge capacities and unit edge length, length bound $h \geq 1$, approximation parameter $\varepsilon > 0$, and source and sink $s, t \in V$, there is an algorithm computing a $1 - \varepsilon$ approximate maximum h -length flow, in the sense that the flow value is at least $1 - \varepsilon$ times the maximum h -length flow from s to t in G . The algorithm takes $\tilde{O}(m \cdot h^{17} \cdot \frac{1}{\varepsilon^9})$ work and $\tilde{O}(h^{17} \cdot \frac{1}{\varepsilon^9})$ depth.*

We use Lemma 6.14 with parameter $\varepsilon = 1/2$, h the same as our h , and the digraph the same as G' with unit length. The guarantee for the output is a flow F' , which is a $1 - \varepsilon$ approximation for maximum h -length flow, which, in our case, is the maximum flow with hop bound h since the graph has unit length. Then, we call the backward mapping algorithm $\mathcal{A}'$ to turn F' into a flow F on G with congestion κ . To make F feasible, we scale the flow on each edge down by a factor of κ , and denote the final flow as F/κ . According to Definition 6.3, every flow on G corresponds to a feasible flow on G' with hop bound h and the same value. Thus, F' has a value of at least $1/2$ times the max flow value on G , which implies that F/κ is a $1/(2\kappa)$ approximate max flow on G .

The algorithm takes an input graph of m edges and n vertices, uses 1 call to $\mathcal{A}$ and $\mathcal{A}'$, and additional $\tilde{O}(\text{poly}(h) \cdot (m + m'))$ work and $\text{poly}(h)$ depth using Lemma 6.14.

Exact max flow. If we have an algorithm computing ε -approximate max flow (here $\varepsilon = 1/(2\kappa)$), then we can use a folklore reduction to get the exact max flow: iteratively compute approximate max flows on the residual graph. In $\tilde{O}(1/\varepsilon)$ iterations the optimal flow value of the residual graph will be decreased to $1/\text{poly}(n)$. Notice that we can also apply the standard rounding algorithm (for example, see [Coh95, §5]) if we want to get an integer flow of the max flow value.

The algorithm uses $\tilde{O}(\kappa)$ calls to the approximate max flow algorithm, which corresponds to $\tilde{O}(\kappa)$ calls to $\mathcal{A}, \mathcal{A}'$, and additional $\tilde{O}(\kappa \cdot \text{poly}(h) \cdot (m + m'))$ work and $\tilde{O}(\kappa \cdot \text{poly}(h))$ depth. $\square$

Part II

Included Papers

Paper A

DAG Projections: Reducing Distance and Flow Problems to DAGs

Bernhard Haeupler, Yonggang Jiang, Thatchaphol Saranurak

Published at ACM Symposium on Theory of Computing (STOC) 2026.

Full version available at <https://arxiv.org/abs/2604.04752>.

Abstract

We show that every directed graph G with n vertices and m edges admits a directed acyclic graph (DAG) with $m^{1+o(1)}$ edges, called a *DAG projection*, that can either $(1 + 1/\text{polylog}(n))$ -approximate distances between all pairs of vertices (s, t) in G , or $n^{o(1)}$ -approximate maximum flow between all pairs of vertex subsets (S, T) in G . Previous similar results suffer a $\Omega(\log n)$ approximation factor for distances [AHW25; Fil26] and, for maximum flow, no prior result of this type is known.

Our DAG projections admit $m^{1+o(1)}$ -time constructions. Further, they admit almost-optimal parallel constructions, i.e., algorithms with $m^{1+o(1)}$ work and $m^{o(1)}$ depth, assuming the ones for approximate shortest path or maximum flow *on DAGs*, even when the input G is not a DAG.

DAG projections immediately transfer results on DAGs, usually simpler and more efficient, to directed graphs. As examples, we improve the state-of-the-art of $(1 + \varepsilon)$ -approximate distance preservers [HXX25] and single-source minimum cut [CLL13], and obtain simpler construction of $(n^{1/3}, \epsilon)$ -hop-set [KP22b; BW23] and combinatorial max flow algorithms [Ber+24; Ber+25a].

Finally, via DAG projections, we reduce major open problems on almost-optimal parallel algorithms for *exact* single-source shortest paths (SSSP) and maximum flow to easier settings:

- From exact directed SSSP to exact *undirected* ones,
- From exact directed SSSP to $(1 + 1/\text{polylog}(n))$ -approximation on DAGs, and
- From exact directed maximum flow to $n^{o(1)}$ -approximation on DAGs.

Distance	From $\rightarrow$ To	Approximation	Total size
[MN07]	undirected $\rightarrow$ tree	$O(k)$	$O(kn^{1+1/k})$
[BFN19]	undirected $\rightarrow$ tree	$O(n^{1/k} \log^{1-1/k} n)$	nk
[AHW25]	directed $\rightarrow$ DAG	$O(\log^3 n \log \log n)$	$O(m \log^3 n)$
[Fil26]	directed $\rightarrow$ DAG	$O(\log n \log \log n)$	$O(m \log^3 n)$
Ours (Theorem A.1)	directed $\rightarrow$ DAG	$(1 + 1/\text{polylog}(n))$	$m^{1+o(1)}$

Maximum flow	From $\rightarrow$ To	Approximation	Total size
[Rac02]	undirected $\rightarrow$ tree	$O(\log^3 n)$	$2n$
[HHR03]	undirected $\rightarrow$ tree	$O(\log^2 n \log \log n)$	$2n$
[Rac08]	undirected $\rightarrow$ tree	$O(\log n)$	$O(mn)$
[RS14]	undirected $\rightarrow$ tree	$O(\log n \log \log n)$	$2n$
Ours (Theorem A.2)	directed $\rightarrow$ DAG	$n^{o(1)}$	$m^{1+o(1)}$

Table A.1: Summary of results on approximating undirected and directed graphs by trees and DAGs, respectively. For distances, we trade $O(m \log^3 n)$ size and $\Omega(\log n)$ approximate for $m^{1+o(1)}$ size and $(1 + o(1))$ approximation. For maximum flow, we give the first such DAG.

A.1 Introduction

Distance and maximum flow structures of *undirected* graphs can be approximated with trees, using *tree covers* [MN07] and *tree cut sparsifier* [Rac02], respectively. Both objects are highly influential across many areas including data structures [MN07; TZ05], fast algorithms [RS14; Van+24], and online algorithm [Bar96a; Alo+06].¹ This motivates the research program on showing analogous results in directed graphs.

In this paper, we make significant progress in this direction and obtain many applications. Below, we first review tree covers and tree cut sparsifier, as well as the state-of-the-art of their directed analogous objects summarized in Table A.1.

For any undirected or directed graph G with n vertices and m edges, vertex pair $s, t \in V(G)$, and vertex subset pair $S, T \subseteq V(G)$, let $\text{dist}_G(s, t)$ denote the distance from s to t and $\text{maxflow}_G(S, T)$ denote the maximum flow value from S to T .

Approximating Distances. A *tree cover* of a graph G is a collection $\mathcal{T}$ of trees where $\text{dist}_G(s, t) \leq \min_{T \in \mathcal{T}} \text{dist}_T(s, t) \leq \alpha \cdot \text{dist}_G(s, t)$ for every $s, t \in V(G)$ and α is called the approximation factor. Every *undirected* graph admits, for any $k \geq 1$, a tree cover $\mathcal{T}$ containing $O(kn^{1/k})$ trees with $O(k)$ approximation and total size $|\sum_{T \in \mathcal{T}} E(T)| = O(kn^{1+1/k})$ [MN07].

Motivated by the fact that directed cyclic graphs (DAGs) are usually more algorithmic friendly than general directed graphs, Assadi, Hoppenworth, and Wein [AHW25] recently introduced a *DAG cover* as a directed analog of a tree cover. A DAG cover of a graph G is a collection $\mathcal{D}$ of DAGs where $\text{dist}_G(s, t) \leq \min_{D \in \mathcal{D}} \text{dist}_D(s, t) \leq \alpha \cdot \text{dist}_G(s, t)$. They showed that every directed graph with edge weights from $\{1, 2, \dots, \text{poly}(n)\}$ admits a DAG cover containing $O(\log n)$ DAGs with $O(\log^3 n \log \log n)$ approximation and total size $|\sum_{D \in \mathcal{D}} E(D)| = O(m \log^3 n)$. They also gave a $\tilde{O}(m)$ -time algorithm to construct it. Later, Filtser [Fil26] strictly improved the approximation factor to $O(\log n \log \log n)$, while using the same size and construction time.

Both [AHW25; Fil26] left as an open problem whether their approximation factor can be improved further.

Approximating Maximum Flow. A *tree cut sparsifier* of G is a tree T where, for every $S, T \subseteq V(G)$, $\text{maxflow}_G(S, T) \leq \text{maxflow}_T(S, T) \leq \alpha \cdot \text{maxflow}_G(S, T)$. Every undirected graph admits a tree cut sparsifier T with $O(\log n \log \log n)$ approximation and size $|E(T)| \leq 2n$ [RS14]. However, it remains unknown if there exists any set of DAGs that could approximate maximum flows of a directed graph.

¹There are probabilistic version of tree covers and tree cut sparsifiers called *probabilistic/stochastic tree embedding* [Bar96a; FRT04] and *probabilistic tree cut sparsifiers* [Rac08], respectively.

A.1.1 Our Structural Results

We make significant progress on both sides. First, for the distance side, we improve the approximation factor of $O(\log n \log \log n)$ [Fil26] down to $(1 + 1/\text{polylog}(n))$ using a DAG with slightly larger $m^{1+o(1)}$ size. Second, for the flow side, we give the first directed analog of tree cut sparsifiers using a DAG of size $m^{1+o(1)}$.

To describe our DAGs more precisely, we need a notion of *DAG projection*. A *partial projection* to G is a graph G' whose vertices of G' are either copies of vertices in G or Steiner vertices. Formally, there exists a mapping $\pi : V(G') \rightarrow V(G) \cup \{\perp\}$. We say $u' \in \pi^{-1}(u)$ is a copy of u and $u' \in \pi^{-1}(\perp)$ is a Steiner vertex.² The *width* of G' is the maximum number of copies, i.e., $\max_{u \in V(G)} |\pi^{-1}(u)|$. If G' is a DAG, then we say G' is a *DAG projection*.

(1 + ϵ)-Distance-Preserving DAG. Our first result is a DAG projection with almost-linear size that can $(1 + 1/\text{polylog}(n))$ -approximate distances.

Theorem A.1. *For any directed graph G with edge weights from $\{0, 1, 2, \dots, \text{poly}(n)\}$ and $\epsilon \geq 1/\text{polylog}(n)$, there exists a DAG projection G' to G of size $|E(G')| = m^{1+o(1)}$ and width $n^{o(1)}$ such that, for every $s, t \in V(G)$,*

$$\text{dist}_G(s, t) \leq \text{dist}_{G'}(\pi^{-1}(s), \pi^{-1}(t)) \leq (1 + \epsilon) \text{dist}_G(s, t).$$

Moreover, there is a randomized algorithm for computing G' in $m^{1+o(1)}$ time.

Recall that $\text{dist}_{G'}(\pi^{-1}(s), \pi^{-1}(t))$ is the distance from any copy of s to any copy of t in G' . It can be easily computed, for example, by adding to G' zero-weight edges from dummy source s_0 to $\pi^{-1}(s)$ and from $\pi^{-1}(t)$ to dummy sink t_0 , and then computing the distance from s_0 to t_0 .

Compared with [AHW25; Fil26], their DAG cover $\mathcal{D}$ only guarantees $O(\log n \log \log n)$ approximation albeit with slightly better $O(m \log^3 n)$ total size.

$n^{o(1)}$ -Congestion-Preserving DAG. Our next result is a DAG projection with almost-linear size that can $n^{o(1)}$ -approximate maximum flows. This is the first directed analog of tree cut sparsifier in undirected graphs.

Theorem A.2. *For any directed graph G with edge capacity from $\{1, 2, \dots, \text{poly}(n)\}$, there exists a DAG projection G' to G of size $|E(G')| = m^{1+o(1)}$ and width $n^{o(1)}$ such that, for every $S, T \subseteq V(G)$,*

$$\text{maxflow}_G(S, T) \leq \text{maxflow}_{G'}(\pi^{-1}(S), \pi^{-1}(T)) \leq n^{o(1)} \text{maxflow}_G(S, T).$$

Moreover, there is a randomized algorithm for computing G' in $m^{1+o(1)}$ time.

Observe that $\text{maxflow}_{G'}(\pi^{-1}(S), \pi^{-1}(T))$ can be computed, for example, by adding to G' infinite-capacity edges from dummy source s_0 to $\pi^{-1}(S)$ and from $\pi^{-1}(T)$ to dummy sink t_0 , and then computing the maximum flow from s_0 to t_0 .

We emphasize that there are exponentially many $\text{maxflow}_G(S, T)$ values that G' preserves. So, it is unclear that G' exists even if we allow $|E(G')| = O(n^2)$. Note that even though our approximation is $n^{o(1)}$, the best approximation for tree cut sparsifiers in undirected graphs is $\Omega(\log n)$.

Almost-Optimal Size. The size of $m^{1+o(1)}$ in both Theorems A.1 and A.2 are optimal up to $m^{o(1)}$ factor. Indeed, since reachability information of m -edge directed bipartite graphs can encode $\Omega(m)$ bits of information, the $\Omega(m)$ size lower bound holds even for arbitrary data structures that can answer reachability only, let alone approximating distances or flow. In contrast, in undirected graphs, tree covers or tree cut sparsifiers may have total size $o(m)$.

²A *projection* has a stricter requirement that $\pi : V(G') \rightarrow V(G)$ is a graph homomorphism. That is, there is no Steiner vertex and if $(u', v') \in E(G')$, then $(u, v) \in E(G)$ where $u = \pi(u')$ and $v = \pi(v')$ (and they have the same edge weight). See Definition A.10, which was also used in [Ash+24a]. We usually omit the term “partial” and distinguish the two concepts only when the difference is crucial.

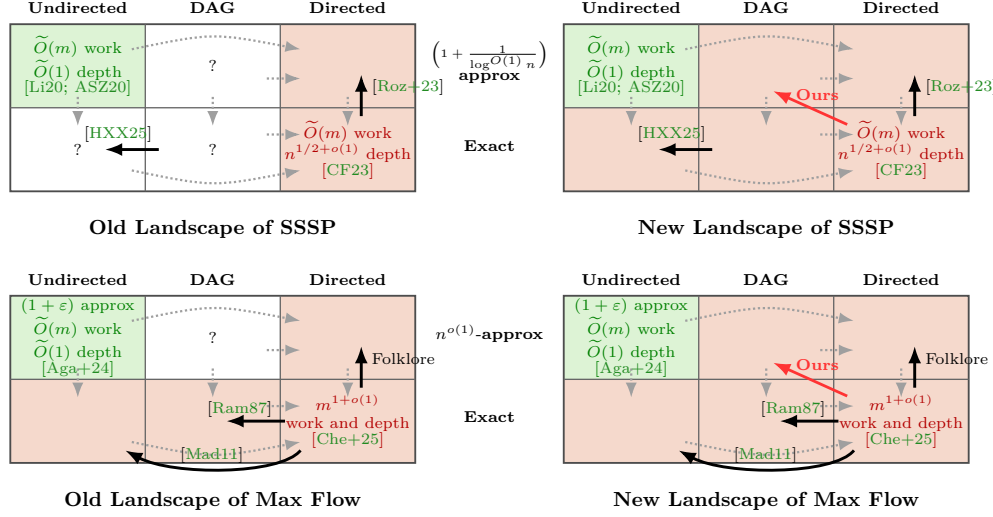


Figure A.1: Old and new landscapes of parallel SSSP and maximum flow algorithms. The green area highlights the settings where near-optimal parallel algorithms are known. The red area highlights the settings as hard as the exact directed setting. The solid arrows represent non-trivial efficient parallel reductions, while the dotted arrows represent trivial ones.

Parallel Construction: Reductions to Approximations on DAGs. Not only we can construct the DAG projections from [Theorems A.1 and A.2](#) in almost-optimal $m^{1+o(1)}$ time in the sequential setting, our constructions are also almost optimal in the parallel settings (i.e., they take $m^{1+o(1)}$ work and $m^{o(1)}$ depth), assuming almost-optimal parallel algorithms for *approximate* shortest paths or maximum flow *on DAGs*.

To formalize this, we say that there is an *efficient parallel reduction* from problem $\mathcal{A}$ to problem $\mathcal{O}$ if, given that $\mathcal{O}$ can be solved in w work and d depth on a graph with m edges, then $\mathcal{A}$ can be solved in $w \cdot m^{o(1)}$ work and $d \cdot m^{o(1)}$ depth on a graph with m edges.

Theorem A.3 (Efficient parallel reductions). *There are efficient parallel reductions from constructing DAG projections in [Theorems A.1 and A.2](#) to, respectively, computing $(1 + 1/\text{polylog}(n))$ -approximate single-source shortest path and $n^{o(1)}$ -approximate maximum flow on a DAG whose topological order is given.*

A.1.2 New Landscape of Parallel Shortest Paths and Maximum Flow

Via the efficient parallel reduction in [Theorem A.3](#), we reduce major open problems of finding almost-optimal parallel algorithms for *exact* single-source shortest paths (SSSP) and maximum flow to easier settings, leading to a clean landscape of both problems. See [Figure A.1](#).

More precisely, using our DAG projections, we obtain efficient parallel reductions

1. From exact directed SSSP to $(1 + 1/\text{polylog}(n))$ -approximation on DAGs,
2. From exact directed SSSP to exact *undirected* ones (using [\[HXX25\]](#)), and
3. From exact directed maximum flow to $n^{o(1)}$ -approximation on DAGs. This reduction was not known even in the classical *sequential* setting.³

Consider algorithms in the six settings based on the following combinations: (1) exact or approximate, and (2) on directed graphs, DAGs, or undirected graphs. Our reductions categorize these six settings for both SSSP and maximum flow into only two regimes. The *easy* regime consists of only the approximate undirected setting, and the *hard* regime contains the remaining five settings. These five settings are equivalent in the sense they all admit algorithms with the same work and depth up to subpolynomial factor.

³It confirms the informal statement in [\[Ber+24\]](#) about computing maximum flow, saying that “the main bottleneck seems to be a fast $n^{o(1)}$ -approximation for DAGs”.

This significantly cleans up the landscape of both problems. With this new landscape, to improve the state-of-the-art of either exact directed SSSP or maximum flow, it suffices to improve either (1) approximation algorithms on DAGs, or (2) exact algorithms on undirected graphs.

We note that we crucially exploit our approximation guarantees. If they were worse, our reductions between exact and approximate settings above would not work with current techniques. For example, they would fail if the distance-preserving DAG projection only guaranteed 1.01-approximation.⁴ Thus, the previous results with $\Omega(\log n)$ approximate [AHW25; Fil26] indeed do not work. Similarly, if the congestion-preserving DAG projection guaranteed $n^{0.1}$ -approximation, then the reduction would have been too inefficient.

A.1.3 Transferring Results from DAGs to Directed Graphs

Lastly, our DAG projections immediately transfer algorithms on DAGs, usually simpler and more efficient, to directed graphs. Below, we list problems that we improve the state-of-the-art or obtain simpler algorithms via DAG projections.

Applications of Distance-Preserving DAG Projections.

Distance preservers: improved. Given a directed graph $G = (V, E)$ with edge lengths and a set of demand pairs $P \subseteq V \times V$, a $(1 + \varepsilon)$ -approximate distance preserver $((1 + \varepsilon)$ -DP) is a subgraph $H \subseteq G$ such that for every $(s, t) \in P$, we have $\text{dist}_H(s, t) \leq (1 + \varepsilon) \cdot \text{dist}_G(s, t)$. A long line of work (e.g. [CE06; Bod17; HXX25]) have been trying to find $(1 + \varepsilon)$ -DPs of smallest size $|E(H)|$.

Currently, the best bounds for DAGs are strictly better than the ones for general directed graphs. To simplify the discussion, we focus on the $O(n)$ -size regime. There exists a $(1 + \varepsilon)$ -DP for DAGs of size $O(n + p\sqrt{n})$, which gives $O(n)$ as long as $p = O(\sqrt{n})$ [CE06; HXX25].⁵ However, the best bound for general directed graphs achieves $O(n)$ size only when $p = O(n^{1/3})$ [Bod17].⁶ Our distance-preserving DAG projection **Theorem A.1** closes this gap up to $n^{o(1)}$ factor.

Corollary A.4. *For $\varepsilon \geq 1/\text{polylog}(n)$, there exists a $(1 + \varepsilon)$ -approximate distance preserver for any n -node directed graph with polynomial edge lengths and p demand pairs of size $(n + p\sqrt{n}) \cdot n^{o(1)}$, which is $n^{1+o(1)}$ for $p = O(\sqrt{n})$.*

Corollary A.4 exploits that the width in **Theorem A.1** is $n^{o(1)}$ so that we do not blow up the number of vertices, and also that G' is actually a *projection* (see **Definition A.10**), not only a partial projection.

Hop-set: simplified. Given a directed graph $G = (V, E)$ with edge lengths, a (β, ε) -hop-set of G is a set of additional edges $H \subseteq V \times V$ added to the graph so that for every $s, t \in V$ we have (i) $\text{dist}_G(s, t) \leq \text{dist}_{G \cup H}(s, t)$, and (2) $\text{dist}_{G \cup H}^{(\beta)}(s, t) \leq (1 + \varepsilon) \cdot \text{dist}_G(s, t)$ where $\text{dist}_{G \cup H}^{(\beta)}(s, t)$ denotes the lengths of the shortest path from s to t using at most β edges.

The breakthrough result of Kogan and Parter [KP22b] showed the existence of linear-sized $(O(n^{1/3}), \varepsilon)$ -hop-sets for DAGs, but only linear-sized $(O(n^{2/5}), \varepsilon)$ -hop-sets for general graphs. With significant effort, Bernstein and Wein [BW23] closed this gap by showing linear-sized $(O(n^{1/3}), \varepsilon)$ -hop-sets for any graph.

Theorem A.1 can bypass this significant effort and close the gap up to $n^{o(1)}$ factor in a black-box way. Indeed, by applying the construction of [KP22b] for DAGs on top of our distance-preserving DAG projection, we immediately obtain $n^{1+o(1)}$ -sized $(O(n^{1/3}), \varepsilon)$ -hop-sets.

Potential applications. There are gaps between DAGs and general directed graphs in several variants of *min-distance* problems [DK21; CZ22]. Also, the recent approximate restricted SSSP algorithm [ABK25] is much simpler on DAGs. Thus, DAG projections could potentially help close these gaps or simplify the algorithms.

⁴It fails because the reduction in **Lemma A.15** by [Roz+23] requires $(1 + o(1/\log n))$ -approximate oracle.

⁵This holds even in the exact setting by applying Lemma 4.6 of [HXX25] to the exact distance preservers on undirected graphs by [CE06].

⁶The bound of $O(n + n^{2/3}p)$ by Bodwin [Bod17] works even in the exact setting.

Applications of Congestion-Preserving DAG Projections.

Combinatorial Max flow: simplified. The combinatorial maximum flow algorithms by [Ber+24; Ber+25a] runs in $\tilde{O}(n^2)$ time, near-optimal on dense graphs. They first showed a very simple push-relabel algorithm that gives $O(1)$ -approximation on DAGs and then generalized to general graphs via expander hierarchies in a white-box way. Our reduction from exact max flow to $n^{o(1)}$ -approximation on DAGs (via Theorem A.2), directly generalizes their algorithms for DAGs to general graphs in a black-box way with an additional $n^{o(1)}$ factor in time.

Bounded Single-Source Max flow: improved. Let G be a graph with positive integral edge capacities and source $s \in V(G)$. Let $\text{maxflow}_G^k(s, t) = \min\{k, \text{maxflow}_G(s, t)\}$ denote the k -bounded maximum flow value from s to t . If G is a DAG, the algorithm based on network coding by [CLL13] can compute $\text{maxflow}_G^k(s, t)$ for all $t \in V(G)$ in $O(k^{\omega-1}m)$ total time, while the best known algorithm on general graphs is to trivially compute $\text{maxflow}_G^k(s, t)$ for every t using $\Omega(mn)$ time.

Using our congestion-preserving DAG projection, we obtain a non-trivial algorithm that beats the mn bound when $k \in (n^{\Omega(1)}, n^{1/\omega-\Omega(1)})$.

Theorem A.5. *There is a randomized algorithm that, given a directed graph with positive integral edge capacities, a source s , and an integer k , computes $n^{o(1)}$ -approximation of $\text{maxflow}_G^k(s, t)$ for all vertices t in $k^\omega m^{1+o(1)}$ time.*

Potential application. Vertex cut sparsifiers for k terminals with optimal $\Theta(k^2)$ size are only known on DAGs [HLW21], but the best construction on general graphs still requires $O(k^3)$ size [KW12]. Can we use congestion-preserving DAG projection help bridging this gap, even with approximation?

A.1.4 Related Work

There has also been a line of work on preserving distances in *undirected* graphs by tree-like graphs with multiple copies of the original vertices. This includes the *multi-embeddings* of Bartal and Mendel [BM03], the *clan embeddings* of Filtser and Le [FL21], and *tree embeddings with copies* [HHZ22]. However, these works incur at least logarithmic distortion and are tailored to undirected graphs.

In contrast, our distance DAG projections apply to directed graphs and achieve $(1 + \epsilon)$ -approximation with almost-linear size. Moreover, for our congestion DAG projection, we are not aware of prior copy-based constructions aimed at preserving congestion or flow.

A.1.5 Organization

In Section A.2, we will sketch the *existence* of our distance DAG projections and congestion DAG projections, without considering the time complexity. In Section A.3, we give a full definition of all the basic notations used in the paper. In Section A.4, we give a formal definition of all DAG projection-related concepts, and show their applications assuming efficient construction. In Section A.5, we will show an efficient algorithm for constructing distance DAG projections assuming SSSP oracles on DAGs. In Section A.6, we will show an efficient algorithm for constructing congestion DAG projections assuming max flow oracles on DAGs.

A.2 Overview

In this section, we sketch the existence of our distance and congestion DAG projections, ignoring running time and focusing only on the idea and existence.

A.2.1 Distance DAG Projections

Our construction is inspired by the $(n^{o(1)}, \epsilon)$ -hop-sets of [Coh00] in *undirected* graphs.

Given a directed graph $G = (V, E)$ with edge lengths $\ell_G : E \rightarrow \mathbb{N}_{\geq 1}$, the goal is to construct a $(1 + \varepsilon)$ -distance-preserving DAG projection D onto G . For convenience, when we say that D *preserves the* (s, t) -distance for $s, t \in V(G)$, we mean: there exist $s', t' \in V(D)$ with $\pi(s') = s$ and $\pi(t') = t$ such that

$$\text{dist}_G(s, t) \leq \text{dist}_D(\pi^{-1}(s), \pi^{-1}(t)) \leq (1 + \varepsilon) \cdot \text{dist}_G(s, t).$$

We will use the following directed low-diameter decomposition (LDD) as a subroutine.

Lemma A.6 ([BCF23; BNW22]). *Let $G = (V, E)$ be a directed graph with edge lengths and let d be a positive integer. There exists a random set of edges E^{rem} (a low-diameter decomposition) satisfying:*

- *each SCC of $G - E^{\text{rem}}$ has diameter at most d ,*
- *for every $e \in E$, $\Pr[e \in E^{\text{rem}}] \leq s_{\text{LDD}} \cdot \ell_G(e)/d$,*

where $s_{\text{LDD}} = \tilde{O}(1)$.

Now, the construction is as follows.

Step 1 (LDD). Compute LDDs of G with diameters 2^i for all $0 \leq i \leq O(\log n)$. Fix one such LDD for a given i with parameter $d = 2^i$, yielding an edge set E^{rem} as in Lemma A.6. We show how to build a DAG projection that preserves (s, t) -distances in G whenever $\text{dist}_G(s, t)$ is on the order of d/ε . The final projection D will be the union of the $O(\log n)$ projections over all i , thereby handling all pairs (we assume edge lengths are polynomial on n).

Let $\mathcal{C}$ be the family of SCCs of $G - E^{\text{rem}}$. By construction, each $C \in \mathcal{C}$ has $\text{diam}(G[C]) \leq d$. Let $\sigma = n^{o(1)}$ (to be fixed later), and classify clusters by size:

- *large* if $|C| \geq n/\sigma$,
- *small* if $|C| < n/\sigma$.

Step 2 (recursive construction for small clusters). For each small $C \in \mathcal{C}$, recursively construct a DAG projection D_C of $G[C]$ that $(1 + \varepsilon)$ -approximately preserves distances for *all pairs* of vertices in $G[C]$ (instead of just length bounded pairs).

Step 3 (shortest-path trees for large clusters). For each large $C \in \mathcal{C}$, pick an arbitrary root $r_C \in C$. In G , compute a shortest-path tree T_C^{out} rooted at r_C and a reversed shortest-path tree T_C^{in} rooted at r_C (i.e., every (s, r_C) -path in T_C^{in} is a shortest (s, r_C) -path). Define D_C to be the DAG projection that combines T_C^{out} and T_C^{in} (as *disjoint copies* of subgraphs of G) by merging the common root r_C .

Step 4 (combining everything). Let $\mathcal{C} = (C_1, \dots, C_z)$ be a topological ordering of the SCCs in $G - E^{\text{rem}}$. Form D' by concatenating the D_{C_i} using copies of edges in G : for every $1 \leq i < j \leq z$, every $u' \in V(D_{C_i})$, and every $v' \in V(D_{C_j})$, if $(\pi(u'), \pi(v')) \in E(G)$ then add the edge (u', v') to D' .

Finally, let $x = s_{\text{LDD}}/\varepsilon$. Create x copies $D'_1, \dots, D'_x$ of D' and concatenate them in the same manner: for every $1 \leq i < j \leq x$, $u' \in V(D'_i)$, and $v' \in V(D'_j)$, if $(\pi(u'), \pi(v')) \in E(G)$ then add (u', v') . The resulting DAG is D .

Distance preservation. It is easy to see that

$$\text{dist}_G(s, t) \leq \text{dist}_D(\pi^{-1}(s), \pi^{-1}(t))$$

since π is a (weight-preserving) graph homomorphism (the only edges we added to D are in Step 3 and 4, which are copies of edges in the original graph): Every path in D connecting a vertex in $\pi^{-1}(s)$ to a vertex in $\pi^{-1}(t)$ can be mapped to a path in G connecting s to t with the same length. Hence the distance in G is at most the distance in D .

In what remains, we will only prove the harder direction

$$\text{dist}_D(\pi^{-1}(s), \pi^{-1}(t)) \leq (1 + \varepsilon) \cdot \text{dist}_G(s, t).$$

Fix $s, t \in V(G)$ with $\text{dist}_G(s, t) = d/\varepsilon$. Let p be an $s \rightarrow t$ path in G of length $\ell_G(p) = d/\varepsilon$. By [Lemma A.6](#), in expectation there are $s_{\text{LDD}}/\varepsilon$ edges of p lie in E^{rem} . Hence we can partition p into subpaths $p_1, p_2, \dots$, each entirely contained in $G - E^{\text{rem}}$. Notice that edges connecting p_i to p_{i+1} is preserved in D according to Step 4.

Let $a, b \in V$ be the endpoints of some subpath p_i . Let p' be a shortest $a \rightarrow b$ path in $G - E^{\text{rem}}$. Decompose p' into maximal subpaths $p'_1, \dots, p'_{y'}$ so that each p'_j lies within a single SCC of $G - E^{\text{rem}}$. Notice that edges connecting p'_i to p'_{i+1} is preserved in D' according to Step 4. If that SCC is a small cluster, then the endpoints' distance is preserved by the recursive construction. Otherwise, it is a large cluster.

Let C_L be the first large cluster meeting p (not p' , we switch our attention to the whole path p) and, similarly, C_R be the last. Let $\tilde{p}$ be the subpath of p from its first vertex in C_L to its last vertex in C_R . Denote these endpoints by $\text{Start}(\tilde{p})$ and $\text{End}(\tilde{p})$. By the definition of D_C for a large cluster C , D_C contains T_C^{in} followed by T_C^{out} , both rooted at r_C . Thus a copy of $\text{Start}(\tilde{p})$ in $T_{C_L}^{\text{in}}$ reaches r_{C_L} within distance at most d (since $G[C_L]$ has diameter d and $T_{C_L}^{\text{in}}$ is a shortest path tree), and from r_{C_L} we can reach a copy of $\text{End}(\tilde{p})$ within distance $\ell_G(\tilde{p})$ plus at most d additional length via $T_{C_R}^{\text{out}}$ (this is because the original distance from C_L to $\text{End}(\tilde{p})$ is at most $d + \ell_G(\tilde{p})$: from C_L we can reach $\text{Start}(\tilde{p})$ within distance d , then we follow $\tilde{p}$ to reach $\text{End}(\tilde{p})$). Hence, the total distance is at most

$$\ell_G(\tilde{p}) + 2d,$$

which has an additive $O(d)$ overhead. Since $\text{dist}_G(s, t) = d/\varepsilon$, this additive term is negligible and absorbed in the $(1 + \Theta(\varepsilon))$ multiplicative guarantee.

Size of the projection. If we write $f(n)$ as the width (maximum number of copies) of the DAG projection returned by the above construction on an n -vertex graph, then $f(n)$ can be calculated as follows.

- (1) In Step 1 and Step 4, we know that D contains $x = \tilde{O}(1/\varepsilon)$ copies of D' .
- (2) In Step 2, each vertex is copied $f(n/\sigma)$ times in D' .
- (3) In Step 3, each vertex is copied $2 \cdot \sigma^2$ times in D' , where the factor 2 comes from the forward and reversed shortest path trees, and the factor σ^2 comes from the fact that there are at most σ large clusters (because each large cluster has at least n/σ vertices).

To summarize, we get the following recursive inequality:

$$f(n) \leq \tilde{O}(1/\varepsilon) \cdot (f(n/\sigma) + 2\sigma^2).$$

Notice that $\varepsilon = 1/\text{polylog}(n)$ according to our assumption. By choosing an appropriate parameter $\sigma = n^{o(1)}$, the desired bound $f(n) = n^{o(1)}$ follows.

Algorithmic Aspect. The algorithm above runs in $m^{1+o(1)}$ time; it takes near-linear time in the size of the output, which is $m^{1+o(1)}$, because we simply call LDD which can be computed using SSSP as a subroutine.

In the parallel setting, our goal is to reduce DAG projection construction to approximate SSSP on DAGs. But, currently, the algorithm needs to call as a subroutine the SSSP algorithm on general graphs, which does not admit efficient parallel algorithms yet. Our strategy is to reduce SSSP on general graphs to SSSP on DAGs using our distance DAG projection itself. This leads to the ‘chicken-and-egg’ situation: (When we are given an SSSP on DAGs oracle), we can reduce distance DAG projection to SSSP on general graphs, and we can reduce SSSP on general graphs to distance DAG projection.

Our strategy (detailed in [Section A.5](#)) is to find a spiral recursion: We define h -length DAG projection that only preserves distances up to h , and we reduce it to $h/n^{o(1)}$ -length SSSP on general graphs which computes distance up to $h/n^{o(1)}$, reducing to $h/n^{o(1)}$ -length DAG projection. This spiral recursion will finally make h small enough so the construction is trivial.

A.2.2 Congestion DAG Projections

Given a graph $G = (V, E)$ with edge capacities denoted by $U_G : E \rightarrow \mathbb{R}_{\geq 0}$, we will show how to construct a $n^{o(1)}$ -congestion-preserving DAG projection.

We consider *expander decomposition* as the analogue of LDD in the congestion setting. We first give some necessary definitions.

For a flow f , we represent the demand routed by f as a pair of vectors (Δ, ∇) , where $\Delta, \nabla : V \rightarrow \mathbb{R}_{\geq 0}$ denote the source demands and sink demands routed by f , respectively; see [Section A.3.2](#) for the formal definition.

We say a demand (Δ, ∇) is **d-respecting** for some function $\mathbf{d} : V \rightarrow \mathbb{R}_{\geq 0}$ if $\Delta(v), \nabla(v) \leq \mathbf{d}(v)$ for every $v \in V$. For an edge set $E' \subseteq E$, define

$$\text{vol}_{E'}(v) = \sum \{ U_G(u, v) \mid (u, v) \in E' \text{ or } (v, u) \in E' \},$$

and for a vertex set $C \subseteq V$ write $\text{vol}_{E'}|_C$ for the restriction of function $\text{vol}_{E'}$ to C .

Definition A.7 (Terminal Expanding). Given a directed graph $G = (V, E)$ with edge capacities and a function $\mathbf{d} : V \rightarrow \mathbb{R}_{\geq 0}$, we say $\mathbf{d}$ is *ϕ -expanding* on G if every $\mathbf{d}$ -respecting demand is routable via a flow f in G with congestion at most $1/\phi$.

We next formalize the hierarchical structure we will use.

Definition A.8 (Expander Hierarchy). Given a directed graph $G = (V, E)$ with edge capacities, an *ϕ -expander hierarchy* with t layers consists of edge sets $E_1, \dots, E_t$ with $E_1 = E$ and $E_t = \emptyset$. Let $E_{>i} = \bigcup_{t \geq j > i} E_j$. For every $1 \leq i < t$ and every strongly connected component (SCC) C of $G - E_{>i}$, the function $\text{vol}_{E_i}|_C$ is ϕ -expanding in $G[C]$.

By [\[Ber+24\]](#), such a hierarchy exists with expansion $\phi = 2^{-O(\sqrt{\log n})}$ and $t = \sqrt{\log n}$. In our actual efficient construction, we will instead use a weaker hierarchy ([Definition A.41](#)) that admit fast construction.

Constructing the congestion DAG projection D . Let $E_1, \dots, E_t$ be the expander hierarchy of G with expansion $\phi = 2^{-O(\sqrt{\log n})}$ and $t = \sqrt{\log n}$. The construction proceeds recursively from top to bottom using the hierarchy. We will only describe the top level.

Fix an SCC C of G . By definition, $\text{vol}_{E_{t-1}}|_C$ is ϕ -expanding in $G[C]$. Let D_C be a congestion DAG embedding of $G[C] - E_{t-1}$ obtained *recursively*. We build a congestion DAG embedding D'_C of $G[C]$ as follows:

- Create two disjoint copies $D_C^{(1)}$ and $D_C^{(2)}$ of D_C , and add a dummy vertex w_C .
- For every $v \in V(D_C^{(1)})$, add an edge (v, w_C) with capacity $\text{vol}_{E_{t-1}}(\pi(v))$.
- For every $v \in V(D_C^{(2)})$, add an edge (w_C, v) with capacity $\text{vol}_{E_{t-1}}(\pi(v))$.

Here π denotes the projection map from the vertices of D_C to G .

Let $C_1, \dots, C_z$ be the SCCs of G in a topological order. We obtain D by concatenating the graphs $D'_{C_1}, \dots, D'_{C_z}$: for every $1 \leq i < j \leq z$ and for every $u \in V(D'_{C_i})$, $v \in V(D'_{C_j})$, add an edge (u, v) in D whenever $(\pi(u), \pi(v)) \in E(G)$.

Size of D . There are $t = O(\sqrt{\log n})$ layers, and in each layer we make two copies of every vertex in the next layer. Hence, every vertex of G (and every dummy vertex w_C) is copied at most $2^t = n^{o(1)}$ times overall. For edges, we only add edges (u, v) in D if $(\pi(u), \pi(v)) \in E$ or one of u, v is a dummy vertex. Since the number of vertex copies is $n^{o(1)}$, it follows that $|E(D)| \leq m^{1+o(1)}$.

Correctness. We first show that $\text{maxflow}_G(S, T) \leq \text{maxflow}_{G'}(\pi^{-1}(S), \pi^{-1}(T))$. Given a flow f in G from S to T , we show how to route f in D from $\pi^{-1}(S)$ to $\pi^{-1}(T)$ with the same value. Because we concatenate $D'_{C_1}, \dots, D'_{C_z}$ using all edges of G respecting the topological order, it suffices to show that the

restriction f_C of f to a fixed SCC C can be routed in D'_C . Consider a flow path p of f_C ; we decompose it into three segments:

1. The prefix of p before it uses any edge of E_{t-1} . This segment is routed inside $D_C^{(1)}$ by the recursive embedding.
2. The subsequence of p from its first edge $(u_1, v_1) \in E_{t-1}$ to its last edge $(u_2, v_2) \in E_{t-1}$. We replace this by the two-hop path (u'_1, w_C, v'_2) where $\pi(u'_1) = u_1$ and $\pi(v'_2) = v_2$. Feasibility holds because the capacity of (u'_1, w_C) is $\text{vol}_{E_{t-1}}(u_1)$, and analogously for (w_C, v'_2) so no congestion larger than 1 is introduced when considering all flow paths of f (a feasible flow in G) passing through E_{t-1} .
3. The suffix of p after its last edge in E_{t-1} , which is routed inside $D_C^{(2)}$ by recursion.

Then we show that $\text{maxflow}_{G'}(\pi^{-1}(S), \pi^{-1}(T)) \leq n^{o(1)} \text{maxflow}_G(S, T)$. Given a flow f^D in D , we map it back to G with congestion at most $n^{o(1)}$. For all portions of f^D that do not traverse dummy vertices, we use the projection π ; since each original vertex has at most $n^{o(1)}$ copies, this incurs at most an $n^{o(1)}$ congestion blow-up.

Next, consider every subpath of flow paths of f^D that are incident to the dummy vertex w_C . This induces a demand that is $n^{o(1)} \cdot \text{vol}_{E_{t-1}}$ -respecting in $G[C]$ (the $n^{o(1)}$ factor comes from the number of vertex copies), hence is routable in $G[C]$ with congestion at most $1/\phi = 2^{O(\sqrt{\log n})} = n^{o(1)}$. Although each w_C may itself be replicated $n^{o(1)}$ times across lower layers, the resulting multiplicative factors still yield total congestion $n^{o(1)}$ overall.

Algorithmic Aspect. Excluding the time for computing the expander hierarchy in [Definition A.8](#), the construction of congestion DAG projection take near-linear time in the output-size, which is $m^{1+o(1)}$ time. To compute the expander hierarchy efficiently, we employ a weaker version as in stated [Definition A.41](#) which admit simpler construction by performing expander decomposition in a bottom-up manner.

In the sequential setting, by using almost-linear time maximum flow algorithm to compute expander decomposition, this gives $m^{1+o(1)}$ total time.

Unfortunately, to obtain efficient parallel reduction, this suffers from the similar bottleneck as in the distance DAG projection algorithm: Expander decomposition usually requires maximum flow on a general graph as a subroutine (which does not admit fast parallel algorithms), and we intend to reduce max flow on general graphs to max flow on DAGs.

The idea is again to design a spiral recursion (detailed in [Section A.6](#)): (Given a max flow on DAGs oracle) We define δ additive-error congestion DAG projection to tolerate some δ additive error, and reduce it to $\delta \cdot n^{o(1)}$ additive-error maximum flow on general graphs, which reduces to $\delta \cdot n^{o(1)}$ additive-error congestion DAG projection. The additive error will be finally large enough so the problem is trivial.

A.3 Preliminaries

We use the following notation. For $a \in \mathbb{N}$, let $[a] = \{1, 2, \dots, a\}$. We write $\tilde{O}(f) = f \cdot \text{polylog} n$ and $\hat{O}(f) = f \cdot n^{o(1)}$. Unless stated otherwise, we use n, m to denote the numbers of nodes and edges of the input graph, respectively. A *randomized algorithm* succeeds with high probability (w.h.p.), meaning that it fails to output the correct answer with probability at most $1/n^c$ for some constant c .

We use $[E]$ to denote the indicator variable of the event E , i.e., $[E] = 1$ if E happens and $[E] = 0$ otherwise. We overload $[\cdot]$ for both ranges and indicators; context will be clear. For functions with the same domain (e.g. $d_1, d_2 : V \rightarrow \mathbb{R}$), we write $d_1 \leq d_2$ to denote $d_1(v) \leq d_2(v)$ for every $v \in V$. Other relations $d_1 = d_2$, $d_1 < d_2$, etc. are defined similarly. For a function $d : V \rightarrow \mathbb{R}$ and $a \in \mathbb{R}$, we write $d + a$ to denote the function $(d + a)(v) = d(v) + a$ for every $v \in V$. For $S \subseteq V$, we write $d(S) = \sum_{v \in S} d(v)$. We write $d|_S$ to denote the function d restricted to S .

An algorithm $\mathcal{A}$ with input size t in this paper may make oracle calls to another algorithm $\mathcal{A}'$. We say the oracle calls are *f-work-efficient* if the sum of all input sizes to $\mathcal{A}'$ over all oracle calls is at most $f \cdot t$, and *f-depth-efficient* if the longest dependency chain among all calls has length at most f .

Graph. All graphs in this paper, unless specified otherwise, are directed graphs $G = (V, E)$ possibly with edge lengths and capacities. We assume edge lengths and capacities are positive integers bounded by W , where W is a polynomial in n , for simplicity. We treat edge lengths and capacities as attributes of edges $e \in E$, so we use $\ell(e)$ and $U(e)$ to denote the length and capacity of e , respectively, instead of writing $G = (V, E, \ell, U)$. We write $\ell_G(e)$ and $U_G(e)$ if we want to stress that the length and capacity are associated with G . For an edge set $E' \subseteq E$ and vertex sets $S, T \subseteq V$, we write

$$E'(S, T) = \{(s, t) \mid s \in S, t \in T, (s, t) \in E'\}.$$

We use standard definitions of graph terminology for directed graphs, including path, distance, strongly connected components (SCCs), and shortest-path tree, which can be found in textbooks. For a path H , we use $\ell(H)$ and $U(H)$ to denote the sum of the lengths or capacities, respectively, of the edges on the path, and we use $|H|$ to denote the number of edges (edges can repeat, in which case they are counted multiple times). We use

$$\delta^+(S) = \{(u, v) \mid u \in S, v \in V \setminus S\} \quad \text{and} \quad \delta^-(S) = \{(u, v) \mid v \in S, u \in V \setminus S\}$$

to denote the outgoing and incoming edge sets of S , respectively. We use $\text{Start}(p)$ and $\text{End}(p)$ to denote the starting vertex and the ending vertex of a path p .

Projection map. A graph $G = (V, E)$ can be assigned a projection map $\pi : V \rightarrow U$ or $\pi : V \rightarrow U \cup \{\perp\}$ (where U can be the vertex set of another graph). We write π_G to stress that π_G is the labeling function for G . We write $\pi^{-1} : U \rightarrow 2^V$ for the inverse mapping, where $\pi^{-1}(u) = \{v \in V \mid \pi(v) = u\}$. For a set $U' \subseteq U$, we write $\pi^{-1}(U') = \bigcup_{u \in U'} \pi^{-1}(u)$.

For a set $S \subseteq V$, we write

$$\pi(S) = \{\pi(s) \mid s \in S\} \setminus \{\perp\},$$

and we write

$$\hat{\pi}(S) = \{u \in U \mid \pi^{-1}(u) \subseteq S\}.$$

In words, $\pi(S)$ contains vertices u of G for which some copy of u is in S , and $\hat{\pi}(S)$ contains vertices u of G for which all copies of u are in S .

Remark on DAG as an input. A DAG refers to a *directed acyclic graph*. In this paper, when we say a DAG is an input, we assume its vertex set is given in a topological order.

A.3.1 Distances and Shortest Paths

Graph distance. We use $\text{dist}_G(s, t)$ to denote the distance from s to t in G (with respect to the length function ℓ_G). When G is clear from the context, we simply write $\text{dist}(s, t)$.

Given a directed graph $G = (V, E)$ and a vertex set $A \subseteq V$, the *weak diameter* of A (in G) is defined as

$$\max_{s, t \in A} \text{dist}_G(s, t).$$

In contrast, the *strong diameter* of A is defined as

$$\max_{s, t \in A} \text{dist}_{G[A]}(s, t),$$

where $G[A]$ is the subgraph of G induced by A . The strong diameter of A is also the diameter of $G[A]$.

Single-source shortest path (SSSP). The single-source shortest path problem is: given a graph with edge lengths and a source s , find a shortest-path tree rooted at s , i.e., every (s, t) -path in the tree has length $\text{dist}_G(s, t)$ for every $t \in V$.

The problem of α -approximate SSSP (denoted by α -ApxSSSP) only requires outputting an approximate shortest-path tree such that the length of any (s, t) -path (denoted by $\tilde{d}(s, t)$) satisfies

$$\text{dist}_G(s, t) \leq \tilde{d}(s, t) \leq \alpha \cdot \text{dist}_G(s, t).$$

The h -length α -ApxSSSP problem only needs to satisfy

1. $\text{dist}_G(s, t) \leq \tilde{d}(s, t)$ for every $t \in V$, and
2. $\tilde{d}(s, t) \leq \alpha \cdot \text{dist}_G(s, t)$ only for those $t \in V$ with $\text{dist}_G(s, t) \leq h$.

We use h -length SSSP to denote the problem of h -length 1-ApxSSSP.

We use α -ApxSSSP-DAG to denote the problem of α -approximate SSSP when the input graphs are restricted to DAGs. We assume that a topological order of the input DAG is given as part of the input.

A.3.2 Flows and Cuts

Cuts. Given a directed graph $G = (V, E)$, a cut is a partition of the vertex set into two sets $(S, \bar{S})$ where $\bar{S} := V \setminus S$ (when V is clear from the context). For convenience, we also use S to denote the cut $(S, \bar{S})$ when there is no ambiguity. The value of a cut $(S, \bar{S})$ is defined as

$$\text{val}_G(S) := U_G(\delta^+(S)) := \sum_{e \in \delta^+(S)} U_G(e),$$

i.e., the total capacity of edges going from S to $\bar{S}$. We allow a cut to be $\emptyset$ or V , in which case the value is defined to be 0.

Flows. Given a directed graph $G = (V, E)$, a (single-commodity) flow f is represented as a collection of paths in G associated with positive flow values. We call each path $p \in f$ a *flow path*, and we denote its value by $f(p) \in \mathbb{R}_{>0}$. If $p \notin f$, we write $f(p) = 0$. For an edge $e \in E$, we define

$$f(e) = \sum_{p: e \in p} f(p).$$

(That is, the flow on an edge is the sum of the flows of all paths that use that edge.) By default, we assume a flow is represented by flows on each edge, so every flow can be represented in $O(m)$ space.

The *congestion* of a flow f in G is

$$\text{Cong}(f) = \max_{e \in E} \frac{f(e)}{U_G(e)}.$$

If $\text{Cong}(f) \leq 1$, we say f is *feasible*. Let

$$f^{\text{last}}(u) = \sum_{(u, v) \in E} f(u, v) \quad \text{and} \quad f^{\text{first}}(u) = \sum_{(v, u) \in E} f(v, u)$$

be the outgoing and incoming flow at a vertex $u \in V$, and let

$$f(u) = f^{\text{last}}(u) - f^{\text{first}}(u)$$

be the net outgoing flow at u .

The following lemma shows it is possible to convert from flow path decomposition and edge representation. It is implied by Section 8 of [Aga+24]. In their paper, they only proved the flow decomposition. The reverse direction that recovers a flow from a set of sink-source demands can be done by a reverse traversal of their algorithm.

Theorem A.9 (Section 8 of [Aga+24]). *Let G be a directed graph with capacities, and let F be a circulation-*

free feasible flow on G . Then there is a parallel algorithm that computes a representation

$$\{(\lambda_i, s_i, t_i)\}_{i \in [k]}$$

such that F can be decomposed into k directed flow paths, where path i has value λ_i and goes from s_i to t_i .

Moreover, given any values $(\lambda'_i)_{i \in [k]}$ satisfying $\lambda'_i \leq \lambda_i$ for every $i \in [k]$, there is a parallel algorithm that computes an edge representation of a feasible flow routing λ'_i units from s_i to t_i for every $i \in [k]$.

Both algorithms run in near-linear work and polylogarithmic depth.

Demands. A (single-commodity) demand is a pair (Δ, ∇) where $\Delta, \nabla : V \rightarrow \mathbb{R}_{\geq 0}$. Given a flow f , the demand routed by the flow, denoted by $\text{Dem}(f) = (\Delta_f, \nabla_f)$, is defined as

$$\Delta_f(u) = \max(f(u), 0) \quad \text{and} \quad \nabla_f(u) = -\min(f(u), 0).$$

The value of a demand (Δ, ∇) is defined as

$$\text{val}(\Delta, \nabla) = \sum_{v \in V} \Delta(v).$$

The value of a flow is $\text{val}(f) = \text{val}(\text{Dem}(f))$. We say f is an (s, t) -flow with value x if f routes the demand

$$\Delta(v) = x \cdot [v = s], \quad \nabla(v) = x \cdot [v = t].$$

Similarly, we say f is an (S, T) -flow with value x for $S, T \subseteq V$ if f routes a demand with source and sink demands being subsets of S, T .

A sub-demand of (Δ, ∇) , denoted by (Δ', ∇') , satisfies $\Delta'(v) \leq \Delta(v)$ and $\nabla'(v) \leq \nabla(v)$ for every $v \in V$, in which case we write $(\Delta', \nabla') \preceq (\Delta, \nabla)$. A flow *partially* routes the demand (Δ, ∇) if it routes a sub-demand of it. We define the support of a demand to be

$$\text{Supp}(\Delta, \nabla) = \{v \in V \mid \Delta(v) + \nabla(v) > 0\}.$$

Let $\pi : V \rightarrow U$ be a projection map. For a demand $D = (\Delta, \nabla)$ on V , we define $\pi(D) = (\Delta', \nabla')$ on U by

$$\Delta'(u) = \sum_{v \in V : \pi(v) = u} \Delta(v) \quad \text{and} \quad \nabla'(u) = \sum_{v \in V : \pi(v) = u} \nabla(v)$$

for every $u \in U$.

Volume. Consider $F \subseteq E$. Let

$$\delta_F^+(v) = \{(v, u) \in F\}, \quad \delta_F^-(v) = \{(u, v) \in F\}, \quad \delta_F(v) = \delta_F^+(v) \cup \delta_F^-(v)$$

be the sets of edges in F incident to v . Define

$$\text{vol}_F^+(v) = \sum_{e \in \delta_F^+(v)} U_G(e), \quad \text{vol}_F^-(v) = \sum_{e \in \delta_F^-(v)} U_G(e), \quad \text{vol}_F(v) = \text{vol}_F^+(v) + \text{vol}_F^-(v).$$

A demand (Δ, ∇) is vol_F -respecting if

$$\Delta(v), \nabla(v) \leq \text{vol}_F(v) \quad \text{for every } v \in V.$$

Max flow. The classical max-flow problem is: given a directed graph with edge capacities $G = (V, E)$ and two vertices s, t , output an (s, t) -flow f with maximum value.

In many cases, however, we need to route a demand instead of fixing the source and sink. Thus, we define the (generalized) max-flow problem as: given a demand (Δ, ∇) , find a feasible flow with maximum value that routes a sub-demand of (Δ, ∇) . Notice that this problem is equivalent to the classical setting by

adding a super source connected to every node v with capacity $\Delta(v)$ and a super sink connected from every node v with capacity $\nabla(v)$.

We also define the α -approximate max-flow problem as finding a feasible flow with value at least a $1/\alpha$ -fraction of the maximum value that routes a sub-demand of (Δ, ∇) .

It is well known that *cuts* are dual to *flows*. Thus, if we can return both a flow and a cut, they will certify the approximation ratio. To make this formal, we define the α -ApxMFMC problem as finding a feasible flow f that routes a sub-demand of (Δ, ∇) and a cut S satisfying

$$\text{val}(S) + \sum_{v \in S} \nabla(v) + \sum_{v \notin S} \Delta(v) \leq \alpha \cdot \text{val}(f). \quad (\text{A.1})$$

In this case, (f, S) is called an α -ApxMFMC pair. We define α -ApxMFMC-DAG as the problem α -ApxMFMC with inputs restricted to DAGs.

Equation (A.1) certifies that f is an α -approximate max flow. To see this, consider adding a super source connected to every node with capacity $\Delta(v)$ and a super sink connected from every node with capacity $\nabla(v)$, and apply the classical max-flow/min-cut theorem on the super source and sink.

For technical reasons, we will also sometimes use additive error. We define the problem (α, δ) -ApxMFMC as finding a feasible flow f that routes a sub-demand of a given (Δ, ∇) and a cut S satisfying

$$\text{val}(S) + \nabla(S) + \Delta(\bar{S}) \leq \alpha \cdot \text{val}(f) + \delta. \quad (\text{A.2})$$

A.4 DAG Projections and Their Applications

In this section, we introduce the notion of distance-preserving DAG projection in [Section A.4.1](#), state our main result ([Theorem A.13](#)), and show its applications in [Section A.4.3](#). Then, we introduce the notion of congestion-preserving DAG projection in [Section A.4.4](#), state our main result ([Theorem A.26](#)), and show its applications in [Section A.4.5](#).

A.4.1 Distance DAG projections

Definition A.10 (Graph Projections). Let $G = (V, E)$ be a graph with edge lengths. A *DAG projection onto G* is a DAG $D = (V', E')$ together with a projection map $\pi : V' \rightarrow V$ that is a *weight-preserving* graph homomorphism. That is, for every $(x, y) \in E'$, we have $(\pi(x), \pi(y)) \in E$ and

$$\ell_D(x, y) = \ell_G(\pi(x), \pi(y)).$$

The *size* of the projection is $|E'|$, and the *width* of the projection is $\max_{v \in V} |\pi^{-1}(v)|$.

Naturally, we would like D to approximately preserve distances in G .

Definition A.11. A DAG projection D onto G is λ -distance-preserving if the following holds: for every $s, t \in V$,

$$\text{dist}_G(s, t) \leq \text{dist}_D(\pi^{-1}(s), \pi^{-1}(t)) \leq \lambda \cdot \text{dist}_G(s, t).$$

The following lemma follows immediately from [Definition A.10](#): every (s', t') -path in D projects to a path in G of the same length.

Lemma A.12. *If D is a DAG projection onto G , then for every $s, t \in V$ we have*

$$\text{dist}_G(s, t) \leq \text{dist}_D(\pi^{-1}(s), \pi^{-1}(t)).$$

We say a projection D onto G is a *distance projection* if it is λ -distance-preserving for some λ .

In [Section A.5](#), we will prove the following key result, which shows that a distance DAG projection can be constructed efficiently in the parallel setting, given only an approximate SSSP algorithm on DAGs. We state here a simplified version for $\varepsilon \geq 1/\text{polylog} n$. For general ε , see [Theorem A.30](#).

Theorem A.13 (Simplified version of [Theorem A.30](#)). *There is a randomized algorithm that, given a directed graph G and a parameter $o(1) > \varepsilon \geq 1/\text{polylog}n$, constructs a $(1 + \varepsilon)$ -distance-preserving DAG projection of G with width $n^{o(1)}$ (and its topological order). The algorithm makes $n^{o(1)}$ work-efficient, $\tilde{O}(1)$ depth-efficient calls to an $(1 + \varepsilon/\log n)$ -ApxSSSP-DAG oracle.*

By plugging in the best parallel algorithm for approximate SSSP [\[CFR20\]](#) into [Theorem A.13](#), we obtain the following bound.

Corollary A.14. *There is a randomized algorithm that, given a directed graph G and a parameter $o(1) > \varepsilon \geq 1/\text{polylog}n$, constructs a $(1 + \varepsilon)$ -distance-preserving DAG projection of G (and its topological order) with width $n^{o(1)}$ in $\tilde{O}(m)$ work and $\tilde{O}(\sqrt{n})$ depth.*

A.4.2 Useful Tools for SSSP

Boosting Approximate SSSP . It is possible to reduce exact SSSP to their approximate variants.

For SSSP, we have the following result from [\[Roz+23\]](#).

Lemma A.15 ([\[Roz+23\]](#)). *Given a directed graph G with edge lengths and a source $s \in V(G)$, there is an algorithm that solves SSSP on G with source s . The algorithm makes $\tilde{O}(1)$ -work-efficient and $\tilde{O}(1)$ -depth-efficient an oracle that solves $O(\max_{t \in V(G)} \text{dist}_G(s, t))$ -length $(1 + o(1/\log n))$ -ApxSSSP.*

[Lemma A.15](#) is an implication of [Theorem 1.7](#) and [Lemma 3.1](#) in [\[Roz+23\]](#). One minor difference is that in [Theorem 1.7](#) they only specify the oracle as a general $(1 + o(1/\log n))$ -ApxSSSP on arbitrary graphs (not necessarily G), instead of an $O(\max_{t \in V(G)} \text{dist}_G(s, t))$ -length $(1 + o(1/\log n))$ -ApxSSSP. However, by examining their algorithm, the oracle is always called on graphs (not necessarily G) whose maximum source distance is $O(\max_{t \in V(G)} \text{dist}_G(s, t))$.

We can strengthen [Lemma A.15](#) by a simple trick so that both the algorithm and the oracle are h -length bounded.

Corollary A.16. *Given a directed graph G with edge lengths, a source $s \in V(G)$, and a length bound h , there is an algorithm solving h -length SSSP on G with source s . The algorithm makes $\tilde{O}(1)$ -work-efficient and $\tilde{O}(1)$ -depth-efficient an oracle solving $O(h)$ -length $(1 + o(1/\log n))$ -ApxSSSP.*

Proof of [Corollary A.16](#). We build a new graph G' by adding an edge of length $2h$ from s to every other node in $V(G)$. Then we run [Lemma A.15](#) on G' and s . According to [Lemma A.15](#), it requires an oracle solving $O(\max_{t \in V(G')} \text{dist}_{G'}(s, t))$ -length $(1 + o(1/\log n))$ -ApxSSSP, where $O(\max_{t \in V(G')} \text{dist}_{G'}(s, t)) \leq 2h$ by the definition of G' . Thus, we can get the exact distances from s in G' by $\tilde{O}(1)$ calls to an $O(h)$ -length $(1 + o(1/\log n))$ -ApxSSSP oracle on graphs with $O(m)$ edges.

To obtain h -length SSSP on G , notice that if $\text{dist}_G(s, t) \leq h$ then $\text{dist}_{G'}(s, t) = \text{dist}_G(s, t)$. If $\text{dist}_G(s, t) > h$, we do not need to return the exact distance in G . Hence, for every $t \in V(G)$, it suffices to return $\text{dist}_G(s, t) = \text{dist}_{G'}(s, t)$ if $\text{dist}_{G'}(s, t) \leq h$, and return $\text{dist}_G(s, t) = +\infty$ otherwise. $\square$

Low Diameter Decomposition A low-diameter decomposition (LDD) decomposes a directed graph into small-diameter clusters.

Definition A.17 (Low-Diameter Decomposition). Let $G = (V, E)$ be a directed graph with edge weights and let d be a positive integer. A low-diameter decomposition (LDD) with diameter d and slack s_{LDD} of G is a sequence of vertex sets $(V_1, V_2, \dots, V_z)$, which is a partition of V , such that

- each V_i has weak diameter at most d in G , and
 - for any edge $(u, v) \in E$, the probability that $u \in V_i$ and $v \in V_j$ with $i > j$ is at most $s_{\text{LDD}} \cdot w(u, v)/d$.
- We call such an edge (u, v) a *reversed edge*.

Ashvinkumar et al. [\[Ash+24a\]](#) show how to compute an LDD with $\tilde{O}(1)$ calls to an SSSP oracle.

Lemma A.18 ([Ash+24a]). *There is a randomized algorithm that, given a directed graph G and a positive integer d , computes an LDD with diameter d and slack $s_{\text{LDD}} = O(\log^2 n)$ using $\tilde{O}(1)$ calls to a $(s_{\text{LDD}} \cdot d)$ -length SSSP oracle on graphs with $O(n)$ vertices and $O(m)$ edges.*

Lemma A.18 follows from Lemma 4 in [Ash+24a], with two minor differences.

1. Lemma 4 in [Ash+24a] does not output the order of the sequence $(V_1, \dots, V_z)$ but only outputs the set of reversed edges. However, this is not a problem since their algorithm inherently specifies the order of the partition.
2. Lemma 4 in [Ash+24a] only specifies the oracle as a general SSSP, instead of a $(s_{\text{LDD}} \cdot d)$ -length SSSP. However, every oracle call in their algorithm only needs the vertices within distance at most $s_{\text{LDD}} \cdot d$ from the source, so it is effectively a $(s_{\text{LDD}} \cdot d)$ -length SSSP oracle.

A.4.3 Applications of Distance DAG Projections

Reducing Exact SSSP to Approximate SSSP on DAGs. The first immediate implication is that we can reduce exact SSSP on general graphs to SSSP on DAGs.

Lemma A.19. *There is a parallel randomized algorithm solving (exact) SSSP on directed graphs, with $n^{o(1)}$ -work-efficient $\tilde{O}(1)$ -depth-efficient oracle calls to a $(1 + o(1/\log n))$ -approximate SSSP oracle on DAGs.⁷*

The main idea is as follows. We use **Theorem A.13** to construct a DAG projection D of G , and then run the oracle to solve SSSP on D . To get the distance between (s, t) in G , we take the minimum over all pairs (s', t') in D with $\pi(s') = s$ and $\pi(t') = t$; this gives an approximate distance in G . Finally, we boost the approximate distance to the exact distance by using **Lemma A.15**. For the formal argument, see **Lemma A.34**.

Reducing Exact SSSP to Undirected Graphs. It is known that SSSP on DAGs can be reduced to (exact) SSSP on undirected graphs (see [HXX25, Lemma 4.6]). Thus, we obtain the following lemma.

Lemma A.20. *There is a parallel randomized algorithm solving (exact) SSSP on directed graphs, with $n^{o(1)}$ -work-efficient $\tilde{O}(1)$ -depth-efficient oracle calls to an exact SSSP oracle on undirected graphs.*

Proof. We apply **Lemma A.19** with the following algorithm that solves exact SSSP on a DAG D using a single oracle call to exact SSSP on an undirected graph D' . We set $V(D) = V(D')$. Suppose D has a topological order $(v_1, \dots, v_n)$.⁸ For every edge $(v_i, v_j) \in E(D)$ with $i < j$, we create an undirected edge $(v_i, v_j) \in E(D')$ with length

$$\ell_{D'}(v_i, v_j) = \ell_D(v_i, v_j) + (j - i) \cdot M,$$

where M is a sufficiently large constant chosen as described in the text. Notice that the (v_i, v_j) -distance in D' is at most $\text{dist}_D(v_i, v_j) + (j - i) \cdot M$ by following the shortest path in D , and any path in D' that does not follow the topological order must incur an extra $(j - i + 1) \cdot M > \text{dist}_D(v_i, v_j) + (j - i) \cdot M$. Thus, running exact SSSP on D' gives exact SSSP on D after subtracting the corresponding $(j - i) \cdot M$ term. $\square$

Reducing Hop-set Construction to DAGs. The definition of DAG projection also immediately gives a hop-set reduction from general graphs to DAGs.

Lemma A.21. *Suppose there is an oracle $\mathcal{O}_{\text{HS}}$ constructing (β, ε) -hopset of size $s(m, n)$ for $o(1) > \varepsilon > 1/\text{polylog}(n)$ on DAGs, then there is a randomized algorithm constructing $(\beta, 3\varepsilon)$ -hopset of size $s(m^{1+o(1)}, n^{1+o(1)})$ on general directed graphs, with $n^{o(1)}$ -work-efficient and $\tilde{O}(1)$ -depth-efficient calls to $(1 + \varepsilon/\log n)$ -approximate SSSP and $\mathcal{O}_{\text{HS}}$.*

⁷The reduction is strong enough that we can assume the DAG oracle is given a topological order as part of the inputs. This is a basic assumption across the paper: When we call an oracle on DAGs, a topological order of the DAG must be given.

⁸The reduction from directed SSSP to DAGs assumes that a topological order for the input DAG is given. This is not always a standard assumption in parallel settings, but when we generate the DAG projection, a corresponding topological order is also produced.

Proof. The algorithm takes a graph G and applies [Theorem A.13](#) with parameter ε , which gives a DAG projection D of G with width $n^{o(1)}$, using $n^{o(1)}$ work-efficient and $\tilde{O}(1)$ depth-efficient calls to $(1 + \varepsilon/\log n)$ -Ap $\times$ SSSP-DAG. Since D is a projection, it has $m^{1+o(1)}$ edges and $n^{1+o(1)}$ vertices.

We then apply the oracle $\mathcal{O}_{HS}$ on D to get a hop set H^D of size $s(m^{1+o(1)}, n^{1+o(1)})$ which is a (β, ε) -hop-set of D . Let

$$H = \{(\pi(u), \pi(v)) \mid (u, v) \in H^D\}.$$

We show that H is a $(\beta, 3\varepsilon)$ -hop-set for G .

First, for any $(u, v) \in H$, pick $(u', v') \in H^D$ with $\pi(u') = u$ and $\pi(v') = v$. By construction, the weight of (u, v) in H is the same as the weight of (u', v') in H^D , and since D is a distance projection, we have

$$\text{dist}_G(u, v) \leq \text{dist}_D(u', v') \leq \ell_{H^D}(u', v') = \ell_H(u, v),$$

so adding H does not create shorter-than-original edges in G .

Now fix $s, t \in V$. By the definition of a distance DAG projection, there exist $s', t' \in V(D)$ with $\pi(s') = s$, $\pi(t') = t$ such that

$$\text{dist}_D(s', t') \leq (1 + \varepsilon) \cdot \text{dist}_G(s, t).$$

Since H^D is a (β, ε) -hop-set for D , there is a path p^D in $D \cup H^D$ with at most β edges and

$$\ell_{D \cup H^D}(p^D) \leq (1 + \varepsilon) \cdot \text{dist}_D(s', t') \leq (1 + \varepsilon)(1 + \varepsilon) \cdot \text{dist}_G(s, t) \leq (1 + 3\varepsilon) \cdot \text{dist}_G(s, t)$$

for $\varepsilon \leq 1$. The projection $\pi(p^D)$ is a path in $G \cup H$ with the same number of edges and the same length, so

$$\text{dist}_{G \cup H}^{(\beta)}(s, t) \leq (1 + 3\varepsilon) \cdot \text{dist}_G(s, t).$$

□

Reducing Distance Preserver Construction to DAGs. We can reduce the construction of $(1 + \varepsilon)$ -approximate distance preservers to DAGs.

Lemma A.22. *Suppose there is an oracle $\mathcal{O}_{DP}$ constructing $(1 + \varepsilon)$ -approximate distance preserver of size $s(n, p)$ for $o(1) > \varepsilon > 1/\text{polylog}(n)$ on n -node DAGs with p demand pairs, then there is a randomized algorithm constructing $(1 + 3\varepsilon)$ -approximate distance preserver of size $s(n^{1+o(1)}, p)$ on n -node general graphs with p demand pairs, with $n^{o(1)}$ -work-efficient and $\tilde{O}(1)$ -depth-efficient calls to $(1 + \varepsilon/\log n)$ -approximate SSSP and $\mathcal{O}_{DP}$.*

Proof. The algorithm takes a graph G and applies [Theorem A.13](#) with parameter ε , which gives a DAG projection D of G with width $n^{o(1)}$, using $n^{o(1)}$ work-efficient and $\tilde{O}(1)$ depth-efficient calls to $(1 + \varepsilon/\log n)$ -Ap $\times$ SSSP-DAG. Again, D has $m^{1+o(1)}$ edges and $n^{1+o(1)}$ vertices.

We modify D as follows: for every $v \in V(G)$, add two vertices v^{first} and v^{last} to D , and for every $v' \in V(D)$ with $\pi(v') = v$ add edges (v^{first}, v') and (v', v^{last}) of length 0. Denote the resulting graph by D' .

We apply the oracle $\mathcal{O}_{DP}$ on D' with demand pairs

$$P^D := \{(u^{\text{first}}, v^{\text{last}}) \mid (u, v) \in P\}$$

to get H^D of size $s(n^{1+o(1)}, p)$. Let

$$H = \{(\pi(u), \pi(v)) \mid (u, v) \in H^D, u \in V(D), v \in V(D)\}.$$

We emphasize that H is well-defined because π is a graph homomorphism according [Definition A.11](#). (A partial projection, as in [Definition A.23](#), is not enough.)

Now, we show that H is a $(1 + 3\varepsilon)$ -approximate distance preserver for G . Fix $(s, t) \in P$. Then $(s^{\text{first}}, t^{\text{last}}) \in P^D$, so

$$\text{dist}_{H^D}(s^{\text{first}}, t^{\text{last}}) \leq (1 + \varepsilon) \cdot \text{dist}_{D'}(s^{\text{first}}, t^{\text{last}}).$$

By the way we added zero-length in/out vertices, there exist $s', t' \in V(D)$ with $\pi(s') = s$ and $\pi(t') = t$ such that

$$\text{dist}_{D'}(s^{\text{first}}, t^{\text{last}}) = \text{dist}_D(s', t').$$

Also, the path witnessing $\text{dist}_{H^D}(s^{\text{first}}, t^{\text{last}})$ can be chosen to stay inside $V(D)$ except for the endpoints, so its projection is a path in H . Therefore,

$$\text{dist}_H(s, t) \leq (1 + \varepsilon) \cdot \text{dist}_D(s', t') \leq (1 + 3\varepsilon) \cdot \text{dist}_G(s, t),$$

where the last inequality uses that D is a $(1 + \varepsilon)$ -distance-preserving DAG projection of G . $\square$

Previously, the best $(1 + \varepsilon)$ -approximate distance preserver has size $O(n + \sqrt{n} \cdot p)$ [CE06; HXX25]⁹ We thus immediately get [Corollary A.4](#) by applying [Lemma A.22](#).

A.4.4 Congestion DAG Projections

We can also consider the case where we want D to preserve flow. For technical reasons, we allow the DAG projection D to contain *dummy vertices* that map to a dummy value $\perp$, and we do not require the projection to be a homomorphism. This can be removed with some efforts, but this version makes the later algorithms cleaner. We call it a *partial projection*, and we will usually omit the word “partial”.

Definition A.23 (Partial Projections). Let $G = (V, E)$ be a graph with edge capacities. A DAG (partial) projection onto G is a DAG $D = (V', E')$ together with a (partial) projection map $\pi : V' \rightarrow V \cup \{\perp\}$.

Now we define congestion-preserving projections.

Definition A.24. A DAG projection D of $G = (V, E)$ is κ -congestion-preserving if for every $S, T \subseteq V(G)$, we have

$$\text{maxflow}_G(S, T) \leq \text{maxflow}_{G'}(\pi^{-1}(S), \pi^{-1}(T)) \leq \kappa \cdot \text{maxflow}_G(S, T).$$

Often we want not only the existential property above, but also an explicit and efficient way to project flows and cuts from D back to G . This is given by the following definition.

Definition A.25 (Projection Algorithm). Let D be a DAG projection of $G = (V, E)$. A κ -congestion-preserving projection algorithm associated with D is an algorithm that, given either

- a flow f^D in D with $\pi(\text{Supp}(\text{Dem}(f^D))) \subseteq V$, returns a flow f in G with congestion at most $\kappa \cdot \text{Cong}(f^D)$ such that $\pi(\text{Dem}(f^D)) = \text{Dem}(f)$; or
- a cut S^D in D , returns a cut S in G with value at most $\text{val}(S^D)$ such that $\hat{\pi}(S^D) \subseteq S \subseteq \pi(S^D)$.

The projection algorithm is *efficient* if it takes $\hat{O}(|E(D)|)$ work and $\hat{O}(1)$ depth.

When we say D is associated with a projection algorithm, we assume the projection is surjective, i.e. $\pi^{-1}(v) \neq \emptyset$ for every $v \in V(G)$.

In [Section A.6](#), we will prove the following theorem, showing that congestion DAG projections can be constructed efficiently in the parallel setting using an approximate max-flow oracle on DAGs.

Theorem A.26. Suppose there is an oracle for α -ApMFC-DAG where $\alpha = n^{o(1)}$. Then there is a randomized algorithm that, given a directed graph $G = (V, E)$ with edge capacities, outputs a DAG projection D of G together with an efficient $n^{o(1)}$ -congestion-preserving projection algorithm, such that $|E(D)| = \hat{O}(|E(G)|)$ (and D 's topological order is returned). The algorithm makes $n^{o(1)}$ work-efficient and $n^{o(1)}$ depth-efficient calls to α -ApMFC-DAG.

⁹Lemma 4.6 of [HXX25] shows that distance preserver on DAGs reduces to (exact) distance preservers on undirected graphs, which exhibits a construction in [CE06].

A.4.5 Applications of Congestion DAG Projections

Reducing Exact Max Flow to Approximate Max Flow on DAGs. We first show the immediate application of reducing exact max flow to DAGs.

Lemma A.27. *There is a parallel randomized algorithm solving (exact) max flow on directed graphs, with $n^{o(1)}$ -work-efficient $n^{o(1)}$ -depth-efficient oracle calls to a $n^{o(1)}$ -approximate max flow oracle¹⁰ on DAGs.*

To prove the lemma, we first show how to use a congestion DAG projection to reduce the max-flow/min-cut problem to DAGs.

Lemma A.28. *Let D be a DAG projection of $G = (V, E)$ with a κ -congestion-preserving efficient projection algorithm. Let $\mathcal{O}$ be an oracle solving α -ApxMFMC on DAGs. Then there is an algorithm solving $\alpha \cdot \kappa$ -ApxMFMC on G , with complexity proportional to one efficient call to $\mathcal{O}$ plus the cost of the projection algorithm.*

Proof. We build a graph D' from D as follows. For every $v \in V$, add a vertex s_v and edges (s_v, v') for every $v' \in V(D)$ with $\pi(v') = v$, each with infinite capacity. Also for every $v \in V$, add a vertex t_v and edges (v', t_v) for every $v' \in V(D)$ with $\pi(v') = v$, each with infinite capacity. Then add a super source s and edges (s, s_v) for every $v \in V$ with capacity $\Delta(v)$, and add a super sink t and edges (t_v, t) for every $v \in V$ with capacity $\nabla(v)$. Denote the resulting graph by D' .

Run the oracle $\mathcal{O}$ on D' with source s and sink t , and let the resulting flow and cut be (f', S') . Then

$$\text{val}(S') \leq \alpha \cdot \text{val}(f').$$

Restrict f' and S' to D to get f^D and S^D . By construction, all source/sink demand of f^D lives on vertices that project to V , i.e. $\pi(\text{Supp}(\text{Dem}(f^D))) \subseteq V$. Apply the projection algorithm to (f^D, S^D) to get a flow f in G and a cut S in G . By the guarantee of the projection algorithm:

$$\text{Cong}(f) \leq \kappa \cdot \text{Cong}(f^D) \leq \kappa,$$

so f/κ is feasible in G , and

$$\text{val}(S) \leq \text{val}(S^D), \quad \hat{\pi}(S^D) \subseteq S \subseteq \pi(S^D).$$

Next we relate the cut values. As in the usual super-source/super-sink reduction, any $v \in V$ for which there exists $v' \in V(D)$ with $v' \notin S^D$ must have $s_v \notin S'$ (otherwise an infinite-capacity edge is cut), so (s, s_v) contributes $\Delta(v)$ to $\text{val}(S')$. Symmetrically, any $v \in V$ for which there exists $v' \in V(D)$ with $v' \in S^D$ must have $t_v \in S'$ (otherwise an infinite-capacity edge is cut), so (t_v, t) contributes $\nabla(v)$ to $\text{val}(S')$. All remaining contribution to $\text{val}(S')$ comes from edges inside D , i.e. from $\text{val}(S^D)$. Hence

$$\text{val}(S') \geq \text{val}(S^D) + \sum_{v \notin \hat{\pi}(S^D)} \Delta(v) + \sum_{v \in \pi(S^D)} \nabla(v).$$

Since $\text{val}(S') \leq \alpha \cdot \text{val}(f')$ and $\text{val}(f) = \text{val}(f')$, and since

$$\hat{\pi}(S^D) \subseteq S \subseteq \pi(S^D),$$

we get

$$\text{val}(S) + \sum_{v \in S} \nabla(v) + \sum_{v \notin S} \Delta(v) \leq \text{val}(S^D) + \sum_{v \in \pi(S^D)} \nabla(v) + \sum_{v \notin \hat{\pi}(S^D)} \Delta(v) \leq \alpha \cdot \text{val}(f).$$

Finally, we output the feasible flow f/κ and the cut S . Multiplying the right-hand side by κ to account for

¹⁰For technical reasons (cut-matching game in expander decomposition), we require this max flow algorithm to return an approximate max flow and min cut pair.

scaling gives

$$\text{val}(S) + \sum_{v \in S} \nabla(v) + \sum_{v \notin S} \Delta(v) \leq \alpha \kappa \cdot \text{val}(f/\kappa),$$

so $(f/\kappa, S)$ is an $\alpha \cdot \kappa$ -ApMFC pair. $\square$

Now we can prove [Lemma A.27](#).

Proof of Lemma A.27. Given a directed graph G , apply [Theorem A.26](#) to construct a DAG projection of G with an $n^{o(1)}$ -congestion-preserving efficient projection algorithm, of size $\hat{O}(|E(G)|)$, and with a topological order. Then apply [Lemma A.28](#) to obtain an $n^{o(1)}$ -approximate max flow on G . It is folklore that an approximate max-flow algorithm can be turned into an exact max-flow algorithm by working on the residual graph and rerunning the approximation, so the lemma follows. $\square$

Single-source Bounded Minimum Cuts. Another application is the single-source k -bounded minimum cuts problem. Previously, efficient algorithms were known only for DAGs. Using our congestion DAG projection, we obtain an algorithm for general graphs.

Lemma A.29. *There is a randomized algorithm given a directed graph with edge capacities, a source s , and an integer k , find $n^{o(1)}$ -approximate (s, t) -minimum cut for all $t \in V$ with $\text{MinCut}_G(s, t) \leq k$ in $k^\omega m^{1+o(1)}$ time.*

Proof. Given a directed graph $G = (V, E)$ with edge capacities, apply [Lemma A.42](#) with $\sigma = 1$ and a directed expander hierarchy whose last layer $E_t = \emptyset$, to construct a DAG projection with an $n^{o(1)}$ -congestion-preserving efficient projection algorithm of size $\hat{O}(|E(G)|)$ and integral capacities. Here we use the almost-linear-time max-flow algorithm of [\[Che+25\]](#) as the oracle and the directed expander-hierarchy algorithm of [\[Ber+25a\]](#) to obtain the hierarchy in $m^{1+o(1)}$ time.

Then, add a vertex s^* to D and connect it to every vertex in $\pi^{-1}(s)$ with capacity k . For every $t \in V \setminus \{s\}$, add a vertex t^* to D and connect every vertex in $\pi^{-1}(t)$ to t^* with capacity k . For every edge $(u, v) \in E(D)$, replace it by $\min(k, U_D(u, v))$ parallel uncapacitated edges. Denote the resulting graph by D' . Then $|E(D')| = k \cdot \hat{O}(|E(G)|)$.

Run the single-source k -edge-connectivity algorithm of [\[CLL13, Theorem 1.2\]](#) on D' from s^* to all t^* to obtain $\lambda_{D'}(s^*, t^*)$ for all t , in time $k^\omega \hat{O}(|E(G)|)$. We claim that $\lambda_{D'}(s^*, t^*)$ is an $n^{o(1)}$ -approximation to $\lambda_G(s, t)$ whenever $\lambda_G(s, t) \leq k$.

For the lower bound, let f' be a max flow from s^* to t^* in D' . Restrict f' to D to obtain f^D whose source demand lies on $\pi^{-1}(s)$ and sink demand lies on $\pi^{-1}(t)$. Because we introduced at most as many parallel edges as the original capacity, the congestion does not increase. Applying the projection algorithm yields a flow f from s to t in G of the same value and congestion at most $n^{o(1)}$. Scaling down by $n^{o(1)}$ makes it feasible in G , so

$$\lambda_G(s, t) \geq \lambda_{D'}(s^*, t^*)/n^{o(1)}.$$

For the upper bound, let S' be a minimum (s^*, t^*) -cut in D' with value $\lambda_{D'}(s^*, t^*)$. If S' cuts an edge adjacent to s^* or t^* , then the cut has value at least $k \geq \lambda_G(s, t)$ (by the assumption $\lambda_G(s, t) \leq k$), so we are done. Otherwise, $\pi^{-1}(s) \subseteq S'$ and $\pi^{-1}(t) \cap \bar{S}' = \emptyset$. Restrict S' to D to get S^D . By construction (replacing each edge by up to k parallels), either the value of S^D increases past k , in which case we are done, or it stays the same. Moreover, $s \in \hat{\pi}(S^D)$ and $t \notin \pi(S^D)$, so applying the projection algorithm to S^D gives a valid (s, t) -cut in G of value at most $\lambda_{D'}(s^*, t^*)$. $\square$

A.5 Distance DAG Projection Construction

In this section, we show an efficient parallel algorithm for constructing a $(1 + \varepsilon)$ -distance-preserving DAG projection, using oracle calls to only an approximate SSSP on DAGs.

Theorem A.30. *There is a randomized algorithm that, given a directed graph G with edge lengths and parameters $0 < \varepsilon < o(1)$ and $0 < \delta < 0.5$, constructs a $(1 + \varepsilon)$ -distance-preserving DAG projection of G with width*

$$w = \left(\tilde{O}(1/\varepsilon)\right)^{\log^\delta n} \cdot n^{1/\log^\delta n}.$$

The algorithm makes $n^{o(1)}w$ -work-efficient $\tilde{O}(1)$ -depth-efficient calls to an $(1 + \varepsilon/\log n)$ -ApxSSSP-DAG oracle on graphs.

High-Level Strategy. By Lemma A.15, to reduce SSSP on general graphs to DAGs, it suffices to construct a $(1 + \varepsilon)$ -distance-preserving DAG projection with $\varepsilon = o(1/\log n)$. The difficulty is that we only know how to construct such a projection using oracle calls to an *exact* SSSP algorithm (to build an LDD), which creates a “chicken-and-egg” situation.

To resolve this issue, as mentioned in the overview, we will first define the h -length version of DAG projection.

Definition A.31. A DAG projection D onto G is h -length λ -distance-preserving if the following holds: for every $s, t \in V$ with $\text{dist}_G(s, t) \leq h$, we have

$$\text{dist}_G(s, t) \leq \text{dist}_D(\pi^{-1}(s), \pi^{-1}(t)) \leq \lambda \cdot \text{dist}_G(s, t).$$

We first observe that, when $h = \tilde{O}(1)$, the problem becomes trivial: making h copies of the vertex set, and build a h -layered graph with original edges connecting adjacent layers suffices (assuming edge length are positive integers).

Our main technical contribution in this section is to show the following spiral reduction to graduate reduce h .

1. Using similar ideas in the overview, we can show how to reduce the algorithm of constructing (a fixed size) h -length distance-preserving DAG projection to h -length SSSP.
2. We can reduce h -length SSSP to h/z -length distance-preserving DAG projection with the help of a DAG oracle for SSSP using Lemma A.34.

Organization. In Section A.5.1, we show the second point of reducing h -length SSSP to h/z -length distance-preserving DAG projection. In later sections, we show the algorithm of reducing constructing a fixed-size DAG projection to an SSSP oracle, and its analysis.

A.5.1 SSSP via DAG Projections for Smaller Length Constraints

The following definition will be used repeatedly to construct DAG projections from smaller ones.

Definition A.32 (Induced DAG Projections). Let $G = (V, E)$ be a directed graph and let $(G_1, G_2, \dots, G_z)$ be a sequence of DAG projections of G . We say that G' is a DAG projection of G *induced* from $(G_1, G_2, \dots, G_z)$ if G' is constructed as follows:

- let $G' = G_1 \cup G_2 \cup \dots \cup G_z$ (i.e. take the disjoint union of the DAGs);
- for every $u \in V(G_i)$ and $v \in V(G_j)$ with $i < j$, if $\pi(u) = \pi(v)$ or $(\pi(u), \pi(v)) \in E(G)$, then we add the edge (u, v) to G' with the same length and capacity as in G .

Intuitively, G' “concatenate” the DAG projections $G_1, \dots, G_z$ together, and then “lift” original edges of G to connect between different G_i .

We first show that a distance DAG projection allows us to reduce approximate SSSP on general graphs to SSSP on DAGs. The reduction uses an additional parameter λ that lets us decrease the length bound.

Lemma A.33. *Suppose $0 < \varepsilon, \varepsilon' < 1$. There is an algorithm that, given*

- *a directed graph G with edge lengths,*
- *an integer $\lambda \geq 1$,*

- an h -length $(1 + \varepsilon)$ -distance-preserving DAG projection D of G with width w , and
- a source $s \in V(G)$,

solves (λh) -length $(1 + \varepsilon)(1 + \varepsilon')$ -ApSSSP on (G, s) by making λw -work-efficient $\tilde{O}(1)$ -depth-efficient calls to an $(1 + \varepsilon')$ -ApSSSP-DAG oracle.

Proof. Let D' be the induced DAG projection induced from a sequence of λ copies of D (cf. [Definition A.32](#)). For every $s' \in V(D')$ with $\pi(s') = s$, we run the $(1 + \varepsilon')$ -ApSSSP-DAG oracle on D' with source s' . Let $\tilde{d}_{D'}(s', v')$ be the resulting approximate distance from s' to v' in D' .

For each $v \in V(G)$, we output

$$d_G(v) := \min\{\tilde{d}_{D'}(s', v') \mid \pi(s') = s, \pi(v') = v\}.$$

Since there are at most w preimages of any vertex in one copy and we have λ copies, this makes at most λw oracle calls. Each such call is on a graph of size $O(\lambda w m)$, so the total extra work is $O(\lambda w m)$ and depth is $\tilde{O}(1)$.

For correctness, note first that D' is still a projection of G , so any path in D' projects to a path of the same length in G . Therefore $d_G(v) \geq \text{dist}_G(s, v)$.

Now let $v \in V(G)$ with $\text{dist}_G(s, v) \leq \lambda h$, and let p be a shortest (s, v) -path in G . Partition p into at most λ subpaths $p_1, \dots, p_\lambda$ such that every p_i has length at most h (this is possible because the total length is at most λh). By the definition of an h -length $(1 + \varepsilon)$ -distance-preserving DAG projection, for each p_i there is a corresponding path p'_i in D with

$$\pi(\text{Start}(p'_i)) = \text{Start}(p_i), \quad \pi(\text{End}(p'_i)) = \text{End}(p_i), \quad \text{and} \quad \ell_D(p'_i) \leq (1 + \varepsilon) \cdot \ell_G(p_i).$$

Let p''_i be the copy of p'_i in the i -th copy of D inside D' . By the construction of the induced projection (we connect later copies after earlier ones when their projections match), the concatenation

$$p'' := p''_1 \oplus p''_2 \oplus \dots \oplus p''_\lambda$$

is a valid path in D' from some $s' \in \pi^{-1}(s)$ in the first copy to some $v' \in \pi^{-1}(v)$ in the last copy. Its length is

$$\ell_{D'}(p'') \leq (1 + \varepsilon) \sum_i \ell_G(p_i) = (1 + \varepsilon) \cdot \ell_G(p) = (1 + \varepsilon) \cdot \text{dist}_G(s, v).$$

The oracle gives a $(1 + \varepsilon')$ -approximation to this distance in D' , so

$$d_G(v) \leq (1 + \varepsilon') \cdot \ell_{D'}(p'') \leq (1 + \varepsilon')(1 + \varepsilon) \cdot \text{dist}_G(s, v).$$

□

Since exact SSSP can be reduced to approximate SSSP, we can now reduce exact SSSP to DAGs.

Lemma A.34. *There is an algorithm that, given an m -edge directed graph G with edge lengths, a source s , and integers $h \geq 1$ and $\lambda \geq 1$, solves (λh) -length exact SSSP. The algorithm makes*

- $O(1)$ -work-efficient $\tilde{O}(1)$ -depth-efficient calls to an algorithm that outputs an h -length $(1 + o(1/\log n))$ -distance-preserving DAG projection of width w on graphs with $O(m)$ edges,
- $O(\lambda w)$ -work-efficient $\tilde{O}(1)$ -depth-efficient calls to an $(1 + o(1/\log n))$ -ApSSSP-DAG oracle.

Proof. By [Corollary A.16](#), to solve (λh) -length exact SSSP it suffices to solve $\tilde{O}(1)$ instances of (λh) -length $(1 + o(1/\log n))$ -ApSSSP on graphs with $\tilde{O}(m)$ edges.

For one such instance on a graph G' with $\tilde{O}(m)$ edges, first run the DAG-projection oracle on G' to obtain an h -length $(1 + o(1/\log n))$ -distance-preserving DAG projection D of G' with width w . Then apply [Lemma A.33](#) with $\varepsilon = o(1/\log n)$ and $\varepsilon' = o(1/\log n)$ to solve the (λh) -length approximate SSSP instance using λw calls to an $(1 + o(1/\log n))$ -ApSSSP-DAG oracle on graphs with $O(\lambda w m)$ edges and with additional $O(\lambda w m)$ work and $\tilde{O}(1)$ depth. □

A.5.2 DAG Projections via SSSP

In this section, we describe a recursive algorithm $\text{DISTDAGPROJ}(G, \varepsilon, h)$ that takes a directed graph G with edge lengths and returns an h -length $(1 + \varepsilon)$ -distance-preserving DAG projection of G with width $\hat{O}(1)$. To obtain [Theorem A.30](#), it suffices to call $\text{DISTDAGPROJ}(G, \varepsilon, nW)$, since every shortest s - t path has length at most nW . Let $\delta \in (0, 1)$ be a constant parameter.

Step 1 (LDD). The algorithm computes $O(\log n)$ independent LDDs of G with diameter 2^i and slack $s_{\text{LDD}} = O(\log^2 n)$, for every

$$0 \leq i \leq \lceil \log h \rceil,$$

and collects them in a family $\mathcal{S}$. These LDDs are computed by combining [Lemma A.18](#) and [Lemma A.34](#), i.e. we reduce the LDD computations to

1. $\tilde{O}(1)$ calls to $\text{DISTDAGPROJ}(G', \varepsilon, h/\lambda)$ where G' has $O(m)$ edges, and
2. $\tilde{O}(1)$ calls to the $(1 + \varepsilon/\log n)$ -ApxSSSP-DAG oracle on graphs with $\tilde{O}(\lambda m)$ edges,

where we set the parameter $\lambda = 2^{\log^{1-\delta} n}$.

By [Definition A.17](#), each $S \in \mathcal{S}$ is a sequence of vertex sets forming a partition of V . Let $\sigma = 2^{\log^{1-\delta} n}$ be a parameter. For each $S \in \mathcal{S}$ and each $C \in S$, we call C

- a *large cluster* if $|C| \geq n/\sigma$, and
- a *small cluster* if $|C| < n/\sigma$.

Step 2 (recursive construction for small clusters). For each $S \in \mathcal{S}$ and each small cluster $C \in S$, we make the recursive call

$$D_C \leftarrow \text{DISTDAGPROJ}(G[C], (1 - \frac{1}{\log n}) \cdot \varepsilon, nW).$$

Step 3 (shortest-path trees for large clusters). We first make a recursive call on the whole graph to get a smaller-length projection:

$$D^\ell \leftarrow \text{DISTDAGPROJ}(G, (1 - \frac{10}{\log n}) \cdot \varepsilon, h/\lambda).$$

For each $S \in \mathcal{S}$ and each large cluster $C \in S$, let $r_C \in C$ be an arbitrary vertex. Using [Lemma A.33](#) with the D^ℓ as the required DAG projection, together with the $(1 + \varepsilon/\log n)$ -ApxSSSP-DAG oracle, we compute an h -length α -approximate shortest-path tree T_C^{out} rooted at r_C , where

$$\alpha = \left(1 + (1 - \frac{10}{\log n})\varepsilon\right) \cdot \left(1 + \frac{\varepsilon}{\log n}\right) \leq 1 + \left(1 - \frac{5}{\log n}\right)\varepsilon,$$

and we also compute a reversed shortest-path tree T_C^{in} rooted at r_C (i.e. every (s, r_C) -path in T_C^{in} is a shortest (s, r_C) -path). Let D_C be the DAG obtained by combining T_C^{out} and T_C^{in} (as *disjoint copies* of subgraphs of G) with their roots r_C identified, and define the vertex labeling to $V(G)$ in the natural way.

Step 4 (combining everything). For each $S = (V_1, V_2, \dots, V_z) \in \mathcal{S}$, let D_S be the DAG projection induced from the sequence

$$(D_{V_1}, D_{V_2}, \dots, D_{V_z}),$$

where D_{V_i} is the DAG constructed in Step 2 or Step 3 depending on whether V_i is small or large.

For each $S \in \mathcal{S}$, we make

$$z := \frac{50(\log n)s_{\text{LDD}}}{\varepsilon}$$

copies of D_S and arrange them into a sequence $(D_S^{(1)}, D_S^{(2)}, \dots, D_S^{(z)})$. Let D'_S be the DAG projection of G induced from this sequence.

Let

$$D' := \bigcup_{S \in \mathcal{S}} D'_S.$$

We make two copies of D' , denoted D'_1, D'_2 , and let the final DAG projection D be the DAG induced from (D'_1, D'_2) . We return D .

Base case. When G has a constant number of vertices or h is a constant, we return the DAG projection D induced from

$$(G^{(1)}, G^{(2)}, \dots, G^{(\max(|V(G)|, h))})$$

(i.e. we repeat G exactly $\max(|V(G)|, h)$ times).

A.5.3 Analysis: Approximation

It is immediate that D is an DAG projection onto G , since D is constructed only by applying [Definition A.32](#) to a sequence consisting of (1) DAG projections onto subgraphs of G and (2) subgraphs of G . Thus, it remains to show that D is an h -length $(1 + \varepsilon)$ -distance-preserving DAG projection of G . By [Lemma A.12](#) and [Definition A.11](#), it is enough to prove that for every $s, t \in V(G)$ with $\text{dist}_G(s, t) \leq h$,

$$\min\{\text{dist}_D(s', t') \mid \pi(s') = s, \pi(t') = t\} \leq (1 + \varepsilon) \cdot \text{dist}_G(s, t).$$

We prove this by induction on $|V(G)|$ and on h . In the base case, when $|V(G)|$ is constant or h is constant, any shortest path of length at most h has at most $\max(|V(G)|, h)$ vertices (the path is simple and edge lengths are positive). Because we stacked $\max(|V(G)|, h)$ copies of G , the induced DAG contains a path from the copy of s in the first layer to the copy of t in the last layer with exactly the same length, so the base case holds.

Lemma A.35 (Induction). *Assume all recursive calls in $D \leftarrow \text{DISTDAGPROJ}(G, \varepsilon, h)$ are correct. Then, with high probability, for every $s, t \in V(G)$ with $\text{dist}_G(s, t) \leq h$, we have*

$$\min\{\text{dist}_D(s', t') \mid \pi(s') = s, \pi(t') = t\} \leq (1 + \varepsilon) \cdot \text{dist}_G(s, t).$$

Proof. Fix $s, t \in V(G)$ with $\text{dist}_G(s, t) \leq h$. We show the inequality holds w.h.p. for this pair; a union bound over all s, t gives the lemma.

Let p be a shortest s - t path in G . We need the following.

Lemma A.36. *With high probability, there exists $S \in \mathcal{S}$ such that*

1. *every cluster in S has diameter at most $\frac{\varepsilon \cdot \ell_G(p)}{9 \log n}$, and*
2. *p contains at most $50(\log n)s_{\text{LDD}}/\varepsilon$ reversed edges in S (see [Definition A.17](#)).*

Proof. If $\ell_G(p) = O((\log n)s_{\text{LDD}}/\varepsilon)$, then $|p| = O((\log n)s_{\text{LDD}}/\varepsilon)$ (edge lengths are positive), and taking the LDD with diameter 1 satisfies the claim: every cluster is a singleton, so diameter 0, and the number of reversed edges is $O((\log n)s_{\text{LDD}}/\varepsilon)$.

Otherwise, $\ell_G(p) = \omega((\log n)s_{\text{LDD}}/\varepsilon)$. Among the $O(\log n)$ diameter scales 2^i we took (up to at least $h \geq \ell_G(p)$), there is one scale d in

$$d \in \left[\frac{\varepsilon \cdot \ell_G(p)}{18 \log n}, \frac{\varepsilon \cdot \ell_G(p)}{9 \log n} \right].$$

For that scale we sampled $O(\log n)$ independent LDDs. By [Definition A.17](#), the expected number of reversed edges on p in such an LDD is at most

$$s_{\text{LDD}} \cdot \frac{\ell_G(p)}{d} \leq 36(\log n)s_{\text{LDD}}/\varepsilon.$$

By Markov's inequality and independence across the $O(\log n)$ trials at that scale, w.h.p. one of them has at most $50(\log n)s_{\text{LDD}}/\varepsilon$ reversed edges and diameter at most $\varepsilon \cdot \ell_G(p)/(9 \log n)$. $\square$

Let $S \in \mathcal{S}$ be as in [Lemma A.36](#), and let $d \leq \varepsilon \cdot \ell_G(p)/(9 \log n)$ be its diameter parameter. Let

$$(C_1, C_2, \dots, C_q)$$

be the sequence of clusters of S that p visits, in order. Let p_i be the subpath of p inside C_i . By Lemma A.36, there are at most

$$q_{\text{rev}} := 50(\log n)s_{\text{LDD}}/\varepsilon$$

reversed edges along p . For each p_i , define $\text{Pre}(p_i)$ to be the number of reversed edges on p before p_i , and $\text{Suf}(p_i)$ the number after p_i .

Let C_x be the first large cluster along this sequence, and C_y the last large cluster (it is possible that $C_x = C_y$; the one-large-cluster case is handled similarly, so we assume $x < y$ for clarity). Let

$$p_{\text{mid}} := p_x \oplus p_{x+1} \oplus \cdots \oplus p_y$$

be the portion of p from the first to the last large cluster.

For every $i < x$, C_i is a small cluster. By the induction hypothesis, D_{C_i} contains a path p'_i that projects to p_i and

$$\ell_{D_{C_i}}(p'_i) \leq \left(1 + \left(1 - \frac{1}{\log n}\right)\varepsilon\right) \cdot \ell_G(p_i).$$

Recall that in Step 4 we made

$$z := \frac{50(\log n)s_{\text{LDD}}}{\varepsilon}$$

copies of D_S , so we can place p'_i in the $\text{Pre}(p_i)$ -th copy of D_S . We do the symmetric construction for every $i > y$ (these go into the second big union). Hence we can replace every p_i with $i < x$ or $i > y$ by a corresponding p'_i with only a factor $(1 + (1 - 1/\log n)\varepsilon)$ blow-up, and the edges linking p'_i to p'_{i+1} exist in D :

- if the edge was reversed, we put p'_{i+1} in a later copy, so the induced construction adds the connecting edge;
- if it was not reversed, the edge is already in the same copy.

It remains to replace p_{mid} in D . By construction of D_{C_x} (large cluster), we have a reversed SPT $T_{C_x}^{\text{in}}$, so there is a path

$$p'_x : \text{Start}(p_{\text{mid}}) \rightarrow r_{C_x}$$

in D_{C_x} . Its projection is the corresponding path in G , and since the cluster diameter is at most $d \leq \varepsilon \ell_G(p)/(9 \log n) \leq h$, and $T_{C_x}^{\text{in}}$ is a $(1 + (1 - 5/\log n)\varepsilon)$ -approximation,

$$\ell(p'_x) \leq (1 + (1 - 5/\log n)\varepsilon) \cdot d \leq \frac{\varepsilon \cdot \ell_G(p)}{8 \log n}.$$

Similarly, in $T_{C_y}^{\text{out}}$ we have a path

$$p'_y : r_{C_y} \rightarrow \text{End}(p_{\text{mid}})$$

with the same kind of bound.

Between the two large clusters, in $T_{C_x}^{\text{out}}$ we have a path

$$p'_{\text{mid}} : r_{C_x} \rightarrow r_{C_y}$$

that is a $(1 + (1 - 5/\log n)\varepsilon)$ -approximate shortest path in G . Also,

$$\text{dist}_G(r_{C_x}, r_{C_y}) \leq \ell_G(p_{\text{mid}}) + 2d$$

because we can go from r_{C_x} to the entry of p_{mid} in C_x in at most d , follow p_{mid} , then go from the exit in C_y to r_{C_y} in at most d .

We place p'_x and p'_{mid} in the $\text{Pre}(p_x)$ -th copy of D_S in the first big union, and p'_y in the $\text{Pre}(p_y)$ -th copy in the second big union. By the induced construction, these connect to form a path

$$p''_{\text{mid}} := p'_x \oplus p'_{\text{mid}} \oplus p'_y$$

in D . Its length satisfies

$$\begin{aligned}\ell_D(p''_{\text{mid}}) &\leq 2 \cdot \frac{\varepsilon \cdot \ell_G(p)}{8 \log n} + (1 + (1 - 5/\log n)\varepsilon) \cdot (\ell_G(p_{\text{mid}}) + 2d) \\ &\leq (1 + (1 - 5/\log n)\varepsilon) \cdot \ell_G(p_{\text{mid}}) + \frac{\varepsilon \cdot \ell_G(p)}{2 \log n}.\end{aligned}$$

Finally, concatenating the replaced prefix, the middle p''_{mid} , and the replaced suffix gives a path in D from some s' with $\pi(s') = s$ to some t' with $\pi(t') = t$ whose length is at most

$$(1 + (1 - \frac{1}{\log n})\varepsilon) \cdot (\ell_G(p) - \ell_G(p_{\text{mid}})) + (1 + (1 - \frac{5}{\log n})\varepsilon) \cdot \ell_G(p_{\text{mid}}) + \frac{\varepsilon \cdot \ell_G(p)}{2 \log n} \leq (1 + \varepsilon) \cdot \ell_G(p),$$

for n sufficiently large. This proves the inductive step. $\square$

A.5.4 Analysis: Size and Complexity

Width of the DAG projection. The width of the DAG projection is bounded as follows.

Lemma A.37. *Suppose $D \leftarrow \text{DISTDAGPROJ}(G, \varepsilon, h)$ where G has n vertices. Then D has width*

$$f(n, \varepsilon) = (\tilde{O}(1/\varepsilon))^{\log^\delta n} \cdot n^{1/\log^\delta n}$$

Proof. The base case is trivial since each vertex is duplicated only a constant number of times.

Recall the structure of D . The final DAG D is induced from two copies of D' . Each D' is the union (over $S \in \mathcal{S}$) of D'_S . Each D'_S is obtained from

$$z := 50(\log n) s_{\text{LDD}}/\varepsilon$$

copies of D_S . Each D_S contains one copy of D_C for every cluster C in S .

For a cluster C we have two cases.

- If C is a *small* cluster, then by the induction hypothesis every vertex in C is duplicated

$$f(|C|, (1 - 1/\log n)\varepsilon)$$

times in D_C .

- If C is a *large* cluster, then we build D_C just from the two trees, so every vertex of G participates in D_C at most twice.

Large clusters have size at least n/σ and are disjoint, so there are at most σ large clusters. Thus, across all large clusters, each vertex is duplicated at most 2σ times. Small clusters are disjoint and have size at most n/σ , so across all small clusters, each vertex is duplicated at most

$$f(n/\sigma, (1 - 1/\log n)\varepsilon)$$

times.

We have $O(\log^2 n)$ LDDs in $\mathcal{S}$ (because we take $O(\log n)$ scales and $O(\log n)$ independent LDDs per scale). Every such LDD is blown up by a factor of $O((\log n) s_{\text{LDD}}/\varepsilon)$. Putting this together, we obtain the recursion

$$f(n, \varepsilon) \leq O((\log n)^5/\varepsilon) \cdot (f(n/\sigma, (1 - 1/\log n)\varepsilon) + 2\sigma).$$

Taking

$$\sigma = 2^{\log^{1-\delta} n}$$

and unwinding for at most $\log^\delta n$ levels (since $n \mapsto n/\sigma$ shrinks by $2^{\log^{1-\delta} n}$ each time), we get

$$f(n, \varepsilon) = (\tilde{O}(1/\varepsilon))^{\log^\delta n} \cdot n^{1/\log^\delta n}.$$

□

Complexity. Let $T(m, n, \varepsilon, h)$ be the work of $\text{DISTDAGPROJ}(G, \varepsilon, h)$ on a graph G with m edges and n nodes. Let $\mathcal{O}_{\text{DAG}}(m)$ denote the work of one call to the DAG-SSSP oracle on a graph with m edges.

Step 1 runs $O(\log^2 n)$ LDD constructions. By [Lemma A.18](#) and [Lemma A.34](#), this costs

$$\tilde{O}(1) \text{ calls to } (1 + \varepsilon/\log n)\text{-ApSSSP-DAG on graphs with } O(\lambda w m) \text{ edges,}$$

plus

$$\tilde{O}(1) \text{ recursive calls } T(m, n, \varepsilon, h/\lambda),$$

Step 2 makes recursive calls on all small clusters. All such calls are in parallel. Since the clusters of a fixed LDD partition V , and we have only $O(\log^2 n)$ LDDs, we have

$$\sum_{C \in \mathcal{S}, S \in \mathcal{S}} |E(G[C])| \leq O(m \log^2 n).$$

So we can upper-bound Step 2 by

$$O(\log^2 n) \cdot T(m, n/\sigma, (1 - 1/\log n)\varepsilon, nW).$$

If these recursive calls internally make more oracle calls, we can batch them (by taking the union of the graphs and adding a super source), so the stated number of oracle calls in [Theorem A.30](#) still holds.

Step 3 makes one recursive call

$$T(m, n, (1 - 10/\log n)\varepsilon, h/\lambda)$$

and, since there are at most σ large clusters, makes $O(\sigma)$ calls to the $(1 + \varepsilon/\log n)$ -ApSSSP-DAG oracle on graphs with $\hat{O}(m)$ edges.

Step 4 costs time proportional to the size of the final DAG, which is $O(w \cdot m)$ by [Lemma A.37](#).

Recall that $\lambda = \sigma = 2^{\log^{1-\delta} n}$. Putting these together, we get the recurrence

$$T(m, n, \varepsilon, h) \leq \tilde{O}(T(m, n, \varepsilon, h/\lambda)) + \sum_i \tilde{O}(T(m_i, n/\sigma, (1 - 1/\log n)\varepsilon, nW)) + \mathcal{O}_{\text{DAG}}(O(\lambda w m)).$$

Where m_i is the number of edges in the i -th small cluster. Observe that, since $\lambda = \sigma = 2^{\log^{1-\delta} n}$, the recursion depth is at most $\log^{2\delta} n$ for expanding both the n part and h part, so the parameter ε passed down the recursion never shrinks below

$$(1 - 1/\log n)^{\log^{2\delta} n} \varepsilon \geq \varepsilon/2$$

for n large enough. Moreover, the total number of oracle calls is

$$(\tilde{O}(1))^{\log^{2\delta} n} = \hat{O}(1),$$

and each is on a graph with

$$\tilde{O}(\lambda w m) = \hat{O}(w m)$$

edges (since w is a function of n, ε and ε stays within a constant factor of the root value). Thus the total extra work from oracle calls is

$$(\tilde{O}(1))^{\log^{2\delta} n} \cdot \lambda w m = \hat{O}(w m).$$

Finally, since the recursion depth is only $\log^{2\delta} n$ and all work inside a level can be parallelized, the overall depth is $\tilde{O}(1)$.

A.6 Congestion DAG Projection Construction

In this section, we provide an efficient parallel algorithm for constructing congestion DAG projections using oracle calls to only approximate max-flow/min-cut on DAGs.

Theorem A.26. *Suppose there is an oracle for α -ApMFC-DAG where $\alpha = n^{o(1)}$. Then there is a randomized algorithm that, given a directed graph $G = (V, E)$ with edge capacities, outputs a DAG projection D of G together with an efficient $n^{o(1)}$ -congestion-preserving projection algorithm, such that $|E(D)| = \tilde{O}(|E(G)|)$ (and D 's topological order is returned). The algorithm makes $n^{o(1)}$ work-efficient and $n^{o(1)}$ depth-efficient calls to α -ApMFC-DAG.*

High-Level Strategy. Assuming an oracle for $n^{o(1)}$ -approximate MFC on DAGs, our strategy for constructing a $n^{o(1)}$ -congestion-preserving DAG projection is as follows.

Similar to the distance-preserving DAG projection case, the difficulty is that, if we follow the idea from the overview, then an algorithm for DAG projection requires solving max flow. Our idea is to define a relaxed notion of (κ, δ) -congestion-preserving DAG projection as follows.

Definition A.38. Let D be a DAG projection of $G = (V, E)$. A (κ, δ) -congestion-preserving projection algorithm associated with D takes as input either

- a flow f^D in D with $\pi(\text{Supp}(\text{Dem}(f^D))) \subseteq V$, and returns a flow f in G with congestion at most $\kappa \cdot \text{Cong}(f^D)$ such that

$$\text{Dem}(f) \preceq \pi(\text{Dem}(f^D)) \quad \text{and} \quad \text{val}(f) \geq \text{val}(f^D) - \delta,$$

or

- a cut S^D in D , and returns a cut S in G with value at most $\text{val}(S^D)$ such that

$$\hat{\pi}(S^D) \setminus \{\perp\} \subseteq S \subseteq \pi(S^D).$$

The projection algorithm is *efficient* if its complexity is at most the complexity of constructing D .

When δ is tiny enough, it is the κ -congestion-preserving DAG projection we need.

We first observe that, when δ is very big, the problem becomes trivial. In particular, $(\kappa = 1, \delta \geq U(E))$ -congestion-preserving DAG projection means the flow projection algorithm does not need to return any flow, so a DAG with two copies of V (denoted by V_1, V_2), and a center vertex c with edge capacities $\deg_G(v)$ connecting from each copies of v in V_1 to c ; connecting to each copy of v in V_2 from c , would work as a DAG satisfying the cut projection.

Our main technical contribution is to show that it is possible to reduce (κ, δ) -congestion-preserving DAG projection to a larger δ DAG projection. In particular, we will show the following spiral recursion.

- We will show how to get (κ, δ) -congestion-preserving DAG projection using roughly (κ, δ) -approximate max flow by following the idea from the overview.
- Then we will show how to get (κ, δ) -approximate max flow by roughly $(\kappa, \delta \cdot z)$ -approximate DAG projection where $z = n^{o(1)}$ is an appropriate parameter. This step requires concatenating many copies of $(\kappa, \delta \cdot z)$ -approximate DAG projection to get a (κ, δ) -approximate DAG projection.

Organization. As preliminaries, we first define Expander Decomposition and Hierarchy in [Section A.6.1](#). To implement the main technical contribution that gives the above spiral reduction, we give the following chain of reduction:

- DAG projection to Expander Hierarchy ([Section A.6.2](#)). The key idea for reducing the additive approximation δ by making copies is presented here.
- Expander Hierarchy to Expander Decomposition ([Section A.6.3](#)).
- Expander Decomposition to approximate max flow ([Section A.6.4](#))

- Approximate max flow on general graphs to approximate max flow on DAGs using DAG projection (Section A.6.5).

And in the end, we combine everything together and finish the spiral reduction in Section A.6.6.

A.6.1 Expander Decomposition and Hierarchy

Although we did *strong expander decomposition* in the intro in the sense that edges are expanding in each induced subgraph of SCCs, we will do *weak expander decomposition*, which only guarantees expanding in the whole graph in this section, because it admits a simpler algorithm.

Basic definitions of ordered vertex-partition. Let $G = (V, E)$ be a directed graph. For a sequence of vertex sets $\mathcal{V} = (V_1, \dots, V_z)$ that partitions V (which we call an ordered vertex-partition), define

$$E^{\text{rev}}(\mathcal{V}) = \{(u, v) \in E \mid u \in V_i, v \in V_j, j < i\}$$

to be the set of *reversed edges* with respect to $\mathcal{V}$.

For two sequences $(V_1, \dots, V_x)$ and $(V'_1, \dots, V'_y)$ that both partition V , we say $(V_1, \dots, V_x)$ *refines* $(V'_1, \dots, V'_y)$ if we can write

$$(V_1, \dots, V_x) = (\mathcal{C}_1, \dots, \mathcal{C}_y),$$

where each $\mathcal{C}_i$ is a subsequence of $(V_1, \dots, V_x)$ that partitions V'_i .

Given $G = (V, E)$ and a ordered vertex-partition $\mathcal{V} = (V_1, \dots, V_z)$, we say a demand (Δ, ∇) is $\mathcal{V}$ -constrained if for every V_i ,

$$\sum_{v \in V_i} \Delta(v) = \sum_{v \in V_i} \nabla(v),$$

i.e. each part is individually balanced.

Definition A.39 (Terminal Expanding). Let $G = (V, E)$ be a directed graph with edge capacities, let $\mathbf{d} : V \rightarrow \mathbb{R}_{\geq 0}$, and let $\mathcal{V}$ be a ordered vertex-partition. We say $\mathbf{d}$ is $\mathcal{V}$ -constrained ϕ -expanding on G if every $\mathbf{d}$ -respecting, $\mathcal{V}$ -constrained demand can be routed in G with congestion at most $1/\phi$.

We say an edge set $F \subseteq E$ is $\mathcal{V}$ -constrained ϕ -expanding on G if vol_F is $\mathcal{V}$ -constrained ϕ -expanding on G .

Definition A.40 (Expander Decomposition). Let $G = (V, E)$ be a directed graph and $F \subseteq E$ an edge set. A (weak) F -expander decomposition with expansion ϕ and slack γ returns a ordered vertex-partition $\mathcal{V}$ such that

- F is ϕ -expanding in the graph $G - E^{\text{rev}}(\mathcal{V})$, and
- $\text{vol}(E^{\text{rev}}(\mathcal{V})) \leq \gamma \phi \cdot \text{vol}(F)$.

An associated *flow routing algorithm* takes a $\mathcal{V}$ -constrained, vol_F -respecting demand and outputs a flow in G routing that demand. It is *efficient* if its complexity is at most the complexity of constructing the decomposition.

Definition A.41 (Expander Hierarchy). Let $G = (V, E)$ be a directed graph with edge capacities. An *expander hierarchy* with t layers consists of edge sets $E_1, \dots, E_t$ and ordered vertex-partitions $\mathcal{C}_0, \dots, \mathcal{C}_t$ such that¹¹

- $\mathcal{C}_0 = (\{v_1\}, \{v_2\}, \dots, \{v_n\})$ consists only of singletons, and $\mathcal{C}_t = (V)$,
- for every $i \in [t]$, the sequence $\mathcal{C}_{i-1}$ refines $\mathcal{C}_i$,
- for every $i \in [t]$, we have $E^{\text{rev}}(\mathcal{C}_{i-1}) \subseteq E_{\geq i}$, where $E_{\geq i} := \bigcup_{j \geq i} E_j$.

We say the hierarchy has expansion $(\phi_1, \dots, \phi_t)$ if, for every $1 \leq i \leq t$, the edge set E_i is $\mathcal{C}_i$ -constrained ϕ_i -expanding on G .

¹¹This deviates slightly from the classical definition in that the $\mathcal{C}_i$'s do not have to be the SCCs of $G - E_{> i}$, and we assume the sequences $\mathcal{C}_i$ are given. This avoids explicitly computing SCCs in parallel.

An *flow routing algorithm* associated with the hierarchy takes as input a layer number i and a $\mathcal{C}_i$ -constrained, vol_{E_i} -respecting demand, and outputs a flow in G routing the demand with congestion at most $1/\phi_i$. It is *efficient* if its complexity is at most the complexity of constructing the hierarchy.

A.6.2 Congestion DAG Projections via Expander Hierarchy

In this section, we assume we are given an expander hierarchy and we show how to construct a congestion DAG projection.

Lemma A.42. *There is an algorithm that takes as input*

- *a directed graph $G = (V, E)$ with edge capacities,*
 - *an oracle that constructs an expander hierarchy of G with t layers and expansion $(\phi_1, \phi_2, \dots, \phi_t)$ where $\phi_i = \phi$ for all $i \leq t-1$,¹² together with an efficient flow routing algorithm, and*
 - *a parameter $\sigma \geq 1$, which controls the trade-off between the projection size and additive-approximation,*
- and returns a DAG projection D of G such that*
- $|E(D)| = O(2^t \sigma \cdot |E(G)|),$
 - D is (κ, δ) -congestion-preserving with

$$\kappa = O(2^t / \phi) \quad \text{and} \quad \delta = U_G(E_t) / \sigma,$$

- *and D has an efficient projection algorithm.*

The algorithm uses one call to the hierarchy oracle and performs additional $\tilde{O}(|E(D)|)$ work and $\tilde{O}(1)$ depth.

Moreover, if the edge capacities of G are integral and $\sigma = 1$, then the edge capacities of D are integral.

Algorithm (constructing D). Let $\mathcal{C}_0, \dots, \mathcal{C}_t$ and $E_1, \dots, E_t$ be the expander hierarchy returned by the oracle. Define

$$G_i := G - E_{>i} \quad \text{for } i = 0, 1, \dots, t,$$

so $G_t = G$ (since $E_{>t} = \emptyset$). For every $i = 0, 1, \dots, t$ and every cluster $C \in \mathcal{C}_i$, we will build a DAG projection for $G_i[C]$, denoted $D_i(C)$. Since $\mathcal{C}_t = (V)$, $D_t(V)$ is a DAG projection for G , and we will return $D := D_t(V)$.

For every original vertex v in $G_i[C]$, we will specify two special copy vertices of v in $D_i(C)$, denoted

$$v_i^{\text{first}}, v_i^{\text{last}} \in V(D_i(C)),$$

with $\pi(v_i^{\text{first}}) = \pi(v_i^{\text{last}}) = v$. We will later see that v_i^{first} and v_i^{last} are basically the first and last copy of vertex v in the topological order of the DAG projection. Intuitively, these are the attachment points when we later connect edges that go in or out of C . Moreover, in the construction, there will be an infinite capacity path from v_i^{first} to v_i^{last} .

Base layer. When $i = 0$, each $C \in \mathcal{C}_0$ is a singleton, say $C = \{v\}$. We let $D_0(C)$ be the one-vertex DAG with label v , and we set $v_0^{\text{first}} = v_0^{\text{last}}$ to be that vertex.

Inductive step. Suppose $i \geq 1$. We will define a parameter

$$\sigma_i = \begin{cases} 1, & \text{if } i < t, \\ \sigma, & \text{if } i = t. \end{cases}$$

Since $\mathcal{C}_{i-1}$ refines $\mathcal{C}_i$, the clusters of $\mathcal{C}_{i-1}$ that are contained in $C \in \mathcal{C}_i$ form a subsequence

$$(Y_1, \dots, Y_z)$$

that partitions C . By the induction hypothesis, for each Y_j we have already built a DAG projection $D_{i-1}(Y_j)$ of $G_{i-1}[Y_j]$.

¹²We do not care about the expansion for the last layer, because the flow routing there is contributed to the additive error.

(1) Concatenating the child DAGs. We first build an intermediate DAG $\tilde{D}_i(C)$ by taking the DAGs

$$D_{i-1}(Y_1), D_{i-1}(Y_2), \dots, D_{i-1}(Y_z)$$

in order, and for every edge $(u, v) \in E(G_i)$ with $u \in Y_x$, $v \in Y_y$, and $x < y$, we add to $\tilde{D}_i(C)$ the edge

$$(u_{i-1}^{\text{last}}, v_{i-1}^{\text{first}})$$

with capacity $U_G(u, v)$. This makes $\tilde{D}_i(C)$ a DAG projection of $G_i[C]$ consistent with the ordering induced by $(Y_1, \dots, Y_z)$.

(2) Replicating to allow congestion projection. To form $D_i(C)$, we make $2\sigma_i$ copies of *this* intermediate DAG $\tilde{D}_i(C)$, all with the same projection maps. For any vertex $x \in V(\tilde{D}_i(C))$, denote its copy in the k -th replica by $x_{(k)}$, for $1 \leq k \leq 2\sigma_i$.

We then add the following edges:

- (i) For every $u \in C$ and every $1 \leq k < 2\sigma_i$, add an edge

$$(u_{i-1, (k)}^{\text{last}}, u_{i-1, (k+1)}^{\text{first}})$$

with *infinite* capacity. This allows flow on u to “walk” forward through all replicas.

- (ii) For every $(u, v) \in E_i$ with $u, v \in C$, and every $1 \leq k < 2\sigma_i$, add an edge

$$(u_{i-1, (k)}^{\text{last}}, v_{i-1, (k+1)}^{\text{first}})$$

with capacity $U_G(u, v)$. This encodes the “fresh” edges of layer i across replicas.

- (iii) Add a dummy vertex w_i^C , i.e., $\pi(w_i^C) = \perp$. For every $u \in C$, add edges

$$(u_{i-1, (\sigma_i)}^{\text{last}}, w_i^C) \quad \text{and} \quad (w_i^C, u_{i-1, (\sigma_i+1)}^{\text{first}})$$

each with capacity $\text{vol}_{E_i}(u)$. This ensures we can “collect” and “redistribute” the E_i -volume of u between the two “halves” of the replicas without violating capacities.

Finally, we scale *all* capacities in $D_i(C)$ by a factor of $1/\sigma_i$ so that the total capacity budget per original vertex is preserved across the $2\sigma_i$ replicas. For every original vertex $u \in C$, we define two special copies in $D_i(C)$

$$u_i^{\text{first}} := u_{i-1, (1)}^{\text{first}} \quad \text{and} \quad u_i^{\text{last}} := u_{i-1, (2\sigma_i)}^{\text{last}}.$$

Since $\mathcal{C}_t = (V)$, the construction for $i = t$ produces $D_t(V)$, which we output as the final DAG projection D . This completes the construction of our congest DAG projection.

Analysis.

The following lemma bounds the size of the DAG projection.

Lemma A.43. *D has size at most $O(2^t \sigma |E(G)|)$.*

Proof. For $i \geq 1$, each $D_i(C)$ (for $C \in \mathcal{C}_i$) is composed of $2\sigma_i$ copies of $\tilde{D}_i(C)$, and $\tilde{D}_i(C)$ is the concatenation of $D_{i-1}(Y_j)$ over the partition $(Y_1, \dots, Y_z)$ of C . Thus,

$$|E(D_i(C))| \leq 2\sigma_i \cdot \left(|E(G[C])| + \sum_{j=1}^z |E(D_{i-1}(Y_j))| \right).$$

Summing this recurrence up the hierarchy (and using that $\sum_{C \in \mathcal{C}_i} |E(G[C])| \leq |E(G)|$ and $\sigma_i = \sigma$ for only layer t while $\sigma_i = 1$ for $i < t$) yields

$$|E(D_t(V))| \leq O(2^t \sigma |E(G)|).$$

□

Projection algorithm (flow). For an edge $(u, v) \in E(G)$, define its D -congestion to be

$$\sum_{(u', v') \in E(D) : \pi(u')=u, \pi(v')=v} U_D(u', v').$$

Lemma A.44. For every $(u, v) \in E(G)$, the D -congestion of (u, v) is at most 2^t .

Proof. We prove by induction on i that, for every $C \in \mathcal{C}_i$, the $D_i(C)$ -congestion of any edge (u, v) is at most 2^i .

For $i = 0$ the claim is trivial, since $D_0(C)$ has no edges.

For $i \geq 1$, $\tilde{D}_i(C)$ is a concatenation of $D_{i-1}(Y_j)$'s. Any edge of G_i that crosses between two different Y_j 's is added at most once in $\tilde{D}_i(C)$, with its original capacity, so such edges contribute congestion 1 at this level. Edges internal to a Y_j come from $D_{i-1}(Y_j)$, and by the induction hypothesis they contribute at most 2^{i-1} per copy.

Now $D_i(C)$ consists of $2\sigma_i$ copies of $\tilde{D}_i(C)$, and finally we scale down capacities by σ_i . Hence, for edges not in E_i , the total congestion is

$$2^{i-1} \cdot \frac{2\sigma_i}{\sigma_i} = 2^i.$$

For edges in E_i , we add one between every consecutive pair of copies, so there are $2\sigma_i$ of them, each scaled by $1/\sigma_i$, giving congestion 2. Taking the maximum over these two cases yields the desired bound 2^i . □

Next, we need to control the demands created at dummy vertices. For every $1 \leq i \leq t$ and $C \in \mathcal{C}_i$, the construction creates many copies of the dummy vertex w_i^C . For example, D_{i+1} makes $2\sigma_i$ copies of w_i^C . Even more copies are inductively created at higher levels. For such a w_i^C : - each incoming edge (u, w_i^C) gives u a w_i^C -source-demand equal to the capacity of that edge; - each outgoing edge (w_i^C, v) gives v a w_i^C -sink-demand equal to the capacity of that edge.

The (C, i) -source-demand of u is the sum of all w_i^C -source-demands over all dummy centers w_i^C of clusters $C \in \mathcal{C}_i$; the (C, i) -sink-demand is defined symmetrically.

Lemma A.45. For all $1 \leq i \leq t$, all $u \in V$, and all $C \in \mathcal{C}_i$, the (C, i) -source-demand (and also the (C, i) -sink-demand) of u is at most

$$2^{t-i} \cdot \frac{\text{vol}_{E_i}(u)}{\sigma_i}.$$

Proof. We prove the following slightly stronger statement by induction: for any $x \geq i$ and any cluster $C' \in \mathcal{C}_x$, the total (C, i) -source-demand (or sink-demand) of u in $D_x(C')$ is at most

$$2^{x-i} \cdot \frac{\text{vol}_{E_i}(u)}{\sigma_i}.$$

When $x = i$, this is exactly how we defined the dummy edges at level i : the (C, i) -demand is $\text{vol}_{E_i}(u)/\sigma_i$.

For $x > i$, the DAG $D_x(C')$ is made from $2\sigma_x$ copies of $\tilde{D}_x(C')$, and each $\tilde{D}_x(C')$ is a concatenation of lower-level DAGs. For a fixed u , only one of those lower-level DAGs contains the copy of u , so by induction that copy contributes at most

$$2^{x-1-i} \cdot \frac{\text{vol}_{E_i}(u)}{\sigma_i}$$

in each replica. After making $2\sigma_x$ replicas and scaling by $1/\sigma_x$, we get

$$2^{x-1-i} \cdot \frac{\text{vol}_{E_i}(u)}{\sigma_i} \cdot \frac{2\sigma_x}{\sigma_x} = 2^{x-i} \cdot \frac{\text{vol}_{E_i}(u)}{\sigma_i},$$

as desired. □

Now we are ready to state the flow projection algorithm. Let f^D be a flow in D whose support does not contain dummy vertices (i.e. all demand is on original vertices). Consider a flow path p^D of f^D that does not pass through the top-level dummy vertex w_t^V . We decompose p^D into subpaths of two types:

1. **Pure subpaths** $\tilde{p}$ that contain no dummy vertex. Every edge (u', v') on such a subpath satisfies either $(\pi(u'), \pi(v')) \in E(G)$ or $\pi(u') = \pi(v')$. Thus $\pi(\tilde{p})$ is a path in G (after removing repetitions). We create a flow path in G along $\pi(\tilde{p})$ with the same flow value.

2. **Dummy vertex subpaths** of the form (u, w_i^C, v) , where w_i^C is a dummy vertex. For each such subpath with value γ , we create a source demand γ on $\pi(u)$ and a sink demand γ on $\pi(v)$ to be later routed by the flow routing associated with layer i .

Doing this for all flow paths of f^D that avoid w_t^V gives us a partial flow f in G plus, for each layer i , a $\mathcal{C}_i$ -constrained demand (Δ_i, ∇_i) . The total flow in f^D that *does* go through w_t^V is at most

$$\sum_{u \in V} \text{vol}_{E_t}(u) = U_G(E_t),$$

because those edges were scaled by $1/\sigma_t = 1/\sigma$. Hence

$$\text{Dem}(f) \preceq \pi(\text{Dem}(f^D)) \quad \text{and} \quad \text{val}(f) \geq \text{val}(f^D) - \delta$$

with $\delta \leq U_G(E_t)/\sigma$, as claimed in Lemma A.42.

To route the dummy-hop demands, for each $i \in [t]$ we collect all tuples (u, w_i^C, v) of value c into a demand (Δ_i, ∇_i) . By Lemma A.45, this demand is $(2^{t-i} \cdot \text{vol}_{E_i}(u)/\sigma_i)$ -respecting, and by construction it is $\mathcal{C}_i$ -constrained. We scale this demand down by a factor $\sigma_i/2^{t-i}$ and apply the flow routing at layer i (which has congestion at most $1/\phi_i$), and finally scale the routed flow back up by $2^{t-i}/\sigma_i$. This gives congestion at most

$$\frac{2^{t-i}}{\sigma_i \phi_i}$$

for the layer- i part.

The final flow in G consists of: - the “trivial” projection of edges, which by Lemma A.44 has congestion at most 2^t , and - the expander-routed part, whose congestion is at most

$$\sum_{i=1}^t \frac{2^{t-i}}{\sigma_i \phi_i} \leq O(2^t/\phi)$$

since $\phi_i = \phi$ for $i \leq t-1$ and $\sigma_i \geq 1$.

The second part dominates the first, so the total congestion is $O(2^t/\phi)$, as stated in Lemma A.42. The running time is also as claimed, since flow routing is assumed to be efficient and the remaining steps are linear in $|E(D)|$.

Projection algorithm (cut). For $0 \leq i < t$, $C \in \mathcal{C}_i$, and the DAG projection $D_i(C)$, define

$$V^{\text{first}}(D_i(C)) := \{u_i^{\text{first}} \in V(D_i(C)) \mid u \in C\}, \quad V^{\text{last}}(D_i(C)) := \{u_i^{\text{last}} \in V(D_i(C)) \mid u \in C\}.$$

We prove the following by induction.

Lemma A.46. *For every $0 \leq i < t$ and every $C \in \mathcal{C}_i$, let S^D be a cut in $D_i(C)$ with finite cut value. Then there exists a set $S_C \subseteq C$ such that*

- $\pi(S^D \cap V^{\text{first}}(D_i(C))) \subseteq S_C \subseteq \pi(S^D \cap V^{\text{last}}(D_i(C)))$,
- $\text{val}_{G_i[C]}(S_C) \leq \text{val}_{D_i(C)}(S^D)$.

Moreover, S_C can be found in $\tilde{O}(|D_i(C)|)$ work and $\tilde{O}(t)$ depth.

Once we have this lemma, the cut-projection algorithm follows immediately: for $i = t$ and $C = V$, we obtain a cut $S_C \subseteq V$ such that

$$\hat{\pi}(S^D) \subseteq \pi(S^D \cap V^{\text{first}}(D_t(V))) \subseteq S_C \subseteq \pi(S^D \cap V^{\text{last}}(D_t(V))) \subseteq \pi(S^D),$$

and

$$\text{val}_G(S_C) \leq \text{val}_D(S^D),$$

as required. (If S^D has infinite value, we can return any valid s - t separating cut in G .) Thus it suffices to prove [Lemma A.46](#).

Proof of Lemma A.46. Base case. For $i = 0$, each $C \in \mathcal{C}_0$ is a singleton, so $D_0(C)$ has one vertex and every cut has value 0. Taking $S_C = C$ or $S_C = \emptyset$ satisfies the statement.

Inductive step. Let $i \geq 1$. The DAG $D_i(C)$ consists of $2\sigma_i$ copies of $\tilde{D}_i(C)$ plus one dummy vertex w_i^C . WLOG assume $w_i^C \notin S^D$; the other case is symmetric (we only mirror the choice of copies).

Let $(Y_1, \dots, Y_z)$ be the subsequence of $\mathcal{C}_{i-1}$ that partitions C . For $x \in [2\sigma_i]$, let $\tilde{D}_i^{(x)}(C)$ denote the x -th copy of $\tilde{D}_i(C)$. Recall that $\tilde{D}_i(C)$ is formed by concatenating $D_{i-1}(Y_1), \dots, D_{i-1}(Y_z)$ in order. Let $D_{i-1}^{(x)}(Y_y)$ denote the copy of $D_{i-1}(Y_y)$ inside $\tilde{D}_i^{(x)}(C)$.

For each $x \in [2\sigma_i]$ and each $y \in [z]$, define

$$S_{x,y}^D := S^D \cap V(D_{i-1}^{(x)}(Y_y)).$$

By the induction hypothesis applied to level $i-1$, there is a set $S_{x,y} \subseteq Y_y$ such that

$$\pi(S_{x,y}^D \cap V^{\text{first}}(D_{i-1}^{(x)}(Y_y))) \subseteq S_{x,y} \subseteq \pi(S_{x,y}^D \cap V^{\text{last}}(D_{i-1}^{(x)}(Y_y))), \quad (\text{A.3})$$

and

$$\text{val}_{G_{i-1}[Y_y]}(S_{x,y}) \leq \text{val}_{D_{i-1}^{(x)}(Y_y)}(S_{x,y}^D). \quad (\text{A.4})$$

Let

$$S_x := \bigcup_{y=1}^z S_{x,y}.$$

We will eventually pick one S_x as S_C . To do that, define

$$x^* = \arg \min_{x \in [2\sigma_i]} \text{val}_{G_i[C]}(S_x) \quad (\text{A.5})$$

where $E_i[C] := E_i \cap E(G_i[C])$. (If $w_i^C \in S^D$, we do the symmetric choice among $x \in \{\sigma_i + 1, \dots, 2\sigma_i\}$.) Set $S_C := S_{x^*}$.

Property 1 (in/out sandwich). We claim that for every $x \in [2\sigma_i]$ and every $y \in [z]$,

$$\pi(S^D \cap V^{\text{first}}(D_{i-1}^{(x)}(Y_y))) \subseteq S_{x,y} \subseteq \pi(S^D \cap V^{\text{last}}(D_{i-1}^{(x)}(Y_y))) \subseteq \pi(S^D \cap V^{\text{first}}(D_{i-1}^{(x+1)}(Y_y))), \quad (\text{A.6})$$

ignoring the last inclusion when $x = 2\sigma_i$. The first two inclusions are exactly [\(A.3\)](#). For the last inclusion: if there were $u \in Y_y$ such that $u_{i-1,(x)}^{\text{last}} \in S^D$ but $u_{i-1,(x+1)}^{\text{first}} \notin S^D$, then the edge

$$(u_{i-1,(x)}^{\text{last}}, u_{i-1,(x+1)}^{\text{first}})$$

of infinite capacity would be in the cut, contradicting the assumption that S^D has finite value. Thus [\(A.6\)](#) holds.

Taking the union over all y and using [\(A.6\)](#) for consecutive copies, we obtain

$$\pi(S^D \cap V^{\text{first}}(D_i(C))) \subseteq S_x \subseteq \pi(S^D \cap V^{\text{last}}(D_i(C)))$$

for every x . In particular, it holds for x^* , so Property 1 is proved.

Property 2 (cut value). We need to compare $\text{val}_{G_i[C]}(S_C)$ to $\text{val}_{D_i(C)}(S^D)$. First we show a per-copy inequality.

Lemma A.47. For every $x \in [2\sigma_i]$,

$$\frac{1}{\sigma_i} \cdot \text{val}_{G_i[C]-E_i}(S_x) \leq \text{val}_{\tilde{D}_i^{(x)}(C)}(S^D \cap V(\tilde{D}_i^{(x)}(C))).$$

Proof. Let

$$E_C^\rightarrow := \{(u, v) \in E(G_i[C]) \mid u \in Y_{k_1}, v \in Y_{k_2}, k_1 < k_2\}$$

be the “forward” edges between different Y ’s (recall $E^{\text{rev}}(\mathcal{C}_{i-1}) \subseteq E_i$, so $G_i[C] - E_i$ only has forward inter- Y edges). Then

$$\text{val}_{G_i[C] - E_i}(S_x) \leq \sum_{y=1}^z \text{val}_{G_{i-1}[Y_y]}(S_{x,y}) + U_G(\{(u, v) \in E_C^\rightarrow \mid u \in S_x, v \notin S_x\}).$$

On the other hand, the cut of S^D in $\tilde{D}_i^{(x)}(C)$ is exactly

$$\sum_{y=1}^z \text{val}_{D_{i-1}^{(x)}(Y_y)}(S_{x,y}) + U_{\tilde{D}_i^{(x)}(C)}(\{(u_{i-1,(x)}^{\text{last}}, v_{i-1,(x)}^{\text{first}}) \mid u_{i-1,(x)}^{\text{last}} \in S^D, v_{i-1,(x)}^{\text{first}} \notin S^D\}).$$

By (A.4), the first part is at least

$$\frac{1}{\sigma_i} \sum_{y=1}^z \text{val}_{G_{i-1}[Y_y]}(S_{x,y}),$$

because capacities in $D_i(C)$ are scaled by $1/\sigma_i$. By (A.3) and the same “if $u \in S_x$ then its out-copy is in S^D , if $v \notin S_x$ then its in-copy is not in S^D ” argument as before, each inter- Y edge in $E_C^\rightarrow(S_x, C \setminus S_x)$ is cut in $\tilde{D}_i^{(x)}(C)$, again up to the $1/\sigma_i$ scaling. Putting these together gives the claim. $\square$

Next, recall that $D_i(C)$ is formed from all $2\sigma_i$ copies plus the cross edges and the dummy edges. Let E_{cross}^D denote the set of edges in $D_i(C)$ that connect *different* copies (including edges adjacent to w_i^C). We relate E_{cross}^D to the $E_i[C]$ part:

Lemma A.48.

$$\sum_{x \in [\sigma_i]} \frac{1}{\sigma_i} \cdot \text{val}_{E_i[C]}(S_x) \leq \text{val}_{E_{\text{cross}}^D}(S^D).$$

Proof. Use Equation (A.6). For each $x \in [\sigma_i]$, partition the edges in $E_i[C](S_x, C \setminus S_x)$ into:

- *Type “Bridge-paid”*: edges (u, v) where $v_{i-1,(x+1)}^{\text{first}} \notin S^D$. Then $u_{i-1,(x)}^{\text{last}} \in S^D$ by Equation (A.6), so the edge $(u_{i-1,(x)}^{\text{last}}, v_{i-1,(x+1)}^{\text{first}})$ (a cross edge) is cut and can pay for this (u, v) .
- *Type “Terminal-paid”*: edges (u, v) where $v_{i-1,(x+1)}^{\text{first}} \in S^D$. Then again by Equation (A.6), $v_{i-1,(\sigma_i)}^{\text{last}} \in S^D$. Since we assumed $w_i^C \notin S^D$, the edge $(v_{i-1,(\sigma_i)}^{\text{last}}, w_i^C)$ is cut and can pay for (u, v) . Moreover, this charging does not double-count edges to w_i^C : once (u, v) is of Type “Terminal-paid” at level x , both endpoints remain inside all later $S_{x'}$, so it cannot be of Type “Terminal-paid” again.

In all cases, we pay for the original capacity only once, and then scale by $1/\sigma_i$ to match the scaled capacity in $D_i(C)$. $\square$

Now we can finish. Let x^* be as in (A.5). Then

$$\begin{aligned} \text{val}_{D_i(C)}(S^D) &= \text{val}_{E_{\text{cross}}^D}(S^D) + \sum_{x \in [2\sigma_i]} \text{val}_{\tilde{D}_i^{(x)}(C)}(S^D \cap V(\tilde{D}_i^{(x)}(C))) \\ &\geq \sum_{x \in [\sigma_i]} \frac{1}{\sigma_i} \text{val}_{E_i[C]}(S_x) + \sum_{x \in [\sigma_i]} \frac{1}{\sigma_i} \text{val}_{G_i[C] - E_i}(S_x) && \text{by Lemmas A.47 and A.48} \\ &= \frac{1}{\sigma_i} \sum_{x \in [\sigma_i]} \text{val}_{G_i[C]}(S_x) \\ &\geq \text{val}_{G_i[C]}(S_{x^*}) = \text{val}_{G_i[C]}(S_C), && \text{as } x^* = \arg \min_{x \in [\sigma_i]} \text{val}_{G_i[C]}(S_x) \end{aligned}$$

The work/depth bound follows because at each level we only need to:

- apply the inductive procedure to $O(1)$ (in fact $O(2\sigma_i)$) sub-DAGs,
- sum/compare the cut values, and

- pick the best x ,

all of which take time subsumed by constructing $D_i(C)$. Over t levels, this gives $\tilde{O}(|D_i(C)|)$ work and $\tilde{O}(t)$ depth. $\square$

A.6.3 Expander Hierarchy via Expander Decomposition

In the next section, we will prove the following expander decomposition lemma.

Lemma A.49. *There is an algorithm that takes as input*

- a directed graph $G = (V, E)$ with edge capacities,
- an edge set $F \subseteq E$,
- a parameter $0 < \phi < 1$,
- an oracle $\mathcal{O}_{\text{MFMC}}$ solving (α, δ) -ApxMFMC,

and outputs an edge set $F' \subseteq F$ and an F' -expander decomposition of G with expansion ϕ and slack γ , together with an associated efficient flow routing algorithm, such that

$$\gamma = \alpha \log^6 n, \quad \delta \leq U_G(F) \cdot \frac{(\phi\gamma)^2}{99 \log^4 n}, \quad U_G(F') \geq (1 - \phi\gamma) \cdot U_G(F),$$

provided $\phi \leq o(1/\gamma)$. The algorithm makes $\tilde{O}(1/(\phi\gamma))$ -efficient calls to $\mathcal{O}_{\text{MFMC}}$ and uses additional $\tilde{O}(|E(G)|/(\phi\gamma))$ work and $\tilde{O}(1/(\phi\gamma))$ depth.

Now we can use [Lemma A.42](#) to get an expander hierarchy. We initialize $E'_1 = E$ and $\mathcal{C}'_0 = (\{v_1\}, \{v_2\}, \dots, \{v_n\})$ where $(v_1, \dots, v_n)$ is an arbitrary order of V . We will show how to start from $E'_i, \mathcal{C}'_i$ to compute $E_i, \mathcal{C}'_{i+1}, E'_{i+1}$. We should think of E'_i as the final $E_{\geq i}$, and $\mathcal{C}'_i$ will be processed in the end to get $\mathcal{C}_i$ to make sure $\mathcal{C}_i$ refines $\mathcal{C}_{i+1}$.

Let

$$\gamma = \alpha \kappa' \log^6 n \quad \phi = 2^{-\log^{0.5} n} / \gamma$$

If the following inequality is satisfied

$$\delta' \leq U_G(E'_i) \cdot (\phi\gamma)^2 / (99 \log^4 n)$$

Then we apply [Lemma A.49](#) on G, E'_i, ϕ to get $E_i \subseteq E'_i$ and a E_i -expander decomposition of G denoted by $\mathcal{C}'_i$ with expansion ϕ and slack γ where

$$U_G(E_i) \geq (1 - \phi\gamma) \cdot U_G(E'_i)$$

This call to [Lemma A.49](#) is valid by checking all the four inequalities described in [Lemma A.49](#).

We let $E'_{i+1} = (E'_i \setminus E_i) \cup E^{\text{rev}}(\mathcal{C}'_i)$.

Otherwise we have

$$\delta' > U_G(E'_i) \cdot (\phi\gamma)^2 / (99 \log^4 n) = U_G(E'_i) \cdot 4^{-\log^{0.5} n} / (99 \log^4 n) \quad (\text{A.7})$$

In this case, we **stop** and let $t = i$ and $E_t = E'_i, \mathcal{C}_t = (V)$. We process $\mathcal{C}'_i$ to get $\mathcal{C}_i$ from $i = t - 1$ to $i = 1$ in the following way (to make sure $\mathcal{C}_j$ refines $\mathcal{C}_{j+1}$): suppose $\mathcal{C}'_i = (C_1, \dots, C_z)$ and $\mathcal{C}_{i+1} = (C'_1, \dots, C'_{z'})$. We define $C_{x,y} = C'_x \cap C_y$. We let $\mathcal{C}_i = (C_{1,1}, C_{1,2}, \dots, C_{1,z}, C_{2,1}, \dots, C_{z',z})$, i.e., order first according to x , then to y . Notice that this step can be done in parallel $\tilde{O}(m)$ work and $\tilde{O}(1)$ step by parallel sorting.

Lemma A.50. $E_1, \dots, E_t$ and $\mathcal{C}_0, \dots, \mathcal{C}_t$ is a valid expander hierarchy with expansion $(\phi_i)_{i \in [t]}$ where $\phi_i = \phi$ for all $i < t$ and

$$U_G(E_t) < \delta' \cdot 2^{3 \log^{0.5} n}$$

It is associated with an efficient projection algorithm.

Proof. The inequality for $U_G(E_t)$ is from [Equation \(A.7\)](#) by taking n to be sufficiently large.

It is clear from the definition of $(\mathcal{C}_i)_{i \in [t]}$ that $\mathcal{C}_i$ refines $\mathcal{C}_{i+1}$ for every $0 \leq i \leq t-1$.

Then we prove that $E^{\text{rev}}(\mathcal{C}_{i-1}) \subseteq E_{\geq i}$ for every $i \in [t]$. According to the definition if E'_i , we always have $E^{\text{rev}}(\mathcal{C}'_{i-1}) \subseteq E'_i$ and $E'_i \subseteq E_{\geq i}$. Moreover, according to the definition of $\mathcal{C}_{i-1}$, we must have $E^{\text{rev}}(\mathcal{C}_{i-1}) \subseteq E^{\text{rev}}(\mathcal{C}'_{i-1}) \cup E^{\text{rev}}(\mathcal{C}_i)$. Expanding repeatedly for $E^{\text{rev}}(\mathcal{C}_i)$ gives us $E^{\text{rev}}(\mathcal{C}_{i-1}) \subseteq \bigcup_{j \geq i} E^{\text{rev}}(\mathcal{C}'_j) \subseteq E_{\geq i}$.

Then we prove the expanding guarantee. According to the definition of E_i -expander decomposition, we have that E_i is $\mathcal{C}'_i$ -constraint ϕ -expanding in G . Notice that $\mathcal{C}_i$ refines $\mathcal{C}'_i$, so any $\mathcal{C}_i$ -constraint demand is also a $\mathcal{C}'_i$ -constraint demand, so E_i is also $\mathcal{C}_i$ -constraint ϕ -expanding in G .

At last, the projection algorithm for each layer of the hierarchy is efficient according to [Lemma A.49](#), combining them gives an efficient projection algorithm for the whole hierarchy. $\square$

Notice that $t = O(\log^{0.5} n)$ since

$$U_G(E'_{i+1}) \leq 2\phi\gamma \cdot U_G(E'_i) = 2 \cdot 2^{-\log^{0.5} n} \cdot U_G(E'_i)$$

and we assume the capacities are polynomially bounded.

A.6.4 Expander Decomposition via MFMC Oracle

In this section, we describe how to compute an expander decomposition given access to a congestion DAG projection. By [Lemma A.56](#), a suitable DAG projection already gives such an oracle, so it is convenient to phrase the lemma directly in terms of an MFMC oracle.

Lemma A.49. *There is an algorithm that takes as input*

- *a directed graph $G = (V, E)$ with edge capacities,*
- *an edge set $F \subseteq E$,*
- *a parameter $0 < \phi < 1$,*
- *an oracle $\mathcal{O}_{\text{MFMC}}$ solving (α, δ) -ApxMFMC,*

and outputs an edge set $F' \subseteq F$ and an F' -expander decomposition of G with expansion ϕ and slack γ , together with an associated efficient flow routing algorithm, such that

$$\gamma = \alpha \log^6 n, \quad \delta \leq U_G(F) \cdot \frac{(\phi\gamma)^2}{99 \log^4 n}, \quad U_G(F') \geq (1 - \phi\gamma) \cdot U_G(F),$$

provided $\phi \leq o(1/\gamma)$. The algorithm makes $\tilde{O}(1/(\phi\gamma))$ -efficient calls to $\mathcal{O}_{\text{MFMC}}$ and uses additional $\tilde{O}(|E(G)|/(\phi\gamma))$ work and $\tilde{O}(1/(\phi\gamma))$ depth.

The algorithm follows the same high-level structure as Section 7 of [\[Ber+25a\]](#) (a directed non-stop cut-matching framework), but we restate the pieces here for completeness.

Definition A.51 (Matching). Let (P, Q) be a partition of V . A set of directed edges $M \subseteq P \times Q$ with capacities $U_M : M \rightarrow \mathbb{R}_{\geq 0}$ is called a (P, Q) -matching. For a demand bound $\mathbf{d} : V \rightarrow \mathbb{R}_{\geq 0}$ and an error $\varepsilon \geq 0$, we say M is $(\mathbf{d}, \varepsilon)$ -perfect if

1. $\text{vol}_M \leq \mathbf{d}$ (i.e. M does not send/receive more than $\mathbf{d}(v)$ out of any vertex), and
2. $\text{vol}_M(V) \geq (1 - \varepsilon) \cdot \mathbf{d}(V)$.

We will use the “non-stop” cut-matching game for directed graphs from [\[FLL25\]](#), which is an extension of the “non-stop” cut-matching game for undirected graphs by [\[RS14; SW19\]](#).

Lemma A.52 ([\[FLL25\]](#)). *Let $\mathbf{d} : V \rightarrow \mathbb{N}$ be polynomially bounded and let $\varepsilon = o(1/\log^2 n)$. Suppose we have an oracle $\mathcal{O}_{\text{match}}$ that, for any partition (P, Q) of V and any $\mathbf{d}' \leq \mathbf{d}$ with $\mathbf{d}'(P) = \mathbf{d}'(Q)$ and $\mathbf{d}'(V) \geq \mathbf{d}(V)/2$, returns a $(\mathbf{d}', \varepsilon)$ -perfect (P, Q) -matching.*

Then there is a randomized algorithm that makes $O(\log^2 n)$ calls to $\mathcal{O}_{\text{match}}$ and $O(m \log^2 n)$ additional work with $\tilde{O}(1)$ depth, and produces $\tilde{\mathbf{d}} \leq \mathbf{d}$ with

$$\tilde{\mathbf{d}}(V) \geq (1 - O(\varepsilon \log^2 n)) \cdot \mathbf{d}(V)$$

such that, with high probability, $\tilde{\mathbf{d}}$ is $\Omega(1)$ -expanding in the graph

$$W := \bigcup_i M_i,$$

where M_i is the matching returned by the i -th call to $\mathcal{O}_{\text{match}}$.

Intuitively, [Lemma A.49](#) will simulate the matching oracle $\mathcal{O}_{\text{match}}$ using the MFMC oracle $\mathcal{O}_{\text{MFMC}}$: each time we need a nearly perfect matching across a cut, we phrase it as a flow instance with capacities restricted to F , call $\mathcal{O}_{\text{MFMC}}$, and either (i) get the matching we wanted, or (ii) discover a cut with small F -capacity and peel it off, charging its capacity to the slack. Repeating this for $O(1/(\phi\gamma))$ rounds preserves almost all of F (the F' guarantee) and yields an F' -expanding decomposition, exactly as in [\[Ber+25a\]](#), but now with the additive loss δ inherited from the MFMC oracle.

Expander Decomposition Algorithm. The algorithm for [Lemma A.49](#) is recursive. We write

$$(F', \mathcal{C}) \leftarrow \text{ED}(G, F, \phi)$$

to denote a subroutine that should satisfy the guarantees of [Lemma A.49](#): namely, $\mathcal{C}$ is an F' -expander decomposition of G with expansion ϕ and slack γ , and $F' \subseteq F$ preserves almost all of F .

We set

$$\mathbf{d} := \text{vol}_F \quad \text{and} \quad \varepsilon := \frac{\phi\gamma}{\log^3 n}.$$

We want to run the cut-matching game of [Lemma A.52](#), so we must implement the matching oracle $\mathcal{O}_{\text{match}}$.

The oracle $\mathcal{O}_{\text{match}}$. An oracle call receives

- a partition (P, Q) of V ,
- a vector $\mathbf{d}' \leq \mathbf{d}$ such that $\mathbf{d}'(P) = \mathbf{d}'(Q)$ and $\mathbf{d}'(V) \geq \mathbf{d}(V)/2$.

We maintain:

- a flow f^* (initially empty), which will accumulate routed demand, and
- a residual demand $\mathbf{d}'' \leq \mathbf{d}'$ (initially $\mathbf{d}'' := \mathbf{d}'$).

We repeat the following “flow-or-cut” step while

$$\mathbf{d}''(V) \geq \varepsilon \cdot \mathbf{d}'(V).$$

Call the MFMC oracle $\mathcal{O}_{\text{MFMC}}$ on G with the bipartite demand

$$(\mathbf{d}''|_P, \mathbf{d}''|_Q),$$

and obtain a flow f and a cut $(S, \bar{S})$ such that

$$\text{val}(S) + \mathbf{d}''(S \cap Q) + \mathbf{d}''(\bar{S} \cap P) \leq \alpha \cdot \text{val}(f) + \delta. \tag{A.8}$$

Define the threshold

$$T := \mathbf{d}''(P) \cdot \frac{\phi\gamma}{9\alpha \log n}.$$

We branch on whether the flow is large.

Case 1: $\text{val}(f) \geq T$. In this case the oracle made good progress routing the current residual demand. We update

$$f^* \leftarrow f^* + f \quad \text{and} \quad \mathbf{d}'' \leftarrow \mathbf{d}'' - \text{Dem}(f),$$

(where we view $\text{Dem}(f)$ as a vector on V , and note that source/sink supports are disjoint). Then we continue the while-loop.

Case 2: $\text{val}(f) < T$. Here the flow is too small; we interpret (A.8) as a certificate of a sparse cut and recurse on both sides.

Define

$$F[S] := \{(u, v) \in F \mid u, v \in S\},$$

$$\widehat{F}[S] := F[S] \cup \{(u, u) \mid \exists v \in \bar{S} \text{ with } (u, v) \in F \text{ or } (v, u) \in F, \text{ and set } U(u, u) := U_G(u, v) \text{ or } U_G(v, u)\},$$

i.e. we add self-loops to preserve the F -volume of vertices of S . Define $\widehat{F}[\bar{S}]$ symmetrically.

Now recurse:

$$(F'_1, \mathcal{C}_1) \leftarrow \text{ED}(G[\bar{S}], \widehat{F}[\bar{S}], \phi) \quad \text{and} \quad (F'_2, \mathcal{C}_2) \leftarrow \text{ED}(G[S], \widehat{F}[S], \phi).$$

We form $F' \subseteq F$ as follows:

- include all edges of F inside S that survived in F'_2 (i.e. corresponding to self-loops of $\widehat{F}[S]$ that were kept),
- include all edges of F inside $\bar{S}$ that survived in F'_1 ,
- for every edge $(u, v) \in F$ with $u \in S, v \in \bar{S}$ (or vice versa), keep (u, v) in F' if the two self-loops of u and v both survived in F'_2 and F'_1 respectively.

Return F' and the vertex-set sequence $(\mathcal{C}_1, \mathcal{C}_2)$ and terminate $\text{ED}(G, F, \phi)$.

If the loop never triggers Case 2. Suppose in the $O(\log^2 n)$ calls required by Lemma A.52 we always land in Case 1. Then the loop ends only because

$$\mathbf{d}''(V) < \varepsilon \cdot \mathbf{d}'(V),$$

and the accumulated f^* routes the demand

$$((\mathbf{d}' - \mathbf{d}'') \mid_P, (\mathbf{d}' - \mathbf{d}'') \mid_Q).$$

We then construct the required $(\mathbf{d}', \varepsilon)$ -perfect (P, Q) -matching by, for every flow path p of f^* , adding the matching edge

$$(\text{Start}(p), \text{End}(p))$$

with capacity equal to the flow on p . This is exactly the matching $\mathcal{O}_{\text{match}}$ must return.

Applying the cut-matching game. We now run the directed cut-matching game of Lemma A.52 using this implementation of $\mathcal{O}_{\text{match}}$. If none of the $O(\log^2 n)$ rounds ever falls into Case 2, then by Lemma A.52 we obtain $\tilde{\mathbf{d}} \leq \mathbf{d}$ with

$$\tilde{\mathbf{d}}(V) \geq (1 - O(\varepsilon \log^2 n)) \mathbf{d}(V),$$

and $\tilde{\mathbf{d}}$ is $\Omega(1)$ -expanding in the union $W := \bigcup_i M_i$ of the matchings we produced. In this situation we define

$$F' := \{(u, v) \in F \mid \tilde{\mathbf{d}}(u) \geq \mathbf{d}(u)/2 \text{ and } \tilde{\mathbf{d}}(v) \geq \mathbf{d}(v)/2\}$$

and return F' and the trivial partition (V) as the F' -expander decomposition.

This matches the guarantees of Lemma A.49.

Correctness (expansion). We prove by induction on the size of the instance passed to $\text{ED}(G, F, \phi)$ that the returned F' is ϕ -expanding and that we get an efficient flow routing algorithm.

When G has a constant number of vertices, the statement is trivial since $1/\phi = \omega(1)$.

Assume now that G has more than one vertex. There are two ways the algorithm can terminate:

1. it finishes the cut-matching game (i.e. every call to $\mathcal{O}_{\text{match}}$ is in Case 1), or

2. it stops early in Case 2 and recurses on S and $\bar{S}$.

First situation (cut-matching game finishes). In this case we obtain $\tilde{\mathbf{d}}$ that is $\Omega(1)$ -expanding in

$$W := \bigcup_i M_i,$$

where M_i is the matching from the i -th call to $\mathcal{O}_{\text{match}}$. Each matching M_i is formed from the flow f_i that $\mathcal{O}_{\text{match}}$ routed in that call. We show that the total congestion of all these flows is small.

Lemma A.53. *The congestion of $\sum_i f_i$ is at most $9/(\phi \log^2 n)$.*

Proof. Each f_i is itself the sum of per-iteration flows f . In Case 1 of $\mathcal{O}_{\text{match}}$ we have

$$\text{val}(f) \geq T = \mathbf{d}''(P) \cdot \frac{\phi\gamma}{9\alpha \log n}.$$

This means in that iteration the residual demand $\mathbf{d}''(V)$ shrinks by at least a $(\phi\gamma)/(9\alpha \log n)$ fraction. We stop when $\mathbf{d}''(V) < \varepsilon \mathbf{d}'(V)$, where

$$\varepsilon = \frac{\phi\gamma}{\log^3 n}.$$

Hence the number of iterations inside *one* call to $\mathcal{O}_{\text{match}}$ is at most

$$\frac{9\alpha \log^2 n}{\phi\gamma} \leq \frac{9}{\phi \log^4 n},$$

using $\gamma = \alpha \log^6 n$. Since the cut-matching game makes $O(\log^2 n)$ calls to $\mathcal{O}_{\text{match}}$, multiplying these two factors gives total congestion at most

$$\frac{9}{\phi \log^2 n}.$$

□

We store all these flows. Now let a demand be given that is $\text{vol}_{F'}$ -respecting and (V) -constrained. By construction of F' , every $(u, v) \in F'$ satisfies

$$\tilde{\mathbf{d}}(u) \geq \mathbf{d}(u)/2 \quad \text{and} \quad \tilde{\mathbf{d}}(v) \geq \mathbf{d}(v)/2,$$

and recall $\mathbf{d} = \text{vol}_F \geq \text{vol}_{F'}$, so the demand is $2\tilde{\mathbf{d}}$ -respecting. Since $\tilde{\mathbf{d}}$ is $\Omega(1)$ -expanding in W , we can apply any standard expander routing on W to route this demand with congestion $O(\log n)$.

Finally, we replace every edge of the routed flow on W by the corresponding flow bundle $\sum_i f_i$ in G by using [Theorem A.9](#), which turns the cumulative edges to source-sink demands and find the corresponding edge representation of flows in $\tilde{O}(m)$ work and $\tilde{O}(1)$ depth.. This multiplies the congestion by at most $9/(\phi \log^2 n)$, so the final congestion is

$$\frac{9}{\phi \log^2 n} \cdot O(\log n) \leq \frac{1}{\phi},$$

as desired. The work is bounded by the work to run the cut-matching game plus linear overhead, and the additional depth is $\tilde{O}(1)$.

Second situation (early cut, Case 2). Here the algorithm returns the union of two recursive calls:

$$(F'_1, \mathcal{C}_1) = \text{ED}(G[\bar{S}], \widehat{F}[\bar{S}], \phi), \quad (F'_2, \mathcal{C}_2) = \text{ED}(G[S], \widehat{F}[S], \phi).$$

By the induction hypothesis,

$$F'_1 \text{ is } \mathcal{C}_1\text{-constrained } \phi\text{-expanding in } G[\bar{S}], \quad F'_2 \text{ is } \mathcal{C}_2\text{-constrained } \phi\text{-expanding in } G[S].$$

By the definition of F' (we keep a cross edge only if both corresponding self-loops survived in the two recursive calls), any $\text{vol}_{F'}$ -respecting $(\mathcal{C}_1, \mathcal{C}_2)$ -constrained demand restricts to a $\text{vol}_{F'_1}$ -respecting demand

on $G[\bar{S}]$ and to a vol_{F_2} -respecting demand on $G[S]$. So we can route separately in the two subgraphs with congestion at most $1/\phi$, and hence F' is $(\mathcal{C}_1, \mathcal{C}_2)$ -constrained ϕ -expanding in G .

Correctness (slack). We now show that the total capacity of reversed edges in the final decomposition is at most $\phi\gamma \cdot U_G(F)$. If we finish in the first situation (full cut-matching game), then $\mathcal{C} = (V)$ and $U_G(E^{\text{rev}}(\mathcal{C})) = 0$, so there is nothing to prove.

So assume we stopped in Case 2 on some cut $(S, \bar{S})$. We use:

Lemma A.54. *The cut S found in Case 2 satisfies*

$$\text{val}(S) \leq \text{vol}_F(S) \cdot \frac{\phi\gamma}{\log n} \quad \text{and} \quad \text{val}(\bar{S}) \leq \text{vol}_F(\bar{S}) \cdot \frac{\phi\gamma}{\log n}.$$

Proof. The condition for Case 2 is $\text{val}(f) \leq T$. From (A.8) we have

$$\begin{aligned} \text{val}(S) &\leq \alpha \cdot \text{val}(f) + \delta \\ &\leq \alpha \cdot \mathbf{d}''(P) \cdot \frac{\phi\gamma}{9\alpha \log n} + U_G(F) \cdot \frac{(\phi\gamma)^2}{99 \log^4 n} \\ &\leq \mathbf{d}''(P) \cdot \frac{\phi\gamma}{4 \log n}, \end{aligned}$$

where the last inequality uses that the additive term is dominated by the main term once we plug in $\varepsilon = \frac{\phi\gamma}{\log^3 n}$ and the lower bound $\mathbf{d}''(P) \geq \varepsilon \mathbf{d}'(P) \geq \varepsilon \mathbf{d}(P)/2$.

On the other hand,

$$\begin{aligned} \text{vol}_F(S) &\geq \mathbf{d}''(S) \\ &\geq \mathbf{d}''(P) - \mathbf{d}''(\bar{S} \cap P) \\ &\geq \mathbf{d}''(P) - (\alpha \text{val}(f) + \delta) \quad (\text{by (A.8)}) \\ &\geq \mathbf{d}''(P) - \mathbf{d}''(P) \cdot \frac{\phi\gamma}{4 \log n} \\ &\geq \mathbf{d}''(P)/2. \end{aligned}$$

Combining the two displays gives

$$\text{val}(S) \leq \text{vol}_F(S) \cdot \frac{\phi\gamma}{\log n}.$$

A symmetric argument, swapping P and Q , gives

$$\text{vol}_F(\bar{S}) \geq \mathbf{d}''(Q)/2 \quad \text{and} \quad \text{val}(\bar{S}) \leq \text{vol}_F(\bar{S}) \cdot \frac{\phi\gamma}{\log n}.$$

□

Therefore, whenever we split on $(S, \bar{S})$, the amount of capacity we “cut off” is at most

$$\min(\text{vol}_F(S), \text{vol}_F(\bar{S})) \cdot \frac{\phi\gamma}{\log n}. \quad (\star)$$

So for the final decomposition $\mathcal{C}$ we have the recursion

$$U_G(E^{\text{rev}}(\mathcal{C})) \leq \min(\text{vol}_F(S), \text{vol}_F(\bar{S})) \cdot \frac{\phi\gamma}{\log n} + U_G(E^{\text{rev}}(\mathcal{C}_1)) + U_G(E^{\text{rev}}(\mathcal{C}_2)),$$

where $\mathcal{C}_1, \mathcal{C}_2$ are the decompositions returned on $G[\bar{S}]$ and $G[S]$. Unrolling this recursion over all cuts in the recursion tree, and noting that each time we lose at most a $\frac{\phi\gamma}{\log n}$ -fraction of the smaller side, we obtain

$$U_G(E^{\text{rev}}(\mathcal{C})) \leq \phi\gamma \cdot U_G(F),$$

as claimed.

Correctness (size of F'). We will show that $U_G(F') \geq (1 - \phi\gamma) \cdot U_G(F)$.

We first show it for the first situation where the cut-matching game finishes.

Lemma A.55. *If the cut-matching game finishes, we have $U_G(F') \geq (1 - O(\phi\gamma/\log n)) \cdot U_G(F)$.*

Proof. We have $\tilde{\mathbf{d}}(V) \geq (1 - O(\varepsilon \log^2 n)) \cdot \mathbf{d}(V)$ according to Lemma A.52. Then we let F' contain all edges $(u, v) \in F$ such that both $\tilde{\mathbf{d}}(u) \geq \mathbf{d}(u)/2, \tilde{\mathbf{d}}(v) \geq \mathbf{d}(v)/2$. In other words, for every edge $(u, v) \in F - F'$, there must exist $x \in \{u, v\}$ such that $\tilde{\mathbf{d}}(x) < \mathbf{d}(x)/2$. We charge $U_G(u, v)$ to x .

Let $K = \{x \in V \mid \tilde{\mathbf{d}}(x) < \mathbf{d}(x)/2\}$. Then

$$\begin{aligned} \sum_{x \in K} \text{vol}_F(x) &\leq 2 \cdot (\text{vol}_F(V) - \text{vol}_{F'}(V)) \\ &\leq O(\varepsilon \log^2 n) \cdot \mathbf{d}(V) \\ &= O(\phi\gamma/\log^2 n) \cdot \mathbf{d}(V), \end{aligned}$$

because $\varepsilon = \phi\gamma/\log^3 n$. Each $x \in K$ can receive at most $\text{vol}_F(x)$ total charge (since we charge edges incident to x), so the total charge, which equals $U_G(F - F')$, is at most

$$O(\phi\gamma/\log^2 n) \cdot \mathbf{d}(V) = O(\phi\gamma/\log^2 n) \cdot U_G(F).$$

This implies

$$U_G(F') \geq (1 - O(\phi\gamma/\log n)) \cdot U_G(F),$$

where we relaxed $1 - O(\phi\gamma/\log^2 n)$ to $1 - O(\phi\gamma/\log n)$. $\square$

Now we consider the second situation where the cut-matching game stops due to Case 2. We get the following recursive inequality

$$U_G(F - F') \leq U_G(\hat{F}[\bar{S}] - F'_1) + U_G(\hat{F}[S] - F'_2),$$

where recall that $\hat{F}[\bar{S}]$ and $\hat{F}[S]$ add self-loops so that $\text{vol}_{\hat{F}[\bar{S}]} = \text{vol}_F \upharpoonright_{\bar{S}}$ and $\text{vol}_{\hat{F}[S]} = \text{vol}_F \upharpoonright_S$. Note that $\hat{F}[\bar{S}] + \hat{F}[S]$ double-counts edges in $F(S, \bar{S})$ (edges of F with one endpoint in S and the other in $\bar{S}$). However, each such edge is double-counted at most once in the recursion, because once we cut along $(S, \bar{S})$, that edge never appears in a deeper subproblem again (the self-loop we use in the subproblems does not create another cross-partition edge).

Thus, unrolling the recursion and using Lemma A.54 (which shows each cut removes only a $\phi\gamma/\log n$ -fraction of the relevant volume), we get

$$U_G(F - F') \leq O(\phi\gamma/\log n) \cdot U_G(F).$$

Finally, by taking n sufficiently large (so the hidden $O(\cdot)$ is at most, say, $1/2$), we get

$$U_G(F') \geq (1 - \phi\gamma) \cdot U_G(F).$$

Complexity. In the first situation (the cut-matching game finishes), the algorithm uses $\tilde{O}(1)$ calls to $\mathcal{O}_{\text{MFC}}$ and $\tilde{O}(m)$ work and $\tilde{O}(1)$ depth.

In the second situation (the algorithm makes recursive calls to $G[S], G[\bar{S}]$), note that from the proof of Lemma A.54 we have

$$\begin{aligned} \text{vol}_F(S) &\geq \mathbf{d}''(P)/2 \geq \varepsilon \cdot \mathbf{d}(P)/4, \\ \text{vol}_F(\bar{S}) &\geq \mathbf{d}''(P)/2 \geq \varepsilon \cdot \mathbf{d}(P)/4. \end{aligned}$$

So each time we recurse, both sides keep at least an ε -fraction (up to constants) of the volume, and the recursion depth is $O(1/\varepsilon)$. Each level of recursion takes $\tilde{O}(m)$ work and $\tilde{O}(1)$ depth in total and makes $\mathcal{O}_{\text{MFC}}$ calls only on induced subgraphs (whose vertex sets are a partition of V), so the calls to $\mathcal{O}_{\text{MFC}}$

are $O(1/\varepsilon)$ -efficient. Hence the total additional work is $\tilde{O}(m/\varepsilon)$ and the depth is $\tilde{O}(1/\varepsilon)$. Plugging in $\varepsilon = \phi\gamma/\log^3 n$ gives the claimed complexity.

A.6.5 MFMC on General Graphs via MFMC on DAGs and DAG Projections

The next lemma shows that such a projection algorithm reduces MFMC on general graphs to MFMC on DAGs, with both a multiplicative and an additive loss that match the DAG oracle.

Lemma A.56. *Let D be a DAG projection of $G = (V, E)$ with an (κ, δ) -congestion-preserving efficient projection algorithm. Let $\mathcal{O}$ be an oracle solving α -ApxMFMC-DAG. Then there is an algorithm solving $(\alpha\kappa, \alpha\delta)$ -ApxMFMC on G with complexity proportional to one efficient call to $\mathcal{O}$ plus the projection algorithm.*

Proof. Build D' from D exactly as in the standard super-source/super-sink reduction: for every $v \in V$, add a vertex s_v and edges (s_v, v') for every $v' \in V(D)$ with $\pi(v') = v$, each with infinite capacity; also add a vertex t_v and edges (v', t_v) for every such v' , each with infinite capacity. Then add a super source s and edges (s, s_v) of capacity $\Delta(v)$ for every $v \in V$, and add a super sink t and edges (t_v, t) of capacity $\nabla(v)$ for every $v \in V$. Let the resulting graph be D' .

Run the oracle $\mathcal{O}$ on D' with source s and sink t , and let it return a flow f' and a cut S' satisfying

$$\text{val}(S') \leq \alpha \cdot \text{val}(f').$$

Restrict f' and S' to D to obtain f^D and S^D . By construction, $\pi(\text{Supp}(\text{Dem}(f^D))) \subseteq V$, so we can invoke the projection algorithm to get a flow f in G and a cut S in G such that

$$\text{Cong}(f) \leq \kappa \cdot \text{Cong}(f^D), \quad \text{Dem}(f) \preceq \pi(\text{Dem}(f^D)), \quad \text{val}(f) \geq \text{val}(f^D) - \delta,$$

and

$$\text{val}(S) \leq \text{val}(S^D), \quad \hat{\pi}(S^D) \setminus \{\perp\} \subseteq S \subseteq \pi(S^D).$$

Since $\text{Cong}(f) \leq \kappa$, the scaled flow f/κ is feasible in G ; we will output f/κ together with S .

Now we relate the cut values. As in the usual argument, for any $v \in V$ for which there exists $v' \in V(D)$ with $v' \notin S^D$, we must have $s_v \notin S'$; otherwise, some infinite-capacity edge (s_v, v') would cross the cut. Thus (s, s_v) contributes $\Delta(v)$ to $\text{val}(S')$. Symmetrically, for any $v \in V$ for which there exists $v' \in V(D)$ with $v' \in S^D$, we must have $t_v \in S'$; otherwise, some infinite-capacity edge (v', t_v) would cross the cut, and so (t_v, t) contributes $\nabla(v)$ to $\text{val}(S')$. All other contributions to $\text{val}(S')$ come from edges of D , i.e.

$$\text{val}(S^D) \leq \text{val}(S') - \sum_{v \notin \hat{\pi}(S^D)} \Delta(v) - \sum_{v \in \pi(S^D)} \nabla(v).$$

We also have $\text{val}(f^D) = \text{val}(f')$ (restriction does not reduce the flow through s, t in D'), and the projection algorithm gives $\text{val}(f) \geq \text{val}(f') - \delta$. Using $\text{val}(S') \leq \alpha \text{val}(f')$, we obtain

$$\text{val}(S^D) + \sum_{v \in \pi(S^D)} \nabla(v) + \sum_{v \notin \hat{\pi}(S^D)} \Delta(v) \leq \alpha(\text{val}(f) + \delta).$$

Since

$$\hat{\pi}(S^D) \setminus \{\perp\} \subseteq S \subseteq \pi(S^D),$$

we can replace the sums over $\pi(S^D)$ and over $V \setminus \hat{\pi}(S^D)$ by sums over S and $V \setminus S$, respectively, and also use $\text{val}(S) \leq \text{val}(S^D)$, to get

$$\text{val}(S) + \sum_{v \in S} \nabla(v) + \sum_{v \notin S} \Delta(v) \leq \alpha(\text{val}(f) + \delta).$$

Finally, we output the feasible flow f/κ and the cut S . Multiplying the right-hand side by κ to account for

scaling gives

$$\text{val}(S) + \sum_{v \in S} \nabla(v) + \sum_{v \notin S} \Delta(v) \leq \alpha\kappa \cdot \text{val}(f/\kappa) + \alpha\delta,$$

so $(f/\kappa, S)$ is an $(\alpha\kappa, \alpha\delta)$ -ApMFC pair. $\square$

A.6.6 Completing the Spiral

In this section, we will combine everything in the previous sections and prove [Theorem A.26](#).

Theorem A.26. *Suppose there is an oracle for α -ApMFC-DAG where $\alpha = n^{o(1)}$. Then there is a randomized algorithm that, given a directed graph $G = (V, E)$ with edge capacities, outputs a DAG projection D of G together with an efficient $n^{o(1)}$ -congestion-preserving projection algorithm, such that $|E(D)| = \hat{O}(|E(G)|)$ (and D 's topological order is returned). The algorithm makes $n^{o(1)}$ work-efficient and $n^{o(1)}$ depth-efficient calls to α -ApMFC-DAG.*

The algorithm is recursive. We will describe an algorithm $\text{CongDAGProj}(G, \kappa, \delta)$ computing a (κ, δ) -congestion preserving DAG projection for G . It suffices to call $\text{CongDAGProj}(G, \kappa^*, 1/n^C)$ for some $\kappa^* = n^{o(1)}$ to be fixed and sufficiently large constant C to get the required algorithm for [Theorem A.26](#) (we will argue in the end why $1/n^C$ suffices). Now we describe $\text{CongDAGProj}(G, \kappa, \delta)$.

Base Case. Suppose $\delta \geq 2 \cdot U_G(E(G))$, then we return $D = (V_1 \cup \{w\} \cup V_2, E_1 \cup E_2)$ where V_1, V_2 are copies of $V(G)$ with a natural trivial projection map and

$$E_1 = \{(v, w) \text{ with } U(v, w) = \text{vol}_{E(G)}(\pi(v)) \mid v \in V_1\}$$

$$E_2 = \{(w, v) \text{ with } U(w, v) = \text{vol}_{E(G)}(\pi(v)) \mid v \in V_2\}$$

Clearly D is a DAG.

The projection algorithm takes a flow in D , and returns an empty flow in G . Since $\delta \geq 2 \cdot U_G(E(G))$, this is a valid projection algorithm.

Suppose the projection algorithm is given a cut S^D in D , if $w \notin S^D$, then return $S = \pi(S^D \cap V_1)$ as a cut in G ; Otherwise return $S = \pi(S^D \cap V_2)$. Clearly we have $\hat{\pi}(S^D) \subseteq S \subseteq \pi(S^D)$. Moreover, according to the definition of $E_1 \cup E_2$, every edge in $\delta_G^+(S)$ in case $w \in S^D$ (or $\delta_G^-(\bar{S})$ in case $w \notin S^D$) has its capacity subsumed by the corresponding part in $E_1 \cup E_2$, so this is a valid projection algorithm.

For the rest of the algorithm, we suppose $\delta < 2 \cdot U_G(E(G))$ and try to solve $\text{CongDAGProj}(G, \kappa, \delta)$ using recursive calls to $\text{CongDAGProj}(G', \kappa', \delta')$ for some G' created by the recursive algorithm and $\kappa' < \kappa$ and $\delta' > \delta$.

The oracle $\mathcal{O}_{\text{MFC}}$. We are intended to use [Lemma A.42](#) but we need a $\mathcal{O}_{\text{MFC}}$ oracle. In this paragraph, we will describe how to get the oracle $\mathcal{O}_{\text{MFC}}$ by using $\text{CongDAGProj}(G', \kappa', \delta')$. We apply [Lemma A.56](#) on the DAG projection output by $\text{CongDAGProj}(G', \kappa', \delta')$. According to [Lemma A.56](#), we get the oracle $\mathcal{O}_{\text{MFC}}$ that solves $(\alpha\kappa', \alpha\delta')$ -ApMFC-DAG with one efficient call to α -ApMFC-DAG.

The expander hierarchy. We use [Lemma A.50](#) to get the expander hierarchy.

Getting the DAG projection. Let $\eta = 1/\sqrt{\log_n \alpha}$. Notice that if $\alpha = n^{o(1)}$, then $\eta = \omega(1)$ and $\alpha^\eta = n^{o(1)}$.

We use [Lemma A.42](#) on the expander hierarchy and parameter

$$\sigma = 2^{2 \cdot \log^{0.7} n} \cdot \alpha^\eta$$

to get a DAG projection D of G with an efficient (κ, δ) -congestion-preserving projection algorithm such that

$$\kappa = O(2^t / \phi) = 2^{O(\log^{0.5} n)} \cdot \alpha\kappa'$$

$$\delta = U_G(E_t)/\sigma = 2^{3 \log^{0.5} n} \cdot \delta'/\sigma \leq \frac{\delta'}{2^{\log^{0.7} n} \cdot \alpha^\eta} \quad (\text{A.9})$$

Congestion analysis. We will run $\text{CongDAGProj}(\kappa^*, 1/n^C)$ for sufficiently large constant C .

According to Equation (A.9), the recursion depth d for $\text{CongDAGProj}(\kappa^*, 1/n^C)$ is

$$d = \frac{O(\log n)}{\log(2^{\log^{0.7} n} \cdot \alpha^\eta)}$$

After depth d , the base case should have $\kappa_0 \geq 1$ and $\delta \geq 2U_G(E(G))$. Thus, we get that

$$\kappa^* = \left(2^{O(\log^{0.5} n)} \cdot \alpha\right)^d \leq 2^{\frac{O(\log^{1.5} n)}{\log^{0.7} n}} \cdot \alpha^{\frac{O(\log n)}{\eta \log \alpha}} \leq n^{o(1)} \quad (\text{A.10})$$

as required.

Removing Tiny Additive Error. Lastly, according to the construction of the DAG projection Lemma A.42, if the input graph G has integer capacity (which is the assumption), the DAG projection D cannot have edges with capacity less than $1/n$ (as the only scaling down capacity part scales down by $\sigma = n^{o(1)}$). Thus, any flow in D can be scaled up to value at least $1/n$ and apply the projection algorithm with additive error $1/n^C$, which is subsumed by the multiplicative error to the original graph. This gives a κ^* -congestion preserving DAG projection without additive error.

Size of the DAG projection. According to Lemma A.42, the size of the DAG projection can always be upper bounded by

$$O(2^t \sigma |E(G)|) = 2^{O(\log^{0.7} n)} \cdot \alpha^\eta \cdot |E(G)| = n^{o(1)} \cdot |E(G)|$$

as required.

Complexity analysis. Notice that G is changing during the recursive calls. However, according to Lemma A.49, each recursive level only boost the total graph size by a factor of $\tilde{O}(1/(\phi\gamma)) = O(2^{\log^{0.5} n})$. Thus, the total graph size among all recursive calls is upper bounded by $n^{o(1)}$ according to the same calculation as in Equation (A.10).

The oracle calls to the α -ApMFC-DAG algorithm is only by Lemma A.56, which has input size proportional to the DAG projection size, upper bounded by $n^{o(1)}$. So the oracle calls are $n^{o(1)}$ -efficient. The additional work and depth according to Lemmas A.42, A.49 and A.56 is at most $m^{1+o(1)}$ and $n^{o(1)}$, as required.

Paper B

Combinatorial Minimum Cost Flow in Almost-Linear Time on Dense Graphs

Bernhard Haeupler, Yonggang Jiang, Thatchaphol Saranurak

In submission.

Abstract

We give a combinatorial algorithm that computes an exact minimum-cost flow in $n^{2+o(1)}$ time. This is the first combinatorial improvement since Goldberg–Tarjan [GT87], and is almost optimal in dense graphs. All previous faster algorithms, including the almost-linear-time algorithm of Chen et al. [Che+25], rely on interior-point methods.

Our main conceptual contribution is a black-box reduction from exact minimum-cost flow on general graphs to its *approximation* on DAGs, with only an $n^{o(1)}$ loss on the running time. This reduction allows future work on combinatorial minimum-cost flow to focus on DAGs, typically much easier to handle than general graphs. Our $n^{2+o(1)}$ -time algorithm follows by combining this reduction with a simple augmenting-path-based approximation algorithm on DAGs.

The reduction is based on a new structural result, called a *flow-cost-preserving DAG projection*. Given an arbitrary directed graph, this construction produces a DAG that approximately preserves both the congestion of flows and their cost up to a small loss.

B.1 Introduction

Minimum-cost flow (MCF) is one of the most fundamental problems in combinatorial optimization. Given a directed graph $G = (V, E)$ with n vertices and m edges, capacities $U \in \mathbb{Z}_{>0}^E$, costs $\ell \in \mathbb{Z}^E$, and a demand vector $D \in \mathbb{R}^V$ satisfying $\sum_{v \in V} D(v) = 0$, the goal is to find a flow $f \in \mathbb{R}^E$ that respects capacities, routes D , and minimizes the total cost $\sum_{e \in E} \ell(e)f(e)$. MCF captures shortest paths, transshipment, bipartite matching, circulation with demands, and many other basic primitives.

Fast algorithms for MCF have been studied since the 1950s; see, e.g., [Dan51; Ful61; Kle67; EK72; Tar85; Or188]. These classical algorithms share a common combinatorial principle: they repeatedly improve the current solution by augmenting flow along a negative cycle or a shortest augmenting path in the *residual graph*. This line culminated in push-relabel-based algorithms [GT87; GT90; Ahu+92] with $\tilde{O}(mn)$ running time.¹

Later work improved this long-standing $\tilde{O}(mn)$ bound via interior-point methods (IPMs) [DS08; LS19; Bra+21; AMV22] and culminated in almost-linear $m^{1+o(1)}$ -time algorithms [Che+25; Van+24]. These algorithms achieve spectacular running times, but they do so by bypassing the classical directed residual graph: their progress is described through barrier functions and the current position in the polytope, and the key subroutines are on *undirected* graphs.

This leaves a striking gap in our understanding of minimum-cost flow. The fastest exact algorithms are almost-linear, but the residual-graph-based approach for exact MCF appeared stuck at the classical $\tilde{O}(mn)$ bound. This raises a natural question: can one beat the classical $\tilde{O}(mn)$ bound for exact minimum-cost flow while staying within the residual-graph paradigm?

Very recent work showed that this barrier can be broken for closely related special cases while staying entirely within the directed residual graph. For bipartite matching [CK24b; CK24a] and maximum flow [Ber+24; Ber+25a], new residual-graph-based algorithms achieved $\tilde{O}(n^2)$ running time, which is near-optimal in dense graphs. Besides the running-time improvement, working with residual graphs leads to new primitives native to directed graphs. For example, [Ber+24] gives a simple and teachable approximate maximum flow algorithm on DAGs. Directed expander hierarchies introduced in [Ber+24; Ber+25a] found applications to directed graph sparsifiers [HSW25], global minimum cut and arborescence packing [Jia+25a], and black-box reductions for maximum flow to approximation on DAGs [HJS26a]. Unfortunately, these recent algorithms do not handle edge costs.

Thus, for exact minimum-cost flow, the fastest residual-graph-based algorithm has remained the $\tilde{O}(mn)$ -time algorithm of Goldberg and Tarjan [GT87] for nearly forty years.

Our result: the first combinatorial improvement since Goldberg–Tarjan. We answer the question above affirmatively: within the residual-graph paradigm, we give the first improvement for exact minimum-cost flow since [GT87].

Theorem B.1. *There is a combinatorial algorithm computing a minimum-cost flow in $n^{2+o(1)}$ time.*

In particular, **Theorem B.1** is almost optimal in dense graphs. Our algorithm is randomized and correct with high probability.

Our approach: exact MCF via weak approximation on DAGs. The conceptual core of the paper is a reduction from exact MCF on general directed graphs to a very weak approximate primal-dual version of MCF on DAGs. Let f be a flow and $(\phi, \tilde{\ell})$ be a dual solution. We say that $(f, \phi, \tilde{\ell})$ is a (κ, λ) -*approximate PD-MCF solution* for demand D if f routes D with congestion at most κ , and the cost of f is at most λ times the value of the dual solution $(\phi, \tilde{\ell})$; see **Section B.3.1** for the formal definition.

We prove **Theorem B.1** in three steps.

1. **Exact-to-Approximate Reduction.** We reduce the problem of computing an exact MCF solution to $n^{o(1)}$ calls to an oracle for computing $(n^{o(1)}, 1 + 1/n^{o(1)})$ -approximate PD-MCF. Thus, exactness

¹We use $\tilde{O}$ to hide $\text{polylog}(n, U_{\max}, \ell_{\max})$ factors, where $U_{\max} = \max_e U(e)$ and $\ell_{\max} = \max_e |\ell(e)|$.

can be bootstrapped from a surprisingly weak approximate solution. This guarantee is much stronger than that of the successive approximation framework in [GT90].

2. **Flow-Cost-Preserving DAG Projection.** We further reduce $(n^{o(1)}, 1 + 1/n^{o(1)})$ -approximate PD-MCF on general directed graphs to the same problem on DAGs. Informally, a flow-cost-preserving DAG projection replaces a general directed graph by a DAG while preserving both the congestion and the cost of flows up to a small loss. This DAG generalizes the recent DAG projections in [HJS26a] that preserve only congestion or only costs, and is our key structural contribution.
3. **Weighted Push-Relabel for MCF on DAGs.** We give a simple augmenting-path-based algorithm for computing approximate PD-MCF on DAGs in $n^{2+o(1)}$ time, generalizing a similar weighted push-relabel algorithm in [Ber+24] for the maximum flow problem, a special case of the minimum-cost flow problem.

Combining these three ingredients yields [Theorem B.1](#). More importantly, the first two ingredients already give a black-box reduction from exact MCF on general graphs to approximate MCF on DAGs.

Theorem B.2. *There is an algorithm computing a minimum-cost flow in time complexity proportional to making oracle calls for computing $(n^{o(1)}, 1 + 1/n^{o(1)})$ -approximate PD-MCF on DAGs with cumulative total size $m^{1+o(1)}$ and cumulative total number of vertices $n^{1+o(1)}$.*

[Theorem B.2](#) is the conceptual core of our framework, and is useful beyond merely implying our main result, [Theorem B.1](#).

1. First, it isolates the main difficulty of exact MCF on general directed graphs into weak approximation on DAGs. Note that a very weak $n^{o(1)}$ approximation on congestion is allowed. Thus, future work on *combinatorial* minimum-cost flow can focus on the substantially simpler setting of approximation on DAGs.
2. Second, it is quite plausible to strengthen [Theorem B.2](#) to work in the deterministic and/or parallel setting. This is because our two main ingredients, DAG projections and directed expander hierarchies, are naturally compatible with these settings, as shown in [HJS25; HJS26a]. Although we do not pursue these directions here, this suggests a promising route toward deterministic and parallel algorithms for minimum-cost flow, again by reducing attention to DAGs.

B.2 Overview

In this section, we sketch the three ingredients behind our $n^{2+o(1)}$ combinatorial algorithm for minimum-cost flow:

1. in [Section B.2.1](#), we reduce exact MCF to the approximate primal-dual version of MCF;
2. in [Section B.2.2](#), we reduce this approximate problem on general directed graphs to the same problem on DAGs via a *flow-cost-preserving DAG projection*;
3. in [Section B.2.3](#), we give a simple augmenting-path-based algorithm for approximate PD-MCF on DAGs.

For a directed graph $G = (V, E)$ and an edge $e \in E$, we write $\ell_G(e)$ for its length and $U_G(e)$ for its capacity. For a vertex $v \in V$, we write $\deg_G(v) \stackrel{\text{def}}{=} \sum_{e \in e} U_G(e)$ for its capacity-weighted degree. When the graph is clear from context, we omit the subscript G . For a flow f , we write its value $\text{val}(f)$ as the amount of flow it sends, and $\text{cost}(f)$ as the cost of the flow defined as $\sum_{e \in E} f(e) \cdot \ell(e)$, i.e., we call $\ell(e)$ both the *length* of an edge and also the *cost* of the edge.²

The demands D in this section are all (single-commodity) demands, where each vertex gets a value: positive means flow should leave this node, and negative means flow should enter this node. A flow routing a demand is defined in the natural way. We write $|D| = \sum_{v \in V, D(v) > 0} D(v)$.

²Although mixing cost and length might be confusing, if one considers minimum cost flow as a combination of shortest path and maximum flow, then the notation of ‘length’ makes sense. It also combines many recent development of length bounded flows.

We say that an algorithm makes g -efficient oracle calls to an oracle $\mathcal{O}$ if:

- the total number of vertices over all instances given to $\mathcal{O}$ is at most $g \cdot n$,
- the total number of edges over all instances given to $\mathcal{O}$ is at most $g \cdot m$, and
- the additional running time outside the oracle calls is at most $g \cdot m$.

B.2.1 Exact-to-Approximation Reduction

Throughout this section, all MCF instances are assumed to have non-negative edge lengths. For *exact* MCF, this is without loss of generality: if an edge (u, v) has negative length, we may reverse it to (v, u) , negate its length, and modify the demands by adding $U(u, v)$ to v and $-U(u, v)$ to u . This folklore reduction transforms the instance into one with non-negative lengths. For multiplicative approximation problems, non-negative lengths are the natural setting.

We first explain how to reduce exact MCF to the approximate primal-dual version of MCF. The full proof is given in [Section B.4](#).

Theorem B.3. *For $\varepsilon = o(1/\log^9 n)$, there is an algorithm solving exact minimum-cost flow using*

$$\tilde{O}\left(n^{O(1/\log(1/\varepsilon))} \cdot \kappa^2 / \varepsilon^3\right)$$

-efficient calls to a $(\kappa, 1 + \varepsilon)$ -approximate PD-MCF oracle.

By setting $\kappa = n^{o(1)}$ and $\varepsilon = 1/n^{o(1)}$, [Theorem B.3](#) implies that exact MCF can be solved using only $n^{o(1)}$ calls to a $(\kappa, 1 + \varepsilon)$ -approximate PD-MCF oracle.

Comparison with successive approximation. The closest prior exact-to-approximate framework for minimum-cost flow is the classical successive-approximation / cost-scaling framework of Goldberg and Tarjan [GT90]. However, as far as we understand, that framework is not a black-box reduction to an arbitrary approximate MCF oracle. Instead, it relies on repeatedly making the most negative reduced-cost edge less negative, from $-\varepsilon$ to $-\varepsilon/2$, over $O(\log n)$ phases.

This is not suitable for our purposes, since our oracle does not provide any direct control over the most negative cost edge. In contrast, [Theorem B.3](#) gives a genuinely black-box reduction from exact minimum-cost flow to a much weaker approximate primal-dual oracle. To the best of our knowledge, this is the first black-box reduction of this flavor for minimum-cost flow. In particular, it can be viewed as the minimum-cost-flow analogue of exact-to-approximate boosting phenomena that are by now standard for problems such as shortest paths and maximum flow. Moreover, our reduction unifies several natural approximate formulations of minimum-cost flow by showing that they are equivalent for the purpose of computing an exact solution.

Two intermediate problems. To prove [Theorem B.3](#), we introduce two intermediate problems.

$(\kappa, 1 + \varepsilon)$ -strongly-approximate PD-MCF. Given a demand D , output a $(1 \pm \varepsilon)$ -approximate primal-dual pair of the MCF LP $(f, \phi, \tilde{\ell})$, where the primal flow routes a sub-demand $D' \preceq D$ with $|D'| \geq |D|/\kappa$ and has no congestion. The MCF LP is defined in [Section B.3.1](#). It tries to route a fixed demand with a minimum cost.³

$(1 - \varepsilon)$ -approximate PD-MBCF. Given a cost budget B , output a $(1 \pm \varepsilon)$ -approximate primal-dual pair of the maximum bounded-cost flow (MBCF) LP. The MBCF LP can be found in [Section B.4.2](#). It outputs the approximate maximum flow given a bounded cost.

The proof of [Theorem B.3](#) is obtained by combining the following three reductions.

³This notion is stronger than $(\kappa, 1 + \varepsilon)$ -approximate PD-MCF, since by enlarging the primal flow by a factor of κ , we get a flow with congestion at most κ and cost at most $(1 + \varepsilon)$ times the optimal, as required by the $(\kappa, 1 + \varepsilon)$ -approximate PD-MCF. The opposite direction generally does not hold.

Step 1: exact PD-MCF to $(\kappa, 1 + \varepsilon)$ -strongly-approximate PD-MCF. We begin with the warm-up case $\kappa = 1$, namely an oracle that routes *all* of the demand and returns an approximate primal-dual pair for the MCF LP. Suppose the oracle returns a flow f together with a dual pair $(\phi, \tilde{\ell})$. We define a residual instance

$$(G_{\text{res}}, D_{\text{res}}) = (G, D)_{f, \phi}$$

in the following way.

Let G_f be the residual graph of G with respect to f . Let $(G_f)_\phi$ be the graph obtained by reweighting residual edges using the potential ϕ , i.e., for every residual edge (u, v) ,

$$\ell_{(G_f)_\phi}(u, v) = \ell_G(u, v) - (\phi(v) - \phi(u)).$$

Let E_{neg} be the set of negative-length edges in $(G_f)_\phi$. We define G_{res} by reversing every edge in E_{neg} and negating its length, so that all edge lengths in G_{res} are non-negative. The demand D_{res} is chosen so that each reversed edge contributes the appropriate amount of source and sink demand.

We first show a simple lemma saying that that solving this residual instance exactly lets us repair the current approximate solution.

Lemma B.4. *Suppose $f, \phi, \tilde{\ell}$ is a feasible primal-dual pair for the minimum-cost flow problem (G, D) , and f_{res}^* is a minimum-cost flow of $(G, D)_{f, \phi}$. Then $f + f_{\text{res}}^*$ is a minimum-cost flow of (G, D) .*

Proof. Since f_{res}^* is a minimum-cost flow of $(G, D)_{f, \phi}$, the residual graph of $f + f_{\text{res}}^*$ contains no negative cycle. Moreover, $f + f_{\text{res}}^*$ is a feasible flow routing D . Hence it is optimal. $\square$

The second key lemma is that if $(f, \phi, \tilde{\ell})$ is a good approximate primal-dual pair, then the residual instance has small optimum.

Lemma B.5. *Suppose $f, \phi, \tilde{\ell}$ is a feasible primal-dual pair of (G, D) with primal-dual value difference Δ . Then the optimal cost of the minimum-cost flow instance $(G, D)_{f, \phi}$ is at most Δ .*

The proof is by a careful calculation of the sum over all negative residual edges, and comparing it with the primal-dual difference. The full calculation appears in [Lemma B.14](#).

Thus, if the oracle returns a $(1 \pm \varepsilon)$ -approximate primal-dual pair with, say, $\varepsilon = 0.1$, then the residual instance has only a constant fraction of the original optimum cost. Repeating this for $O(\log n)$ rounds and applying [Lemma B.4](#) yields an exact algorithm.

We next extend this to general κ . Now the oracle may route only a sub-demand $D' \preceq D$ with $|D'| \geq |D|/\kappa$. We call the oracle repeatedly. In the i -th round it returns a flow f_i and dual $(\phi_i, \tilde{\ell}_i)$ on the remaining demand. After each round we form the corresponding residual graph, reweight by ϕ_i , and simply round every negative edge length up to 0 before making the next call (because we assume the oracle can only work on graphs with non-negative lengths). Since each call routes at least a $1/\kappa$ fraction of the remaining demand, after $O(\kappa \log n)$ calls the total routed flow already covers all of D .

The subtle point is to show that these repeated roundings do not destroy the reduction. Intuitively, each round contributes only an $O(\varepsilon)$ fraction of its own routed cost to the final residual error, so the total accumulated rounding loss remains small. This is formalized in [Lemma B.15](#).

Step 2: $(\kappa, 1 + \varepsilon)$ -strongly-approximate PD-MCF from $(1 - \varepsilon)$ -approximate PD-MBCF. We now explain how to use an approximate bounded-cost-flow oracle to obtain a strongly-approximate PD-MCF solution.

Fix a demand D and, for simplicity, assume it is an s - t demand of total value 1. Suppose we are given an oracle for maximum bounded-cost flow: for any budget B , it returns an approximately maximum s - t flow of cost at most B , together with an approximate dual certificate.

A first idea is to perform binary searches over the budget $B_i = (1 + \varepsilon)^i$ and find the first budget at which the oracle routes a constant fraction of the demand, say at least $1/2$. However, this fails if the cost-value tradeoff has a sharp jump: it may happen that routing value μ costs B , while routing value $(1 + \varepsilon)\mu$ already

costs $100B$. In that case, the first budget that certifies a value of at least μ is not necessarily close to the true cost of routing μ . The bad case is when the budget grows a lot but the flow value grows only slightly.

Our way around this is to look for a *good scale* where the routed value grows noticeably while the budget only increases by a $(1 + \alpha)$ factor. We query the MBCF oracle on a geometric sequence of budgets

$$B_i = (1 + \alpha)^{i-2}, \quad \text{where } \alpha = \varepsilon^{1/3},$$

and let q_i be the amount of flow returned at budget B_i . The “good point” we need is: the budget only changes from B_a to $B_{a+1} = (1 + \alpha)B_a$, but the routed flow value jumps by a multiplicative $(1 + \beta)$ factor. We will set $\alpha \gg \beta$, so that if all the points are not good, then the cost will increase by too much (controlled by $(1 + \alpha)$) until reaching the trivial upper bound $\sum_{e \in E} \ell(e) \cdot U(e)$, which we assume to be $\text{poly}(n)$, while the flow value only increase by tiny (not exceeding 1). This is a contradiction since we can definitely route 1 unit of flow with some cost below $\sum_{e \in E} \ell(e) \cdot U(e)$. So there must be a good point, and we can also use binary search to find that. The detailed proof is given in [Lemma B.20](#).

Step 3: $(1 - \varepsilon)$ -approximate PD-MBCF from $(\kappa, 1 + \varepsilon)$ -approximate PD-MCF. The final step is based on the flow-boosting template of [[Hae+24](#), Theorem 9.2], which uses multiplicative weight updates (MWU) to reduce bounded-cost flow to repeated calls to an approximate PD-MCF oracle.

The main extra work on our side is to recover a suitable dual solution for the bounded-cost-flow LP. The dual involves the concept of *moving cut*, which serves as a continuous extension of the normal cut in a graph. The goal of the dual is to block every path from the sources to sinks using a small moving cut.

The MWU template already constructs a moving cut that blocks a large fraction of the relevant paths, but not necessarily all of them. We show that the remaining uncovered part is small: it accounts for only an $O(\varepsilon)$ fraction of the optimum. Therefore, one additional $(\kappa, 1 + \varepsilon)$ -approximate PD-MCF call (which actually gives a $O(\kappa)$ -approximate optimal moving cut) suffices to cover the remaining part, at a cost that can be absorbed into the approximation factor by choosing ε sufficiently small (much smaller than $1/\kappa$). This yields the desired primal-dual pair for bounded-cost flow. The detailed proof can be found in [Section B.4.2](#).

B.2.2 Flow-Cost-Preserving DAG Projection

We next explain how to reduce approximate PD-MCF on a general directed graph to approximate PD-MCF on a DAG. This is the main structural contribution of the paper.

For convenience, write $\mathcal{O}_{\kappa, \lambda}^{\text{PDMCF}}$ for an oracle that computes a (κ, λ) -approximate PD-MCF solution. A *flow-cost-preserving DAG projection* of a directed graph G is defined as follows.

Definition B.6. A flow-cost-preserving DAG projection of $G = (V, E)$ is a DAG G' along with a projection map $\pi : V(G') \rightarrow V$. We call it (κ, λ) -flow-cost-preserving DAG projection, or say it has congestion slack κ and length slack λ if

1. For every demand D' routable in G' by a feasible flow with cost B , $\pi(D')$ is routable in G with congestion κ and cost B and,
2. For every demand D routable in G by a feasible flow with cost B , $\hat{\pi}^{-1}(D)$ is routable in G' by a feasible flow with cost at most $\lambda \cdot B$.

$\pi(D')$ is defined in the natural way: $\pi(D')(v) = \sum_{\pi(v')=v} D'(v')$. $\hat{\pi}^{-1}(D)$ is a bit trickier: we fix a topological order of G' , if $D(v) > 0$, then we choose the first vertex v' in the topological order with $\pi(v') = v$ and let $\hat{\pi}^{-1}(D)(v') = D(v)$; if $D(v) < 0$, then we choose the last vertex v' in the topological order with $\pi(v') = v$ and let $\hat{\pi}^{-1}(D)(v') = D(v)$; otherwise we let $\hat{\pi}^{-1}(D)(v') = 0$. i.e., we lift the demand to be a demand on the DAG from the first copy of sources to the last copy of sinks.

The following lemma shows the power of DAG projection.

Lemma B.7 (Informal version of [Lemma B.24](#)). *Let G be a directed graph, and suppose G' is a (κ', λ') -flow-cost-preserving DAG projection of G . Then one call to $\mathcal{O}_{\kappa, \lambda}^{\text{PDMCF}}$ on G' can be transformed into a solution to $\mathcal{O}_{\kappa \kappa', \lambda \lambda'}^{\text{PDMCF}}$ on G .*

The formal statement additionally requires a projection algorithm that maps primal and dual objects between G and G' . For the overview, the key intuition is simple: if G' approximately preserves all relevant flows and their costs, then solving PD-MCF on G' is essentially as good as solving it on G .

We now sketch how such a DAG projection is constructed.

Existence

The construction relies on two primitives: a length-constrained expander hierarchy, and a bounded-width path cover.

Primitive 1: LC expander hierarchy. A *moving cut* C assigns a non-negative value to every edge. Each moving cut is associated with a length parameter denoted by $\text{len}(C)$. When we view C as an h -length moving cut, we denote graph $G - C$ by the graph increasing the length of each edge e in G by $C(e) \cdot h$.

For such a cut C , define $\text{vol}_C(v) \stackrel{\text{def}}{=} \sum_{e \in E} C(e) \cdot U(e)$.

We say that a demand D is *vol $_C$ -respecting* if $|D(v)| \leq \text{vol}_C(v)$ for every vertex v .

We say that a demand is *h -length symmetrically constrained* if it can be decomposed into pieces, each supported on a strongly connected subgraph of diameter at most h .⁴

An (h, s) -length ϕ -expander with respect to vol_C is a graph in which every vol_C -respecting h -length symmetrically constrained demand can be routed with congestion at most $1/\phi$ and average cost at most hs . Here s is the *length slack*.

A ϕ -expander hierarchy with length slack s is a sequence of moving cuts $C_1, \dots, C_t$ such that for every scale h and every level i , if we interpret $C_{>i} \stackrel{\text{def}}{=} \sum_{j>i} C_j$ as an h -length moving cut, then $G - C_{>i}$ is an (h, s) -length ϕ -expander with respect to vol_{C_i} .

We require $C_1(e) = 1$ for all $e \in E$, and in our setting we have $s = n^{o(1)}$ and $\phi = 1/n^{o(1)}$. Such hierarchies are known to exist, and in [Section B.6](#) we also give an algorithm to construct them.

Primitive 2: bounded-width path cover. Path covers were introduced in [\[HJS25\]](#) as a directed analogue of neighborhood covers on undirected graphs.

A d_{cov} -path cover of G with diameter d_{diam} is a graph G' , together with a projection map $\pi : V(G') \rightarrow V(G)$, such that:

1. every edge of G' projects to an edge of G (so π is essentially a graph homomorphism from G' to G);
2. every path p of length at most d_{cov} in G lifts to a path in G' , i.e., there is a path p' such that $\pi(p') = p$ (under the natural definition of $\pi(p')$); and
3. every SCC of G' projects to an induced subgraph of G and has strong diameter at most d_{diam} .

The *width* of the path cover is

$$\omega \stackrel{\text{def}}{=} \max_{v \in V(G)} |\pi^{-1}(v)|,$$

that is, the maximum number of copies of a vertex. Bounding the width is crucial for controlling congestion.

The construction in [\[HJS25\]](#) gives near-linear size but does not control the width. For our purposes this is not enough. We therefore design a new construction, inspired by multiplicative weight updates, that penalizes vertices that are copied too often. The result is a path cover of width $n^{o(1)}$ with parameters

$$d_{\text{cov}} = n^{o(1)} \cdot d_{\text{diam}}.$$

The full construction is given in [Section B.10](#).⁵

⁴The formal definition can be found in [Section B.3](#). Here we only give a simplified version, which is not fully rigorous.

⁵In [\[HJS25\]](#), they only construct a path cover satisfying $d_{\text{cov}} < d_{\text{diam}}$. In comparison, this relation $d_{\text{cov}} = n^{o(1)} \cdot d_{\text{diam}}$ might look surprising at first glance. However, we show that this is easy to achieve by concatenating many copies of the same path cover into a single one (as shown in [Lemma B.55](#)).

One-layer hierarchy. To illustrate the main idea, first assume that the input graph G itself is already an (h, s) -length ϕ -expander with respect to $\deg_G$, for every scale h .

For each dyadic scale h' , we construct a DAG $\widehat{G}^{(h')}$ that is responsible for routing flow paths of cost roughly $h' \cdot n^{o(1)}$. The final DAG projection is the disjoint union over all dyadic scales.

We start from a bounded-width path cover $G_{\text{cov}}^{(h')}$ with diameter parameter $h_{\text{diam}} = h'$, covering parameter $h_{\text{cov}} = h' \cdot n^{o(1)}$, and width $\omega = n^{o(1)}$. Similar to the idea of [Coh00], we look at its SCCs and classify them as follows.

Light SCCs. If $|S| < n/\sigma$, where $\sigma = n^{o(1)}$ is a parameter, we recurse on $G[S]$ and replace S by its recursively constructed DAG projection.

Heavy SCCs. If $|S| \geq n/\sigma$, we do not recurse. Instead, we use the expander property: since $G[S]$ has strong diameter at most h' , any sufficiently local demand inside S can be routed at low congestion and average cost $h's$. We therefore replace S by a simple *star gadget*: one copy of S for entering the gadget, one central hub vertex v_S , and one copy of S for exiting it. The edges incident to the hub have capacity $\deg_G(\cdot)$ and length $h's$.

After replacing every SCC in this way, the SCC structure disappears and we obtain a DAG $D^{(h')}$. The crossing SCC edges can also be defined naturally in $D^{(h')}$.

The final step is to add *bridges* between heavy SCCs. We duplicate $D^{(h')}$ into two copies and, for every pair of heavy SCCs (S_1, S_2) , we add $O(\log n)$ edges from the hub of S_1 to the hub of S_2 . The i -th bridge has capacity equal to the maximum flow that can be sent from S_1 to S_2 in G with average cost at most $r_i = (1 + \varepsilon)^i$.

Forward mapping. Given a feasible flow F in G , we lift each flow path P to the appropriate scale $\widehat{G}^{(h')}$ according to its cost. Because the path cover lifts all paths up to length h_{cov} , the path P lifts to a path in $G_{\text{cov}}^{(h')}$, which decomposes into pieces inside SCCs.

For a light SCC, we can simply recurse. So the case that all SCCs are light is easy to solve.

If the path passes through at least one heavy SCC, let the first one be S_1 and last one be S_2 . For each S_1, S_2 , we route through the corresponding star gadget. We connect the two hub vertices v_{S_1}, v_{S_2} using the appropriate bridge. The bridge length approximates the cost of the corresponding subpath from S_1 to S_2 up to a factor of $(1 + \varepsilon)$, while the additional star cost is only $h's$, which can be made negligible compared to the total cost of the path by parameter choice (since $h's = h' \cdot n^{o(1)}$ and the path length $\text{cost}(P) = h' \cdot n^{o(1)}$, we could make $hs = \varepsilon \cdot \text{cost}(P)$). Thus, the total loss is only a $(1 + O(\varepsilon))$ factor.

Feasibility follows because star capacities are chosen according to vertex degree, and bridge capacities are chosen according to the maximum amount of flow that can be routed at the corresponding average-cost scale.

Backward mapping. Conversely, a flow in the DAG can be mapped back to G :

- the light-SCC part is handled recursively,
- the star gadgets are implemented using the expander property inside the corresponding heavy SCC, and
- each bridge is realized by the definition of its capacity.

This gives the backward projection.

Multiple layers. For a full LC-expander hierarchy, the construction recurses through the hierarchy levels. At a high level, instead of building a star over each heavy cluster, we replace each heavy cluster by a lower layer DAG projection, then build the star and bridges. The analysis follows a similar flavour but is much more complicated. We defer the full proof to [Section B.7](#).

Connection to previous works [Hae+25a]. The construction has several similar ideas to the *LC-flow shortcut* construction in [Hae+25a]. In their work, a shortcut on *undirected* graph is constructed so that every flow has a corresponding low-step flow on the graph after adding this shortcut. Our construction

of DAG projection is *not a shortcut* and is working on *directed graphs*. This leads to several difficulties compared to the shortcut construction.

The construction of [Hae+25a] uses *neighborhood cover* on an undirected graph, but there is no such object that covers every neighbor in a directed graph. There is an object called *path cover* with bounded width that we have introduced, but it is weaker than covering neighborhoods, and specifically, it does not allow the argument of covering one low-length path by one SCC - a critical requirement needed for [Hae+25a]. Thus, a more careful analysis of cumulative SCC covering along a path is needed.

More importantly, our construction requires a *cut projection*, as we will explain in the next paragraph.

Flow and cut projection. Remember how we use DAG projection to reduce computing flow on a general graph to a DAG: we compute a flow on the DAG projection, and then convert it into a flow on the original graph using the backward mapping

Fast construction

The existence proof above is not yet algorithmic; the main bottleneck is that it seems to require exactly the approximate MCF oracle we are trying to obtain. To be precise, there are two places that require an MCF oracle:

1. constructing the LC-expander hierarchy usually requires a MCF oracle for the cut-matching game, and
2. computing the capacities of the bridges between heavy SCCs requires at least an approximate MCF.

This results in a 'chicken-and-egg' situation: we want to compute DAG projection to get a MCF algorithm, but it requires MCF as a subroutine.

A similar issue also appears in [HJS26a]. For their purpose, they wish to construct a DAG projection that only preserves distance so that it can help the algorithm for single-source shortest path (SSSP), but the construction requires (1) low-diameter decomposition (LDD), which requires an SSSP oracle, (2) computing bridges similar to our case, which requires an SSSP oracle. Their solution is to introduce an *h-length* version of the projection problem: instead of preserving all distances, the projection only needs to preserve distances up to scale h . They can also define the *h-length* version of LDD and SSSP. Then they try to reduce the *h-length* problem to a smaller-scale h/z -length problem for a suitable parameter $z = n^{o(1)}$, and repeat this $O(\log n)$ times to get the final algorithm.

We apply the same philosophy here. We define:

- an *h-length* DAG projection, which only needs to preserve flows of average cost at most h ;
- an *h-length* approximate MCF problem; and
- an *h-length* LC-expander hierarchy.

The formal definitions are more subtle and appear in [Section B.5](#).

At a high level, we obtain the following cycle of reductions:

1. reduce *h-length* DAG projection to *h-length* LC-expander hierarchy plus *h-length* approximate MCF (in [Section B.7](#));
2. reduce *h-length* LC-expander hierarchy to *h-length* approximate MCF (in [Section B.6](#));
3. reduce *h-length* approximate MCF to *h-length* DAG projection with the help of the DAG MCF oracle (see [Lemma B.24](#));
4. reduce *h-length* DAG projection to (h/z) -length DAG projection using concatenation (see [Lemma B.55](#)).

Taken together, these reductions decrease the length scale from h to h/z , and the recursion terminates once h becomes small enough to handle directly.

A natural question is why we do not simply apply the last reduction repeatedly, since it already decreases the length scale. The reason is that the concatenation lemma [Lemma B.55](#) enlarges the size of the DAG by a factor of z each time. Repeating this for $O(\log_z n)$ levels would lead to a polynomial blow-up. In contrast, the four-step recursion above avoids this: the parameter z enters the *running time* of the recursion, but not the final approximation guarantees or the final size of the projection.

LC-expander hierarchy algorithm. We also briefly mention the algorithmic novelties in [Section B.6](#). Previous work gave existence results for directed LC-expanders [[Hae+25b](#)], and algorithmic constructions only in undirected graphs [[HHT24](#)]. In order to combine the previous ideas into a directed LC-expander hierarchy algorithm, two important ingredients were missing before our work:

1. a *fat low-diameter decomposition*, namely a decomposition whose clusters are well separated in round-trip distance compared to their diameter; and
2. a suitable directed cut-matching game with length constraints.

We obtain the first by a reduction to its undirected variant in [Section B.9.2](#). The second contains more details, and we leave the full proof to [Section B.6](#).

B.2.3 Weighted Push-Relabel Algorithm for MCF on DAGs

Given the first two ingredients, it remains to solve approximate PD-MCF on DAGs. We prove the following in [Section B.8](#).

Lemma B.8. *There exists an **augmenting-path-based** algorithm on DAGs that returns a $(3, 1 + \varepsilon)$ -approximate PD-MCF solution.*⁶

The algorithm extends the weighted push-relabel method for maximum flow on DAGs from [[Ber+24](#)] to the minimum-cost flow setting.

As in classical push-relabel for MCF [[GT90](#)], we maintain a potential function $\phi : V \rightarrow \mathbb{R}$. For an edge (u, v) , its reduced length is

$$\ell_\phi(u, v) \stackrel{\text{def}}{=} \ell(u, v) - (\phi(u) - \phi(v)).$$

The algorithm alternates between:

- *relabel* steps, which decrease vertex potentials, and
- *push* steps along *admissible* edges, i.e., edges whose reduced length is sufficiently negative.

The classical Goldberg–Tarjan algorithm takes $\tilde{O}(mn)$ time essentially because an edge can become admissible up to $\tilde{O}(n)$ different times. Our key idea is to define admissibility using a carefully chosen weight function $w : E \rightarrow \mathbb{R}_{>0}$.

Following [[Ber+24](#)], on a DAG one can assign weights so that:

1. every flow f of interest satisfies

$$\frac{w(f)}{\text{val}(f)} = O(n), \quad \text{where } w(f) \stackrel{\text{def}}{=} \sum_{e \in E} f(e) w(e),$$

and

- 2.

$$\sum_{e \in E} \frac{1}{w(e)} = O(n).$$

We then declare an edge e admissible when its reduced length is roughly $-\Theta(w(e))$. Saturating such an edge changes the relevant potentials by at least $w(e)$. Since the total potential range is $O(n)$, edge e can be saturated only $O\left(\frac{n}{w(e)}\right)$ times. Summing over all edges gives

$$\sum_{e \in E} O\left(\frac{n}{w(e)}\right) = O\left(n \sum_{e \in E} \frac{1}{w(e)}\right) = O(n^2),$$

which is the desired bound.

Although this idea is morally similar to [[Ber+24](#)], extending it from max flow to min-cost flow is not automatic. Two main difficulties arise:

⁶In fact, the weighted push-relabel algorithm on DAGs returns a $(3, 1 + \varepsilon)$ -strongly-approximate PD-MCF solution, which is stronger than what the rest of the framework requires.

1. In max flow, admissible paths move monotonically down the level structure. In min-cost flow, the relevant notion of level is given by potentials, and admissible paths need not be monotone in any obvious sense.
2. In max flow, once the source label becomes small enough, one can directly certify that the current flow is good. In the min-cost setting this argument is more delicate, and we exploit the special structure of DAGs to make it work.

These issues are resolved in [Section B.8](#), completing the third ingredient of the framework.

B.3 Preliminaries

We always use n and m to denote the number of vertices and edges of the original graph in the relevant context (for example, for each main theorem, the original graph is its input graph). Throughout the paper, all input graphs have length (cost) and capacity functions, and we assume that all lengths and capacities are *integers* upper bounded by $N = \text{poly}(n)$ and lower bounded by $-N$. We further assume that capacities are positive.

Directed Graphs. Let $G = (V, E)$ be a directed graph. In this paper, G can be a multigraph containing parallel edges. Let $\ell_G : E \rightarrow \mathbb{R}$ denote the edge length (or *cost*, in the minimum-cost-flow terminology) function of G . A path from vertex v to vertex w is called a (v, w) -path. For any path P , let $\ell_G(P) = \sum_{e \in P} \ell_G(e)$. The *distance* between vertices v and w is $\text{dist}_G(v, w) = \min_{P: (v, w)\text{-path}} \ell_G(P)$. A *ball* of radius r around a vertex v is $\text{ball}_G(v, r) = \{w \mid \text{dist}_G(v, w) \leq r\}$. We further define $\text{dist}_G(v, w) = \infty$ if vertex w cannot be reached from vertex v .

Let $U_G : E \rightarrow \mathbb{R}_{\geq 0}$ denote the edge capacity function of G . Sometimes we use $U(e)$ to denote the capacity of edge e when G is explicitly mentioned. We further define the directed degrees of a vertex v in G , denoted by $\deg_G^{\text{in}}(v)$ and $\deg_G^{\text{out}}(v)$, as $\sum_{(w, v) \in E} U_G(w, v)$ and $\sum_{(v, w) \in E} U_G(v, w)$, respectively. Namely, they represent the total capacities of edges incoming to v and outgoing from v . We use $\deg_G(v) = \deg_G^{\text{in}}(v) + \deg_G^{\text{out}}(v)$ to denote the total capacity of all edges incident to v .

Flows. A *flow/routing* in G is a function $F : \mathcal{P} \rightarrow \mathbb{R}_{\geq 0}$ (where $\mathcal{P}$ denotes the set of simple paths in G) that assigns each simple path P in G a flow value $F(P) \geq 0$. We define P to be a *flow path* of F if $F(P) > 0$. Further, P is a (v, w) -*flow path* of F if P is both a (v, w) -path and a flow path of F . Let $\text{Path}(F)$ denote the set of all paths P with $F(P) > 0$. The *value* of F is denoted by $\text{val}(F) = \sum_{P \in \text{Path}(F)} F(P)$.

The *edge representation* of a flow is a function from edges to the amount of flow passing through each edge, i.e., $F(e) = \sum_{P: P \ni e} F(P)$. Since a flow-path representation can usually have superlinear size, whenever we store a flow, we store it in edge representation. The *cost* of a flow is defined as $\text{cost}(F) = \sum_{e \in E} F(e) \cdot \ell(e)$. The *congestion* of F on an edge e is $\text{cong}_F(e) = \frac{F(e)}{U_G(e)}$. We define a flow to be feasible if its congestion does not exceed one.

It is possible to perform a *flow-path decomposition* of any flow given in edge representation to obtain one of its path representations, as stated in the following theorem.

Theorem B.9 (Lemma C.1 of [\[Ber+25a\]](#)). *Given a flow F on an m -edge graph G with edge capacities and lengths, there is an $O(m \log m)$ -time algorithm that computes a representation $(L_i, \lambda_i, s_i, t_i)_{i \in [k]}$ such that F can be decomposed into k flow paths, each with value λ_i from s_i to t_i and with length L_i .*

Moreover, given any subset $I \subseteq [k]$, there is an algorithm that computes an edge representation routing λ_i amount of flow from s_i to t_i for every $i \in I$ such that the total cost is $\sum_{i \in I} L_i \cdot \lambda_i$.

A flow is called *h -length bounded* if there exists a path decomposition of the flow such that the length of every flow path is at most h . The triple $(\lambda_i, s_i, t_i)_{i \in [k]}$ is called an *implicit flow-path decomposition* if the flow can be decomposed into k flow paths, each from s_i to t_i with value λ_i .

Demands. A (single-commodity) demand $D : V \rightarrow \mathbb{R}$ assigns a value to each vertex such that $\sum_{v \in V} D(v) = 0$. We say the demand D is

- *symmetric h -length* if for every u, v with $D(u), D(v) \neq 0$, we have $\text{dist}_G(u, v) \leq h$ and $\text{dist}_G(v, u) \leq h$,
- *A -respecting*: if for any $v \in V$, $|D(v)| \leq A(v)$.
- *h -length symmetrically constrained* if there exists a partition of D into $D = D_1 + D_2 + \dots + D_k$ such that for every $i \in [k]$, D_i is symmetric h -length. We call the partition $D_1, D_2, \dots, D_k$ a *witness* that D is an h -length symmetrically constrained demand.
- *A -respecting h -length symmetrically constrained*: For the witness sequence of demands $D_1, \dots, D_k$, if for any $v \in V$, $\sum_{i \in [k]} |D_i(v)| \leq A(v)$, then we say D is A -respecting h -length symmetrically constrained. Notice that this is different from saying D is A -respecting, since different D_i with different signs might cancel out.
- *feasible* if there is a feasible flow routing this demand.

We write the size of a demand as $|D| = \sum_{v \in V, D(v) > 0} D(v)$.

We call D an s - t demand if only $D(s) > 0, D(t) < 0$ are nonzero.

Moving cuts. We now give the definition of moving cuts.

Definition B.10 (Length-Constrained Moving Cut). An h -length moving cut $C : E \mapsto \mathbb{R}_{\geq 0}$ assigns to each edge e a non-negative integer length increase $\ell_C = h \cdot C(e)$. We use $\text{len}(C) = h$ to denote the length parameter of C (as an inherited parameter of the moving cut). The *size* of C is defined as

$$|C| = \sum_{e \in E} U(e) \cdot C(e).$$

We define the degree of a moving cut at a vertex v to be

$$\deg_C(v) = \sum_{e \ni v} U(e) \cdot C(e).$$

The graph $G - C$ denotes the graph using $\ell_G + h \cdot C$ as the length function. We write $C' \preceq C$ if $C'(u, v) \leq C(u, v)$ for all $(u, v) \in E$.

B.3.1 Minimum cost flow.

Given a graph $G = (V, E)$ with edge capacities and lengths and a single-commodity demand D , the *minimum cost flow (MCF)* problem asks for a flow routing D with minimum cost.

Minimum cost flow has the following primal-dual LP.

$$\begin{array}{ll}
 \text{(P)} & \min \sum_{e \in E} \ell(e) \cdot f(e) \\
 \text{s.t.} & \sum_{e \in \delta^+(u)} f(e) - \sum_{e \in \delta^-(u)} f(e) = D(u) \quad \forall u \in V, \\
 & 0 \leq f(e) \leq U(e) \quad \forall e \in E. \\
 \text{(D)} & \max \sum_{u \in V} (-D(u)) \cdot \phi(u) - \sum_{e \in E} U(e) \cdot \tilde{\ell}(e) \\
 \text{s.t.} & \phi(v) - \phi(u) \leq \ell(e) + \tilde{\ell}(e) \quad \forall e = (u, v) \in E, \\
 & \tilde{\ell}(e) \geq 0 \quad \forall e \in E.
 \end{array}$$

We can view $\tilde{\ell} : E \rightarrow \mathbb{R}_{\geq 0}$ as a length increase. We can view ϕ as a distance function (from source to sink). The goal of the dual is to enlarge the distance from source to sink by applying a small length increase. We write

$$\text{Dual}(\phi, \tilde{\ell}) = \sum_{u \in V} (-D(u)) \cdot \phi(u) - \sum_{e \in E} U(e) \cdot \tilde{\ell}(e).$$

We say a primal-dual triple $(f, \phi, \tilde{\ell})$ is a (κ, λ) -approximate PD-MCF if (i) f has congestion at most κ , and (ii) $\text{cost}(f) \leq \lambda \cdot \text{Dual}(\phi, \tilde{\ell})$.

The problem of (κ, λ) -approximate PD-MCF is: given a graph G with positive, polynomially bounded integer capacities and lengths, find a (κ, λ) -approximate PD-MCF. We denote an oracle solving the (κ, λ) -approximate PD-MCF by $\mathcal{O}_{\kappa, \lambda}^{\text{PDMCF}}$.

The goal of this paper is to reduce the exact minimum-cost-flow problem on general graphs to $\mathcal{O}_{\kappa,\lambda}^{\text{PDMCF}}$ on DAGs, and to solve $\mathcal{O}_{\kappa,\lambda}^{\text{PDMCF}}$ on DAGs.

When an algorithm makes oracle calls, we say that they are f -efficient if

- the cumulative number of vertices in the graphs on which the oracle runs is at most $f \cdot n$,
- the cumulative number of edges in the graphs on which the oracle runs is at most $f \cdot m$,
- the additional running time is at most $f \cdot m$.

B.3.2 Projection map.

We will extensively use the concept of *projections*, introduced in [HJS25; HJS26a], throughout this paper.

A graph $G' = (V', E')$ can be assigned a projection map $\pi : V' \rightarrow V \cup \{\perp\}$ (where V may be the vertex set of another graph $G = (V, E)$). We write $\pi^{-1} : V \rightarrow 2^{V'}$ for the inverse mapping, where $\pi^{-1}(u) = \{v \in V' \mid \pi(v) = u\}$. For a set $S \subseteq V$, we write $\pi^{-1}(S) = \bigcup_{u \in S} \pi^{-1}(u)$.

For a set $S \subseteq V'$, we write

$$\pi(S) = \{\pi(s) \mid s \in S\} \setminus \{\perp\}.$$

For a path $(v_1, \dots, v_k)$ in G' , we write $\pi(p)$ for the (potentially valid) path in another graph given by $(\pi(v_1), \dots, \pi(v_k))$.

When G' is a DAG, for each $v \in V$, write $v^{\text{first}}, v^{\text{last}}$ for the first and last vertex v' in the topological order of G' with $\pi(v') = v$, respectively. For a demand D' on G' , we write $D = \pi(D')$ for the demand $D(v) = \sum_{\pi(v')=v} D'(v')$. For a demand D on G , we write the lifted demand from first to last vertex on G' as $D' = \hat{\pi}^{-1}(D)$ where $D'(v^{\text{first}}) = D(v)$ for all $D(v) > 0$ and $D'(v^{\text{last}}) = D(v)$ for all $D(v) < 0$, and all other nodes have $D'(v) = 0$.

In [HJS25], they introduced a useful notion of *path cover*, which we state as follows.

Definition B.11. A h_{cov} -path cover with diameter h_{diam} of a graph $G = (V, E)$ is a graph G' along with a projection map π such that

- for every $(u, v) \in E(G')$, we have $(\pi(u), \pi(v)) \in E(G)$ with the same edge length, and $\pi(u) \neq \perp$ for every $u \in V(G')$,
- for every path p of length at most h_{cov} in G , there is a path p' in G' with $\pi(p') = p$,
- for every strongly connected component (SCC) C of G' , $G'[C]$ has diameter at most h_{diam} .

The size of the path cover is $|E(G')|$. We say G' is *induced* if for every $(u, v) \in E(G')$ we have $(\pi(u), \pi(v)) \in E(G)$, and every SCC of G' maps to an induced subgraph of G under π . The *width* of the path cover is defined as $\max_{v \in V} |\pi^{-1}(v)|$.

In [HJS25], they only show how to construct a path cover with bounded size, but with unbounded width. We will show in Section B.10 how to extend their idea to obtain a bounded-width path cover. We summarize the result as follows.

Lemma B.12. *There exists a deterministic algorithm that, given a directed graph $G = (V, E)$ with non-negative edge weights, a diameter parameter d , a parameter $\varepsilon \in (\log \log n / \log n, 1)$ and a stretch parameter $q \in \mathbb{N}_+$, constructs a qd -path cover H of G such that*

1. *every SCC of H has strong diameter $d \cdot \exp(O(1/\varepsilon))$,*
2. *the width is at most $q \cdot n^{O(\varepsilon)}$,*

in $q \cdot m^{1+o(\varepsilon)}$ time.

B.4 Boosting Approximate Minimum Cost Flow

In this section we will show how to get an exact minimum cost flow by oracle calls to approximate minimum cost flow.

Theorem B.3. For $\varepsilon = o(1/\log^9 n)$, there is an algorithm solving exact minimum-cost flow using

$$\tilde{O}\left(n^{O(1/\log(1/\varepsilon))} \cdot \kappa^2/\varepsilon^3\right)$$

-efficient calls to a $(\kappa, 1 + \varepsilon)$ -approximate PD-MCF oracle.

In [Section B.4.1](#), we will show that if we have an oracle solving approximate primal-dual to a MCF LP, then there is a simple way to get an exact algorithm. This boosting holds even when the MCF LP only routes a small fraction of the required demands.

However, a $(\kappa, 1 + \varepsilon)$ -approximate PD-MCF oracle does not directly give a solution to the MCF LP. In [Section B.4.2](#), we will show how to use MWU to turn the $(\kappa, 1 + \varepsilon)$ -approximate PD-MCF oracle into a MCF LP solution and thus get the exact algorithm.

Proof of Theorem B.3. By combining [Lemmas B.18](#) and [B.20](#), we get the required complexity. $\square$

B.4.1 Exact Flow with Approximate Cost

Primal-dual pair of minimum cost flow. We call following primal dual LP for minimum cost flow the MCF LP, as defined in the preliminary.

$$\begin{array}{ll} \text{(P)} & \min \sum_{e \in E} \ell(e) \cdot f(e) \\ \text{s.t.} & \sum_{e \in \delta^+(u)} f(e) - \sum_{e \in \delta^-(u)} f(e) = D(u) \quad \forall u \in V, \\ & 0 \leq f(e) \leq U(e) \quad \forall e \in E \end{array} \qquad \begin{array}{ll} \text{(D)} & \max \sum_{u \in V} (-D(u)) \cdot \phi(u) - \sum_{e \in E} U(e) \cdot \tilde{\ell}(e) \\ \text{s.t.} & \phi(v) - \phi(u) \leq \tilde{\ell}(e) + \ell(e) \quad \forall e = (u, v) \in E, \\ & \tilde{\ell}(e) \geq 0 \quad \forall e \in E. \end{array}$$

Residual minimum cost flow problem. Given a primal-dual pair $f, \phi, \tilde{\ell}$, we define the residual problem $(G_{\text{res}}, D_{\text{res}}) = (G, D)_{f, \phi}$ as follows (notice that $\tilde{\ell}$ should also be in the dual solution, but we omit it here because the residual problem only depends on f and ϕ).

Let G_f be the residual graph of G on f .⁷ Let $(G_f)_\phi$ be the graph G_f applying ϕ as a potential to change the edge length (cost). To be precise, for every edge $(u, v) \in E((G_f)_\phi)$, we have $\ell_{(G_f)_\phi}(u, v) = \ell_G(u, v) - (\phi(v) - \phi(u))$ (for convenience, if $(u, v) \notin E$ but $(u, v) \in E(G_f)$, we define $\ell_G(u, v) = -\ell_G(v, u)$).

Suppose E_{neg} contains all edges in $(G_f)_\phi$ that has negative length, then G_{res} is defined as reversing all edges $(u, v) \in E_{\text{neg}}$ (and change the weight to $-\ell_{(G_f)_\phi}(u, v)$). Notice that G_{res} has only non-negative lengths. D_{res} is defined as the demand such that every edge $(u, v) \in E_{\text{neg}}$ contribute $U(u, v)$ unit source demands $+1$ to v and unit sink demands -1 to u .

The following lemma shows that the exact minimum cost flow can be solved by finding a flow plus an optimal flow for the residual problem.

Lemma B.13. Suppose $f, \phi, \tilde{\ell}$ is a feasible primal-dual pair for the minimum cost flow problem (G, D) , and f_{res}^* is a minimum cost flow of $(G, D)_{f, \phi}$, then $f + f_{\text{res}}^*$ is a minimum cost flow of (G, D) .

Proof. Since f_{res}^* is a minimum cost flow of $(G, D)_{f, \phi}$, we have that $(G_f)_{f_{\text{res}}^*} = G_{f+f_{\text{res}}^*}$ has no negative cycle. Notice that $f + f_{\text{res}}^*$ is a feasible flow routing D according to definition, thus, $f + f_{\text{res}}^*$ is optimal. $\square$

The following lemma shows that the residual problem has a small optimal value if the primal-dual pair is a good approximation.

⁷Notice that G_f might contain parallel edges: for an edge $e \in E$, if f sends x amount of flow on e , then the residual graph will create a reversed edge with cost $-\ell_G(e)$ and capacity x , which might be in parallel with some existing edges with different costs.

Lemma B.14. Suppose $f, \phi, \tilde{\ell}$ is a feasible primal-dual pair of (G, D) with primal-dual value difference Δ , then the optimal cost of the minimum cost flow instance $(G, D)_{f, \phi}$ is at most Δ . More precisely, we have that

$$\sum_{\substack{e \in (G_f)_\phi \\ \ell_{(G_f)_\phi}(e) < 0}} |\ell_{(G_f)_\phi}(e)| \leq \Delta$$

Proof. We use the same notation $G_f, (G_f)_\phi, E_{\text{neg}}, G_{\text{res}}, E_{\text{res}}$ as in the definition of residual problem. For convenience, we write $\ell(\cdot), U(\cdot)$ as the length and capacity function of the original graph G . We write $\ell_\phi(u, v)$ as the length function of $(G_f)_\phi$. We write $U_f(\cdot)$ as the capacity function on the residual graph G_f .

By definition, we can route D_{res} by sending $U_f(v, u)$ units of flow for each edge $(u, v) \in E_{\text{neg}}$. Thus, the optimal cost of $(G_{\text{res}}, D_{\text{res}})$ is at most $\sum_{e \in E_{\text{neg}}} (-U_f(e) \cdot \ell_\phi(e))$. For each $e \in E_{\text{neg}}$, there are two cases,

- (i) if $e = (u, v) \in E$, then since $\phi, \tilde{\ell}$ is a feasible dual solution, we have $-\ell_\phi(e) = -(\ell(e) - \phi(v) + \phi(u)) \leq \tilde{\ell}(e)$,
- (ii) if $e = (v, u) \notin E$ but $e \in E(G_f)$ (which means f sends flows by (u, v) in G so $(u, v) \in E$), then we have $-\ell_\phi(v, u) = \ell(u, v) - (\phi(v) - \phi(u))$. Notice that in this case we must have $U_f(v, u) = f(u, v) = f(e)$.

Thus, we have

$$\begin{aligned} \sum_{e \in E_{\text{neg}}} (-U_f(e) \cdot \ell_\phi(e)) &\leq \sum_{e \in E} U(e) \cdot \tilde{\ell}(e) + \sum_{(u, v) \in E} f(u, v) \cdot (\ell(u, v) - \phi(v) + \phi(u)) \\ &= \sum_{e \in E} f(e) \cdot \ell(e) + \sum_{(u, v) \in E} f(u, v) \cdot (\phi(u) - \phi(v)) + \sum_{e \in E} U(e) \cdot \tilde{\ell}(e) \\ &= \sum_{e \in E} f(e) \cdot \ell(e) + \sum_{u \in V} \phi(u) \cdot \left(\sum_{e \in \delta^+(u)} f(e) - \sum_{e \in \delta^-(u)} f(e) \right) + \sum_{e \in E} U(e) \cdot \tilde{\ell}(e) \\ &= \sum_{e \in E} f(e) \cdot \ell(e) - \left(\sum_{u \in V} \phi(u) \cdot (-D(u)) - \sum_{e \in E} U(e) \cdot \tilde{\ell}(e) \right) \end{aligned}$$

Notice that this is exactly the difference between primal and dual values of $f, \phi, \tilde{\ell}$. $\square$

Lemma B.14 implies that, if we have an algorithm that gives $1 \pm \varepsilon$ approximate primal dual pair for minimum cost flow, then there is an exact algorithm using $O(\log n)$ calls of the approximation algorithm whenever $\varepsilon < 0.5$.

In many situations, we cannot route D perfectly. The following lemma shows how to deal with this case.

Lemma B.15 (Boosting fractional $(1 \pm \varepsilon)$ primal-dual to exact MCF). Suppose there is an algorithm $\mathcal{A}$ which, given a non-negative minimum-cost flow instance (G, D) , returns (i) a feasible flow f routing a sub-demand $D' \preceq D$ with $|D'| \geq |D|/\kappa$ (where $\kappa \geq 1$), and (ii) a dual pair $(\phi, \tilde{\ell})$ such that $(f, \phi, \tilde{\ell})$ is a $(1 \pm \varepsilon)$ approximate primal-dual pair for routing D' in G with $0 \leq \varepsilon \leq 0.1$. Then there is an algorithm that computes an exact minimum-cost flow routing all of D in G using at most $\tilde{O}(\kappa)$ -efficient calls to $\mathcal{A}$.

Proof. We describe a recursive procedure $\mathcal{B}(G, D)$ that returns an exact optimal flow routing D in G . We assume the demand D contains only one source s and one sink t for technical reasons. This assumption is w.o.l.g. since we can add a super source s and super sink t connecting to every vertex with positive/negative demand with the corresponding capacities and length 0.

The algorithm initially sets $G^{(0)} = G$ and $D^{(0)} = D$. Starting from $i = 0$, the algorithm will repeat the following procedure to compute flows.

For each i , the algorithm makes one call to $\mathcal{A}$ to compute a flow f_i routing $\bar{D}^{(i)} \preceq D^{(i)}$ and an approximate dual $(\phi_i, \tilde{\ell}_i)$. Set $G^{(i+1)} = \left((G_{f_i}^{(i)})_{\phi_i} \right)_{\geq 0}$, i.e., the residual graph after routing the flow f_i and applying the potential function ϕ_i , then round each negative weight edge up to 0. Set $D^{(i+1)} = D^{(i)} - \bar{D}^{(i)}$.

We repeat the above operation until $D^{(i)}$ has sufficiently small size (less than $1/\text{poly}(n)$), in which case we let f_i be an arbitrary flow routing $D^{(i)}$ and end the iterations.

Let $f^* = \sum_{i \geq 0} f_i$ and $\phi^* = \sum_{i \geq 0} \phi_i$. Consider the residual instance $(G_{\text{res}}, D_{\text{res}}) = (G, D)_{f^*, \phi^*}$, we recursively call $\mathcal{B}(G_{\text{res}}, D_{\text{res}})$ to find a flow f' routing D_{res} on G_{res} . The final flow we return is $f^* + f'$.

The base case of the recursion is when the residual instance $\mathcal{B}(G_{\text{res}}, D_{\text{res}})$ can be routed by cost sufficiently small (less than $1/\text{poly}(n)$), then we can let f' be an arbitrary flow routing D_{res} , and use a standard rounding lemma⁸ to get an exact optimal solution.

Correctness. The correctness for the non-base case follows from [Lemma B.13](#) since f^*, ϕ^* is a feasible primal and dual solution and f' is an optimal solution for $(G_{\text{res}}, D_{\text{res}})$. In the base case, the flow is a $(1 + 1/\text{poly}(n))$ approximation to the optimal solution, and the correctness is by the rounding lemma in [\[Bra+20a\]](#).

Complexity. The complexity relies on the following lemma.

Lemma B.16. *Let the optimal cost for routing D on G be $\text{OPT}_G(D)$. We have that*

$$\sum_{i \geq 0} \ell_{G^{(i)}}(f_i) \leq 2 \cdot \text{OPT}_G(D)$$

Proof. We will prove by induction that

$$\text{OPT}_{G^{(i)}}(D^{(i)}) \leq \text{OPT}_G(D) - 0.9 \cdot \sum_{j < i} \ell_{G^{(j)}}(f_j) \quad (\text{B.1})$$

Notice that if [Equation \(B.1\)](#) is correct, then we are done since $\text{OPT}_{G^{(i)}}(D^{(i)}) \geq 0$ ($G^{(i)}$ always contains positive edge lengths).

When $i = 0$, we have that $\text{OPT}_{G^{(i)}}(D^{(i)}) = \text{OPT}_G(D)$, so [Equation \(B.1\)](#) holds.

Suppose $i > 0$. We have that

$$\text{OPT}_{G^{(i-1)}}(D^{(i-1)}) \leq \text{OPT}_G(D) - 0.8 \cdot \sum_{j < i-1} \ell_{G^{(j)}}(f_j)$$

by induction hypothesis. Let $\hat{f}$ be the optimal flow routing $D^{(i-1)}$ in $G^{(i-1)}$. Recall that f_{i-1} routes $\bar{D}^{(i-1)}$ in $G^{(i-1)}$. Thus, there is a flow $f = \hat{f} - f_{i-1}$ routing $D^{(i)} = D^{(i-1)} - \bar{D}^{(i-1)}$ in the residual graph $G_{f_{i-1}}^{(i-1)}$, with cost at most $\text{OPT}_{G^{(i-1)}}(D^{(i-1)}) - \ell_{G^{(i-1)}}(f_{i-1})$. Now consider the cost of f on the graph $(G_{f_{i-1}}^{(i-1)})_{\phi_{i-1}}$, we claim that the cost does not increase after applying ϕ_{i-1} : since we are working on single source s and single sink t , the cost will not increase if $\phi_{i-1}(t) \geq \phi_{i-1}(s)$, which can be guaranteed w.o.l.g. if we set ϕ_{i-1} to be the distance function with source s and non-negative edge weights $\ell_{G^{(i-1)}} + \tilde{\ell}_{i-1}$. This adjustment to ϕ_{i-1} makes the difference between $\phi(t)$ and $\phi(s)$ the maximum, and does not change $\tilde{\ell}_{i-1}$, so the dual solution does not decrease. Finally, we have that $G^{(i)} = \left((G_{f_{i-1}}^{(i-1)})_{\phi_{i-1}} \right)_{\geq 0}$, for which the total cost increase for f is at most $2\varepsilon \cdot \ell_{G^{(i-1)}}(f_{i-1})$ according to [Lemma B.14](#). Thus, we get [Equation \(B.1\)](#). $\square$

We also need the following lemma to bound the cost for the recursive call.

Lemma B.17. *Let E_{neg} denote the negative edges of G_{f^*, ϕ^*} , we have that*

$$\sum_{e \in E_{\text{neg}}} |\ell_{G_{f^*, \phi^*}}(e)| \leq 3\varepsilon \cdot \sum_{i \geq 0} \ell_{G^{(i)}}(f_i)$$

Proof. Let $G^{(k)}$ be the graph in the last iteration, we have that $G^{(k)}$ only contains non-negative cost edges. For each edge e of $G^{(k)}$, the difference

$$U_{G^{(k)}}(e) \cdot |\ell_{G^{(k)}}(e) - \ell_{G_{f^*, \phi^*}}(e)|$$

⁸See Lemma 8.10 of [\[Bra+20a\]](#)

comes from a rounding up operation that rounds some negative edges to 0 in some iterations. Consider the i -th iteration, the total amount of rounding up depends on the total size of the negative edges

$$\sum_{\ell_{G_{\phi_i}}^{(i)}(e) < 0} U_{G_{\phi_i}}^{(i)}(e) \cdot |\ell_{G_{\phi_i}}^{(i)}(e)|$$

which according to [Lemma B.14](#), is at most

$$3\varepsilon \cdot \ell_{G^{(i)}}(f_i)$$

Summing over all iterations gives the lemma. $\square$

[Lemma B.17](#) combines with [Lemma B.16](#) gives an upper bound on the optimal cost of the recursive instance to be $6\varepsilon \cdot \text{OPT}_G(D)$. Thus, the recursion has depth at most $O(\log_{1/\varepsilon} n)$ given that $\text{OPT}_G(D) = \text{poly}(n)$ as our assumption. Moreover, the algorithm makes at most $O(\kappa \log n)$ calls to $\mathcal{A}$ since each call to $\mathcal{A}$ reduce the remaining demand $D^{(i)}$ by a fraction of $(1 - 1/\kappa)$. Thus, the total number of calls to $\mathcal{A}$ is $\tilde{O}(\kappa)$ as long as $\varepsilon < 0.1$. $\square$

B.4.2 Flow Boosting

In this section, we show how to use flow boosting to obtain a $(1 + \varepsilon)$ approximation to the following primal-dual form of minimum cost flow, using oracle calls to $(\kappa, (1 + \varepsilon))$ -approximate PD-MCF. Notice that the following formalization is weaker than the MCF LP stated in the last section (for example, an approximation to this LP does not solve max flow exactly while the strong version does). We call it the *MBCF LP*. For convenience, we only consider s - t flow, and let $\mathcal{F}_{s,t}$ contain all s - t feasible flows. Remember the definition of moving cut [Definition B.10](#). We consider C as a moving cut of length h .

$$\begin{aligned} \max \quad & \text{val}(F) \\ \text{s.t.} \quad & \text{cost}(F) \leq B \\ & F \in \mathcal{F}_{s,t} \\ \\ \min \quad & \sum_{e \in E} U(e) \cdot C(e) + B/h \\ \text{s.t.} \quad & \text{dist}_{G-C}(s, t) \geq h \\ & C \geq 0, h > 0 \\ & C \text{ is a } h\text{-length moving cut} \end{aligned}$$

We prove the following lemma.

Lemma B.18. *There is an algorithm that, given a graph G with non-negative edge capacities and lengths, computes a $(1 + \varepsilon)$ -approximate primal-dual pair of the MBCF LP, making $\tilde{O}(\kappa^2/\varepsilon^3)$ -efficient calls to a $(\kappa, o(\varepsilon))$ -approximate PD-MCF oracle.*

One benefit of the MBCF LP is that it admits the following flow boosting template using multiplicative weight update (MWU). The following flow boosting template is from [\[Hae+24\]](#).

Theorem B.19 (Theorem 9.2 of [\[Hae+24\]](#)). *Let $G = (V, E)$ be a graph with capacity function U_G and cost function ℓ_G , and let $B \geq 0$ be a cost budget. Let $\mathcal{F}$ be any convex set of flows in G containing at least one capacity-respecting flow, and let $\varepsilon, s, \kappa \geq 0$ be parameters. Suppose an algorithm is given an oracle $\mathcal{O}$ that, given any integral edge length function $\ell : E \rightarrow \{1, 2, \dots, O(m^{1/\varepsilon} N/\varepsilon)\}$, computes the edge representation of a (not necessarily capacity-respecting) flow $F \in \mathcal{F}$ such that*

- Length slack s : $\sum_{e \in E} F(e) \cdot \ell(e) \leq s \cdot \sum_{e \in E} F^*(e) \cdot \ell(e)$ for any flow $F^* \in \mathcal{F}$ that is also capacity-respecting, and
- Congestion slack κ : $F(e) \leq \kappa U(e)$ for all $e \in E$, i.e., $F(e)/\kappa$ is capacity-respecting.

Then, there is a deterministic algorithm that outputs the edge representation of a flow $\bar{F} \in \mathcal{F}$ and scalar $\lambda \geq 0$ such that

1. Feasibility: The flow $\lambda \bar{F}$ is capacity-respecting with cost at most B , and
2. Approximation factor $\approx s$: Let λ^* be the maximum value such that there exists flow $F^* \in \mathcal{F}$ where $\lambda^* F^*$ is capacity-respecting with cost at most B . Then, $\lambda \geq \frac{1-O(\varepsilon)}{s} \lambda^*$.
3. Dual solution h, C : A length value $h > 1$ and an h -length moving cut C such that (i) for every capacity-respecting flow $F \in \mathcal{F}$ we have $\text{cost}_{G-C}(F) \geq h$, (ii) $|C| + B/h \leq (1 + \varepsilon)\lambda$.

The algorithm takes $\tilde{O}(\lambda \kappa \varepsilon^{-2})$ calls to the oracle and $\tilde{O}(\lambda \kappa \varepsilon^{-2} m)$ time outside of the oracle calls.

Now we are ready to prove [Lemma B.18](#).

Proof of Lemma B.18. Let P be the optimal value for the MBCF LP. We assume we know an approximation $\tilde{P} \in [P, 2P]$, which can be guessed by powers of 2. We scale down the capacities of every edge and B by a factor of $\tilde{P} \cdot \varepsilon'$ where ε' is a sufficiently small parameter which will be decided later. Now it remains to solve the MBCF LP on this scaled down graph to get F, C, h where F has value roughly $1/\varepsilon'$. After which we can get the solution to the original graph by scaling up F and the capacity U both by $\tilde{P} \cdot \varepsilon'$, which makes the primal solution $\tilde{P} \cdot \varepsilon'$ times larger and the dual solution $\sum_{e \in E} U(e) \cdot C(e) + B/h$ also $\tilde{P} \cdot \varepsilon'$ times larger, so it remains a $(1 + \varepsilon)$ -approximate primal-dual pair. Now we focus on the scaled down graph and assume that the optimal solution has value roughly $1/\varepsilon'$.

We first apply [Theorem B.19](#) by setting $\mathcal{F}$ to contain all s - t flows with value 1. To obtain the oracle $\mathcal{O}$ for [Theorem B.19](#), we apply the $(\kappa, 1 + o(\varepsilon))$ -approximate PD-MCF algorithm on G and the single-commodity demand with 1 unit of demand from s to t (although the algorithm only works on integer-weighted graphs, we can scale up the fractional capacity to make it integral). It will return a flow F with length at most $s = 1 + o(\varepsilon)$ times the cost of the optimal capacity-respecting flow in $\mathcal{F}$, and congestion at most κ , so it is a valid oracle.

After calling [Theorem B.19](#), we obtain $\bar{F}$ and h, C . Notice that $\bar{F}$ is a convex combination of flows in $\mathcal{F}$, so $\bar{F}$ has value 1. Since $\lambda \bar{F}$ is capacity-respecting with cost at most B and we know that the largest such flow has value $1/\varepsilon'$, we have that $\lambda \leq 1/\varepsilon'$. Moreover, $\lambda \bar{F}$ is a $(1 - o(\varepsilon))/(1 + o(\varepsilon))$ approximation to the optimal capacity-respecting flow with cost at most B , by [Theorem B.19Item 2](#). We conclude that $\lambda \bar{F}$ is a $(1 - \varepsilon)$ -approximate primal solution to the MBCF LP.

It remains to compute a dual solution for the MBCF LP. Recall that [Theorem B.19](#) returns h, C where C is an h -length cut satisfying (i) for every capacity-respecting flow F with value 1 we have $\text{cost}_{G-C}(F) \geq h$, and (ii) $|C| + B/h \leq (1 + o(\varepsilon))\lambda$.

We apply the $(\kappa, (1 + o(\varepsilon)))$ -approximate PD-MCF algorithm on $G - C$ and the demand sending κ from s to t , and denote the returned primal-dual triple by $(f, \phi, \tilde{\ell})$. Since f has congestion at most κ , f/κ is a feasible flow with value 1, which by property (i) has cost at least h . Thus, $\text{cost}_{G-C}(f) \geq h \cdot \kappa$. Moreover, by the definition of $(\kappa, (1 + o(\varepsilon)))$ -approximate PD-MCF, the dual value satisfies

$$\kappa \cdot (\phi(t) - \phi(s)) - \sum_{e \in E} U(e) \cdot \tilde{\ell}(e) \geq h \cdot \kappa / (1 + o(\varepsilon)).$$

Let $h' = h/(1 + o(\varepsilon))$. We have $\kappa \cdot (\phi(t) - \phi(s)) \geq h' \cdot \kappa$, so $\phi(t) - \phi(s) \geq h'$.

Let C' be an h' -length cut defined as follows: (i) if $\text{dist}_{G-C}(s, t) \geq h'$, then $C' = 0$; otherwise, (ii) let $C' = (\tilde{\ell}/h')/T$ where T is the largest value such that when we interpret C' as an h' -length cut, we get $\text{dist}_{G-C-C'}(s, t) = h'$ (such a T must exist since $\text{dist}_{G-C}(s, t) < h'$ and $\text{dist}_{G-C+\tilde{\ell}}(s, t) \geq \phi(t) - \phi(s) \geq h'$). Moreover, we must have $\phi(t) - \phi(s) \leq h' \cdot T$ since scaling up C' by T while keeping the edge length of G the

same does not increase the distance to more than T times larger. Thus, we get

$$|C'| = \sum_{e \in E} U(e) \cdot \tilde{\ell}(e)/(Th') \leq \frac{\kappa(\phi(t) - \phi(s)) - h \cdot \kappa/(1 + o(\varepsilon))}{Th'} \leq \kappa.$$

Let C^* be an h' -length cut defined as $C^* = C \cdot (h/h') + C'$ so that $G - C^* = (G - C) - C'$. It follows from $|C| + B/h \leq (1 + o(\varepsilon))\lambda$ that

$$|C^*| + B/(h') \leq (1 + o(\varepsilon)) \cdot ((1 + o(\varepsilon))\lambda + \kappa).$$

Notice that λ is roughly $1/\varepsilon'$. By setting $\varepsilon' = o(\varepsilon/\kappa)$, we get that

$$|C^*| + B/h' \leq (1 + \varepsilon)\lambda \quad \text{and} \quad \text{dist}_{G-C^*}(s, t) \geq h',$$

so C^*, h' is a valid $(1 + \varepsilon)$ -approximate dual.

According to [Theorem B.19](#), the algorithm makes $\tilde{O}(\lambda\kappa\varepsilon^{-2}) = \tilde{O}(\kappa^2\varepsilon^{-3})$ calls to the oracle, and an additional $\tilde{O}(\kappa^2\varepsilon^{-3}m)$ time. $\square$

B.4.3 Converting MCF solution: From MBCF to MCF

Now we show how to convert solutions to the MBCF LP into the one needed in the previous section.

Lemma B.20. *Suppose there is an algorithm $\mathcal{A}$ that, for every cost bound B , returns a primal-dual pair (f, C, h) for the MBCF LP with approximation ratio $(1 + \varepsilon)$, where $\varepsilon < 1/\log^9 n$. Then there is an algorithm that makes $2^{O(\log_{1/\varepsilon}^{1/3} n)}$ calls to $\mathcal{A}$ and computes an exact minimum cost flow routing D in G , assuming that there exists a feasible flow routing D in G .*

Proof. We describe an oracle $\mathcal{A}'$ that can be plugged into [Lemma B.15](#). For convenience, we first assume D is a single-source single-sink demand from s to t .

Budget schedule and the index a . Let

$$\alpha \stackrel{\text{def}}{=} \varepsilon^{1/3} \quad \text{and} \quad \beta \stackrel{\text{def}}{=} \varepsilon^{2/3}.$$

We will call $\mathcal{A}$ on a geometric sequence of budgets. Define

$$B_i \stackrel{\text{def}}{=} (1 + \alpha)^{i-2} \quad \text{for } i = 0, 1, 2, \dots, I,$$

where I is the smallest index such that $B_I \geq B_{\max}$ and

$$B_{\max} \stackrel{\text{def}}{=} \sum_{e \in E} U(e) \cdot \ell(e).$$

Note that $B_{\max}$ is a valid a priori upper bound on the cost of *any* feasible (capacity-respecting) flow.

For each $i \in \{0, 1, \dots, I\}$, run $\mathcal{A}$ with budget B_i and obtain (f_i, C_i, h_i) . Let $q_i \stackrel{\text{def}}{=} \text{val}(f_i)$.

Since $B_I \geq B_{\max}$ and the instance is feasible, the MBCF optimum at budget B_I equals $|D|$, hence by the approximation guarantee of $\mathcal{A}$ we have

$$q_I \geq \frac{1}{1 + \varepsilon} \cdot |D| > |D|/2.$$

Moreover, we can assume $q_0 \leq |D|/2$. Otherwise, it implies that with a cost less than $1/4$, we can already route half of the demands. We can assume edge lengths and capacities, the demand D are all non-negative integral by scaling. Thus, this implies that by routing through only 0 length edge, we can already route at least $3/4$ fraction of the demands, which finishes the oracle.

Define

$$i_0 \stackrel{\text{def}}{=} \max\{i \in \{0, 1, \dots, I-1\} : q_i \leq |D|/2\}.$$

This i_0 is well-defined and satisfies $q_{i_0} \leq |D|/2$ and $q_{i_0+1} > |D|/2$.

Now choose an index

$$a \in \arg \max_{i \in \{i_0, i_0+1, \dots, I-1\}} \frac{q_{i+1}}{q_i},$$

with the convention that $\frac{q_{i+1}}{q_i} = +\infty$ if $q_i = 0$. Then $q_{a+1} > |D|/2$ (since $a \geq i_0$) and moreover we claim that

$$q_{a+1} \geq (1 + \beta) q_a. \quad (\text{B.2})$$

To see (B.2), note that

$$\frac{q_I}{q_{i_0}} = \prod_{i=i_0}^{I-1} \frac{q_{i+1}}{q_i} \leq \left(\max_{i \in \{i_0, \dots, I-1\}} \frac{q_{i+1}}{q_i} \right)^{I-i_0} = \left(\frac{q_{a+1}}{q_a} \right)^{I-i_0}.$$

Rearranging gives

$$\frac{q_{a+1}}{q_a} \geq \left(\frac{q_I}{q_{i_0}} \right)^{1/(I-i_0)} \geq \left(\frac{(1+\varepsilon)^{-1}|D|}{|D|/2} \right)^{1/I} \geq \left(\frac{2}{1+\varepsilon} \right)^{1/I}.$$

Since $B_{\max}$ is polynomially bounded (lengths and capacities are polynomially bounded integers), we have $I = \Theta(\log B_{\max}/\alpha) = O(\log n/\alpha)$. Therefore,

$$\left(\frac{2}{1+\varepsilon} \right)^{1/I} \geq \exp\left(\frac{\ln(3/2)}{I} \right) \geq 1 + \Omega\left(\frac{1}{I} \right) \geq 1 + \Omega\left(\frac{\alpha}{\log n} \right).$$

Finally, since $\varepsilon < 1/\log^9 n$, we have $\alpha = \varepsilon^{1/3} \leq 1/\log^3 n$ and thus $\alpha/\log n \geq \varepsilon^{2/3} = \beta$ for all sufficiently large n . This implies (B.2).

Output primal and dual. We set the primal flow output by $\mathcal{A}'$ to be

$$f \stackrel{\text{def}}{=} f_{a+1},$$

so $\text{val}(f) = q_{a+1} > |D|/2$ and $\text{cost}(f) \leq B_{a+1}$.

It remains to produce a dual $(\phi, \tilde{\ell})$ for an s - t demand of value $\text{val}(f) = q_{a+1}$ satisfying $\text{cost}(f) \leq 1.1 \text{Dual}(\phi, \tilde{\ell})$.

Let

$$\tilde{\ell}(e) \stackrel{\text{def}}{=} h_a \cdot C_a(e) \quad \forall e \in E,$$

and let $\phi(\cdot)$ be the shortest-path distance from s in the graph with length function $\ell + \tilde{\ell}$ (equivalently, the distance function in $G - C_a$). Then $(\phi, \tilde{\ell})$ is feasible for the MCF dual, and

$$\phi(t) - \phi(s) = \text{dist}_{G-C_a}(s, t) \geq h_a$$

by dual feasibility of (C_a, h_a) in the MBCF LP.

For the s - t demand of value q_{a+1} , the MCF dual objective equals

$$\text{Dual}(\phi, \tilde{\ell}) = q_{a+1} \cdot (\phi(t) - \phi(s)) - \sum_{e \in E} U(e) \cdot \tilde{\ell}(e) \geq q_{a+1} \cdot h_a - h_a \cdot |C_a|.$$

Using (B.2) and the MBCF LP primal-dual guarantee of $\mathcal{A}$ at budget B_a ,

$$|C_a| + \frac{B_a}{h_a} \leq (1 + \varepsilon) q_a,$$

we obtain

$$\text{Dual}(\phi, \tilde{\ell}) \geq (1 + \beta) q_a h_a - |C_a| h_a \geq (1 + \beta) (q_a h_a - |C_a| h_a) \geq \frac{1 + \beta}{1 + \varepsilon} B_a.$$

Since $\text{cost}(f) \leq B_{a+1} = (1 + \alpha)B_a$, it follows that

$$\text{cost}(f) \leq (1 + \alpha)B_a \leq \frac{(1 + \alpha)(1 + \varepsilon)}{1 + \beta} \text{Dual}(\phi, \tilde{\ell}).$$

With $\alpha = \varepsilon^{1/3}$ and $\beta = \varepsilon^{2/3}$, the multiplicative factor above is $1 + O(\varepsilon^{1/3})$, hence it is at most 1.1 for all sufficiently large n (using $\varepsilon < 1/\log^9 n$). Therefore,

$$\text{cost}(f) \leq 1.1 \text{Dual}(\phi, \tilde{\ell}),$$

as required. This completes the construction of $\mathcal{A}'$.

General demands. For a general demand vector D , let $|D| \stackrel{\text{def}}{=} \sum_{u: D(u) > 0} D(u)$. Add a super-source s and edges (s, u) of capacity $D(u)$ and length 0 for each u with $D(u) > 0$, and add a super-sink t and edges (v, t) of capacity $-D(v)$ and length 0 for each v with $D(v) < 0$. Then routing D in G is equivalent to routing $|D|$ units from s to t in the augmented graph; restricting any s - t flow to the original edges gives a feasible routing of D . The primal/dual objects produced above transfer to the original instance by restriction (primal) and by restricting $(\phi, \tilde{\ell})$ to the original vertices/edges.

Finally, plugging $\mathcal{A}'$ into [Lemma B.15](#) yields the claimed algorithm and the stated bound on the number of calls to $\mathcal{A}$. $\square$

B.5 Flow-Cost-Preserving DAG Projections

Below, we start by introducing the definition of flow-cost-preserving DAG projections, state the main construction algorithm for our DAG projection in [Theorem B.23](#), which will be proved later via the constructions in [Sections B.6](#) and [B.7](#). Instead, we show how to use our DAG projection to reduce computing a minimum-cost flow on general graphs to solving it on DAGs.

We take the definition of DAG projections from [\[HJS26a\]](#) and extend it to preserve both distance and congestions. We call it *flow-cost-preserving DAG projection*.

Definition B.21. A flow-cost-preserving DAG projection of $G = (V, E)$ is a DAG G' along with a projection map $\pi : V(G') \rightarrow V$. We call it (κ, λ) -flow-cost-preserving DAG projection, or say it has congestion slack κ and length slack λ if

1. For every demand D' routable in G' with congestion 1 and cost B , $\pi(D')$ is routable in G with congestion κ and cost B and,
2. For every demand D routable in G with congestion 1 and cost B , $\hat{\pi}^{-1}(D)$ is routable in G' with cost $\lambda \cdot B$ and congestion 1.

When $h = \infty$, write *h-length flow-cost-preserving DAG projection* simply as *flow-cost-preserving DAG projection*.

Often, we want not only the existential property above, but also an explicit and efficient way to project flows and moving cuts from D back to G . This is given by the following definitions. We also add a h -length bound for technical reasons.

Definition B.22 (Projection Algorithm). Let G' be a DAG projection of $G = (V, E)$. A *h-length projection algorithm* with congestion slack κ and length slack λ associated with G' (or (κ, λ) -projection algorithm) is an algorithm that, given either

- a flow F' on G' with congestion 1 and cost B , returns a flow F on G with congestion κ and cost B such that $\pi(\text{Dem}(F')) = \text{Dem}(F)$; or
- a moving cut C' on G' , returns a moving cut C on G with the same size and length $\text{len}(C')/\lambda$ such that for every $u, v \in V$ with $\text{dist}_{G-C}(u, v) \leq h$, we have $\text{dist}_{G'-C'}(u^{\text{first}}, v^{\text{last}}) \leq \lambda \cdot \text{dist}_{G-C}(u, v)$.

If $h = \sum_{e \in E} U(e) \cdot \ell(e)$ is sufficiently large, we simply say that the algorithm is a *projection algorithm* without the h -length bound.

When we say D is associated with a projection algorithm, we assume the projection is surjective, i.e. $\pi^{-1}(v) \neq \emptyset$ for every $v \in V(G)$.

Note that the second property of the projection algorithm is stronger than the second property in the definition of flow-cost-preserving DAG projection. This stronger form is by design and is necessary for our reduction algorithm later in the paper.

We denote the oracle of computing a flow-cost-preserving DAG projection associated with a h -length (κ, λ) -projection algorithm as $\mathcal{O}_{h,\kappa,\lambda}^{\text{Proj}}$. With a bit of overload of notations, we also use $\mathcal{O}_{h,\kappa,\lambda}^{\text{Proj}}$ to denote its associated h -length (κ, λ) -projection algorithm.

In [Section B.7](#), we will show the following main theorem about computing a DAG projection.

Theorem B.23. *There is an algorithm given a directed graph $G = (V, E)$ with non-negative polynomially bounded integral edge capacities and lengths, parameters $\varepsilon_f \in (0, 1/\log n)$, $\delta \in (1/\log n, 1)$ and $\kappa_f \in \mathbb{N}_+$, with $n^\delta \cdot (1/\varepsilon_f)^{\text{poly}(1/\delta)}$ -efficient oracle calls to $\mathcal{O}_{\kappa_f, 1+\varepsilon_f}^{\text{PDMCF}}$, output a DAG projection of size*

$$m^{1+\delta} \cdot (1/\varepsilon_f)^{\text{poly}(1/\delta)}$$

with a (κ, λ) -projection algorithm where

$$\kappa = \kappa_f^{1/\delta} \cdot n^\delta \cdot (1/\varepsilon_f)^{\text{poly}(1/\delta)} \quad \lambda = 1 + \varepsilon_f \cdot (1/\delta)^{O(1/\delta)}$$

The projection algorithm takes time $m^{1+\delta} \cdot (1/\varepsilon_f)^{\text{poly}(1/\delta)}$.

The power of a DAG projection above is that it allows us to reduce approximate primal-dual minimum-cost flow on a general graph to approximate one on a DAG.

Lemma B.24. *Let G be a directed graph with positive polynomially bounded integer edge capacities and lengths, let D be a demand vector on G , and let G' be a DAG equipped with a surjective projection map to G and associated with a (κ', λ') -projection algorithm. Then one can compute a $(\kappa\kappa', \lambda\lambda')$ -approximate PD-MCF solution for (G, D) using $O(1)$ -efficient calls to $\mathcal{O}_{\kappa,\lambda}^{\text{PDMCF}}$ and the associated projection algorithm.*

Combining with [Theorem B.3](#), we can further reduce exact minimum cost flow computation from general graphs to approximate computation on DAGs.

Theorem B.25. *For $\kappa \in \mathbb{N}_+$, $\delta \in (1/\log n, 1)$, $\varepsilon \in (0, 1/\log n)$ with $\varepsilon < \delta^{2/\delta}$, there is an algorithm solving the exact minimum cost flow problem with time complexity proportional to making $\kappa^{1/\delta} \cdot (1/\varepsilon)^{\text{poly}(1/\delta)}$ -efficient calls to $\mathcal{O}_{\kappa, 1+\varepsilon}^{\text{PDMCF}}$.*

Proof. We wish to apply [Theorem B.3](#), which requires an oracle solving PD-MCF. To get that, we compute a flow-cost preserving DAG projection using [Theorem B.23](#), and apply [Lemma B.24](#) along with the $\mathcal{O}_{\kappa, 1+\varepsilon}^{\text{PDMCF}}$ oracle on DAGs to get the required $\mathcal{O}_{\kappa', 1+\varepsilon'}^{\text{PDMCF}}$ oracle on general graphs where

$$\kappa' = \kappa^{O(1/\delta)} \cdot n^\delta \cdot (1/\varepsilon)^{\text{poly}(1/\delta)} \quad \varepsilon' = 1 + \varepsilon \cdot (1/\delta)^{(1/\delta)} = 1 + O(\varepsilon)$$

and the oracle calls to $\mathcal{O}_{\kappa, 1+\varepsilon}^{\text{PDMCF}}$ are on graphs with total size $m^{1+\delta} \cdot (1/\varepsilon)^{\text{poly}(1/\delta)}$ (additional running time is subsumed by this).

Now we can apply [Theorem B.3](#). Notice that since $\varepsilon < \delta^{2/\delta}$, we have that $O(1/\log(1/\varepsilon)) < \delta$. Thus, the total calls to $\mathcal{O}_{\kappa, 1+\varepsilon}^{\text{PDMCF}}$ on DAGs has size (after rescaling δ to hide constants in $\text{poly}(1/\delta)$)

$$m^{1+\delta} \cdot \kappa^{1/\delta} \cdot (1/\varepsilon)^{\text{poly}(1/\delta)}$$

□

Let c be a sufficiently small constant in $(0, 1)$, by choosing $\delta = 1/\log^c n$, we get the following corollary.

Corollary B.26. *For a sufficiently small constant $c \in (0, 1)$ and*

$$\kappa < \exp(\log^{1-2c} n) \quad \varepsilon < \exp(-\log^c n \log \log n)$$

There is an algorithm solving the exact minimum cost flow problem with time complexity proportional to making oracle calls to $\mathcal{O}_{\kappa, 1+\varepsilon}^{\text{PDMCF}}$ on DAGs with cumulative total size $m^{1+o(1)}$ and number of nodes $n^{1+o(1)}$.

B.5.1 Reducing PD-MCF on General Graphs to DAGs

This subsection is dedicated for proving [Lemma B.24](#).

The next lemma isolates the two properties of a projection algorithm that we will use later: it transfers primal flows from the DAG back to the original graph, and it also transfers dual certificates in the reverse direction. As an immediate consequence, any DAG equipped with such a projection algorithm is indeed a flow-cost-preserving DAG projection.

Lemma B.27 (Projection algorithms transfer primal and dual certificates). *Let G' be a DAG together with a surjective projection map $\pi : V(G') \rightarrow V(G)$. Suppose G' is associated with a projection algorithm with congestion slack κ and length slack λ . Then the following hold.*

1. **Primal transfer.** *For every flow F' on G' with congestion at most 1 and cost B , the projection algorithm returns a flow F on G with*

$$\text{Dem}(F) = \pi(\text{Dem}(F')),$$

congestion at most κ , and cost exactly B .

2. **Dual transfer.** *For every demand D on G , and every feasible dual solution $(\phi, \tilde{\ell})$ for the minimum-cost flow instance $(G', \hat{\pi}^{-1}(D))$, one can construct a feasible dual solution (ψ, C) for the minimum-cost flow instance (G, D) such that*

$$\text{Dual}(\psi, C) \geq \frac{1}{\lambda} \text{Dual}(\phi, \tilde{\ell}).$$

Consequently, G' is a (κ, λ) -flow-cost-preserving DAG projection of G .

Proof. The first item is exactly the primal part of the projection algorithm.

We now prove the second item. Fix any demand D on G , and let $(\phi, \tilde{\ell})$ be any feasible dual solution for the instance $(G', \hat{\pi}^{-1}(D))$. Let

$$P := \{u \in V(G) \mid D(u) > 0\}, \quad N := \{v \in V(G) \mid D(v) < 0\}.$$

Since $\hat{\pi}^{-1}(D)$ is supported only on first and last copies, we have

$$\text{Dual}(\phi, \tilde{\ell}) = \sum_{v \in N} (-D(v))\phi(v^{\text{last}}) - \sum_{u \in P} D(u)\phi(u^{\text{first}}) - \sum_{e \in E(G')} U(e)\tilde{\ell}(e).$$

View $\tilde{\ell}$ as a moving cut C' on G' by setting $C'(e) := \tilde{\ell}(e)$ for every $e \in E(G')$. By dual feasibility,

$$\phi(y) - \phi(x) \leq \text{dist}_{G'-C'}(x, y) \quad \forall x, y \in V(G').$$

In particular, for every $u \in P$ and $v \in N$,

$$\phi(v^{\text{last}}) - \phi(u^{\text{first}}) \leq \text{dist}_{G'-C'}(u^{\text{first}}, v^{\text{last}}).$$

Apply the dual part of the projection algorithm to C' . It returns a moving cut C on G with the same size,

$$\sum_{e \in E(G)} U(e)C(e) = \sum_{e \in E(G')} U(e)C'(e) = \sum_{e \in E(G')} U(e)\tilde{\ell}(e),$$

and such that for every $u, v \in V(G)$,

$$\text{dist}_{G'-C'}(u^{\text{first}}, v^{\text{last}}) \leq \lambda \cdot \text{dist}_{G-C}(u, v).$$

Hence, for every $u \in P$ and $v \in N$,

$$\phi(v^{\text{last}}) - \phi(u^{\text{first}}) \leq \lambda \cdot \text{dist}_{G-C}(u, v).$$

Now define

$$\psi(x) := \min_{u \in P} \left\{ \frac{\phi(u^{\text{first}})}{\lambda} + \text{dist}_{G-C}(u, x) \right\}.$$

Since $G - C$ uses the length function $\ell + C$, the pair (ψ, C) is dual feasible for the minimum-cost flow LP on G .

Moreover, for every $u \in P$,

$$\psi(u) \leq \frac{\phi(u^{\text{first}})}{\lambda},$$

while for every $v \in N$, the previous inequality implies that for every $u \in P$,

$$\frac{\phi(u^{\text{first}})}{\lambda} + \text{dist}_{G-C}(u, v) \geq \frac{\phi(v^{\text{last}})}{\lambda},$$

and therefore

$$\psi(v) \geq \frac{\phi(v^{\text{last}})}{\lambda}.$$

Using these bounds, we get

$$\begin{aligned} \text{Dual}(\psi, C) &= \sum_{x \in V(G)} (-D(x))\psi(x) - \sum_{e \in E(G)} U(e)C(e) \\ &\geq \frac{1}{\lambda} \left(\sum_{v \in N} (-D(v))\phi(v^{\text{last}}) - \sum_{u \in P} D(u)\phi(u^{\text{first}}) \right) - \sum_{e \in E(G)} U(e)C(e) \\ &= \frac{1}{\lambda} \left(\sum_{v \in N} (-D(v))\phi(v^{\text{last}}) - \sum_{u \in P} D(u)\phi(u^{\text{first}}) - \sum_{e \in E(G')} U(e)\tilde{\ell}(e) \right) \\ &= \frac{1}{\lambda} \text{Dual}(\phi, \tilde{\ell}). \end{aligned}$$

This proves the second item.

It remains to prove the third item, namely that G' is a (κ, λ) -flow-cost-preserving DAG projection of G .

For the first property of [Definition B.6](#), let D' be any demand routable in G' with congestion 1 and cost B , and let F' be a corresponding flow. By the first item, F' projects to a flow F on G with

$$\text{Dem}(F) = \pi(\text{Dem}(F')) = \pi(D'),$$

the same cost B , and congestion at most κ . Hence $\pi(D')$ is routable in G with congestion κ and cost B .

For the second property, let D be any demand routable in G with congestion 1 and cost B . Suppose for contradiction that $\hat{\pi}^{-1}(D)$ is not routable in G' with congestion 1 and cost at most λB . By LP duality, there exists a feasible dual solution $(\phi, \tilde{\ell})$ for $(G', \hat{\pi}^{-1}(D))$ with

$$\text{Dual}(\phi, \tilde{\ell}) > \lambda B.$$

By the second item, this yields a feasible dual solution (ψ, C) for (G, D) such that

$$\text{Dual}(\psi, C) > \frac{1}{\lambda} \cdot \lambda B = B.$$

By weak duality, every feasible flow routing D in G has cost greater than B , contradicting the assumption. Therefore $\hat{\pi}^{-1}(D)$ is routable in G' with congestion 1 and cost at most λB .

So G' is a (κ, λ) -flow-cost-preserving DAG projection of G . □

Now, we are ready to prove [Lemma B.24](#).

Proof of [Lemma B.24](#). Let

$$D' := \hat{\pi}^{-1}(D).$$

Run the oracle $\mathcal{O}_{\kappa, \lambda}^{\text{PDMCF}}$ on the instance (G', D') , and let $(\hat{F}, \hat{\phi}, \hat{\ell})$ be the returned primal-dual triple. Then $\hat{F}$ routes D' in G' with congestion at most κ , and

$$\text{cost}(\hat{F}) \leq \lambda \cdot \text{Dual}(\hat{\phi}, \hat{\ell}).$$

By [Lemma B.27](#)(1), projecting $\hat{F}$ back to G gives a flow F routing

$$\pi(D') = \pi(\hat{\pi}^{-1}(D)) = D,$$

with the same cost and congestion at most $\kappa\kappa'$. Hence

$$\text{cost}(F) = \text{cost}(\hat{F}).$$

By [Lemma B.27](#)(2), the feasible dual solution $(\hat{\phi}, \hat{\ell})$ for (G', D') can be transferred to a feasible dual solution (ψ, C) for (G, D) satisfying

$$\text{Dual}(\psi, C) \geq \frac{1}{\lambda'} \text{Dual}(\hat{\phi}, \hat{\ell}).$$

Therefore,

$$\text{cost}(F) = \text{cost}(\hat{F}) \leq \lambda \cdot \text{Dual}(\hat{\phi}, \hat{\ell}) \leq \lambda\lambda' \cdot \text{Dual}(\psi, C).$$

Thus (F, ψ, C) is a $(\kappa\kappa', \lambda\lambda')$ -approximate PD-MCF solution for (G, D) , as claimed. □

B.6 Length Constrained Expander Hierarchy Construction

The goal of this section is to prove [Lemma B.36](#) on the construction of weak length-constrained expander hierarchy. This will be used for our DAG projection construction in the next section. Before stating the lemma, required background and definitions are given below.

Basic definitions. The following definitions of directed length-constraint expansion are identical to [\[Hae+25b\]](#). We put them here for completeness. We first need to define multi-commodity demand only for the purpose of LC-expander, as follows.

A multi-commodity *demand* $D : V \times V \rightarrow \mathbb{R}_{\geq 0}$ assigns a non-negative value $D(v, w) \geq 0$ to each ordered pair of vertices in V , specifying that vertex v wants to send $D(v, w)$ units of demand to vertex w . We note that it is always safe to assume $D(u, u) = 0$. The size of a demand is written as $|D|$ and is defined as $\sum_{v, w} D(v, w)$. For each vertex v , we let $D(v, \cdot) = \sum_w D(v, w)$ and $D(\cdot, v) = \sum_w D(w, v)$ denote the total demand starting from and ending at v , respectively. We say a demand D is

- *h-length constrained*: if for each $v, w \in V$ with $D(v, w) > 0$, we have $\text{dist}_G(v, w) \leq h$;
- *symmetric*: if for any pair of vertices $v, w \in V$, $D(v, w) = D(w, v)$;
- *a sub-demand of D'* : if for any pair of vertices $v, w \in V$, $D(v, w) \leq D'(v, w)$;
- *A-respecting for some node-weighting A* : if for any $v \in V$, $\max\{D(v, \cdot), D(\cdot, v)\} \leq A(v)$.

We say a flow F routes the demand D if there exists a path decomposition of F such that for every $D(v, w) \geq 0$, there is a flow path of F from v to w with value $D(v, w)$.

Definition B.28 (Node-Weighting). Given a directed graph $G = (V, E)$ with edge capacities, a *node-weighting* is a function $A : V \rightarrow \mathbb{R}_{\geq 0}$. Notice that $\deg_G$ is a node weighting. For a single-commodity demand D , we write A_D as the node-weighting $A_D(v) = |D(v)|$.

Definition B.29 (*h -Length Separated Demand*). For any multi-commodity demand D and any h -length moving cut C , we define the amount of h' -length separated demand as the sum of demands between vertices that are h' -length separated by C . We denote this quantity with $\text{sep}_{h'}(C, D)$, i.e.,

$$\text{sep}_{h'}(C, D) = \sum_{u, v: \text{dist}_{G-C}(u, v) > h'} D(u, v).$$

Definition B.30 (*h -Length Sparsity of a Cut C for Demand D*). For any multi-commodity demand D and any h -length moving cut C and h' with $\text{sep}_{h'}(C, D) > 0$, the h' -length sparsity of C with respect to D is the ratio of C 's size to how much demand it h' -length separates i.e.,

$$\text{spars}_{h'}(C, D) = \frac{|C|}{\text{sep}_{h'}(C, D)}.$$

Above we generalize the definition of length-constrained moving cut w.r.t arbitrary directed h -length demand. However, for the definition of a directed length-constrained expander, we restrict to symmetric h -length demands.

Definition B.31 (*(h, s) -Length Sparsity of a Cut w.r.t. a Node-Weighting*). The (h, s) -length sparsity of any $h \cdot s$ -length moving cut C with respect to a node-weighting A is defined as:

$$\text{spars}_{(h, s)}(C, A) = \min_{A\text{-respecting } h\text{-length symmetric demand } D} \text{spars}_{h \cdot s}(C, D).$$

Sequence of sparse moving cuts. We can upper bound the total size of a sequence of sparse moving cuts.

Definition B.32 (*Sequence of Moving Cuts*). Given a directed graph $G = (V, E)$, and node-weighting A , let $(C_1, C_2, \dots, C_n)$ be a sequence of $h \cdot s$ moving cuts, let $G - \sum_{j < i} C_j$ denote the graph that is applied with cuts from C_1 to C_{i-1} . We define $(C_1, C_2, \dots, C_n)$ as a sequence of (h, s) -length ϕ -sparse moving cuts if and only if the (h, s) -length sparsity of C_i w.r.t A in $G - \sum_{j < i} C_j$ is at most ϕ .

We will prove the following lemma in [Section B.9.1](#).

Lemma B.33. *Let A be a node-weighting on a graph G with positive integral edge lengths and capacities. Let $C_1, \dots, C_k$ be a sequence of moving cuts. Let $h_1, \dots, h_k > 1$, $\phi \in (0, 1)$, $1 \leq \alpha \leq \log n$, and*

$$s > \frac{8 \log_2 n}{\alpha}.$$

Suppose that for every $i \in [k]$, the cut C_i is (h_i, s) -length ϕ -sparse with respect to A in the graph

$$G - \sum_{j < i} C_j.$$

Then

$$\sum_{i=1}^k |C_i| \leq O(2^{8\alpha+2} \phi \log^2 n) \cdot |A|.$$

Decomposition. We are only going to use weak expander decomposition in this paper, which is defined as follows.

Definition B.34 (*Weak Length-Constraint Expander (LC-ED) Decomposition*). Given a graph G with positive integral capacities and lengths, let C be a h -length moving cut and A be a node-weighting.

We say A is (h, s) -length C -constraint ϕ -expanding if every A -respecting h -length symmetrically constrained single-commodity demand D on $G - C$ can be routed on G by a flow with length bound $h \cdot s$ and congestion $1/\phi$.

Moreover, we say A is $(\leq h, s)$ -length C -constraint ϕ -expanding with length slack s if for every $h' \leq h$, every A -respecting h' -length symmetrically constrained single-commodity demand D on $G - C$ can be routed

on G by a flow with length bound $h' \cdot s$ and congestion $1/\phi$.

An *associated routing algorithm* for A is an algorithm with input a A -respecting h -length (or h' -length with $h' \leq h$ in the second case) symmetrically constrained single-commodity demand D on $G - C$ with a witness, output a flow routing the demand.

We define a weak LC-ED hierarchy as follows. For a sequence of moving cuts $C_1, \dots, C_t$, we write $C_{>i} = \sum_{j>i} C_j$.

Definition B.35 (Weak LC-ED hierarchy). Given a graph G with positive integral capacities and lengths, a $\leq h$ -length (ϕ, s) -expanding t -layer weak LC-ED hierarchy is a sequence of moving cuts $C_1, \dots, C_t$ such that $\sum_{i \in [t]} C_i(e) \geq 1$ for all $e \in E$, and for every $h' \leq h$, we interpret $C_{>i}$ as an h' -length moving cut, then $\deg_{C_i}$ is $\leq h'$ -length $C_{>i}$ -constraint (ϕ, s) -expanding.

Now, we are ready to state the goal of this section.

Lemma B.36. Let $G = (V, E)$ be a directed graph with positive integral capacities and lengths, let $h \geq 1$, let $\varepsilon \in (\log \log n / \log n, 1)$, $\phi \in (1/\text{poly}(n), \frac{1}{\kappa \kappa' n^{c\varepsilon}})$ for some sufficiently large constant c , then there exists $s = (\log n)^{O(1/\varepsilon)}$ and an algorithm that outputs a $\leq h$ -length (ϕ, s) -expanding t -layer weak LC-expander hierarchy together with efficient associated routing algorithms, where

$$t = O\left(\frac{\log m}{\log(1/(\kappa \kappa' n^{O(\varepsilon)} \phi))}\right).$$

The algorithm makes $\frac{m^{1+O(\varepsilon)}}{\phi^3}$ -efficient calls to $\mathcal{O}_{(hs, \kappa', \lambda')}^{\text{Proj}}$ and $\mathcal{O}_{\kappa, \lambda}^{\text{PDMCF}}$ where $\lambda, \lambda' \in [1, 2]$

B.6.1 FatLDD for Round-Trip Distance

In this section, we will describe an important subroutine that extends the normal LDD on directed graphs to allow big separation between clusters, which we call FatLDD, stated in [Lemma B.40](#).

The separation will be based on the symmetric round-trip distance, which is defined as follows.

Definition B.37 (Round-trip distance). On a directed graph $G = (V, E)$ with length ℓ , the *round-trip* distance between two vertices u, v is defined as

$$\widehat{\text{dist}}_G(u, v) = \text{dist}_G(u, v) + \text{dist}_G(v, u)$$

We will use the existing FatLDD construction for undirected graphs on the graph G' . We begin with the notions of *clustering* and *neighborhood cover*.

Definition B.38 (Clustering). Given a graph $G = (V, E)$ with distance function $\text{dist} : V \times V \rightarrow \mathbb{R}_{\geq 0}$, a *clustering* $\mathcal{S} = \{S_1, \dots, S_{|\mathcal{S}|}\}$ is a collection of pairwise disjoint vertex sets, called *clusters*. We say that $\mathcal{S}$ has *diameter* h_{diam} (with respect to dist) if

$$\text{dist}(u, v) \leq h_{\text{diam}} \quad \text{for all } i \in [|\mathcal{S}|] \text{ and all } u, v \in S_i.$$

We say that $\mathcal{S}$ has *separation factor* s if

$$\text{dist}(S_i, S_j) \stackrel{\text{def}}{=} \min_{u \in S_i, v \in S_j} \text{dist}(u, v) \geq s \cdot h_{\text{diam}} \quad \text{for all } i \neq j.$$

A *separation witness* for separation factor s is a collection of pairwise disjoint vertex sets $S'_1, \dots, S'_{|\mathcal{S}|}$ such that

$$\text{ball}_{\text{dist}}\left(S_i, \frac{s h_{\text{diam}}}{2}\right) \subseteq S'_i \quad \text{for every } i \in [|\mathcal{S}|].$$

Intuitively, the sets S'_i certify that different clusters are well separated. As usual, one may replace the constant $1/2$ above by any smaller absolute constant by adjusting s by a constant factor.

Definition B.39 (Neighborhood Cover). Given a graph $G = (V, E)$ with distance function $\text{dist} : V \times V \rightarrow \mathbb{R}_{\geq 0}$, a *neighborhood cover* $\mathcal{N}$ with *width* ω , *covering radius* h_{cov} , *diameter bound* h_{diam} , and *separation factor* s (with respect to dist) is a collection of ω clusterings

$$\mathcal{N} = \{\mathcal{S}_1, \dots, \mathcal{S}_\omega\},$$

where each $\mathcal{S}_i$ has diameter $h_i \leq h_{\text{diam}}$ and separation factor s , such that for every vertex $v \in V$, there exists a cluster S in some $\mathcal{S}_i$ satisfying

$$\text{ball}_{\text{dist}}(v, h_{\text{cov}}) \subseteq S.$$

When we say that the cover comes *with separation witnesses*, we mean that each clustering $\mathcal{S}_i$ is accompanied by a separation witness.

We have the following lemma. The proof is deferred to [Section B.9.2](#).

Lemma B.40. *For any $h_{\text{cov}} > 0$, $s > 1$, and $\varepsilon \in (0, 1)$, and any directed graph $G = (V, E)$ with edge lengths ℓ , there is a randomized algorithm that computes a neighborhood cover with respect to $\widehat{\text{dist}}_G$, together with separation witnesses, with*

- *covering radius* h_{cov} ,
- *separation factor* s ,
- *diameter bound*

$$h_{\text{diam}} = \frac{1}{\varepsilon} \cdot O(s \log^2 n)^{O(1/\varepsilon)} \cdot h_{\text{cov}},$$

- *width*

$$\omega = n^{O(\varepsilon)} \cdot \log n.$$

The algorithm runs in $\hat{O}(m)$ time given that both $h_{\text{diam}}/h_{\text{cov}}$ and ω are in the order of $n^{o(1)}$.

B.6.2 LC-Expander Decomposition

In this section, we will show an algorithm that use cut-matching game to either find a cut that separates a large amount of demands, or certify that there is a large amount of node-weighting that is expanding. See [Lemma B.44](#) for the main result of this section. We first provide some helpful lemmas.

The algorithm involves the following non-stop cut matching game from [\[FLL25\]](#).

Definition B.41 (Matching). Let (P, Q) be a partition of V . A set of directed edges $M \subseteq P \times Q$ with capacities $U_M : M \rightarrow \mathbb{R}_{\geq 0}$ is called a (P, Q) -*matching*. For a demand bound $\mathbf{d} : V \rightarrow \mathbb{R}_{\geq 0}$ and an error $\varepsilon \geq 0$, we say M is $(\mathbf{d}, \varepsilon)$ -*perfect* if

1. $\text{vol}_M \leq \mathbf{d}$ (i.e. M does not send more than $\mathbf{d}(v)$ out of any vertex), and
2. $\text{vol}_M(V) \geq (1 - \varepsilon) \cdot \mathbf{d}(V)$.

Lemma B.42 ([\[FLL25\]](#)). *Let $\mathbf{d} : V \rightarrow \mathbb{N}$ be polynomially bounded and let $\varepsilon = o(1/\log^2 n)$. Suppose we have an oracle $\mathcal{O}_{\text{match}}$ that, for any partition (P, Q) of V and any $\mathbf{d}' \leq \mathbf{d}$ with $\mathbf{d}'(P) = \mathbf{d}'(Q)$ and $\mathbf{d}'(V) \geq \mathbf{d}(V)/2$, returns a $(\mathbf{d}', \varepsilon)$ -perfect (P, Q) -matching.*

Then there is a randomized algorithm that makes $O(\log^2 n)$ calls to $\mathcal{O}_{\text{match}}$ and $O(m \log^2 n)$ additional work with $\tilde{O}(1)$ depth, and produces $\tilde{\mathbf{d}} \leq \mathbf{d}$ with

$$\tilde{\mathbf{d}}(V) \geq (1 - O(\varepsilon \log^2 n)) \cdot \mathbf{d}(V)$$

such that, with high probability, $\tilde{\mathbf{d}}$ is $\Omega(1)$ -expanding in the graph

$$W := \bigcup_i M_i,$$

where M_i is the matching returned by the i -th call to $\mathcal{O}_{\text{match}}$.

We also need the following lemma showing how to compute cut-match using calls to a length bounded MCF algorithm.

Lemma B.43. *Given a directed graph G with positive integral edge capacities and lengths, an h -length single-commodity demand D , and parameters $\phi, \varepsilon \in (0, 0.1)$ and $s > 1$, there is an algorithm making $O\left(\frac{\log(1/\varepsilon)}{\phi}\right)$ -efficient oracle calls to $\mathcal{O}_{hs, \kappa', \lambda'}^{\text{Proj}}$ and to $\mathcal{O}_{\kappa, \lambda}^{\text{PDMCF}}$, that either*

- *outputs an implicit flow-path decomposition of an hs -length flow routing some demand $D' \preceq D$ with congestion*

$$O\left(\frac{\kappa \kappa' \log(1/\varepsilon)}{\phi}\right),$$

such that

$$|D'| \geq (1 - \varepsilon) \cdot |D|,$$

or

- *outputs a $\frac{hs}{4\lambda\lambda'}$ -length moving cut C that is $(h, \frac{s}{4\lambda\lambda'})$ -length ϕ -sparse with respect to A_D and has size at least*

$$\frac{\phi\varepsilon}{3} \cdot |D|.$$

Lemma B.44. *Let $G = (V, E)$ be a directed graph with positive integral capacities and integral edge lengths, let $h \geq 1$ and $\lambda, \lambda' \in [1, 2]$, let $\varepsilon \in (\log \log n / \log n, 1)$, $\phi \in (1/\text{poly}(n), 1)$, and let $A : V \rightarrow \mathbb{R}_{\geq 0}$ be a node-weighting.*

Then there is a length slack

$$s = (\log n)^{O(1/\varepsilon)}$$

and an algorithm that makes $\frac{n^{O(\varepsilon)}}{\phi^3}$ -efficient calls to $\mathcal{O}_{hs, \kappa', \lambda'}^{\text{Proj}}$ and $\mathcal{O}_{\kappa, \lambda}^{\text{PDMCF}}$ and the same additional time, and outputs a hs -length moving cut C and a node-weighting $A' \preceq A$ such that:

1. *A' is (h, s) -length C -constraint ϕ -expanding with an efficient associated routing algorithm,*
2. *both $|C|$ and $|A| - |A'|$ satisfy*

$$|C|, |A| - |A'| \leq \Delta \cdot \phi \cdot |A|, \quad \text{where} \quad \Delta = \kappa \kappa' n^{O(\varepsilon)}$$

Proof. We first define the parameters used in the proof:

$$\begin{aligned} s_{\text{sep}} &:= 512 \log n, & s_0 &:= 256 \log n, & \sigma &:= \frac{s_0}{4\lambda\lambda'}, \\ \omega &:= n^{O(\varepsilon)} \log n, & \Gamma &:= \frac{1}{\varepsilon} \cdot O(\log^3 n)^{O(1/\varepsilon)}, & s &:= c_s \Gamma \log^2 n, \\ \varepsilon_{\text{cm}} &:= \frac{\phi}{100\omega \log^2 n}, & \beta &:= \frac{\phi}{100\omega}, & L_\phi &:= \log \frac{100\omega \log^2 n}{\phi}, \\ \phi_0 &:= c_0 \omega \kappa \kappa' L_\phi \cdot \phi, & \Delta &:= c_1 \omega \kappa \kappa' L_\phi \log^2 n. \end{aligned}$$

Here s_{sep} is the separation factor used in the FatLDD cover, s_0 is the slack passed to [Lemma B.43](#), and σ is the resulting sparsity parameter of the cuts returned by [Lemma B.43](#). The parameters ω and Γ are respectively the width and diameter blow-up of the FatLDD cover. The final slack s is chosen large enough to absorb the $O(\Gamma \log^2 n)$ routing-length blow-up. The parameter ε_{cm} is the error used in the cut-matching game, β is the threshold for deciding whether a round is non-terminal, and ϕ_0 is the sparsity parameter given to [Lemma B.43](#). Finally, Δ is the loss parameter that will upper bound both the total cut size and the total mass loss.

Since $\phi \in (1/\text{poly}(n), 1)$ and $\varepsilon > \log \log n / \log n$, all polylogarithmic factors are absorbed by $n^{O(\varepsilon)}$, and thus

$$s = (\log n)^{O(1/\varepsilon)} \quad \text{and} \quad \Delta = \kappa \kappa' n^{O(\varepsilon)}.$$

The algorithm. Instead of maintaining a single moving cut during the execution, we maintain a cumulative length-increase function

$$\Lambda : E \rightarrow \mathbb{R}_{\geq 0},$$

initially $\Lambda \equiv 0$. At the beginning of each round, let G° denote the graph on the same vertex and edge sets as G , with edge-length function

$$\ell_{G^\circ}(e) = \ell_G(e) + \Lambda(e).$$

Apply [Lemma B.40](#) to G° with parameters

$$(h_{\text{cov}}, s, \varepsilon) = (2h, s_{\text{sep}}, \varepsilon),$$

obtaining a neighborhood cover $\mathcal{N}$ with respect to $\widehat{\text{dist}}_{G^\circ}$, with width ω , and with each clustering $\mathcal{S} \in \mathcal{N}$ having diameter at most

$$h_{\mathcal{S}} \leq \Gamma h.$$

Fix a clustering $\mathcal{S} \in \mathcal{N}$ and a cluster $S \in \mathcal{S}$. Let $h_S \leq \Gamma h$ be the diameter bound of S , and let S' be its separation witness, so that

$$\text{ball}_{\widehat{\text{dist}}_{G^\circ}}(S, s_{\text{sep}} h_S / 2) \subseteq S'.$$

We run the cut-matching game of [Lemma B.42](#) on S with demand bound

$$\mathbf{d}(v) := A(v) \quad (v \in S).$$

We implement the matching oracle $\mathcal{O}_{\text{match}}$ as follows. Given a partition (P, Q) of S and a vector $\mathbf{d}' \leq \mathbf{d}$ with

$$\mathbf{d}'(P) = \mathbf{d}'(Q) \quad \text{and} \quad \mathbf{d}'(V) \geq \mathbf{d}(V)/2,$$

define the single-commodity demand

$$D(v) := \begin{cases} \mathbf{d}'(v) & v \in P, \\ -\mathbf{d}'(v) & v \in Q, \\ 0 & v \notin S. \end{cases}$$

We first prove the key geometric claim that paths of length at most $h_S \sigma$ between vertices of S cannot leave S' after adding any cut supported on S' .

Claim B.45. *Let $u, v \in S$, and let C' be any moving cut supported on edges of $G^\circ[S']$. Then every (u, v) -path in $G^\circ - C'$ of length at most $h_S \sigma$ is contained in S' . Consequently, if*

$$\text{dist}_{G^\circ[S'] - C'}(u, v) > h_S \sigma,$$

then also

$$\text{dist}_{G^\circ - C'}(u, v) > h_S \sigma.$$

Proof. Let P be a (u, v) -path in $G^\circ - C'$ of length at most $h_S \sigma$, and let x be any vertex on P . Since C' only adds nonnegative lengths, the prefix of P from u to x has length at most $h_S \sigma$ already in G° , and similarly the suffix from x to v has length at most $h_S \sigma$ in G° . Hence

$$\text{dist}_{G^\circ}(u, x) \leq h_S \sigma$$

and

$$\text{dist}_{G^\circ}(x, v) \leq h_S \sigma.$$

Since $u, v \in S$ and S has round-trip diameter at most h_S , we also have

$$\text{dist}_{G^\circ}(v, u) \leq \widehat{\text{dist}}_{G^\circ}(u, v) \leq h_S.$$

Therefore

$$\text{dist}_{G^\circ}(x, u) \leq \text{dist}_{G^\circ}(x, v) + \text{dist}_{G^\circ}(v, u) \leq h_S \sigma + h_S,$$

and thus

$$\widehat{\text{dist}}_{G^\circ}(x, u) \leq h_S \sigma + (h_S \sigma + h_S) = (2\sigma + 1)h_S.$$

Because $\sigma \leq 64 \log n$ and $s_{\text{sep}} = 512 \log n$, we have

$$2\sigma + 1 < s_{\text{sep}}/2.$$

Hence

$$\widehat{\text{dist}}_{G^\circ}(x, u) < s_{\text{sep}} h_S / 2,$$

so

$$x \in \text{ball}_{\widehat{\text{dist}}_{G^\circ}}(S, s_{\text{sep}} h_S / 2) \subseteq S'.$$

Thus every vertex of P lies in S' , proving the first part.

The second part follows immediately: if there were a (u, v) -path in $G^\circ - C'$ of length at most $h_S \sigma$, then by the first part it would lie entirely in S' , contradicting

$$\text{dist}_{G^\circ[S'] - C'}(u, v) > h_S \sigma. \quad \square$$

Now every positive vertex and every negative vertex of D lie in S . Since S has round-trip diameter at most h_S , every positive vertex and every negative vertex are at directed distance at most h_S in G° . As $\sigma > 1$, **Claim B.45** implies that every path of length at most h_S between vertices of S is contained in S' . Hence D is an h_S -length single-commodity demand on $G^\circ[S']$.

We therefore invoke **Lemma B.43** on $G^\circ[S']$ with parameters

$$(h, \phi, \varepsilon, s) \leftarrow (h_S, \phi_0, \varepsilon_{\text{cm}}, s_0).$$

Since

$$h_S s_0 \leq \Gamma h \cdot 256 \log n \leq h s$$

by the choice of s , every oracle call made by **Lemma B.43** is a valid call to $\mathcal{O}_{(h_S, \kappa', \lambda')}^{\text{Proj}}$.

If **Lemma B.43** outputs an implicit flow-path decomposition

$$F_S = (L_i, \lambda_i, s_i, t_i)_{i \in [k]},$$

then each path goes from some vertex in P to some vertex in Q . We interpret the pairs (s_i, t_i) with capacities λ_i as the matching returned by $\mathcal{O}_{\text{match}}$, and store the corresponding path realizations in $G^\circ[S']$.

If **Lemma B.43** instead outputs a moving cut Q_S , then we declare S to be *bad*, terminate the cut-matching game on S , and define

$$\tilde{A}_S(v) := 0 \quad \text{for all } v \in S.$$

If S is not bad, then **Lemma B.42** returns a node-weighting

$$\tilde{A}_S \preceq A|_S$$

such that

$$|\tilde{A}_S| \geq (1 - O(\varepsilon_{\text{cm}} \log^2 n)) \cdot |A|_S,$$

and such that the union W_S of the matchings produced during the game is an $\Omega(1)$ -expander with respect to $\tilde{A}_S$.

After this is done for every cluster in every clustering of $\mathcal{N}$, we inspect the clusterings. If there exists some clustering $\mathcal{S} \in \mathcal{N}$ such that the total A -mass of its bad clusters is at least

$$\beta |A|,$$

then for every bad cluster S in this clustering we add the returned cut Q_S to the cumulative length increase, namely

$$\Lambda(e) \leftarrow \Lambda(e) + h_S \sigma \cdot Q_S(e) \quad \text{for all } e \in E,$$

and start the next round.

Otherwise the algorithm terminates. For every clustering $\mathcal{S}_j \in \mathcal{N}$, define

$$A_j(v) := \tilde{A}_{\mathcal{S}_j(v)}(v),$$

where $\mathcal{S}_j(v)$ is the unique cluster of $\mathcal{S}_j$ containing v . Finally define

$$A'(v) := \min_j A_j(v).$$

Correctness (associated routing algorithm). We show that A' is (h, s) -length C -constraint ϕ -expanding and give an associated routing algorithm.

Let D be any A' -respecting h -length symmetrically constrained single-commodity demand on the final graph $G - \Lambda$, together with a witness

$$D = D_1 + \dots + D_k,$$

where each D_i is symmetric h -length and the witness is cumulatively A' -respecting, namely

$$\sum_{i=1}^k |D_i(v)| \leq A'(v) \quad \text{for every } v \in V.$$

For each witness demand D_i , choose any vertex u_i in the support of D_i . Since D_i is symmetric h -length, for every vertex v in the support of D_i we have

$$\text{dist}_{G-\Lambda}(u_i, v) \leq h \quad \text{and} \quad \text{dist}_{G-\Lambda}(v, u_i) \leq h.$$

Hence

$$\widehat{\text{dist}}_{G-\Lambda}(u_i, v) \leq 2h.$$

Since the final FatLDD cover has covering radius $2h$, for every i there exists a cluster S_i in some clustering $\mathcal{S}_{j(i)} \in \mathcal{N}$ such that

$$\text{supp}(D_i) \subseteq \text{ball}_{\widehat{\text{dist}}_{G-\Lambda}}(u_i, 2h) \subseteq S_i.$$

For each cluster S appearing in this assignment, let

$$D_S := \sum_{i: S_i=S} D_i.$$

Then the demands D_S sum to D . Moreover, each D_S is supported on S , and for every $v \in S$,

$$|D_S(v)| = \left| \sum_{i: S_i=S} D_i(v) \right| \leq \sum_{i: S_i=S} |D_i(v)| \leq \sum_{i=1}^k |D_i(v)| \leq A'(v) \leq \tilde{A}_S(v).$$

Thus D_S is $\tilde{A}_S$ -respecting.

Because S is not bad, the cut-matching game produced an $\Omega(1)$ -expander W_S with respect to $\tilde{A}_S$. By standard routing on constant expanders, one can route D_S inside W_S with $O(1)$ congestion and with paths using $O(\log n)$ expander edges.

Each edge of W_S came from one successful call to [Lemma B.43](#), and is therefore realized by a path in $G[S']$ of length at most

$$h_S s_0 \leq \Gamma h \cdot 256 \log n,$$

and of congestion

$$O\left(\frac{\kappa \kappa' \log(1/\varepsilon_{\text{cm}})}{\phi_0}\right) = O\left(\frac{1}{\omega \phi}\right)$$

by the definition of ϕ_0 . Replacing each expander edge by a flow path (this can be done cumulatively for all paths using [Theorem B.9](#) in nearly linear time) lifts the routing of D_S from W_S to the original graph. Since the expander paths use only $O(\log n)$ edges, the lifted routing has length

$$O(\Gamma h \log^2 n) \leq hs$$

by the choice of s .

Within any fixed clustering $\mathcal{S}_j$, the witness sets S' are pairwise disjoint, so routings inside different clusters of $\mathcal{S}_j$ do not interfere. Across different clusterings there are only ω possibilities. Hence after summing the lifted routings over all chosen clusters S , the total congestion is

$$O(1) \cdot O\left(\frac{1}{\omega\phi}\right) \cdot \omega = O(1/\phi).$$

After adjusting constants, this gives a routing with congestion at most $1/\phi$ and length bound hs . Thus A' is (h, s) -length C -constraint ϕ -expanding.

Correctness (upper bound on $|A| - |A'|$). Fix one clustering $\mathcal{S}_j$ in the final round. By termination, the bad clusters in $\mathcal{S}_j$ have total A -mass strictly less than $\beta|A|$. For every good cluster $S \in \mathcal{S}_j$, the cut-matching game loses only an $O(\varepsilon_{\text{cm}} \log^2 n)$ fraction of the A -mass on S . Therefore

$$|A| - |A_j| \leq \beta|A| + O(\varepsilon_{\text{cm}} \log^2 n) \cdot |A| = O\left(\frac{\phi}{\omega}\right) \cdot |A|.$$

Since

$$A'(v) = \min_j A_j(v),$$

we have pointwise

$$A(v) - A'(v) \leq \sum_j (A(v) - A_j(v)).$$

Summing over all vertices gives

$$|A| - |A'| \leq \sum_j (|A| - |A_j|) \leq \omega \cdot O\left(\frac{\phi}{\omega}\right) \cdot |A| = O(\phi) \cdot |A| \leq \Delta\phi \cdot |A|.$$

Correctness (upper bound on $|C|$). Fix a bad cluster S that appears in some round, and let G° denote the current graph in that round. The invocation of [Lemma B.43](#) on the induced subgraph $G^\circ[S']$ returns a moving cut Q_S that is

$$(h_S, \sigma)\text{-length } \phi_0\text{-sparse}$$

with respect to A_D , where $A_D(v) = |D(v)|$ and $A_D \preceq A|_S \preceq A$. By [Definition B.31](#), there exists an A_D -respecting h_S -length symmetric demand $\widehat{D}$ on $G^\circ[S']$ such that

$$\text{spars}_{h_S\sigma}(Q_S, \widehat{D}; G^\circ[S']) \leq \phi_0.$$

Since $A_D \preceq A$, the demand $\widehat{D}$ is also A -respecting. Moreover, $\widehat{D}$ is supported on pairs inside S , so by [Claim B.45](#), every pair that is $h_S\sigma$ -separated by Q_S inside $G^\circ[S']$ is also $h_S\sigma$ -separated by Q_S inside the whole graph G° . Hence

$$\text{sep}_{h_S\sigma}(Q_S, \widehat{D}; G^\circ) \geq \text{sep}_{h_S\sigma}(Q_S, \widehat{D}; G^\circ[S']),$$

and therefore

$$\text{spars}_{h_S\sigma}(Q_S, \widehat{D}; G^\circ) \leq \phi_0.$$

Equivalently,

$$\text{spars}_{(h_S, \sigma)}(Q_S, A; G^\circ) \leq \phi_0.$$

Thus every bad cut is sparse in the current whole graph with respect to the same global node-weighting A .

We now order all bad cuts added by the algorithm into a single sequence

$$Q_1, Q_2, \dots, Q_q$$

as follows: we first order them by rounds, and within each round we order the bad clusters of the chosen clustering arbitrarily. For each r , let

$$G_r := G - \sum_{j < r} Q_j$$

denote the graph obtained after applying the previous cuts in this order.

We claim that each Q_r is (h_r, σ) -length ϕ_0 -sparse with respect to A in G_r , where h_r is the diameter bound of the corresponding cluster. This is immediate for cuts coming from different rounds, since G_r is exactly the graph in which the cut is produced, possibly with additional earlier cuts already applied.

The only point to check is when several bad cuts come from the same round. Suppose Q_r comes from a bad cluster S_r with separation witness S'_r , and let Q_j be any earlier cut from the same round. Then Q_j comes from another bad cluster in the same clustering, so its witness set is disjoint from S'_r . Since each cut is produced on the induced subgraph of its own witness set, Q_j is supported on edges of S'_j and therefore has disjoint support from Q_r . More importantly, by [Claim B.45](#), every path of length at most $h_r \sigma$ between vertices of S_r is contained in S'_r . Hence adding the earlier same-round cuts does not affect:

1. the fact that the witness demand $\widehat{D}_r$ is h_r -length, and
2. the value of $\text{sep}_{h_r \sigma}(Q_r, \widehat{D}_r)$.

Therefore the same witness demand still certifies that Q_r is (h_r, σ) -length ϕ_0 -sparse in G_r .

We may therefore apply [Lemma B.33](#) to the sequence $Q_1, \dots, Q_q$ with $\alpha = 1$. This yields

$$\sum_{r=1}^q |Q_r| \leq O(\phi_0 \log^2 n) \cdot |A|.$$

Substituting the definition of ϕ_0 gives

$$\sum_{r=1}^q |Q_r| \leq O(\omega \kappa \kappa' L_\phi \log^2 n) \cdot \phi |A| \leq \Delta \phi \cdot |A|.$$

Let Λ be the cumulative length increase added by all bad cuts during the algorithm, i.e.,

$$\Lambda(e) = \sum_{r=1}^q h_r \sigma \cdot Q_r(e).$$

We define the output cut C to be the hs -length moving cut satisfying

$$hs \cdot C(e) = \Lambda(e), \quad \text{equivalently} \quad C(e) := \frac{\Lambda(e)}{hs} = \frac{1}{hs} \sum_{r=1}^q h_r \sigma \cdot Q_r(e).$$

Since $h_r \leq \Gamma h$ for every r and $\sigma = O(\log n)$, while $s = c_s \Gamma \log^2 n$, choosing c_s sufficiently large ensures that

$$h_r \sigma \leq hs \quad \text{for every } r.$$

Thus C is indeed a valid hs -length moving cut. Finally,

$$|C| = \sum_e U(e) C(e) = \frac{1}{hs} \sum_{r=1}^q h_r \sigma \cdot |Q_r| \leq \sum_{r=1}^q |Q_r| \leq \Delta \phi \cdot |A|.$$

This proves the desired upper bound on $|C|$.

Complexity. We first show the number of rounds by the algorithm. Consider any non-terminal round. By definition, there exists some clustering $\mathcal{S}$ such that the bad clusters in $\mathcal{S}$ have total A -mass at least $\beta|A|$.

For every such bad cluster S , the corresponding call to [Lemma B.43](#) is made on a vector $\mathbf{d}' \leq A|_S$ with $\mathbf{d}'(V) \geq |A|_S/2$. Hence the returned cut has size at least

$$\Omega(\phi_0 \varepsilon_{\text{cm}} \cdot |A|_S).$$

Therefore the total cut added in this round has size at least

$$\Omega(\phi_0 \varepsilon_{\text{cm}} \beta) \cdot |A|.$$

Since the total cut size over the whole execution is at most $\Delta \phi |A|$, the number of rounds is at most

$$O\left(\frac{\Delta \phi}{\phi_0 \varepsilon_{\text{cm}} \beta}\right) = O\left(\frac{\omega^2 \log^4 n}{\phi^2}\right) = \frac{n^{O(\varepsilon)}}{\phi^2}.$$

Fix one round. For any fixed clustering $\mathcal{S}$, the witness sets S' are pairwise disjoint, so the total size of all induced subgraphs $G^\circ[S']$ over clusters $S \in \mathcal{S}$ is $O(m)$. Each cluster runs one cut-matching game, which makes $O(\log^2 n)$ calls to $\mathcal{O}_{\text{match}}$. Each such call invokes [Lemma B.43](#), and [Lemma B.43](#) makes

$$O\left(\frac{L_\phi}{\phi_0}\right)$$

calls to $\mathcal{O}_{(hs, \kappa', \lambda')}^{\text{Proj}}$ and $\mathcal{O}_{\kappa, \lambda}^{\text{PDMCF}}$.

Hence over one clustering, the total size of all oracle instances is

$$O\left(m \log^2 n \cdot \frac{L_\phi}{\phi_0}\right),$$

and over all ω clusterings in the cover, one round contributes total oracle-instance size

$$O\left(\omega m \log^2 n \cdot \frac{L_\phi}{\phi_0}\right) = O\left(\frac{m \log^2 n}{\phi}\right).$$

Multiplying by the number of rounds gives total oracle-instance size

$$O\left(\frac{\omega^2 m \log^6 n}{\phi^3}\right) = \frac{m \cdot n^{O(\varepsilon)}}{\phi^3}.$$

Under the standard regime $n = O(m)$, this is

$$\frac{m^{1+O(\varepsilon)}}{\phi^3}.$$

The same bound also dominates the additional non-oracle work, including the FatLDD construction and the bookkeeping inside the cut-matching games.

Finally, in one round there are at most $O(\omega n \log^2 n)$ invocations of [Lemma B.43](#), and each such invocation makes $O(L_\phi/\phi_0)$ calls to the two underlying oracles. Multiplying again by the number of rounds yields

$$O\left(\frac{\omega^2 n \log^6 n}{\phi^3}\right) = \frac{n^{1+O(\varepsilon)}}{\phi^3}$$

total calls to $\mathcal{O}_{(hs, \kappa', \lambda')}^{\text{Proj}}$ and $\mathcal{O}_{\kappa, \lambda}^{\text{PDMCF}}$. This proves the lemma. $\square$

B.6.3 LC-Expander Hierarchy

In this section, we prove the following lemma that shows how to construct a LC-expander Hierarchy as defined in [Definition B.35](#).

Lemma B.36. *Let $G = (V, E)$ be a directed graph with positive integral capacities and lengths, let $h \geq 1$, let $\varepsilon \in (\log \log n / \log n, 1)$, $\phi \in (1/\text{poly}(n), \frac{1}{\kappa n^{c\varepsilon}})$ for some sufficiently large constant c , then there exists*

$s = (\log n)^{O(1/\varepsilon)}$ and an algorithm that outputs a $\leq h$ -length (ϕ, s) -expanding t -layer weak LC-expander hierarchy together with efficient associated routing algorithms, where

$$t = O\left(\frac{\log m}{\log(1/(\kappa\kappa'n^{O(\varepsilon)}\phi))}\right).$$

The algorithm makes $\frac{m^{1+O(\varepsilon)}}{\phi^3}$ -efficient calls to $\mathcal{O}_{(hs,\kappa',\lambda')}^{\text{Proj}}$ and $\mathcal{O}_{\kappa,\lambda}^{\text{PDMCF}}$ where $\lambda, \lambda' \in [1, 2]$

Proof. Let

$$\mathcal{H} := \{2^0, 2^1, \dots, 2^{\lceil \log_2 h \rceil}\}$$

be the set of dyadic scales up to h . Since all edge lengths are positive integers and $h \leq \text{poly}(n)$, we have

$$|\mathcal{H}| = O(\log n).$$

We will construct cuts

$$C_1, \dots, C_t$$

together with auxiliary residual cuts

$$C'_1, C'_2, \dots, C'_t.$$

Initially, set

$$C'_1(e) := 1 \quad \text{for all } e \in E.$$

For every $i \geq 1$, define

$$A_i := \deg_{C'_i}.$$

Note that

$$|A_i| = 2|C'_i|.$$

We also fix

$$\Delta_0 := \kappa\kappa'n^{O(\varepsilon)},$$

where the hidden constant is large enough to absorb every logarithmic loss arising below, including the $O(\log n)$ dyadic scales. Since

$$\phi < \frac{1}{\kappa\kappa'n^{c\varepsilon}},$$

for a sufficiently large constant c we may assume

$$4\Delta_0\phi \leq \frac{1}{2}.$$

The iterative step. Suppose C'_i has already been defined.

If

$$|A_i| \leq 1,$$

then we terminate the construction and set

$$t := i, \quad C_t := C'_t.$$

We will show below that this last layer is trivially expanding.

Otherwise, assume

$$|A_i| > 1.$$

Step 1: make A_i expanding simultaneously at all scales. For every dyadic scale $\eta \in \mathcal{H}$, apply [Lemma B.44](#) to the graph G , the length bound η , the parameter ϕ , the parameter ε , and the node-weighting A_i . This returns an ηs -length moving cut $D_{i,\eta}$ and a node-weighting

$$A'_{i,\eta} \preceq A_i$$

such that:

1. $A'_{i,\eta}$ is (η, s) -length $D_{i,\eta}$ -constraint ϕ -expanding;
- 2.

$$|D_{i,\eta}|, |A_i| - |A'_{i,\eta}| \leq \Delta_0 \phi |A_i|.$$

We unify these guarantees over all scales by defining

$$A'_i(v) := \min_{\eta \in \mathcal{H}} A'_{i,\eta}(v) \quad \text{for all } v \in V,$$

and

$$C''_i(e) := 2 \max_{\eta \in \mathcal{H}} D_{i,\eta}(e) \quad \text{for all } e \in E.$$

Since $|\mathcal{H}| = O(\log n)$ and all logarithmic factors are absorbed into $n^{O(\epsilon)}$, after enlarging the hidden constant in Δ_0 if necessary we still have

$$|C''_i|, |A_i| - |A'_i| \leq \Delta_0 \phi |A_i|.$$

We claim that A'_i is $(\leq h, s)$ -length C''_i -constraint ϕ -expanding. Indeed, let $h' \leq h$, and choose $\eta \in \mathcal{H}$ such that

$$h' \leq \eta \leq 2h'.$$

If we interpret C''_i as an h' -length moving cut, then for every edge e ,

$$h' C''_i(e) \geq 2h' D_{i,\eta}(e) \geq \eta D_{i,\eta}(e).$$

Therefore every demand that is h' -length in $G - C''_i$ is also η -length in $G - D_{i,\eta}$. Since $A'_i \preceq A'_{i,\eta}$, the routing algorithm for $A'_{i,\eta}$ routes every A'_i -respecting h' -length symmetrically constrained demand with length bound at most

$$\eta s \leq 2h' s.$$

Absorbing the constant factor 2 into the hidden constant in the definition of s proves the claim.

Step 2: realize most of A'_i by a subcut of C'_i . Let $C_i \preceq C'_i$ be a maximal subcut satisfying

$$\deg_{C_i} \preceq A'_i.$$

which can be found by checking each edge and increase it C_i value until one of its end point get saturated in A'_i . Let

$$R_i := C'_i - C_i$$

be the leftover cut.

Since $\deg_{C_i} \preceq A'_i$ and A'_i is $(\leq h, s)$ -length C''_i -constraint ϕ -expanding, the same routing algorithm certifies that $\deg_{C_i}$ is also $(\leq h, s)$ -length C''_i -constraint ϕ -expanding.

We now bound the remainder. By maximality of C_i , every edge e with $R_i(e) > 0$ has at least one endpoint v with

$$\deg_{C_i}(v) = A'_i(v).$$

Charge the quantity $U(e) \cdot R_i(e)$ to one such saturated endpoint. For any fixed vertex v , the total charge sent to v is at most

$$\deg_{R_i}(v) = A_i(v) - \deg_{C_i}(v) \leq A_i(v) - A'_i(v).$$

Summing over all vertices yields

$$|R_i| \leq |A_i| - |A'_i| \leq \Delta_0 \phi |A_i|.$$

Step 3: define the next residual cut. Set

$$C'_{i+1} := C''_i + R_i.$$

Then

$$|C'_{i+1}| \leq |C''_i| + |R_i| \leq 2\Delta_0 \phi |A_i| = 4\Delta_0 \phi |C'_i|.$$

Termination and the number of layers. Because $|C'_1| = \text{poly}(n)$ and

$$|C'_{i+1}| \leq 4\Delta_0\phi|C'_i|,$$

after

$$t = O\left(\frac{\log m}{\log(1/(4\Delta_0\phi))}\right) = O\left(\frac{\log m}{\log(1/(\kappa\kappa'n^{O(\epsilon)}\phi))}\right)$$

iterations we reach an index t with

$$|C'_t| \leq \frac{1}{2},$$

and hence

$$|A_t| = 2|C'_t| \leq 1.$$

This is exactly our stopping rule.

The last layer is expanding. We now verify that the final layer $C_t = C'_t$ is $(\leq h, s)$ -length 0-constraint ϕ -expanding.

Fix any $h' \leq h$, and let D be any $\deg_{C_t}$ -respecting h' -length symmetrically constrained single-commodity demand on G , with witness clustering

$$\{V_1, \dots, V_k\}.$$

For each cluster V_a , choose a root $r_a \in V_a$. Because V_a has weak diameter at most h' , every vertex of V_a can send flow to r_a and receive flow from r_a along paths of length at most h' . Route the demand inside each cluster by first sending all positive mass to the root and then sending it from the root to the negative vertices. This produces a routing of length at most

$$2h' \leq h's$$

since $s \geq 2$.

Moreover, since D is $\deg_{C_t}$ -respecting,

$$|D| := \sum_{v \in V} |D(v)| \leq \sum_{v \in V} \deg_{C_t}(v) = |A_t| \leq 1.$$

Thus the total amount of flow routed in this construction is at most $|D| \leq 1$. As every edge has capacity at least 1, the congestion on every edge is at most $1 \leq 1/\phi$. Therefore $\deg_{C_t}$ is $(\leq h, s)$ -length 0-constraint ϕ -expanding.

Correctness of the hierarchy. We show that

$$C_1, \dots, C_t$$

is a $\leq h$ -length (ϕ, s) -expanding weak LC-ED hierarchy.

First, we prove the covering property

$$\sum_{i=1}^t C_i(e) \geq 1 \quad \text{for every } e \in E.$$

In fact, we prove by backward induction that for every $i \in [t]$,

$$\sum_{j \geq i} C_j \succeq C'_i.$$

The base case $i = t$ is immediate since $C_t = C'_t$. For the inductive step,

$$\sum_{j \geq i} C_j = C_i + \sum_{j \geq i+1} C_j \succeq C_i + C'_{i+1} = C_i + C''_i + R_i = C'_i + C''_i \succeq C'_i.$$

Applying this with $i = 1$ and using $C'_1 \equiv 1$ proves the required covering property.

Next, fix any $i < t$ and any $h' \leq h$. By the claim above, $\deg_{C_i}$ is $(\leq h, s)$ -length C''_i -constraint ϕ -expanding. Also, from the covering property just proved,

$$C_{>i} := \sum_{j>i} C_j \succeq C'_{i+1} \succeq C''_i.$$

If both cuts are interpreted as h' -length moving cuts, then $C_{>i}$ imposes at least as much length increase as C''_i . Therefore every demand that is h' -length in $G - C_{>i}$ is also h' -length in $G - C''_i$. So the same routing algorithm certifies that $\deg_{C_i}$ is $(\leq h, s)$ -length $C_{>i}$ -constraint ϕ -expanding.

For $i = t$, the future cut is empty, and we already proved that $\deg_{C_t}$ is $(\leq h, s)$ -length 0-constraint ϕ -expanding. Hence $C_1, \dots, C_t$ is a $\leq h$ -length (ϕ, s) -expanding t -layer weak LC-ED hierarchy.

Associated routing algorithms. For every layer $i < t$, we simply use the routing algorithm inherited from the node-weighting A'_i and the domination

$$\deg_{C_i} \preceq A'_i.$$

For the last layer $i = t$, we use the explicit root-routing algorithm described above. All these routing algorithms are efficient.

Complexity. Each iteration performs $|\mathcal{H}| = O(\log n)$ calls to [Lemma B.44](#). By [Lemma B.44](#), one such call makes oracle calls to $\mathcal{O}_{(hs, \kappa', \lambda')}^{\text{Proj}}$ and $\mathcal{O}_{\kappa, \lambda}^{\text{PDMCF}}$ on instances of total size

$$\frac{m^{1+O(\varepsilon)}}{\phi^3},$$

and the same additional time. Since

$$t = O\left(\frac{\log m}{\log(1/(\kappa \kappa' n^{O(\varepsilon)} \phi))}\right) \leq O(\log n),$$

and all logarithmic factors are absorbed by $n^{O(\varepsilon)}$, the overall complexity remains. $\square$

B.7 DAG Projection Construction via LC Expander Hierarchy

In this section, we will show the following lemma about constructing a h -length flow-cost-preserving DAG projection.

Lemma B.46. *There is an algorithm given a directed graph $G = (V, E)$ with non-negative polynomially bounded integral edge capacities and lengths, a length bound h , parameters $\varepsilon, \varepsilon_p, \varepsilon_f \in (0, 1/\log n)$, $\delta \in (1/\log n, 1)$ and $z, \kappa_p, \kappa_f \in \mathbb{N}_+$, with oracle calls to $\mathcal{O}_{h/z, \kappa_p, 1+\varepsilon_p}^{\text{Proj}}$ and $\mathcal{O}_{\kappa_f, 1+\varepsilon_f}^{\text{PDMCF}}$, write*

$$S(n, \delta, \varepsilon) = n^{O(\delta)} \cdot (1/\varepsilon)^{O(1/\delta^5)}$$

output a DAG projection of size

$$m \cdot S(n, \delta, \varepsilon)$$

with a (κ, λ) -projection algorithm where

$$\kappa = \kappa_f \kappa_p \cdot S(n, \delta, \varepsilon) \quad \lambda = 1 + O(\varepsilon + \varepsilon_p + \varepsilon_f)/\delta^2$$

The projection algorithm takes time $zm \cdot S(n, \delta, \varepsilon)$ time. The construction algorithm makes $z \cdot S(n, \delta, \varepsilon)$ -efficient oracle calls.

[Lemma B.46](#) will imply [Theorem B.23](#) through recursive construction. Both of them will be proved in [Section B.7.5](#).

B.7.1 An Algorithm for Constructing the DAG Projection

In this section we describe a recursive algorithm for constructing an h -length flow-cost-preserving DAG projection.

We fix two global parameters:

- a stretch parameter $\varepsilon \in (0, 1/\log n)$, and
- a recursion parameter $\delta \in (0, 1)$.

Let

$$\sigma := n^\delta.$$

We also write

$$s = (\log n)^{O(1/\delta)} \quad \text{and} \quad t = O(1/\delta)$$

for the length slack and the number of layers guaranteed by [Lemma B.36](#) for our chosen parameters.

The construction assumes oracle access to

$$\mathcal{O}_{hs, \kappa_p, 1+\varepsilon_p}^{\text{Proj}} \quad \text{and} \quad \mathcal{O}_{\kappa_f, 1+\varepsilon_f}^{\text{PDMCF}},$$

where

$$0 \leq \varepsilon_p, \varepsilon_f \leq 0.1.$$

Notice that $\mathcal{O}_{hs, \kappa_p, 1+\varepsilon_p}^{\text{Proj}}$ is a stronger oracle than the $\mathcal{O}_{h/z, \kappa_p, 1+\varepsilon_p}^{\text{Proj}}$ claimed in [Lemma B.46](#), but this is fine: we can use [Lemma B.55](#) to get the stronger oracle from the weaker one. This will be elaborated in [Section B.7.5](#).

A single global weak LC-ED hierarchy. At the beginning of the algorithm we apply [Lemma B.36](#) to the original graph G , with length parameter h , with the lemma parameter ε set to δ , and with

$$\phi := \frac{1}{\kappa_f \kappa_p n^{c\delta}}$$

for a sufficiently large constant c . This yields a $\leq h$ -length (ϕ, s) -expanding t -layer weak LC-ED hierarchy

$$C_1, \dots, C_t.$$

For a vertex set $S \subseteq V$, when we write

$$G[S] - C_{>i}, \quad \text{where } C_{>i} := \sum_{j>i} C_j,$$

we mean the induced subgraph $G[S]$ equipped with the length function obtained by restricting the moving cut $C_{>i}$ to the edges of $G[S]$.

The construction is recursive. For every $S \subseteq V$ and $i \in \{0, 1, \dots, t\}$, let

$$\text{PROJECTION}(S, i)$$

denote the recursive routine, and note that the top-level call is

$$\text{PROJECTION}(V, t).$$

Intuitively, $\text{PROJECTION}(S, i)$ constructs a DAG projection for the induced subproblem on S , after the cut $C_{>i}$ has already been applied.

Base cases. The recursive routine $\text{PROJECTION}(S, i)$ has the following base cases.

- If $i = 0$, return the edgeless graph on the vertex set S .
- If $|S| = 1$, return the single-vertex graph.

We now describe $\text{PROJECTION}(S, i)$ for $i > 0$ and $|S| > 1$.

Path covers at all dyadic scales. For each dyadic scale $h' \leq h$, let

$$G_{S,i}^{(h')} := G[S] - C_{>i},$$

where $C_{>i}$ is interpreted as an h' -length moving cut. Let

$$q = O(\log h) = O(\log n)$$

denote the number of dyadic scales.

Applying [Lemma B.12](#) to $G_{S,i}^{(h')}$ with appropriate parameters, we can get an induced

$$\frac{h's}{\varepsilon}\text{-path cover}$$

with SCC diameter at most h' , such that the width is at most $\omega = (s/\varepsilon) \cdot \exp(O(1/\delta^3)) \cdot n^{O(\delta^3)}$.

We denote this path cover by

$$P^{(h')}.$$

The graph $P^{(h')}$ is not necessarily a DAG. However, every SCC of $P^{(h')}$ maps to an induced subgraph of $G_{S,i}^{(h')}$, and every such SCC has weak diameter at most h' in $G_{S,i}^{(h')}$. For notational convenience, if C is an SCC of $P^{(h')}$, then we also write C for its image inside G .

Light and heavy SCCs. Let C be an SCC of $P^{(h')}$.

- If

$$|V(C)| \leq |S|/\sigma,$$

then C is called *light*, and we recursively call

$$\text{PROJECTION}(V(C), t)$$

to obtain a DAG projection D_C .

- If

$$|V(C)| > |S|/\sigma,$$

then C is called *heavy*, and we recursively call

$$\text{PROJECTION}(V(C), i - 1)$$

to obtain a DAG projection D_C .

Assembling one DAG layer. For each SCC C of $P^{(h')}$, let D_C be the recursively constructed DAG projection of C . Since D_C is a DAG equipped with a projection map to $V(C)$, for every $u \in V(C)$ we denote by

$$u_C^{\text{first}} \quad \text{and} \quad u_C^{\text{last}}$$

the first and last lifted copies of u in the topological order of D_C .

We now replace each SCC C of $P^{(h')}$ by its DAG projection D_C . For every edge $(u, v) \in E(P^{(h')})$ whose endpoints lie in two different SCCs, say $u \in C_u$ and $v \in C_v$ with $C_u \neq C_v$, we add an edge

$$(u_{C_u}^{\text{last}}, v_{C_v}^{\text{first}})$$

with the same length and capacity as (u, v) . Since the condensation of $P^{(h')}$ is a DAG and each D_C is a

DAG, the resulting graph is again a DAG. We denote it by

$$D^{(h')}.$$

Duplicating the DAG. Create two disjoint copies $D_1^{(h')}$ and $D_2^{(h')}$ of $D^{(h')}$. If $x \in V(D^{(h')})$, then x_1 and x_2 denote its copies in the two DAGs.

For each heavy SCC C , create two fresh interface vertices

$$v_C, \quad v'_C.$$

For every $u \in V(C)$, add the interface edges

$$((u_C^{\text{last}})_1, v_C) \quad \text{and} \quad (v'_C, (u_C^{\text{first}})_2),$$

each of length $h's$, and each of capacity $\deg_{C_i}(u)$.

Auxiliary graph for heavy SCCs. Construct an auxiliary graph $G_{S,i}^{\text{aux},(h')}$ as follows. Start from $G[S] - C_{>i}$. For every heavy SCC C , add a fresh star vertex s_C , and for every $u \in V(C)$ add the two zero-length edges

$$(s_C, u) \quad \text{and} \quad (u, s_C),$$

each with capacity $\deg_{C_i}(u)$.

Invoke the oracle $\mathcal{O}_{hs, \kappa_p, 1+\varepsilon_p}^{\text{Proj}}$ on $G_{S,i}^{\text{aux},(h')}$ and obtain a DAG projection of this auxiliary graph.

For every ordered pair of heavy SCCs (A, B) and every power-of-two demand value 2^j , apply [Corollary B.72](#) to the projected auxiliary graph with an $s_A - s_B$ demand of value 2^j . For each such call, store both:

- the returned flow $g_{A,B,j}^{\text{pow}}$ on $G_{S,i}^{\text{aux},(h')}$, and
- the returned moving cut $C_{A,B,j}^{\text{pow}}$ on $G_{S,i}^{\text{aux},(h')}$.

Let

$$B_0 := \sum_{e \in E(G)} U(e) \cdot |\ell(e)|, \quad R := O(\log n / \varepsilon).$$

The specific value of R will be set later. For every ordered pair of heavy SCCs (A, B) and every $r \in [R]$, let

$$L_r := (1 + \varepsilon)^{r-1}.$$

Among all stored power-of-two flows from s_A to s_B , let

$$\mu_{A,B,r}$$

be the largest value 2^j for which the returned flow has average cost at most L_r . If no such value exists, set $\mu_{A,B,r} := 0$. We let R to be the first time $\mu_{A,B,R}$ reach the maximum value, i.e., the maximum amount of $s_A - s_B$ flow is at most $\mu_{A,B,R}$ under the power 2 scales. In the case of there exists a flow with value 2^j and average cost at most L_r , denote the stored flow $g_{A,B,j}^{\text{pow}}$ by $g_{A,B,r}^*$, and the store cut $C_{A,B,j+1}^{\text{pow}}$ by $C_{A,B,r}^*$ in case $r < R$. Notice that $C_{A,B,r}^*$ corresponds to a flow with average cost at least L_r , and size at most 2 times the flow value $g_{A,B,r}^*$.

Bridge edges between heavy SCCs. For every ordered pair of heavy SCCs (A, B) and every $r \in [R]$ with $\mu_{A,B,r} > 0$, add a bridge edge

$$(v_A, v'_B)$$

of length L_r and capacity

$$2\mu_{A,B,r}/\varepsilon.$$

The DAG at scale h' . Let $\widehat{D}^{(h')}$ denote the DAG obtained by taking the union of the two copies $D_1^{(h')}, D_2^{(h')}$, all interface vertices, and all bridge edges added above.

The final output and the projection map. The output of $\text{PROJECTION}(S, i)$ is the disjoint union

$$D_{S,i}^* := \bigsqcup_{h'} \widehat{D}^{(h')}.$$

Fix any topological order of $D_{S,i}^*$. For every $u \in S$, let

$$x_1 \prec x_2 \prec \cdots \prec x_k$$

be all vertices of $D_{S,i}^*$ whose projection is u , listed in this topological order. Add zero-length, infinite-capacity edges

$$(x_1, x_2), (x_2, x_3), \dots, (x_{k-1}, x_k).$$

The projection map π on $D_{S,i}^*$ is defined as follows.

- On every recursively constructed DAG D_C , use the recursively returned projection map.
- On the two copies $D_1^{(h')}, D_2^{(h')}$, inherit the same projection map.
- For every heavy SCC C , define

$$\pi(v_C) = \pi(v'_C) := \perp.$$

B.7.2 Size and Complexity of the Construction

We now calculate the explicit size of the output DAG and the non-oracle work of the recursive construction.

Lemma B.47 (Size of the recursive projection). *For the top-level recursive call $\text{PROJECTION}(V, t)$, the returned DAG satisfies*

$$|D_{V,t}^*| \leq m^{1+O(\delta)} \cdot (1/\varepsilon)^{O(1/\delta^5)}$$

Proof. We will upper bound the size of the graph $D_{S,i}^*$ returned by a general recursive call $\text{PROJECTION}(S, i)$, write down the recurrence, and then solve it.

Fix one dyadic scale $h' \leq h$.

First, the path cover $P^{(h')}$ has size $\widetilde{O}(\omega^2 |E(G[S])|)$. After duplicating the DAG and adding the inter-SCC edges, this remains $\widetilde{O}(\omega^2 |E(G[S])|)$.

Second, the total number of lifted copies of original vertices in $P^{(h')}$ is at most $\omega |S|$, so the interface vertices and interface edges contribute at most $\widetilde{O}(\omega |S|)$ size.

Third, every heavy SCC contains more than $|S|/\sigma$ original vertices, while the total multiplicity of original vertices over all SCC images at this scale is at most $\omega |S|$. Hence the number of heavy SCCs at this scale is at most $\omega \sigma$. Therefore the bridge edges at this scale contribute at most

$$O((\omega \sigma)^2 R).$$

Thus the explicit graph material created in the current call at one scale is

$$\widetilde{O}(\omega^2 |E(G[S])| + \omega |S| + (\omega \sigma)^2 R).$$

Summed over all q dyadic scales, this becomes

$$q \cdot \widetilde{O}(\omega^2 |E(G[S])| + \omega |S| + (\omega \sigma)^2 R).$$

It remains to account for the recursive SCC gadgets. At one fixed scale, every original vertex of S appears in at most ω SCC images. Each light SCC C satisfies $|V(C)| \leq |S|/\sigma$, and its recursive child is $\text{PROJECTION}(V(C), t)$. Each heavy SCC C satisfies $|V(C)| \leq |S|$, and its recursive child is $\text{PROJECTION}(V(C), i-1)$. Therefore, if we let $\Lambda(\eta, i)$ denote the maximum size of $D_{S,i}^*$ over all $|S| \leq \eta$, for

$\eta > 1, i > 0$,

$$\Lambda(\eta, i) \leq q \cdot \tilde{O}(\omega^2 m + \omega\eta + (\omega\sigma)^2 R) + q\omega \cdot \left(\sum_{S' \text{ is light}} \Lambda(|S'|, t) + \Lambda(\eta, i-1) \right).$$

Solving this recurrence and substituting

$$\begin{aligned} \sigma &= n^\delta, & t &= O(1/\delta), & q &= O(\log n), \\ \omega &= (s/\varepsilon) \cdot \exp(O(1/\delta^3)) \cdot n^{O(\delta^3)}, & s &= (\log n)^{O(1/\delta)}, & R &= O(\log n/\varepsilon), \end{aligned}$$

gives

$$\Lambda(n, t) \leq m^{1+O(\delta)} \cdot \varepsilon^{-O(1/\delta^5)}.$$

which proves the lemma (remember that $\varepsilon < 1/\log n$, and $m \geq n$ w.o.l.g.). $\square$

Lemma B.48 (Construction complexity). *The total non-oracle running time of constructing the top-level DAG $\text{PROJECTION}(V, t)$ and the cumulative graph size the oracles $\mathcal{O}_{hs, \kappa_p, 1+\varepsilon_p}^{\text{Proj}}$, $\mathcal{O}_{\kappa_f, 1+\varepsilon_f}^{\text{PDMCF}}$ runs on is at most $m^{1+O(\delta)} \cdot (1/\varepsilon)^{O(1/\delta^5)}$.*

Proof. The proof is identical in form to [Lemma B.47](#). Let $T(\eta, i)$ denote the maximum non-oracle running time of a recursive call $\text{PROJECTION}(S, i)$ over all $|S| \leq \eta$.

For one fixed dyadic scale h' , the non-oracle time in the current call consists of constructing the path cover, computing SCCs, duplicating the DAG, adding the interface and bridge edges, and adding the final zero-length trivial edges. This is linear, up to polylogarithmic factors, in the explicit graph material created in this call. Hence

$$T(\eta, i) \leq q \cdot \tilde{O}(\omega^2 m + \omega\eta + (\omega\sigma)^2 R) + q\omega \cdot \left(\sum_{S \text{ is light}} T(|S|, t) + T(\eta, i-1) \right)$$

This is the same recurrence as above, so solving it and substituting the same parameter bounds yields the lemma as claim.

Moreover, the total cumulative graph size for the oracle call $\mathcal{O}_{hs, \kappa_p, 1+\varepsilon_p}^{\text{Proj}}$ is on the auxiliary graphs over all layers of recursion and all dyadic scales, with size $G[S]$ each, so the total size is no larger than $|D_{V,t}^*|$. The oracle call $\mathcal{O}_{\kappa_f, 1+\varepsilon_f}^{\text{PDMCF}}$ is also on the auxiliary graph over all layers of recursion and all dyadic scales with size $G[S]$, but repeat for all pairs of large clusters ($\omega\sigma^2$ many of them, which is at most a factor of $n^{O(\delta)} \cdot \exp(1/\delta^3)$). $\square$

B.7.3 Projection Algorithm for Flows

In this section, we prove that the construction above admits a projection algorithm for flows.

Lemma B.49. *Let $D_{V,t}^*$ be the DAG projection returned by $\text{PROJECTION}(V, t)$. Then there is an algorithm that, given a feasible flow F^D on $D_{V,t}^*$, computes a flow F on G such that*

$$\text{Dem}(F) = \pi(\text{Dem}(F^D)), \quad \text{cost}(F) \leq \text{cost}(F^D),$$

and the congestion of F is at most

$$\kappa = \kappa_f \kappa_p \cdot n^{O(\delta)} \cdot (1/\varepsilon)^{O(1/\delta^5)}$$

Moreover, the running time is at most $\tilde{O}(|D_{V,t}^*|)$.

The proof is based on a recursive local reduction. The non-trivial part is that the demands created at a recursive call $\text{PROJECTION}(S, i)$ are *not* routed inside $G[S]$. Instead, each call produces a local flow on $G[S]$

together with a collection of deferred demands. These deferred demands are later routed *globally on G* , layer by layer, using the weak LC-expander hierarchy of the original graph.

Definition B.50 (Global heavy-SCC families). Consider the full recursion tree of the recursive call $\text{PROJECTION}(S, i)$, and collect all heavy SCC vertex sets generated by all recursive calls in this subtree. We denote this family by

$$\mathcal{H}_{S,i}.$$

For every $A \in \mathcal{H}_{S,i}$, let

$$j(A) \in \{1, \dots, t\} \quad \text{and} \quad \hat{h}(A) \leq h$$

denote respectively the second parameter of the recursion call involves A , and the dyadic scale at which the heavy SCC A is generated.

Let $\chi_{S,i}$ be the maximum number of times a vertex v being included in sets of $\mathcal{H}_{S,i}$.

Lemma B.51. *For every recursive call $\text{PROJECTION}(S, i)$, we have*

$$\chi_{S,i} \leq (qw)^{O(t \log_\sigma |S|)}.$$

Proof. We prove the first bound by induction on the recursion. For convenience, write $\Xi(|S|, i)$ for an upper bound on $\chi_{S,i}$.

The base cases are immediate. If $i = 0$, then $\text{PROJECTION}(S, 0)$ returns the edgeless graph on S , so no heavy SCC is generated and $\chi_{S,0} = 0$. If $|S| = 1$, again no heavy SCC is generated.

Now consider a non-base call $\text{PROJECTION}(S, i)$ with $|S| > 1$ and $i > 0$. For each dyadic scale $h' \leq h$, the path cover $P^{(h')}$ has width w . Since there are q dyadic scales, each vertex is included in at most qw many clusters.

For every light SCC C , the recursive child is $\text{PROJECTION}(V(C), t)$, and by definition of light,

$$|V(C)| \leq |S|/\sigma.$$

Hence, by induction, the maximum duplication of vertices in the recursive tree rooted at C is at most

$$\Xi(\lceil |S|/\sigma \rceil, t)$$

Similarly, for every heavy SCC C , the recursive child is $\text{PROJECTION}(V(C), i - 1)$. Since $|V(C)| \leq |S|$, the induction hypothesis gives a upper bound

$$\Xi(|S|, i - 1)$$

Therefore

$$\chi_{S,i} \leq qw \cdot \left(1 + \Xi(\lceil |S|/\sigma \rceil, t) + \Xi(|S|, i - 1)\right).$$

Solving this recurrence gives

$$\chi_{S,i} \leq (qw)^{O(t \log_\sigma |S|)}.$$

The partition can be constructed recursively together with the construction of $D_{S,i}^*$. □

We next define the recursive congestion bound for the local-flow part.

The following lemma performs the recursive local reduction. For each deferred demand D created from a heavy SCC A , we define the *budget* as

$$b_A(D) := \hat{h}(A)s \cdot |D|,$$

Lemma B.52 (Recursive flow mapping). *Fix a recursive call $\text{PROJECTION}(S, i)$, and let $D_{S,i}^*$ be the DAG returned by this call. There is an algorithm that, given a feasible flow F^D on $D_{S,i}^*$, outputs*

- a flow F^{loc} on $G[S]$, and
- for every heavy SCC $A \in \mathcal{H}_{S,i}$, two deferred demands $D_A^{\text{out}}, D_A^{\text{in}}$ supported on $V(A)$,

such that:

1. each $D_A^{\text{out}}, D_A^{\text{in}}$ is $\deg_{C_{j(A)}}$ -respecting and symmetric $\hat{h}(A)$ -length in $G - C_{>j(A)}$;
- 2.

$$\text{Dem}(F^{\text{loc}}) + \sum_{A \in \mathcal{H}_{S,i}} (D_A^{\text{out}} + D_A^{\text{in}}) = \pi(\text{Dem}(F^D));$$

- 3.

$$\text{cost}(F^{\text{loc}}) + \sum_{A \in \mathcal{H}_{S,i}} (b_A(D_A^{\text{out}}) + b_A(D_A^{\text{in}})) \leq \text{cost}(F^D);$$

4. the congestion of F^{loc} is at most $\Gamma(|S|, i)$ where

$$\Gamma(|S|, i) \leq (\omega\sigma)^2 R\kappa_f\kappa_p/\varepsilon \cdot (O(\omega^2))^{i+t\lceil \log_\sigma |S| \rceil}$$

Moreover, the reduction runs in time $\tilde{O}(|D_{S,i}^*|)$.

Proof. We induct on $(|S|, i)$ lexicographically. The base cases $i = 0$ and $|S| = 1$ are trivial.

Write

$$D_{S,i}^* = \left(\bigsqcup_{\hat{h} \leq h} \hat{D}^{(\hat{h})} \right) \cup E_{\text{triv}},$$

where E_{triv} denotes the set of zero-length trivial edges added between copies of the same original vertex.

For every dyadic scale $\hat{h} \leq h$, let $F^{D,(\hat{h})}$ be the restriction of F^D to the edge set of $\hat{D}^{(\hat{h})}$, i.e., we simply ignore all trivial edges and keep only the edges inside the $\hat{h}$ -scale gadget.

Since the gadgets $\hat{D}^{(\hat{h})}$ are edge-disjoint, we have

$$F^D|_{E(D_{S,i}^*) \setminus E_{\text{triv}}} = \sum_{\hat{h} \leq h} F^{D,(\hat{h})}.$$

Moreover, every trivial edge $e = (x, y) \in E_{\text{triv}}$ satisfies $\pi(x) = \pi(y)$, so it contributes zero to the projected demand. Therefore

$$\sum_{\hat{h} \leq h} \pi(\text{Dem}(F^{D,(\hat{h})})) = \pi(\text{Dem}(F^D)).$$

Also, all trivial edges have length 0, hence

$$\sum_{\hat{h} \leq h} \text{cost}(F^{D,(\hat{h})}) \leq \text{cost}(F^D).$$

We now process each scale independently and then sum the results.

Processing one fixed scale $\hat{h}$. Fix one dyadic scale $\hat{h} \leq h$.

Let $F_{\text{stay}}^{D,(\hat{h})}$ be the restriction of $F^{D,(\hat{h})}$ to the edges inside the two copies $D_1^{(\hat{h})}, D_2^{(\hat{h})}$, including the inter-SCC edges of $D^{(\hat{h})}$, but excluding interface and bridge edges.

For every SCC C of $P^{(\hat{h})}$, and for each copy of the recursive gadget D_C , let F_C^D be the restriction of $F_{\text{stay}}^{D,(\hat{h})}$ to that copy. If C is light, apply the induction hypothesis to the recursive call $\text{PROJECTION}(V(C), t)$; if C is heavy, apply it to $\text{PROJECTION}(V(C), i - 1)$.

Every inter-SCC edge

$$(u_{C_u}^{\text{last}}, v_{C_v}^{\text{first}})$$

maps directly to the original edge $(u, v) \in E(G[S])$ with the same length and capacity.

Summing the recursively returned local flows over all copies of all SCC gadgets, together with the direct images of all inter-SCC edges, gives a flow $F_{\text{stay,loc}}^{(\hat{h})}$ on $G[S]$, and a collection of descendant deferred demands $\mathcal{D}_{\text{desc}}^{(\hat{h})}$, such that

$$\text{Dem}(F_{\text{stay,loc}}^{(\hat{h})}) + \sum_{D \in \mathcal{D}_{\text{desc}}^{(\hat{h})}} D = \pi(\text{Dem}(F_{\text{stay}}^{D,(\hat{h})})),$$

and

$$\text{cost}(F_{\text{stay,loc}}^{(\hat{h})}) + \sum_{D \in \mathcal{D}_{\text{desc}}^{(\hat{h})}} b_{A(D)}(D) \leq \text{cost}(F_{\text{stay}}^{D,(\hat{h})}),$$

where $A(D)$ denotes the heavy SCC that created the deferred demand D .

Its congestion is at most

$$2\omega^2 \max\{1, \Gamma(\lceil |S|/\sigma \rceil, t), \Gamma(|S|, i-1)\},$$

because in each of the two copies an original edge can be represented by at most ω^2 lifted edges (each node is copied at most ω times, and the path cover is *induced*).

Handling the interface and bridge edges. We next handle the transfer gadget.

For every heavy SCC A created in the current call at scale $\hat{h}$, and every $u \in V(A)$, define

$$p_A(u) := F^{D,(\hat{h})}((u_A^{\text{last}})_1, v_A), \quad q_A(u) := F^{D,(\hat{h})}(v'_A, (u_A^{\text{first}})_2).$$

For every ordered pair (A, B) and every bridge bucket r , let

$$x_{A,B,r} := F^{D,(\hat{h})}(v_A, v'_B).$$

Since $F^{D,(\hat{h})}$ is feasible and the bridge edge (v_A, v'_B) has capacity $2\mu_{A,B,r}/\varepsilon$, we have

$$x_{A,B,r} \leq 2\mu_{A,B,r}/\varepsilon.$$

We now scale the stored flow by the amount of bridge flow actually used:

$$\bar{g}_{A,B,r} := \begin{cases} \frac{x_{A,B,r}}{\mu_{A,B,r}} g_{A,B,r}^*, & x_{A,B,r} > 0, \\ 0, & x_{A,B,r} = 0. \end{cases}$$

Then $\bar{g}_{A,B,r}$ has value $x_{A,B,r}$, average cost at most L_r , and congestion at most $2\kappa_f \kappa_p/\varepsilon$.

Define

$$\alpha_{A,B,r}(u) := \bar{g}_{A,B,r}(s_A, u), \quad \beta_{A,B,r}(v) := \bar{g}_{A,B,r}(v, s_B),$$

and aggregate over all bridge types incident to A :

$$\alpha_A(u) := \sum_{B,r} \alpha_{A,B,r}(u), \quad \beta_A(u) := \sum_{B,r} \beta_{A,B,r}(u).$$

By flow conservation at the interface vertices v_A, v'_A ,

$$\sum_u p_A(u) = \sum_u \alpha_A(u), \quad \sum_u q_A(u) = \sum_u \beta_A(u).$$

Define deferred demands

$$D_A^{\text{out}}(u) := p_A(u) - \alpha_A(u), \quad D_A^{\text{in}}(u) := \beta_A(u) - q_A(u).$$

These are single-commodity demands summing to zero, supported on $V(A)$. Moreover, for every $u \in V(A)$,

$$|D_A^{\text{out}}(u)| \leq p_A(u) + \alpha_A(u) \leq \deg_{C_i}(u), \quad |D_A^{\text{in}}(u)| \leq q_A(u) + \beta_A(u) \leq \deg_{C_i}(u),$$

because both the interface edges and the star edges adjacent to u have capacity $\deg_{C_i}(u)$. Since A is an SCC of $P^{(\hat{h})}$, it has diameter at most $\hat{h}$ in $G[S] - C_{>i}$. Thus $D_A^{\text{out}}, D_A^{\text{in}}$ are $\deg_{C_i}$ -respecting and symmetric $\hat{h}$ -length in $G - C_{>i}$, witnessed by the clustering consisting of $V(A)$ together with singleton clusters outside A . Since $j(A) = i$ and $\hat{h}(A) = \hat{h}$, this proves Item (1) for heavy SCCs created in the current call; the corresponding claim for descendant heavy SCCs holds by induction.

Let

$$M^{(\hat{h})} := \sum_{A,B,r} \bar{g}_{A,B,r} \big|_{E(G[S])},$$

that is, the sum of the restrictions of all scaled auxiliary flows to the original edges of $G[S]$. Since there are at most $\omega\sigma$ heavy SCCs in the current call at the fixed scale $\hat{h}$, there are at most

$$(\omega\sigma)^2 R$$

bridge types. Therefore $M^{(\hat{h})}$ has congestion at most

$$2(\omega\sigma)^2 R \kappa_f \kappa_p / \varepsilon.$$

Furthermore,

$$\text{Dem}(M^{(\hat{h})}) + \sum_{A \in \mathcal{H}_{S,i,\hat{h}}^{\text{cur}}} (D_A^{\text{out}} + D_A^{\text{in}}) = \pi(\text{Dem}(F_{\text{tr}}^{D,(\hat{h})})),$$

where $F_{\text{tr}}^{D,(\hat{h})}$ denotes the restriction of $F^{D,(\hat{h})}$ to the interface and bridge edges, and

$$\mathcal{H}_{S,i,\hat{h}}^{\text{cur}} \subseteq \mathcal{H}_{S,i}$$

denotes the heavy SCCs created in the current call at scale $\hat{h}$.

For the cost, each $\bar{g}_{A,B,r}$ has average cost at most L_r , hence

$$\text{cost}(M^{(\hat{h})}) \leq \sum_{A,B,r} x_{A,B,r} L_r = \text{cost}(F^{D,(\hat{h})} \big|_{\text{bridge edges}}).$$

Also,

$$|D_A^{\text{out}}| \leq \sum_u p_A(u), \quad |D_A^{\text{in}}| \leq \sum_u q_A(u),$$

so

$$b_A(D_A^{\text{out}}) \leq \hat{h}s \sum_u p_A(u), \quad b_A(D_A^{\text{in}}) \leq \hat{h}s \sum_u q_A(u).$$

These are bounded by the costs of the outgoing and incoming interface-edge families of A , since every such interface edge has length $\hat{h}s$. Hence

$$\text{cost}(M^{(\hat{h})}) + \sum_{A \in \mathcal{H}_{S,i,\hat{h}}^{\text{cur}}} (b_A(D_A^{\text{out}}) + b_A(D_A^{\text{in}})) \leq \text{cost}(F_{\text{tr}}^{D,(\hat{h})}).$$

Now define

$$F_{\text{loc}}^{(\hat{h})} := F_{\text{stay,loc}}^{(\hat{h})} + M^{(\hat{h})},$$

and

$$\mathcal{D}^{(\hat{h})} := \mathcal{D}_{\text{desc}}^{(\hat{h})} \cup \{D_A^{\text{out}}, D_A^{\text{in}} \mid A \in \mathcal{H}_{S,i,\hat{h}}^{\text{cur}}\}.$$

Then

$$\text{Dem}(F_{\text{loc}}^{(\hat{h})}) + \sum_{D \in \mathcal{D}^{(\hat{h})}} D = \pi(\text{Dem}(F^{D,(\hat{h})})),$$

and

$$\text{cost}(F_{\text{loc}}^{(\hat{h})}) + \sum_{D \in \mathcal{D}^{(\hat{h})}} b_{A(D)}(D) \leq \text{cost}(F^{D,(\hat{h})}).$$

Its congestion is at most

$$\Gamma(|S|, i) \leq 2\omega^2 \max\{1, \Gamma(\lceil |S|/\sigma \rceil, i), \Gamma(|S|, i-1)\} + 2(\omega\sigma)^2 R \kappa_f \kappa_p / \varepsilon.$$

Solving the recurrence gives the lemma.

Summing over all scales. Finally set

$$F^{\text{loc}} := \sum_{\hat{h} \leq h} F_{\text{loc}}^{(\hat{h})},$$

and take the union of all deferred-demand collections over all scales. Since there are q dyadic scales, the congestion multiplies by at most q . Therefore

$$\cong (F^{\text{loc}}) \leq \Gamma(|S|, i)$$

after adjusting the constant hidden in the definition of Γ . The demand and cost identities add over different scales, because

$$\sum_{\hat{h} \leq h} \pi(\text{Dem}(F^{D,(\hat{h})})) = \pi(\text{Dem}(F^D))$$

and

$$\sum_{\hat{h} \leq h} \text{cost}(F^{D,(\hat{h})}) \leq \text{cost}(F^D).$$

This proves the lemma. $\square$

Now we are ready to prove [Lemma B.49](#).

Proof of Lemma B.49. Apply [Lemma B.52](#) to the top-level call $\text{PROJECTION}(V, t)$ and the input feasible flow F^D . This gives a local flow F^{loc} on G and, for every heavy SCC $A \in \mathcal{H}_{V,t}$, two deferred demands $D_A^{\text{out}}, D_A^{\text{in}}$, such that

$$\begin{aligned} \text{Dem}(F^{\text{loc}}) + \sum_{A \in \mathcal{H}_{V,t}} (D_A^{\text{out}} + D_A^{\text{in}}) &= \pi(\text{Dem}(F^D)), \\ \text{cost}(F^{\text{loc}}) + \sum_{A \in \mathcal{H}_{V,t}} (b_A(D_A^{\text{out}}) + b_A(D_A^{\text{in}})) &\leq \text{cost}(F^D), \end{aligned}$$

and

$$\cong (F^{\text{loc}}) \leq \Gamma(n, t).$$

For every layer $j \in [t]$ and dyadic scale $\hat{h} \leq h$, define the aggregate deferred demands

$$\Delta_{j,\hat{h}}^{\text{out}} := \sum_{\substack{A \in \mathcal{H}_{V,t} \\ j(A)=j, \hat{h}(A)=\hat{h}}} D_A^{\text{out}}, \quad \Delta_{j,\hat{h}}^{\text{in}} := \sum_{\substack{A \in \mathcal{H}_{V,t} \\ j(A)=j, \hat{h}(A)=\hat{h}}} D_A^{\text{in}}.$$

By Item (1) of [Lemma B.52](#), every $D_A^{\text{out}}, D_A^{\text{in}}$ in the above sum is $\deg_{C_j}$ -respecting and symmetric $\hat{h}$ -length in $G - C_{>j}$. Since each vertex belongs to at most $\chi_{V,t}$ sets in $\mathcal{H}_{V,t}$, the aggregate demands $\Delta_{j,\hat{h}}^{\text{out}}, \Delta_{j,\hat{h}}^{\text{in}}$ are $\chi_{V,t} \deg_{C_j}$ -respecting and symmetric $\hat{h}$ -length in $G - C_{>j}$.

Therefore

$$\frac{1}{\chi_{V,t}} \Delta_{j,\hat{h}}^{\text{out}}, \quad \frac{1}{\chi_{V,t}} \Delta_{j,\hat{h}}^{\text{in}}$$

are $\deg_{C_j}$ -respecting and symmetric $\hat{h}$ -length. Since $\deg_{C_j}$ is $(\leq h, s)$ -length $C_{>j}$ -constraint (ϕ, s) -expanding, the associated routing algorithm outputs flows routing these scaled demands with congestion at most $1/\phi$ and cost at most

$$\hat{h}s \cdot \left| \frac{1}{\chi_{V,t}} \Delta_{j,\hat{h}}^{\text{out}} \right|, \quad \hat{h}s \cdot \left| \frac{1}{\chi_{V,t}} \Delta_{j,\hat{h}}^{\text{in}} \right|.$$

Scaling the two routings back by $\chi_{V,t}$, we obtain flows $R_{j,\hat{h}}^{\text{out}}$ and $R_{j,\hat{h}}^{\text{in}}$ routing $\Delta_{j,\hat{h}}^{\text{out}}$ and $\Delta_{j,\hat{h}}^{\text{in}}$, each with congestion at most $\chi_{V,t}/\phi$, and cost at most

$$\hat{h}s \cdot |\Delta_{j,\hat{h}}^{\text{out}}|, \quad \hat{h}s \cdot |\Delta_{j,\hat{h}}^{\text{in}}|.$$

Since

$$|\Delta_{j,\hat{h}}^{\text{out}}| \leq \sum_{\substack{A \in \mathcal{H}_{V,t} \\ j(A)=j, \hat{h}(A)=\hat{h}}} |D_A^{\text{out}}|, \quad |\Delta_{j,\hat{h}}^{\text{in}}| \leq \sum_{\substack{A \in \mathcal{H}_{V,t} \\ j(A)=j, \hat{h}(A)=\hat{h}}} |D_A^{\text{in}}|,$$

we get

$$\text{cost}(R_{j,\hat{h}}^{\text{out}}) + \text{cost}(R_{j,\hat{h}}^{\text{in}}) \leq \sum_{\substack{A \in \mathcal{H}_{V,t} \\ j(A)=j, \hat{h}(A)=\hat{h}}} (b_A(D_A^{\text{out}}) + b_A(D_A^{\text{in}})).$$

Now define

$$F := F^{\text{loc}} + \sum_{j,\hat{h}} R_{j,\hat{h}}^{\text{out}} + \sum_{j,\hat{h}} R_{j,\hat{h}}^{\text{in}}.$$

The deferred demands cancel exactly, so

$$\text{Dem}(F) = \pi(\text{Dem}(F^D)).$$

Also, summing the above cost bounds over all $(j, \hat{h})$ and using [Lemma B.52](#), we get

$$\text{cost}(F) \leq \text{cost}(F^D).$$

For the congestion,

$$\cong(F) \leq \Gamma(n, t) + \frac{2tq\chi_{V,t}}{\phi}.$$

By [Lemma B.52](#),

$$\Gamma(n, t) \leq (\omega\sigma)^2 R\kappa_f \kappa_p / \varepsilon \cdot (O(\omega^2))^{t+t\lceil \log_\sigma n \rceil},$$

and by [Lemma B.51](#),

$$\chi_{V,t} \leq (qw)^{O(t \log_\sigma n)}$$

Substituting the correct parameters yields the lemma.

Finally, the local reduction takes time $\tilde{O}(|D_{V,t}^*|)$, and for each of the $2tq$ aggregate demands we make one call to the associated routing algorithm. It can be verified that the associated routing algorithm takes time subsumed by $\tilde{O}(|D_{V,t}^*|)$. This proves the lemma. $\square$

B.7.4 Projection Algorithm for Moving Cuts

In this subsection we prove that the recursive construction above admits a projection algorithm for moving cuts.

Lemma B.53. *Let $D_{V,t}^*$ be the DAG returned by $\text{PROJECTION}(V, t)$. There is an algorithm that, given a moving cut C^D on $D_{V,t}^*$, return a moving cut C on G with the same size and length, such that for every $u, v \in V$ with $\text{dist}_{G-C}(u, v) \leq h$, we have $\text{dist}_{D_{V,t}^*-C^D}(u^{\text{first}}, v^{\text{last}}) \leq (1 + \varepsilon^*) \cdot \text{dist}_{G-C}(u, v)$ where*

$$\varepsilon^* = O(\varepsilon + \varepsilon_p + \varepsilon_f) / \delta^2$$

It is technically more convenient to work with *length increases* instead of moving cuts.

For a graph H , a *length increase* on H is a nonnegative function $\tilde{\ell} : E(H) \rightarrow \mathbb{R}_{\geq 0}$. We write $H + \tilde{\ell}$ for the graph obtained by replacing the length of every edge e by $\ell_H(e) + \tilde{\ell}(e)$, and define the size of $\tilde{\ell}$ by

$$|\tilde{\ell}| := \sum_{e \in E(H)} U_H(e) \cdot \tilde{\ell}(e).$$

We define

$$\Lambda_i := (1 + \varepsilon/s)^{t-i},$$

and fix a sufficiently large absolute constant c_0 . We also define

$$\eta(S, i) := \left(1 + c_0(\varepsilon + \varepsilon_p + \varepsilon_f)\right)^{(t+1)\lceil \log_\sigma |S| \rceil + i}.$$

Lemma B.54. *Let $D_{S,i}^*$ be the DAG returned by $\text{PROJECTION}(S, i)$. There is an algorithm that, given a length increase $\tilde{\ell}^D$ on $D_{S,i}^*$, returns a length increase $\tilde{\ell}$ on $G[S]$ such that:*

1. $|\tilde{\ell}| \leq |\tilde{\ell}^D|$,
2. for every dyadic $h' \leq h$, if $C_{>i}$ is interpreted as an h' -length moving cut and

$$G_{S,i}^{(h')} := G[S] - C_{>i},$$

then for every $u, v \in S$ satisfying

$$\text{dist}_{G_{S,i}^{(h')} + \tilde{\ell}}(u, v) \leq \Lambda_i \cdot \frac{h' s}{\varepsilon},$$

we have

$$\text{dist}_{D_{S,i}^* + \tilde{\ell}^D}(u^{\text{first}}, v^{\text{last}}) \leq \eta(S, i) \cdot \text{dist}_{G_{S,i}^{(h')} + \tilde{\ell}}(u, v).$$

Moreover, the algorithm runs in $\tilde{O}(|E(D_{S,i}^*)|)$ time.

Notice that [Lemma B.53](#) is a directly corollary of [Lemma B.54](#) by interpreting $\tilde{\ell}, \tilde{\ell}^D$ as moving cuts, and set $S = V, i = t, h' = h$ (w.o.l.g. we assume h is also powers of 2) so that $G_{S,i}^{(h')} = G$.

Proof. We prove the lemma by induction on the quantity

$$M(S, i) := (t + 1) \lceil \log_\sigma |S| \rceil + i.$$

Base cases. If $i = 0$, then $D_{S,0}^*$ is edgeless, so there is nothing to do. If $|S| = 1$, again there are no nontrivial edges and the claim is vacuous.

We now assume $i > 0$ and $|S| > 1$.

Construction of $\tilde{\ell}$. We define $\tilde{\ell}$ by summing three kinds of contributions.

(a) Recursive edges. For every SCC C appearing in every scale $P^{(h'')}$, consider the subgraph $D_C \subseteq D_{S,i}^*$. Restrict $\tilde{\ell}^D$ to $E(D_C)$, apply the inductive algorithm to this restriction, and obtain a length increase $\tilde{\ell}_C$ on $G[V(C)]$. We add all these contributions to $\tilde{\ell}$.

(b) Interface edges. Let

$$e = ((u_C^{\text{last}})_1, v_C) \quad \text{or} \quad e = (v'_C, (u_C^{\text{first}})_2)$$

be an interface edge, and let

$$\alpha_e := \tilde{\ell}^D(e).$$

For every original edge f incident to u with $C_i(f) > 0$, add

$$C_i(f) \cdot \alpha_e$$

to $\tilde{\ell}(f)$.

(c) Bridge bundles. Fix a scale h'' , and fix an ordered pair (A, B) of heavy SCCs at this scale. Let

$$e_r = (v_A, v'_B)$$

denote the bridge edge of original length L_r , and write

$$\alpha_r := \tilde{\ell}^D(e_r).$$

Define

$$\beta_{A,B} := \min_r (L_r + \alpha_r),$$

and let r^* be the unique index such that

$$L_{r^*} \leq \beta_{A,B} < L_{r^*+1},$$

If $\mu_{A,B,r^*-1} = 0$ then do nothing. Otherwise notice that $r^* - 1 < R$, so consider the stored cut C_{A,B,r^*-1}^* , and let

$$\tau_{r^*-1} := \min \left\{ h, \frac{L_{r^*-1}}{(1+\varepsilon_p)(1+\varepsilon_f)} \right\}.$$

Interpret C_{A,B,r^*-1}^* as a length increase

$$\tilde{\ell}_{A,B}^{\text{aux}} := \tau_{r^*-1} \cdot C_{A,B,r^*-1}^*$$

on the auxiliary graph $G_{S,i}^{\text{aux},(h'')}$.

We now transfer $\tilde{\ell}_{A,B}^{\text{aux}}$ back to $G[S]$ as follows.

- If an auxiliary edge is an original edge of $G[S]$, we add the same length increase to that edge.
- If an auxiliary edge is a star edge (s_A, u) or (u, s_A) , then for every original edge f incident to u , we add

$$C_i(f) \cdot \tilde{\ell}_{A,B}^{\text{aux}}(s_A, u)$$

to $\tilde{\ell}(f)$.

- We do the same for star edges adjacent to s_B .

Summing all recursive, interface, and bridge-bundle contributions gives the final length increase $\tilde{\ell}$.

Correctness: the size bound. We first bound $|\tilde{\ell}|$.

For the recursive parts, $\tilde{\ell}$ does not apply more size than $\tilde{\ell}^D$ by the inductive hypothesis.

For an interface edge

$$e = ((u_C^{\text{last}})_1, v_C) \quad \text{or} \quad e = (v'_C, (u_C^{\text{first}})_2),$$

the size contributed by e to $|\tilde{\ell}^D|$ is

$$U_D(e) \cdot \alpha_e = \deg_{C_i}(u) \cdot \alpha_e.$$

On the other hand, the translated contribution to $\tilde{\ell}$ has size

$$\sum_{f \ni u} U(f) \cdot C_i(f) \cdot \alpha_e = \deg_{C_i}(u) \cdot \alpha_e.$$

Thus each interface edge preserves size exactly.

Now consider a bridge bundle (A, B) at some scale h'' , and suppose $r^* > 1$. Since

$$L_{r^*} \leq \beta_{A,B} < L_{r^*+1} = (1+\varepsilon)L_{r^*},$$

and since $\beta_{A,B} \leq L_{r^*-1} + \alpha_{r^*-1}$, we get

$$\alpha_{r^*-1} \geq \beta_{A,B} - L_{r^*-1} \geq \varepsilon L_{r^*-1}.$$

Recall that the capacity of the bridge edge e_{r^*-1} is

$$U_D(e_{r^*-1}) = 2\mu_{A,B,r^*-1}/\varepsilon.$$

Therefore the contribution of e_{r^*-1} to $|\tilde{\ell}^D|$ is at least

$$U_D(e_{r^*-1}) \cdot \alpha_{r^*-1} \geq (2\mu_{A,B,r^*-1}/\varepsilon) \cdot \varepsilon L_{r^*-1} = 2\mu_{A,B,r^*-1} L_{r^*-1}.$$

The translated auxiliary cut contributes size exactly

$$|\text{translation of } \tilde{\ell}_{A,B}^{\text{aux}}| = |\tilde{\ell}_{A,B}^{\text{aux}}|,$$

because original auxiliary edges preserve size trivially, and each translated star edge preserves size by the same computation as above. Moreover,

$$|\tilde{\ell}_{A,B}^{\text{aux}}| = \tau_{r^*-1} \cdot |C_{A,B,r^*-1}^*| \leq L_{r^*-1} \cdot 2\mu_{A,B,r^*-1},$$

Hence every translated bridge bundle contributes no more size than was already paid by the corresponding bridge edge in $\tilde{\ell}^D$. Summing everything, we obtain

$$|\tilde{\ell}| \leq |\tilde{\ell}^D|.$$

Correctness: the length bound. Fix a dyadic $h' \leq h$, interpret $C_{>i}$ as an h' -length cut, and let

$$d := \text{dist}_{G_{S,i}^{(h')} + \tilde{\ell}}(u, v) \leq \Lambda_i \cdot \frac{h's}{\varepsilon}.$$

Let $h'' \leq h'$ be the smallest dyadic scale such that

$$d \leq \Lambda_i \cdot \frac{h''s}{\varepsilon}.$$

Then by minimality of h'' ,

$$h''s \leq 2\varepsilon d.$$

Let P be a shortest (u, v) -path in $G_{S,i}^{(h'')} + \tilde{\ell}$. We will construct a path from u^{first} to v^{last} in $\widehat{D}^{(h'')} + \tilde{\ell}^D$ of length at most $(1 + \eta(S, i))d$.

(a) Decomposing the lifted path. Since removing $\tilde{\ell}$ only decreases lengths, the same path P has length at most $d \leq \Lambda_i h''s/\varepsilon$ in $G_{S,i}^{(h'')}$. Hence, by the path-cover property, P lifts to a path in $P^{(h'')}$.

Decompose the lifted path into maximal pieces

$$Q_1, Q_2, \dots, Q_z$$

such that each Q_j is contained in a single SCC of $P^{(h'')}$. Let a_j, b_j denote the endpoints of Q_j , and let S_j denote the SCC containing Q_j .

(b) Lifting light pieces recursively. If S_j is light, let

$$d_j := \text{dist}_{G[V(S_j)] - C_{>i} + \tilde{\ell}}(a_j, b_j).$$

Choose the smallest dyadic $\bar{h}_j \in \{1, 2, 4, \dots, h\}$ such that

$$d_j \leq \frac{\bar{h}_j s}{\varepsilon}.$$

This exists because $d_j \leq d \leq \Lambda_i h's/\varepsilon \leq 2hs/\varepsilon$.

Since the child call is $\text{PROJECTION}(V(S_j), t)$, the inductive hypothesis gives a path in $D_{S_j} + \tilde{\ell}^D$ from $(a_j^{\text{first}})_{S_j}$ to $(b_j^{\text{last}})_{S_j}$ of length at most

$$(1 + \eta(V(S_j), t)) \cdot d_j.$$

(c) Lifting non-critical heavy pieces recursively. If S_j is heavy, write

$$\gamma_j := C_i(Q_j) := \sum_{e \in E(Q_j)} C_i(e).$$

Define

$$\Gamma_j^{\leftarrow} := \sum_{\substack{j' \leq j \\ S_{j'} \text{ heavy}}} \gamma_{j'}, \quad \Gamma_j^{\rightarrow} := \sum_{\substack{j' \geq j \\ S_{j'} \text{ heavy}}} \gamma_{j'}.$$

We say Q_j is *left-non-critical* if $\Gamma_j^{\leftarrow} \leq 1$, and *right-non-critical* if $\Gamma_j^{\rightarrow} \leq 1$.

Suppose first that

$$\sum_{\substack{1 \leq j \leq z \\ S_j \text{ heavy}}} \gamma_j \leq 1.$$

Then every heavy piece is both left- and right-non-critical. For such a piece, passing from $G_{S,i}^{(h'')}$ to $G_{S,i-1}^{(h'')}$ adds at most $\gamma_j h''$ along Q_j . Hence

$$\text{dist}_{G[V(S_j)]-C_{>i-1}+\tilde{\ell}}(a_j, b_j) \leq \text{dist}_{G[V(S_j)]-C_{>i}+\tilde{\ell}}(a_j, b_j) + \gamma_j h''.$$

Since

$$\text{dist}_{G[V(S_j)]-C_{>i}+\tilde{\ell}}(a_j, b_j) \leq d \leq \Lambda_i \frac{h'' s}{\varepsilon}$$

and $\gamma_j \leq 1$, we obtain

$$\text{dist}_{G[V(S_j)]-C_{>i-1}+\tilde{\ell}}(a_j, b_j) \leq \Lambda_i \frac{h'' s}{\varepsilon} + h'' = \Lambda_{i-1} \frac{h'' s}{\varepsilon}.$$

Therefore the inductive hypothesis applies to the heavy child $\text{PROJECTION}(V(S_j), i-1)$, and yields a path in $D_{S_j} + \tilde{\ell}^D$ of length at most

$$(1 + \eta(V(S_j), i-1)) \cdot (\text{dist}_{G[V(S_j)]-C_{>i}+\tilde{\ell}}(a_j, b_j) + \gamma_j h'').$$

Summing over all heavy pieces, the total additive loss from the $\gamma_j h''$ terms is at most

$$h'' \cdot \sum_{\substack{1 \leq j \leq z \\ S_j \text{ heavy}}} \gamma_j \leq h''.$$

After concatenating all recursively lifted pieces with the inter-SCC edges inside the first copy $D_1^{(h'')}$, we obtain a path of length at most

$$(1 + \eta(S, i))d,$$

provided c_0 is chosen large enough to absorb the extra $O(\varepsilon)$ term coming from $h'' s \leq 2\varepsilon d$. So in this case we are done.

We may therefore assume

$$\sum_{\substack{1 \leq j \leq z \\ S_j \text{ heavy}}} \gamma_j > 1.$$

Let a be the smallest index such that S_a is heavy and

$$\Gamma_a^{\leftarrow} > 1,$$

and let b be the largest index such that S_b is heavy and

$$\Gamma_b^{\rightarrow} > 1.$$

Then every heavy piece strictly before Q_a is left-non-critical, and every heavy piece strictly after Q_b is right-non-critical. By the same argument as above, all pieces strictly before Q_a and strictly after Q_b can be lifted recursively, and the total additive loss from the corresponding $\gamma_j h''$ terms is at most

$$2h''.$$

(d) Choosing the exit and entry points. Let

$$\Gamma_{a-} := \sum_{\substack{j < a \\ S_j \text{ heavy}}} \gamma_j.$$

Traverse the edges of Q_a from a_a to b_a , and let

$$e_L = (p_L, q_L)$$

be the first edge of Q_a such that

$$\Gamma_{a-} + C_i(Q_a[a_a, p_L]) + C_i(e_L) > 1.$$

Thus

$$\Gamma_{a-} + C_i(Q_a[a_a, p_L]) \leq 1.$$

Define the left candidate set

$$X := \left(\bigcup_{j < a, S_j \text{ heavy}} V(Q_j) \right) \cup V(Q_a[a_a, p_L]).$$

Symmetrically, let

$$\Gamma_{b+} := \sum_{\substack{j > b \\ S_j \text{ heavy}}} \gamma_j,$$

and let

$$e_R = (p_R, q_R)$$

be the last edge of Q_b such that

$$\Gamma_{b+} + C_i(Q_b[q_R, b_b]) + C_i(e_R) > 1.$$

Thus

$$\Gamma_{b+} + C_i(Q_b[q_R, b_b]) \leq 1.$$

Define the right candidate set

$$Y := V(Q_b[q_R, b_b]) \cup \left(\bigcup_{j > b, S_j \text{ heavy}} V(Q_j) \right).$$

For every pair $x \in X$, $y \in Y$, let $A := S(x)$ and $B := S(y)$, and define

$$\Psi(x, y) := \tilde{\ell}^D((x_A^{\text{last}})_1, v_A) + \tilde{\ell}_{A,B}^{\text{aux}}(s_A, x) + \beta_{A,B} + \tilde{\ell}_{A,B}^{\text{aux}}(y, s_B) + \tilde{\ell}^D(v'_B, (y_B^{\text{first}})_2).$$

Choose a pair $(x, y) \in X \times Y$ minimizing $\Psi(x, y)$, and let

$$A := S(x), \quad B := S(y).$$

By construction of X , the total C_i -mass on the left window

$$W_L := \left(\bigcup_{j < a, S_j \text{ heavy}} Q_j \right) \cup Q_a[a_a, p_L] \cup \{e_L\}$$

is strictly larger than 1. Every edge of W_L receives in $\tilde{\ell}$ at least

$$C_i(e) \cdot \left(\tilde{\ell}^D((x_A^{\text{last}})_1, v_A) + \tilde{\ell}_{A,B}^{\text{aux}}(s_A, x) \right)$$

from the combination of interface and left-star translations. Hence

$$\tilde{\ell}(W_L) > \tilde{\ell}^D((x_A^{\text{last}})_1, v_A) + \tilde{\ell}_{A,B}^{\text{aux}}(s_A, x).$$

Similarly, for the right window

$$W_R := \{e_R\} \cup Q_b[q_R, b_b] \cup \left(\bigcup_{j>b, S_j \text{ heavy}} Q_j \right),$$

we have

$$\tilde{\ell}(W_R) > \tilde{\ell}_{A,B}^{\text{aux}}(y, s_B) + \tilde{\ell}^D(v'_B, (y_B^{\text{first}})_2).$$

Thus the extra interface lengths and the two relevant auxiliary-star lengths are already charged to the $\tilde{\ell}$ -length of the original path.

Moreover, each interface edge has original length $h''s$, so the two interface edges contribute an additive

$$2h''s \leq 4\epsilon d.$$

Finally, by construction of X and Y , the total C_i -mass on the heavy pieces traversed before reaching x , and after leaving y , is at most 1. Therefore the same non-critical-heavy-piece argument as above lifts the prefix from u to x inside the first copy $D_1^{(h'')}$, and the suffix from y to v inside the second copy $D_2^{(h'')}$, with an additional additive loss at most $2h''$.

(e) The bridge edge.

By [Corollary B.72](#), in the auxiliary graph

$$G_{S,i}^{\text{aux},(h'')} + \tilde{\ell}_{A,B}^{\text{aux}},$$

the distance from s_A to s_B is at least

$$\tau_{r^*-1} = \min \left\{ h, \frac{L_{r^*-1}}{(1+\epsilon_p)(1+\epsilon_f)} \right\}.$$

Now the subpath $P[x, y]$, together with the two star edges

$$(s_A, x), \quad (y, s_B),$$

forms an s_A - s_B walk in the auxiliary graph. Therefore

$$\tau_{r^*-1} \leq \text{dist}_{G_{S,i}^{(h'')} + \tilde{\ell}}(x, y) + \tilde{\ell}_{A,B}^{\text{aux}}(s_A, x) + \tilde{\ell}_{A,B}^{\text{aux}}(y, s_B).$$

But the two auxiliary-star terms were already charged to $\tilde{\ell}(W_L)$ and $\tilde{\ell}(W_R)$ as argued already in (d) by the choosing of entry point, respectively. Hence they create no additional uncharged loss in the final comparison.

Since

$$\beta_{A,B} < L_{r^*+1} = (1+\epsilon)L_{r^*} \leq (1+\epsilon)^2 L_{r^*-1},$$

we conclude that

$$\beta_{A,B} \leq (1+\epsilon)^2(1+\epsilon_p)(1+\epsilon_f) \cdot \left(\text{dist}_{G_{S,i}^{(h'')} + \tilde{\ell}}(x, y) + \tilde{\ell}_{A,B}^{\text{aux}}(s_A, x) + \tilde{\ell}_{A,B}^{\text{aux}}(y, s_B) \right).$$

After charging the two star terms to the left and right windows, the bridge contributes only a multiplicative $1 + O(\epsilon + \epsilon_p + \epsilon_f)$ loss on the skipped middle part.

(f) Concatenating all parts. We now concatenate:

- the recursively lifted prefix from u to x inside the first copy $D_1^{(h'')}$,
- the interface edge $((x_A^{\text{last}})_1, v_A)$,
- the cheapest bridge (v_A, v'_B) ,
- the interface edge $(v'_B, (y_B^{\text{first}})_2)$,
- the recursively lifted suffix from y to v inside the second copy $D_2^{(h'')}$.

All concatenations are valid because $D_1^{(h'')}$ and $D_2^{(h'')}$ contain the lifted inter-SCC edges of $P^{(h'')}$, and the final trivial zero-length edges in $D_{S,i}^*$ allow us to move from the global u^{first} to the copy of u^{first} in $\hat{D}^{(h'')}$,

and from the copy of v^{last} in $\widehat{D}^{(h'')}$ to the global v^{last} .

Each recursively lifted subpiece contributes at most the factor

$$\max\{\eta(V(S_j), t), \eta(V(S_j), i-1)\} \leq \frac{\eta(S, i)}{1 + c_0(\varepsilon + \varepsilon_p + \varepsilon_f)}.$$

The additive losses from the non-critical heavy parts and the two interface edges are $O(h'' + h''s) = O((\varepsilon + \varepsilon_p + \varepsilon_f)d)$, and the bridge contributes a multiplicative $1 + O(\varepsilon + \varepsilon_p + \varepsilon_f)$ factor on the skipped middle segment. Choosing c_0 large enough, all these losses are absorbed into the definition of $\eta(S, i)$, and we obtain

$$\text{dist}_{D_{S,i}^* + \bar{\ell}D}(u^{\text{first}}, v^{\text{last}}) \leq \eta(S, i) \cdot d.$$

This proves the second item of the lemma.

The running time is $\tilde{O}(|E(D_{S,i}^*)|)$, since each recursive edge, interface edge, and bridge bundle is processed only once up to polylogarithmic overhead. $\square$

B.7.5 Combine Everything

We first prove the following lemma.

Lemma B.55. *Let $q \in \mathbb{N}$. Given a directed graph G with non-negative edge capacities and lengths, suppose $\mathcal{O}_{h,\kappa,\lambda}^{\text{Proj}}$ computes a h -length flow-cost-preserving DAG projection G' of G together with an efficient (κ, λ) -projection algorithm. Then there is an algorithm that computes a qh -length flow-cost-preserving DAG projection $\widehat{G}$ of size $O(q|G'|)$ associated with an efficient $(4\kappa, \lambda)$ -projection algorithm. Its running time is $O(q)$ times that of the projection algorithm for G' .*

Proof. Let G' be the DAG projection returned by $\mathcal{O}_{h,\kappa,\lambda}^{\text{Proj}}$, with projection map $\pi : V(G') \rightarrow V(G)$. For each $i \in [2q]$, let $G^{(i)}$ be a copy of G' , and for each vertex $x \in V(G')$ write $x^{(i)}$ for its copy in $G^{(i)}$.

We construct $\widehat{G}$ as follows.

- For every internal edge of every copy $G^{(i)}$, keep the same length as in G' , but divide its capacity by q .
- For every $i \in [2q-1]$ and every edge $(u, v) \in E(G)$, add an edge from $(u^{\text{last}})^{(i)}$ to $(v^{\text{first}})^{(i+1)}$ with length $\ell_G(u, v)$ and capacity $U_G(u, v)/q$.
- For every $i \in [2q-1]$ and every $u \in V(G)$, add an edge from $(u^{\text{last}})^{(i)}$ to $(u^{\text{first}})^{(i+1)}$ with length 0 and infinite capacity.

Let $\hat{\pi}$ be the natural projection map on $\widehat{G}$, namely $\hat{\pi}(x^{(i)}) = \pi(x)$.

Since each $G^{(i)}$ is a DAG and all added edges go from layer i to layer $i+1$, the graph $\widehat{G}$ is also a DAG. Its size is clearly $O(q|G'|)$.

We now define a qh -length $(4\kappa, \lambda)$ -projection algorithm for $\widehat{G}$.

Primal part. Let F be a flow on $\widehat{G}$ with congestion 1 and cost B . For each $i \in [2q]$, let F_i be the restriction of F to the internal edges of $G^{(i)}$. Also, for each $i \in [2q-1]$, let H_i be the restriction of F to the inter-copy edges from layer i to layer $i+1$ corresponding to original edges of G ; we ignore the zero-length infinite-capacity edges, since they project to the same vertex and thus contribute zero projected demand.

View each F_i as a flow on the original graph G' . Because capacities in $G^{(i)}$ were scaled down by a factor of q , each F_i has congestion at most $1/q$ on G' . Applying the primal part of the projection algorithm for G' to each F_i , we obtain a flow f_i on G with the same cost as F_i , congestion at most κ/q , and demand $\text{Dem}(f_i) = \pi(\text{Dem}(F_i))$.

Next, map each H_i directly to a flow g_i on G by sending exactly the same edge-flow on the corresponding edge of G . Since those inter-copy edges have capacities scaled down by q , each g_i has congestion at most $1/q$ on G , and the same cost as H_i .

Now define

$$f := \sum_{i=1}^{2q} f_i + \sum_{i=1}^{2q-1} g_i.$$

By construction,

$$\text{Dem}(f) = \hat{\pi}(\text{Dem}(F)).$$

Also, $\text{cost}(f) = \text{cost}(F) = B$.

Finally, the congestion of f is at most

$$\sum_{i=1}^{2q} \frac{\kappa}{q} + \sum_{i=1}^{2q-1} \frac{1}{q} \leq 2\kappa + 2 \leq 4\kappa,$$

where we use $\kappa \geq 1$. Hence the primal part has congestion slack 4κ .

Dual part. Let C be an L -length moving cut on $\widehat{G}$. Since the zero-length inter-copy edges have infinite capacity, any finite-size moving cut assigns value 0 to them. Thus C is supported only on the internal edges of the copies and on the inter-copy edges corresponding to edges of G .

For each $i \in [2q]$, let C_i be the restriction of C to the internal edges of $G^{(i)}$. For each $i \in [2q-1]$, let D_i be the restriction of C to the inter-copy edges from layer i to layer $i+1$ corresponding to original edges of G .

For each i , view C_i as a moving cut on the original graph G' . Since capacities in $G^{(i)}$ were scaled down by a factor of q , its size in G' is $q|C_i|$. Applying the dual part of the projection algorithm for G' to C_i , we obtain an (L/λ) -length moving cut C_i^* on G with

$$|C_i^*| = q|C_i|.$$

For each $i \in [2q-1]$, define a cut D_i^* on G by assigning to each edge $(u, v) \in E(G)$ exactly the cut value that D_i assigns to the inter-copy edge from $(u^{\text{last}})^{(i)}$ to $(v^{\text{first}})^{(i+1)}$. Then D_i^* is an (L/λ) -length moving cut on G , and because the capacities of those inter-copy edges were scaled down by q , we have

$$|D_i^*| = q|D_i|.$$

Now define

$$\widehat{C} := \frac{1}{q} \left(\sum_{i=1}^{2q} C_i^* + \sum_{i=1}^{2q-1} D_i^* \right).$$

Then $\widehat{C}$ is an (L/λ) -length moving cut on G , and

$$|\widehat{C}| = \frac{1}{q} \left(\sum_{i=1}^{2q} |C_i^*| + \sum_{i=1}^{2q-1} |D_i^*| \right) = \sum_{i=1}^{2q} |C_i| + \sum_{i=1}^{2q-1} |D_i| = |C|.$$

It remains to prove the distance guarantee. Fix any $u, v \in V(G)$ with $\text{dist}_G(u, v) \leq qh$, and let P be a shortest path from u to v in G .

We first claim that P can be written as

$$P = Q_1 \circ e_1 \circ Q_2 \circ e_2 \circ \cdots \circ e_{r-1} \circ Q_r,$$

where each Q_j is a (possibly empty) subpath of length at most h , each e_j is a single edge, and $r \leq q$. Indeed, greedily take Q_1 to be the longest initial subpath of P of length at most h ; if $Q_1 \neq P$, let e_1 be the next edge, and continue. For every $j < r$, we then have $\ell(Q_j) + \ell(e_j) > h$. Summing over $j < r$ gives

$$(r-1)h < \sum_{j < r} (\ell(Q_j) + \ell(e_j)) \leq \ell(P) \leq qh,$$

so $r \leq q$.

Let s_j, t_j be the endpoints of Q_j , so $s_1 = u$, $t_r = v$, and $e_j = (t_j, s_{j+1})$ for every $j < r$.

For each offset $a \in [q]$, construct a walk W_a in $\widehat{G} - C$ as follows:

- move from $(u^{\text{first}})^{(1)}$ to $(u^{\text{first}})^{(a)}$ using only zero-length inter-copy edges and zero-length paths inside the intermediate copies;
- for each $j = 1, \dots, r$, traverse inside copy $G^{(a+j-1)}$ a shortest path from $(s_j^{\text{first}})^{(a+j-1)}$ to $(t_j^{\text{last}})^{(a+j-1)}$;

- for each $j < r$, traverse the inter-copy edge from $(t_j^{\text{last}})^{(a+j-1)}$ to $(s_{j+1}^{\text{first}})^{(a+j)}$ corresponding to the edge $e_j = (t_j, s_{j+1})$;
- finally move from $(v^{\text{last}})^{(a+r-1)}$ to $(v^{\text{last}})^{(2q)}$ using only zero-length inter-copy edges and zero-length paths inside the later copies.

This is well-defined because $a + r - 1 \leq 2q - 1$.

We justify the zero-length paths inside one copy. Applying the dual guarantee for the base projection to the pair (x, x) gives

$$\text{dist}_{G^{(i)}-C_i}((x^{\text{first}})^{(i)}, (x^{\text{last}})^{(i)}) \leq \lambda \cdot \text{dist}_{G-C_i^*}(x, x) = 0.$$

Hence such a path exists and has length 0.

For each j , since $\text{dist}_G(s_j, t_j) \leq \ell(Q_j) \leq h$, the dual guarantee for the base projection yields

$$\text{dist}_{G^{(a+j-1)}-C_{a+j-1}}((s_j^{\text{first}})^{(a+j-1)}, (t_j^{\text{last}})^{(a+j-1)}) \leq \lambda \cdot \text{dist}_{G-C_{a+j-1}^*}(s_j, t_j) \leq \lambda \cdot \ell_{G-C_{a+j-1}^*}(Q_j).$$

Also, for each $j < r$, the inter-copy edge corresponding to e_j has length

$$\ell_G(e_j) + L \cdot D_{a+j-1}(e_j) \leq \lambda \left(\ell_G(e_j) + \frac{L}{\lambda} D_{a+j-1}^*(e_j) \right) = \lambda \cdot \ell_{G-D_{a+j-1}^*}(e_j),$$

where we use $\lambda \geq 1$ and $\ell_G(e_j) \geq 0$.

Therefore,

$$\ell_{\widehat{G}-C}(W_a) \leq \lambda \sum_{j=1}^r \ell_{G-C_{a+j-1}^*}(Q_j) + \lambda \sum_{j=1}^{r-1} \ell_{G-D_{a+j-1}^*}(e_j).$$

Now average over all $a \in [q]$. We get

$$\begin{aligned} \frac{1}{q} \sum_{a=1}^q \ell_{\widehat{G}-C}(W_a) &\leq \lambda \sum_{j=1}^r \left(\ell_G(Q_j) + \frac{L}{\lambda} \cdot \frac{1}{q} \sum_{a=1}^q C_{a+j-1}^*(Q_j) \right) \\ &\quad + \lambda \sum_{j=1}^{r-1} \left(\ell_G(e_j) + \frac{L}{\lambda} \cdot \frac{1}{q} \sum_{a=1}^q D_{a+j-1}^*(e_j) \right) \\ &\leq \lambda \cdot \ell_{G-\widehat{C}}(P). \end{aligned}$$

Hence for some $a \in [q]$,

$$\ell_{\widehat{G}-C}(W_a) \leq \lambda \cdot \ell_{G-\widehat{C}}(P).$$

Since W_a is a walk from $(u^{\text{first}})^{(1)}$ to $(v^{\text{last}})^{(2q)}$, this implies

$$\text{dist}_{\widehat{G}-C}((u^{\text{first}})^{(1)}, (v^{\text{last}})^{(2q)}) \leq \lambda \cdot \ell_{G-\widehat{C}}(P) = \lambda \cdot \text{dist}_{G-\widehat{C}}(u, v).$$

Thus the dual part also holds. Therefore $\widehat{G}$ is associated with a qh -length $(4\kappa, \lambda)$ -projection algorithm. By [Lemma B.27](#), $\widehat{G}$ is a qh -length flow-cost-preserving DAG projection of G . $\square$

Now we are ready to prove the two lemmas stated at the beginning of this section.

Lemma B.46. *There is an algorithm given a directed graph $G = (V, E)$ with non-negative polynomially bounded integral edge capacities and lengths, a length bound h , parameters $\varepsilon, \varepsilon_p, \varepsilon_f \in (0, 1/\log n)$, $\delta \in (1/\log n, 1)$ and $z, \kappa_p, \kappa_f \in \mathbb{N}_+$, with oracle calls to $\mathcal{O}_{h/z, \kappa_p, 1+\varepsilon_p}^{\text{Proj}}$ and $\mathcal{O}_{\kappa_f, 1+\varepsilon_f}^{\text{PDMCF}}$, write*

$$S(n, \delta, \varepsilon) = n^{O(\delta)} \cdot (1/\varepsilon)^{O(1/\delta^5)}$$

output a DAG projection of size

$$m \cdot S(n, \delta, \varepsilon)$$

with a (κ, λ) -projection algorithm where

$$\kappa = \kappa_f \kappa_p \cdot S(n, \delta, \varepsilon) \quad \lambda = 1 + O(\varepsilon + \varepsilon_p + \varepsilon_f)/\delta^2$$

The projection algorithm takes time $zm \cdot S(n, \delta, \varepsilon)$ time. The construction algorithm makes $z \cdot S(n, \delta, \varepsilon)$ -efficient oracle calls.

Proof. We call the algorithm stated in [Section B.7.1](#) and apply [Lemmas B.47](#) to [B.49](#) and [B.53](#). There are two things to notice in order to get [Lemma B.46](#).

- [Section B.7.1](#) makes oracle calls to $\mathcal{O}_{hs, \kappa_p, 1+\varepsilon_p}^{\text{Proj}}$, i.e., a hs -length algorithm but not h/z . However, we can use [Lemma B.55](#) to get the oracle by concatenating zs copies of the projection returned by $\mathcal{O}_{h/z, \kappa_p, 1+\varepsilon_p}^{\text{Proj}}$, which only increase all the time complexity part by a factor of z (the $s = (\log n)^{1/\delta}$ factor is absorbed in $S(n, \delta, \varepsilon)$).
- [Lemma B.53](#) is slightly different from the definition [Definition B.6](#): the cut C^D, C has the same length while [Definition B.6](#) has a different length (differ by the length slack λ of the projection). This can be easily solved by multiplying all the capacities of the edges in the DAG by λ , so the cut C^D will increase its length by a factor λ . Then we can interpretate C as a $\text{length}(C^D)/\lambda$ cut, the size keep the same and the congestion only increase by a factor of at most $O(1/\delta^2)$.

□

Theorem B.23. *There is an algorithm given a directed graph $G = (V, E)$ with non-negative polynomially bounded integral edge capacities and lengths, parameters $\varepsilon_f \in (0, 1/\log n)$, $\delta \in (1/\log n, 1)$ and $\kappa_f \in \mathbb{N}_+$, with $n^\delta \cdot (1/\varepsilon_f)^{\text{poly}(1/\delta)}$ -efficient oracle calls to $\mathcal{O}_{\kappa_f, 1+\varepsilon_f}^{\text{PDMCF}}$, output a DAG projection of size*

$$m^{1+\delta} \cdot (1/\varepsilon_f)^{\text{poly}(1/\delta)}$$

with a (κ, λ) -projection algorithm where

$$\kappa = \kappa_f^{1/\delta} \cdot n^\delta \cdot (1/\varepsilon_f)^{\text{poly}(1/\delta)} \quad \lambda = 1 + \varepsilon_f \cdot (1/\delta)^{O(1/\delta)}$$

The projection algorithm takes time $m^{1+\delta} \cdot (1/\varepsilon_f)^{\text{poly}(1/\delta)}$.

Proof. Notice that when h is sufficiently small (like $\text{polylog}(n)$), a h -length DAG projection is trivial to get according to [Lemma B.55](#).

[Lemma B.46](#) shows how to enlarge the length by a factor of q with oracle $\mathcal{O}_{\kappa_f, 1+\varepsilon_f}^{\text{PDMCF}}$ and

- enlarging the congestion slack by a factor of $\kappa_f \cdot n^{O(\delta)} \cdot (1/\varepsilon)^{O(1/\delta^5)}$,
- enlarging the length slack minus 1 by a factor of $O(1/\delta^2)$ if we always choose $\varepsilon = \varepsilon_f$ and $\varepsilon_f < \varepsilon_p$,
- with a fix complexity $qm \cdot S(n, \delta, \varepsilon)$ and fixed size $m \cdot S(n, \delta, \varepsilon)$.

Thus, we can choose the parameters q to be $n^{\sqrt{\delta}}$, then up to $O(1/\sqrt{\delta})$ iterations, we get sufficient large length bound h , and the congestion becomes (redefine $\delta = \sqrt{\delta}$)

$$\kappa_f^{O(1/\delta)} \cdot n^{O(\delta)} \cdot (1/\varepsilon_f)^{O(1/\delta^{11})}$$

the length slack becomes

$$1 + \varepsilon_f \cdot (1/\delta)^{O(1/\delta)}$$

The complexity also matches.

□

B.8 Minimum Cost Flow Algorithm on DAGs

In this section we give an algorithm that returns an approximate primal-dual solution for minimum-cost s - t flow in a DAG. The algorithm is inspired by the weighted push-relabel framework for max flow [\[Ber+24\]](#).

Theorem B.56. *Given a DAG $G = (V, E)$ with source s , sink t , and a parameter $\varepsilon \in (0, 0.1)$, suppose there exists an s - t flow f^* on G . Then there is an algorithm that computes a $(1 \pm 6\varepsilon)$ primal-dual minimum-cost flow pair (f, ϕ, C) with value $\text{val}(f) \geq \text{val}(f^*)/3$ in time $\tilde{O}(n^2/\varepsilon)$.*

Notice that [Theorem B.56](#) implies our main theorem about combinatorial minimum cost flow on general graphs.

Theorem B.1. *There is a combinatorial algorithm computing a minimum-cost flow in $n^{2+o(1)}$ time.*

Proof. We apply [Corollary B.26](#) which requires a $\mathcal{O}_{\kappa, 1+\varepsilon}^{\text{PDMCF}}$ oracle. We use [Theorem B.56](#) with ε set to be $\exp(-\log^c n \log \log n)$, which outputs a flow routing $1/3$ of the demands. We repeat this procedure $O(\log n)$ times on the original graph and the remaining demands until all the demands have been routed, summing over all the flows gives a flow with congestion $O(\log n)$. Moreover, suppose the i -th iteration routes a_i amount of demands and the total demand is A , we have that the i -th flow's cost is at most $(1 + O(\varepsilon)) \cdot \text{OPT} \cdot \frac{a_i}{A}$ where OPT is the optimal cost for routing all the demands (this is because we can scale down the optimal flow routing all demands by $\frac{a_i}{A}$ and get a flow with cost $\text{OPT} \cdot \frac{a_i}{A}$ routing a_i demands, and according to the guarantee of [Theorem B.56](#)). Summing over all the costs gives $(1 + O(\varepsilon)) \cdot \text{OPT}$, so the required oracle for [Corollary B.26](#) is satisfied with $\kappa = O(\log n)$. The total running time is $n^{2+o(1)}$. $\square$

B.8.1 Algorithm

Guessing the scale. We first guess (within a factor of 2) the value and cost of f^* . Assume we are given integers q, α such that

$$\text{val}(f^*) \in [q, 2q] \quad \text{and} \quad \ell_G(f^*) \in [\alpha n q, 2\alpha n q].$$

These guesses can be obtained by trying powers of 2 for q and α .

Topological weights. Fix a topological order $(v_1, \dots, v_n)$ of V . For any (directed) edge $e = (v_i, v_j)$ that appears in the residual graph (including reverse residual edges), define its *topological weight*

$$w(e) := |j - i|.$$

(Thus reverse residual edges are also assigned weights by the same formula.)

Residual graph and reduced lengths. We maintain a feasible flow f (initially the zero flow) and its residual graph G_f . Each residual edge $e = (u, v) \in E(G_f)$ has residual capacity $U_{G_f}(e) > 0$ and original length/cost $\ell_G(e)$ (inherited from G for forward residual edges and negated for reverse residual edges, as usual). We also maintain a potential $\phi : V \rightarrow \mathbb{R}$ (initially $\phi \equiv 0$). For a residual edge $e = (u, v)$ define its *reduced length*

$$\ell_\phi(e) := \ell_G(e) + \phi(u) - \phi(v).$$

Admissible edges with lazy updates. Each residual edge is marked either *admissible* or *inadmissible*. Initially all residual edges are inadmissible. We use the following rule to (re)compute the status of an edge, but we do not recompute it on every potential update; instead we only recompute it at discrete “checkpoints” to control the total number of status changes.

Changing edge state (lazy checking). For a residual edge $e = (u, v) \in E(G_f)$, whenever either $\phi(u)$ or $\phi(v)$ hits a multiple of $\varepsilon \alpha \cdot w(e)$, we *check* e and set e as admissible if

$$\ell_\phi(e) \leq -2\varepsilon \alpha \cdot w(e)$$

and otherwise mark e as inadmissible.

The purpose of the checkpointing (multiples of $\varepsilon \alpha w(e)$) is to ensure that each edge is checked only $O\left(\frac{n}{\varepsilon w(e)}\right)$ times over the entire execution (formalized in [Section B.8.3](#)).

Active vertices. We call a vertex v *active* if $v \neq t$ and

$$\phi(v) \geq -99\alpha n.$$

Intuitively, we keep relabeling active vertices until admissible structure emerges; the algorithm terminates once the source potential becomes sufficiently negative (see below).

Operations. The algorithm alternates between the following two types of operations.

Relabel. While there exists an active vertex u with *no* outgoing admissible residual edge, perform

$$\phi(u) \leftarrow \phi(u) - \varepsilon\alpha,$$

and apply the lazy-checking rule above to any incident edges whose endpoint potentials hit a checkpoint.

Push / Augment. Once every active vertex has at least one outgoing admissible residual edge, each active vertex picks an arbitrary outgoing admissible edge. Starting from s , follow the picked admissible edges. [Lemma B.57](#) guarantees that we will not end up with a non-active vertex. Then either we reach t (yielding an admissible s - t path), or we repeat a vertex (yielding an admissible directed cycle). In either case, let P denote the resulting admissible s - t path or directed cycle, and augment f along P by

$$\Delta := \min_{e \in P} u_f(e),$$

updating residual capacities and the flow accordingly. (Implementation details and how we find P efficiently are deferred to [Section B.8.3](#).)

The algorithm terminates when $\phi(s) < -50\alpha n$.

Lemma B.57 (Proof in [Section B.8.2](#)). *Repeatedly following an admissible outgoing edge starting from s does not reach a non-active vertex other than t .*

Recovering the primal and the dual. The primal solution is the flow f at the end of the algorithm.

To construct the dual, define an auxiliary penalty C' on *backward* residual edges as follows. For each residual edge $e = (v_i, v_j) \in E(G_f)$ with $i > j$ (i.e., e goes backward in the fixed topological order), set

$$C'(e) := 6\varepsilon\alpha \cdot w(e),$$

and set $C'(e) = 0$ otherwise. The next lemma (proved in [Section B.8.2](#)) shows that this eliminates negative cycles.

Lemma B.58. *The length function $\ell_\phi(\cdot) - C'(\cdot)$ on G_f has no negative cycle. Equivalently, the residual graph $G_{f,\phi} - C'$ contains no negative cycle.*

Given [Lemma B.58](#), we can run a single-source shortest path algorithm for graphs with negative edges but no negative cycles (e.g., nearly-linear-time implementations as in [\[HJS25\]](#)) on G_f with edge lengths $\ell_\phi(\cdot) - C'(\cdot)$ and source s . Let $\phi' : V \rightarrow \mathbb{R}$ be the resulting distance labels.

Finally, we define the dual slack variables $C : E(G_f) \rightarrow \mathbb{R}_{\geq 0}$ by

$$C(e) := \begin{cases} \phi(v) - \phi(u) - \ell_G(e), & \text{if } e = (u, v) \text{ is saturated in } f \text{ and } \ell_G(e) < \phi(v) - \phi(u), \\ 0, & \text{otherwise,} \end{cases}$$

and return $(f, \phi + \phi', C)$ as the primal-dual output.

B.8.2 Correctness

Recall our guessing assumption: there exists an s - t flow f^* with

$$\text{val}(f^*) \in [q, 2q] \quad \text{and} \quad \ell_G(f^*) \in [\alpha n q, 2\alpha n q].$$

We first prove an invariant of the algorithm maintained by the lazy admissibility checks.

Lemma B.59. *At any time during the algorithm, every residual edge $e \in E(G_f)$ satisfies*

$$\ell_\phi(e) \geq -3\varepsilon\alpha \cdot w(e).$$

Moreover, if e is currently marked admissible, then

$$\ell_\phi(e) \leq -\varepsilon\alpha \cdot w(e).$$

Proof. Fix a residual edge $e = (u, v)$ and let τ be the most recent time at which e was *checked* under the “Changing edge state” rule. At time τ , by definition of the marking rule,

$$e \text{ admissible at time } \tau \implies \ell_\phi^\tau(e) \leq -2\varepsilon\alpha w(e), \quad e \text{ inadmissible at time } \tau \implies \ell_\phi^\tau(e) > -2\varepsilon\alpha w(e),$$

where ℓ_ϕ^τ denotes the reduced length at time τ .

Drift between checks. Between τ and the next check of e , each endpoint potential decreases by at most $\varepsilon\alpha w(e)$ in absolute value, and in particular

$$|(\phi(u) - \phi(v)) - (\phi^\tau(u) - \phi^\tau(v))| \leq \varepsilon\alpha \cdot w(e). \quad (\text{B.3})$$

Since $\ell_\phi(e) = \ell_G(e) + \phi(u) - \phi(v)$, the same bound holds for the change in $\ell_\phi(e)$:

$$|\ell_\phi(e) - \ell_\phi^\tau(e)| \leq \varepsilon\alpha \cdot w(e). \quad (\text{B.4})$$

Lower bound for all residual edges. If e was inadmissible at time τ , then $\ell_\phi^\tau(e) > -2\varepsilon\alpha w(e)$, and by (B.4),

$$\ell_\phi(e) \geq \ell_\phi^\tau(e) - \varepsilon\alpha w(e) > -3\varepsilon\alpha w(e).$$

If e was admissible at time τ , then $\ell_\phi(e)$ cannot decrease since the potential of edges with at least one admissible outgoing edge cannot decrease. Moreover, certainly $\ell_\phi^\tau(e) \leq -2\varepsilon\alpha w(e)$, and again by (B.4),

$$\ell_\phi(e) \leq \ell_\phi^\tau(e) + \varepsilon\alpha w(e) \leq -\varepsilon\alpha w(e).$$

Thus, the upper bound holds.

Effect of push operations. A push only changes the residual graph by (i) decreasing residual capacities on traversed edges and (ii) possibly creating reverse residual edges. Reduced lengths on existing residual edges are unchanged (potentials do not change during a push). For a forward residual edge $a \rightarrow b$, then the newly created reverse residual edge is $b \rightarrow a$ with

$$\ell_\phi(b, a) = -\ell_G(a, b) + \phi(b) - \phi(a) = -(\ell_G(a, b) + \phi(a) - \phi(b)) = -\ell_\phi(a, b).$$

Since the push traverses only admissible edges, $\ell_\phi(a, b) \leq -\varepsilon\alpha w(a, b)$, hence $\ell_\phi(b, a) \geq \varepsilon\alpha w(a, b) \geq -3\varepsilon\alpha w(b, a)$. Thus the invariant is preserved by pushes as well. $\square$

Admissible paths from s cannot enter inactive vertices (proof of Lemma B.57). We now show that before termination, following admissible edges from s cannot reach an inactive vertex; this implies that once every active vertex has an outgoing admissible edge, we can always find either an admissible s - t path or an admissible cycle.

Proof of Lemma B.57. Consider any time during the algorithm at which $\phi(s) \geq -50\alpha n$ (in particular, this holds whenever we execute a Push step). Suppose for contradiction that there exists a directed path P in G_f from s to some vertex v with $\phi(v) < -99\alpha n$ such that every edge of P is admissible.

Let $e = (b, a)$ be the *last* backward residual edge on P (i.e., e satisfies $b = v_i$, $a = v_j$ with $i > j$ in the fixed topological order). Such an edge must exist: since all edges of P are admissible, $\ell_\phi(P) \leq 0$, and telescoping yields

$$\ell_{G_f}(P) = \ell_\phi(P) + \phi(v) - \phi(s) \leq \phi(v) - \phi(s) < -49\alpha n,$$

so P must contain a negative-cost residual edge. Because G is a DAG with nonnegative edge costs, negative-cost residual edges can only be backward residual edges (reverse edges created by the current flow).

Since (b, a) is a backward residual edge, the forward edge (a, b) in G currently carries positive flow in f . Because G is a DAG, the support of f decomposes into s - t paths; hence (a, b) lies on some s - t flow path in this decomposition. Let Q denote the prefix of that path from s to a . Every edge on Q carries positive flow, so every edge of Q appears in G_f as a *backward* residual edge; reversing Q gives a residual path Q^{rev} from a to s consisting only of backward residual edges. Moreover, since Q follows the topological order, its total topological weight satisfies $w(Q) \leq n$, hence also $w(Q^{\text{rev}}) \leq n$.

Applying [Lemma B.59](#) to each edge of Q^{rev} and summing, we obtain

$$\ell_\phi(Q^{\text{rev}}) \geq -3\varepsilon\alpha \cdot w(Q^{\text{rev}}) \geq -3\varepsilon\alpha n.$$

On the other hand, telescoping along Q^{rev} yields

$$\ell_\phi(Q^{\text{rev}}) = \ell_{G_f}(Q^{\text{rev}}) + \phi(a) - \phi(s) \leq \phi(a) - \phi(s),$$

because every edge of Q^{rev} is a backward residual edge and thus has nonpositive original residual cost. Combining the two displays gives

$$\phi(a) - \phi(s) \geq -3\varepsilon\alpha n, \quad \text{i.e.,} \quad \phi(a) \geq \phi(s) - 3\varepsilon\alpha n \geq -50\alpha n - 3\varepsilon\alpha n > -99\alpha n,$$

using $\varepsilon \leq 0.1$. Thus a is active.

Now consider the suffix $P[a \rightsquigarrow v]$ of P starting at a and ending at v . By the choice of (b, a) as the last backward residual edge on P , every edge of $P[a \rightsquigarrow v]$ is a *forward* residual edge, hence has nonnegative original cost. For any forward residual admissible edge $x \rightarrow y$ we have

$$\ell_\phi(x, y) = \ell_G(x, y) + \phi(x) - \phi(y) \leq -\varepsilon\alpha w(x, y) \leq 0,$$

and since $\ell_G(x, y) \geq 0$, it follows that $\phi(y) \geq \phi(x)$. Therefore, potentials are nondecreasing along $P[a \rightsquigarrow v]$, and in particular $\phi(v) \geq \phi(a) > -99\alpha n$. This contradicts $\phi(v) < -99\alpha n$ and completes the proof. $\square$

The algorithm routes $\Omega(q)$ flow. We now show that upon termination we have routed a constant fraction of the guessed value q .

Lemma B.60. *At termination, the algorithm outputs a flow f with $\text{val}(f) \geq 0.9q$.*

Proof. Let termination mean $\phi(s) < -50\alpha n$. Note also that $\phi(t)$ never changes (we never relabel t), hence $\phi(t) = 0$.

Assume for contradiction that $\text{val}(f) < 0.9q$. Since $\text{val}(f^*) \geq q$, the value gap satisfies

$$\Delta := \text{val}(f^*) - \text{val}(f) \geq 0.1q.$$

Let g be the standard residual representation of the difference $f^* - f$ in G_f : for each original edge e we send $\max\{f^*(e) - f(e), 0\}$ units on the forward residual edge e , and $\max\{f(e) - f^*(e), 0\}$ units on the backward residual edge e^{rev} . Then g is a feasible s - t flow in G_f of value $\text{val}(g) = \Delta$. Moreover, since ℓ_G is nonnegative on original edges,

$$\ell_{G_f}(g) \leq \ell_G(f^*) \leq 2\alpha nq.$$

(The inequality $\ell_{G_f}(g) \leq \ell_G(f^*)$ holds because each backward residual edge has nonpositive cost, and $\ell_G(f) \geq 0$.)

We now derive a lower bound on $\ell_{G_f}(g)$ that contradicts this. By flow conservation and telescoping of potentials, for any s - t flow g we have

$$\ell_\phi(g) = \ell_{G_f}(g) + \phi(s)\text{val}(g) - \phi(t)\text{val}(g) = \ell_{G_f}(g) + (\phi(s) - \phi(t))\Delta. \quad (\text{B.5})$$

Rearranging and using $\phi(t) = 0$ and $\phi(s) < -50\alpha n$ gives

$$\ell_{G_f}(g) = \ell_\phi(g) + (\phi(t) - \phi(s))\Delta \geq \ell_\phi(g) + 50\alpha n \cdot \Delta. \quad (\text{B.6})$$

Next, apply [Lemma B.59](#) to lower bound $\ell_\phi(g)$:

$$\ell_\phi(g) = \sum_{e \in E(G_f)} g(e) \ell_\phi(e) \geq -3\varepsilon\alpha \sum_{e \in E(G_f)} g(e) w(e).$$

It remains to upper bound the weighted sum $\sum_e g(e)w(e)$. For any s - t flow h on a DAG, define its w -weight by $W(h) := \sum_e h(e)w(e)$. Since G is a DAG, h decomposes into s - t paths; for any such path p ,

$$\sum_{(v_i, v_j) \in p} w(v_i, v_j) = \sum_{(v_i, v_j) \in p} (j - i) = (\text{index}(t) - \text{index}(s)) \leq n,$$

hence $W(h) \leq n \cdot \text{val}(h)$. Applying this to f^* and f and noting that g uses at most the forward part of f^* and the backward part of f , we obtain

$$\sum_e g(e)w(e) \leq W(f^*) + W(f) \leq n\text{val}(f^*) + n\text{val}(f) \leq n \cdot (2q + 0.9q) < 3nq.$$

Therefore,

$$\ell_\phi(g) \geq -3\varepsilon\alpha \cdot 3nq = -9\varepsilon\alpha nq.$$

Plugging into [\(B.6\)](#) and using $\Delta \geq 0.1q$ yields

$$\ell_{G_f}(g) \geq -9\varepsilon\alpha nq + 50\alpha n \cdot 0.1q = (5 - 9\varepsilon)\alpha nq > 2\alpha nq,$$

for $\varepsilon \leq 0.1$. This contradicts $\ell_{G_f}(g) \leq 2\alpha nq$, completing the proof. $\square$

Primal–dual gap. We will use the following exact (and standard) identity, which cleanly formalizes the “slackness $\Rightarrow$ small gap” step.

Lemma B.61 (Exact primal–dual gap identity). *Consider the min-cost flow primal/dual pair:*

$$(P) \quad \min \sum_{e \in E} \ell(e)f(e) \quad \text{s.t. flow conservation and } 0 \leq f(e) \leq U(e),$$

$$(D) \quad \max \sum_{u \in V} (-D(u))\psi(u) - \sum_{e \in E} U(e)C(e) \quad \text{s.t. } \psi(v) - \psi(u) \leq \ell(e) + C(e), \quad C(e) \geq 0.$$

For any primal-feasible f and any dual-feasible (ψ, C) ,

$$\sum_{e \in E} \ell(e)f(e) - \left(\sum_{u \in V} (-D(u))\psi(u) - \sum_{e \in E} U(e)C(e) \right) = \sum_{e \in E} f(e)\gamma(e) + \sum_{e \in E} (U(e) - f(e))C(e), \quad (\text{B.7})$$

where $\gamma(e) := \ell(e) + C(e) - (\psi(v) - \psi(u)) \geq 0$ is the dual slack on $e = (u, v)$.

Proof. Expand the dual objective and use flow conservation:

$$\sum_{u \in V} D(u)\psi(u) = \sum_{u \in V} \psi(u) \left(\sum_{e \in \delta^+(u)} f(e) - \sum_{e \in \delta^-(u)} f(e) \right) = \sum_{e=(u,v) \in E} f(e)(\psi(u) - \psi(v)).$$

Hence

$$\sum_e \ell(e)f(e) + \sum_u D(u)\psi(u) + \sum_e U(e)C(e) = \sum_e f(e)(\ell(e) + \psi(u) - \psi(v)) + \sum_e U(e)C(e).$$

Add and subtract $\sum_e f(e)C(e)$ and rearrange to obtain [\(B.7\)](#). $\square$

Lemma B.62. *Let $\phi^* := \phi + \phi'$, where ϕ' is computed as in the analysis (with $\phi'(s) = 0$), and let C be the algorithm's output. Then the primal–dual gap satisfies*

$$\ell_G(f) - \left(\sum_{u \in V} (-D(u))\phi^*(u) - \sum_{e \in E} U(e)C(e) \right) \leq 6\varepsilon\alpha \sum_{e \in E} f(e)w(e) \leq 6\varepsilon\alpha n \cdot \text{val}(f).$$

Proof. We apply Lemma B.61 with $(\psi, C) = (\phi^*, C)$. By construction, $C(e) \geq 0$ and (ϕ^*, C) is dual-feasible (the algorithm sets $C(e)$ exactly to fix the upper-capacity constraints on saturated edges).

It remains to upper bound the two nonnegative terms in (B.7). We claim:

(i) **Saturated edges do not contribute to the second term.** If $C(e) > 0$, then e is saturated, i.e. $f(e) = U(e)$, hence $(U(e) - f(e))C(e) = 0$. Therefore the second sum in (B.7) vanishes.

(ii) **Edges with $f(e) > 0$ have slack $\gamma(e) \leq 6\varepsilon\alpha w(e)$.** Fix an original edge $e = (u, v)$ with $f(e) > 0$. Then the reverse residual edge (v, u) exists in G_f . By the way ϕ' is defined in the analysis (as a distance potential on the modified residual graph), we have for this residual edge the inequality

$$(-\ell(e)) + \phi^*(v) - \phi^*(u) \geq -6\varepsilon\alpha w(e),$$

equivalently,

$$\phi^*(v) - \phi^*(u) \geq \ell(e) - 6\varepsilon\alpha w(e). \quad (\text{B.8})$$

On the other hand, if $f(e) < U(e)$ then the forward residual edge (u, v) also exists, and the same reasoning gives $\ell(e) + \phi^*(u) - \phi^*(v) \geq 0$, i.e. $\phi^*(v) - \phi^*(u) \leq \ell(e)$. If $f(e) = U(e)$, dual feasibility ensures $\phi^*(v) - \phi^*(u) \leq \ell(e) + C(e)$. In both cases,

$$\gamma(e) = \ell(e) + C(e) - (\phi^*(v) - \phi^*(u)) \leq \ell(e) + C(e) - (\ell(e) - 6\varepsilon\alpha w(e)) = C(e) + 6\varepsilon\alpha w(e).$$

Moreover, when $f(e) > 0$ and $f(e) < U(e)$ we have $C(e) = 0$ (by construction, C is only needed for saturated edges), so $\gamma(e) \leq 6\varepsilon\alpha w(e)$; when $f(e) = U(e)$ we also have $\gamma(e) = 0$ by definition of $C(e)$. Thus in all cases with $f(e) > 0$ we have $\gamma(e) \leq 6\varepsilon\alpha w(e)$.

Combining (i) and (ii) with (B.7) gives

$$\ell_G(f) - \text{Dual}(\phi^*, C) = \sum_e f(e)\gamma(e) \leq 6\varepsilon\alpha \sum_e f(e)w(e).$$

Finally, every unit of s – t flow can be decomposed into an s – t path following the topological order, and along such a path the w -weights telescope to at most n ; hence $\sum_e f(e)w(e) \leq n \cdot \text{val}(f)$. This yields the stated bound. $\square$

B.8.3 Complexity (Implementation Details)

We implement the algorithm almost identically to Appendix A of [Ber+24] using dynamic trees (link–cut trees). For completeness we spell out the implementation details here. The only new feature is that, when a newly chosen admissible edge would create a directed cycle in the maintained structure, we immediately augment along that cycle (cycle-cancel) before continuing.

Bounding the number of augmentations. We first bound how many augmentations (Push operations) occur.

Lemma B.63. *For any directed graph on vertices ordered as $(v_1, \dots, v_n)$, letting $w(v_i, v_j) = |j - i|$, we have*

$$\sum_{e \in E} \frac{1}{w(e)} \leq \sum_{d=1}^{n-1} \frac{n-d}{d} = O(n \log n) = \tilde{O}(n).$$

Proof. Group edges by their topological distance $d = w(e)$. For a fixed $d \in \{1, \dots, n-1\}$, there are at most

$n - d$ ordered pairs $(i, i + d)$, hence at most $n - d$ edges with $w(e) = d$. Therefore

$$\sum_{e \in E} \frac{1}{w(e)} \leq \sum_{d=1}^{n-1} \frac{n-d}{d} = n \sum_{d=1}^{n-1} \frac{1}{d} - \sum_{d=1}^{n-1} 1 = O(n \log n).$$

□

Lemma B.64. *Throughout the algorithm, the total number of augmentations is $\tilde{O}(n^2/\varepsilon)$. Moreover, any fixed original edge e (equivalently, its two residual directions over time) becomes saturated at most $O\left(\frac{n}{\varepsilon w(e)}\right)$ times.*

Proof. Every augmentation (along an s - t path or a directed cycle) sends

$$\Delta = \min_{a \in P} u_f(a)$$

units and therefore saturates at least one residual edge on the augmented object P . Thus it suffices to bound the total number of saturation events.

Fix an original edge $e = (v_i, v_j)$ and write $d := w(e) = |j - i|$. Consider two consecutive times at which *the same direction* of e becomes saturated as a residual edge. At the moment a residual edge $a = (x, y)$ is saturated in an augmentation, it must be admissible, hence by [Lemma B.59](#)

$$\ell_\phi(a) \leq -\varepsilon\alpha w(a) = -\varepsilon\alpha d.$$

When a is saturated, the reverse residual edge $a^{\text{rev}} = (y, x)$ is created (or its capacity increases), and

$$\ell_\phi(a^{\text{rev}}) = -\ell_\phi(a) \geq \varepsilon\alpha d.$$

For a^{rev} to become admissible (and hence be eligible to be saturated in some later augmentation), we must have $\ell_\phi(a^{\text{rev}}) \leq -\varepsilon\alpha d$ at that later time. Therefore, between the creation of a^{rev} and the time it is next saturated, its reduced length must drop by at least $2\varepsilon\alpha d$.

Now observe that for a fixed residual edge (y, x) ,

$$\ell_\phi(y, x) = \ell_G(y, x) + \phi(y) - \phi(x),$$

and potentials *never increase* (they only decrease under relabels). Hence the only way $\ell_\phi(y, x)$ can decrease by $2\varepsilon\alpha d$ is that $\phi(y)$ decreases by at least $2\varepsilon\alpha d$ (since $\phi(x)$ cannot increase to help). Each vertex potential decreases in steps of size $\varepsilon\alpha$, and a vertex is relabeled only while active, so over the entire run each vertex u satisfies

$$0 \geq \phi(u) \geq -99\alpha n,$$

i.e., $\phi(u)$ decreases by at most $99\alpha n$ in total. Charging $2\varepsilon\alpha d$ units of potential decrease of the tail endpoint each time e is saturated (in the appropriate direction), we conclude that e can be saturated at most

$$O\left(\frac{99\alpha n}{2\varepsilon\alpha d}\right) = O\left(\frac{n}{\varepsilon w(e)}\right)$$

times overall.

Summing over all edges and using [Lemma B.63](#), the total number of saturation events (hence augmentations) is

$$O\left(\sum_{e \in E} \frac{n}{\varepsilon w(e)}\right) = O\left(\frac{n}{\varepsilon}\right) \cdot \sum_{e \in E} \frac{1}{w(e)} = \tilde{O}(n^2/\varepsilon).$$

□

Dynamic trees. We use a standard link-cut tree data structure [[ST83](#); [GT88](#)].

Lemma B.65 (Dynamic trees [ST83; GT88]). *There is a data structure that maintains a collection of rooted trees $\mathcal{J}$ together with values $\nu \in \mathbb{R}^{E(\mathcal{J})}$ on the tree edges. It supports the following operations in amortized $O(\log n)$ time:*

- **LINK**(u, v, ν): *add an edge making v the parent of u with stored value $\nu(u, v) \leftarrow \nu$. Requires that u is currently a root and u, v are in different trees.*
- **CUT**(u): *delete the parent edge of u (if any), making u a root.*
- **FINDROOT**(u): *return the root of the tree containing u .*
- **FINDMIN**(u): *return an edge of minimum ν on the path from u to its root (ties broken toward u).*
- **ADD**(u, x): *add x to $\nu(e)$ for every edge e on the path from u to its root.*

Maintaining an admissible pseudoforest. During a Push phase (i.e., when no more relabels are possible), every active vertex has at least one outgoing admissible residual edge. For each active vertex $u \neq t$, we maintain *one chosen admissible outgoing edge* $p(u) = (u, v)$, called its *parent edge*. The chosen parent edges define a directed pseudoforest (outdegree at most 1). We maintain this structure inside dynamic trees as an *actual forest* by immediately canceling any cycle that would be created:

- When u becomes active and currently has no parent edge in $\mathcal{J}$, choose some admissible outgoing residual edge $e = (u, v)$. If u is a root and u and v are in different trees, we insert it via **LINK**($u, v, u_f(e)$).
- If inserting $e = (u, v)$ would create a directed cycle (i.e., u is a root but **FINDROOT**(v) = u), then the cycle consists of the path from v to u in the current tree plus the edge (u, v) . We augment along this directed cycle by the bottleneck residual capacity (found via **FINDMIN**(v) and the residual of (u, v)), update residual capacities using **ADD**($v, -\Delta$) along the tree path, and decrease the residual of (u, v) by Δ . At least one edge on the cycle becomes saturated; we delete every saturated tree edge by cutting it from $\mathcal{J}$. After the cycle is broken, we retry linking u to some admissible outgoing edge (possibly the same one if it remains residual).

Whenever an edge is marked inadmissible (or its residual capacity drops to 0), if it is currently used as a parent edge in $\mathcal{J}$ we remove it using **CUT**($\cdot$) and then pick a replacement admissible outgoing edge for that child (if the child is active). All such changes take $O(\log n)$ time per link/cut/update.

Augmenting on an s – t path. When the structure is cycle-free in the component containing s , the chosen parent pointers define a (tree) path from s to its root. By **Lemma B.57** (and the cycle-cancel rule above), this root is t . We then augment along the unique tree path from s to t : we compute the bottleneck Δ via **FINDMIN**(s) and apply **ADD**($s, -\Delta$). After this update, at least one tree edge on the path has residual capacity 0; we repeatedly call **FINDMIN**(s) and delete any returned edge with $\nu = 0$ (cutting it from $\mathcal{J}$ and updating the corresponding residual reverse edge in G_f). Each such deletion corresponds to a saturation event.

Total running time. We bound the total number of expensive events.

- **Relabels.** Each relabel decreases $\phi(u)$ by $\varepsilon\alpha$, and $\phi(u)$ decreases from 0 to below $-99\alpha n$ at most. Hence each vertex is relabeled $O(n/\varepsilon)$ times and the total number of relabels is $O(n^2/\varepsilon)$.
- **Edge checks.** An edge $e = (u, v)$ is checked only when $\phi(u)$ or $\phi(v)$ hits a multiple of $\varepsilon\alpha w(e)$. Since each endpoint potential decreases by at most $O(\alpha n)$, the number of checks of e is $O\left(\frac{\alpha n}{\varepsilon\alpha w(e)}\right) = O\left(\frac{n}{\varepsilon w(e)}\right)$. Summing over edges and using **Lemma B.63** gives a total of $\tilde{O}(n^2/\varepsilon)$ edge checks. Each check causes $O(1)$ admissibility work plus $O(\log n)$ dynamic-tree updates in the worst case.
- **Augmentations and saturated-edge deletions.** By **Lemma B.64**, the total number of augmentation steps is $\tilde{O}(n^2/\varepsilon)$, and the total number of saturated-edge deletions from $\mathcal{J}$ is also $\tilde{O}(n^2/\varepsilon)$. Each such event costs $O(\log n)$ amortized time.

Putting everything together, the total running time is $\tilde{O}(n^2/\varepsilon)$.

B.9 Missing Proofs in Section B.6

B.9.1 Union of Sparse Moving Cuts

In this section, we will prove Lemma B.33.

We first need the following lemma from [Hae+25b].

Lemma B.66 (Lemma 3.17 of [Hae+25b]). *Let $C_1, \dots, C_k$ be an sequence of (h, s) -length ϕ -sparse for some node-weighting A in the graph G where $h > 1$, $\phi < 1$, $1 \leq \alpha \leq \log n$ and $s > \frac{4 \log_2 n}{\alpha}$, then $\sum_i |C_i| \leq (2^{8\alpha+2} \phi \ln n) \cdot |A|$.*

We can generalize Lemma B.66 to allow the cut sequence to be a subsequence as follows.

Lemma B.67. *Let A be a node-weighting on a graph G with positive integral edge lengths and capacities. Let $C_1, \dots, C_k$ be a sequence of moving cuts, and let*

$$C_{a_1}, \dots, C_{a_z}$$

be a subsequence of $C_1, \dots, C_k$. Let $h > 1$, $\phi \in (0, 1)$, $1 \leq \alpha \leq \log n$, and

$$s > \frac{4 \log_2 n}{\alpha}.$$

Suppose that for every $r \in [z]$, the cut C_{a_r} is (h, s) -length ϕ -sparse with respect to A in the graph

$$G - \sum_{j < a_r} C_j.$$

Then

$$\sum_{r=1}^z |C_{a_r}| \leq (2^{8\alpha+2} \phi \ln n) \cdot |A|.$$

Proof. This is proved by the same potential argument as in the proof of Lemma 3.17 of [Hae+25b], with only one change: instead of summing the potential drops over all cuts, we sum only over the subsequence $C_{a_1}, \dots, C_{a_z}$.

More precisely, define

$$G_1 := G, \quad G_i := G - \sum_{j < i} C_j \quad \text{for every } i \in [k+1].$$

For each i , let w_i denote the exponential distance weight w_h^α with respect to the graph G_i , and let D_i denote the corresponding exponential demand $D_{h,A}^i$. Also define the potential

$$P_i(u) := A(u) \ln(w_i(u)) \quad \forall u \in V.$$

Fix any $r \in [z]$. Since C_{a_r} is (h, s) -length ϕ -sparse with respect to A in G_{a_r} , the same argument as in Lemma 3.17 of [Hae+25b] gives

$$\text{sep}_{h \cdot s/2}^{a_r}(C_{a_r}, D_{a_r}) \geq 2^{-8\alpha-1} \cdot \frac{|C_{a_r}|}{\phi},$$

and therefore

$$\sum_{u \in V} (P_{a_r}(u) - P_{a_r+1}(u)) \geq 2^{-8\alpha-2} \cdot \frac{|C_{a_r}|}{\phi}.$$

Equivalently,

$$|C_{a_r}| \leq 2^{8\alpha+2} \phi \sum_{u \in V} (P_{a_r}(u) - P_{a_r+1}(u)).$$

Summing over all $r \in [z]$, we obtain

$$\sum_{r=1}^z |C_{a_r}| \leq 2^{8\alpha+2} \phi \sum_{r=1}^z \sum_{u \in V} (P_{a_r}(u) - P_{a_r+1}(u)).$$

Now the key observation is that the potential is monotone under arbitrary length increases: for every i and every $u \in V$,

$$P_i(u) \geq P_{i+1}(u),$$

since passing from G_i to G_{i+1} only increases edge lengths. Hence all terms in the above sum are nonnegative, and so

$$\sum_{r=1}^z \sum_{u \in V} (P_{a_r}(u) - P_{a_r+1}(u)) \leq \sum_{i=1}^k \sum_{u \in V} (P_i(u) - P_{i+1}(u)).$$

The latter telescopes to

$$\sum_{i=1}^k \sum_{u \in V} (P_i(u) - P_{i+1}(u)) \leq \sum_{u \in V} P_1(u).$$

Therefore,

$$\sum_{r=1}^z |C_{a_r}| \leq 2^{8\alpha+2} \phi \sum_{u \in V} P_1(u) = 2^{8\alpha+2} \phi \sum_{u \in V} A(u) \ln(w_1(u)).$$

Finally, as in Lemma 3.17 of [Hae+25b], we have $\ln(w_1(u)) \leq \ln n$ for all u , and so

$$\sum_{r=1}^z |C_{a_r}| \leq (2^{8\alpha+2} \phi \ln n) \cdot |A|.$$

□

Now we are ready to prove [Lemma B.33](#).

Lemma B.33. *Let A be a node-weighting on a graph G with positive integral edge lengths and capacities. Let $C_1, \dots, C_k$ be a sequence of moving cuts. Let $h_1, \dots, h_k > 1$, $\phi \in (0, 1)$, $1 \leq \alpha \leq \log n$, and*

$$s > \frac{8 \log_2 n}{\alpha}.$$

Suppose that for every $i \in [k]$, the cut C_i is (h_i, s) -length ϕ -sparse with respect to A in the graph

$$G - \sum_{j < i} C_j.$$

Then

$$\sum_{i=1}^k |C_i| \leq O(2^{8\alpha+2} \phi \log^2 n) \cdot |A|.$$

Proof. Partition the index set $[k]$ into dyadic buckets according to the values h_i . For each integer ℓ , let

$$I_\ell := \{i \in [k] : 2^\ell < h_i \leq 2^{\ell+1}\}.$$

Only $O(\log H)$ of these buckets are nonempty.

Fix one nonempty bucket I_ℓ , and let

$$\widehat{h}_\ell := 2^{\ell+1}.$$

For each $i \in I_\ell$, define a rescaled cut

$$\widehat{C}_i := \frac{2h_i}{\widehat{h}_\ell} C_i = \frac{h_i}{2^\ell} C_i.$$

Since $2^\ell < h_i \leq 2^{\ell+1}$, we have

$$1 < \frac{2h_i}{\widehat{h}_\ell} \leq 2,$$

and therefore

$$|C_i| \leq |\widehat{C}_i| \leq 2|C_i|.$$

We now view $\widehat{C}_i$ as a $\widehat{h}_\ell \cdot (s/2)$ -length moving cut. Indeed,

$$\widehat{h}_\ell \cdot \frac{s}{2} \cdot \widehat{C}_i(e) = 2^{\ell+1} \cdot \frac{s}{2} \cdot \frac{h_i}{2^\ell} C_i(e) = h_i s C_i(e),$$

so $\widehat{C}_i$ induces exactly the same edge-length increases as C_i . Hence

$$G - \widehat{C}_i = G - C_i.$$

Next, let D_i^* be a witnessing A -respecting h_i -length symmetric demand showing that C_i is (h_i, s) -length ϕ -sparse. Since $h_i \leq \widehat{h}_\ell$, the same demand D_i^* is also A -respecting $\widehat{h}_\ell$ -length symmetric. Moreover,

$$\widehat{h}_\ell \cdot \frac{s}{2} = 2^\ell s \leq h_i s.$$

Therefore every demand pair that is $h_i s$ -separated by C_i is also $\widehat{h}_\ell(s/2)$ -separated by $\widehat{C}_i$. Hence

$$\text{sep}_{\widehat{h}_\ell(s/2)}(\widehat{C}_i, D_i^*) \geq \text{sep}_{h_i s}(C_i, D_i^*).$$

Combining this with $|\widehat{C}_i| \leq 2|C_i|$, we obtain

$$|\widehat{C}_i| \leq 2|C_i| \leq 2\phi \cdot \text{sep}_{h_i s}(C_i, D_i^*) \leq 2\phi \cdot \text{sep}_{\widehat{h}_\ell(s/2)}(\widehat{C}_i, D_i^*).$$

Thus $\widehat{C}_i$ is $(\widehat{h}_\ell, s/2)$ -length 2ϕ -sparse with respect to A in the graph

$$G - \sum_{j < i} C_j = G - \sum_{j < i} \widehat{C}_j,$$

where equality again holds because C_j and $\widehat{C}_j$ induce the same metric.

Now apply [Lemma B.67](#) to the subsequence

$$\{\widehat{C}_i : i \in I_\ell\}$$

with parameters $\widehat{h}_\ell$, 2ϕ , and $s/2$. This is valid because

$$\frac{s}{2} > \frac{4 \log_2 n}{\alpha}$$

by assumption. Therefore,

$$\sum_{i \in I_\ell} |\widehat{C}_i| \leq (2^{8\alpha+2} \cdot 2\phi \ln n) \cdot |A| = O(2^{8\alpha+2} \phi \ln n) \cdot |A|.$$

Since $|C_i| \leq |\widehat{C}_i|$, we also get

$$\sum_{i \in I_\ell} |C_i| \leq O(2^{8\alpha+2} \phi \ln n) \cdot |A|.$$

Finally, summing over all nonempty buckets gives

$$\sum_{i=1}^k |C_i| \leq O(\log H) \cdot O(2^{8\alpha+2} \phi \ln n) \cdot |A| = O(2^{8\alpha+2} \phi \ln n \cdot \log H) \cdot |A|.$$

If $H \leq n^{O(1)}$, then $\log H = O(\log n)$, and hence

$$\sum_{i=1}^k |C_i| \leq O(2^{8\alpha+2} \phi \log^2 n) \cdot |A|.$$

B.9.2 FatLDD on Directed Graphs

In this section, we will fill in the missing proofs in [Section B.6.1](#).

We first show that it is possible to construct an undirected graph approximating the round-trip distance of a directed graph. We need the following lemma about randomized low-diameter decomposition.

Lemma B.68 ([Ash+24a]). *There is a randomized algorithm that, given a directed graph $G = (V, E)$ with edge length ℓ and a parameter d , makes $\tilde{O}(1)$ calls to an exact SSSP oracle and computes a distribution of a partition of the vertex set $V = V_1 \uplus V_2 \dots \uplus V_k$ such that*

- *for each edge $e = (u, v)$, the probability that e is reversed, i.e., $u \in V_i, v \in V_j$ with $j < i$, is at most $O(\log^2 n) \cdot w(e)/d$.*
- *for every $i \in [k]$ and every $u, v \in V_i$, we have $\text{dist}_G(u, v) \leq d$.*

Lemma B.69. *There is a randomized algorithm that, given a directed graph $G = (V, E)$ with edge lengths ℓ , constructs an undirected graph $G' = (V, E')$ using $\tilde{O}(1)$ calls to an exact directed SSSP oracle such that, with high probability, for every $u, v \in V$,*

$$\widehat{\text{dist}}_G(u, v) \leq \text{dist}_{G'}(u, v) \leq O(\log^2 n) \cdot \widehat{\text{dist}}_G(u, v).$$

Proof. For each $d = 2^i$ from $i = 1$ to $i = C \cdot \log n$ for sufficiently large C , we construct $O(\log n)$ independent LDDs with parameter d using [Lemma B.68](#). For each computed LDD $V_1, \dots, V_k$ with parameter d , and each $j \in [k]$, we add edges from every node in V_j to a single node $w \in V_j$ with length d , which results in the final graph $G' = (V, E)$. Notice that constructing LDD makes $\tilde{O}(1)$ calls to exact directed SSSP oracle, and connecting the edges has complexity subsumed by the LDD computation. It remains to prove the correctness.

Correctness. Since for every two nodes $u, v \in V_j$, it is from the LDD property that $\text{dist}_G(u, v) \leq d$, we immediately get one side of the inequality $\widehat{\text{dist}}_G(u, v) \leq \text{dist}_{G'}(u, v)$. For every $u, v \in V$, let us focus on the LDD with parameter $d = C \log^2 n \cdot \widehat{\text{dist}}(u, v)$ for sufficiently large constant C , denoted by $V_1, \dots, V_k$. By the choice of d , such d must exist. Consider the shortest path $p_{u,v}$ from u to v and the shortest path $p_{v,u}$ from v to u . We have that $\ell(p_{u,v}) + \ell(p_{v,u}) = \widehat{\text{dist}}(u, v)$. According to the property of LDD [Lemma B.68](#), with probability at least 0.1, no edges in either $p_{u,v}$ or $p_{v,u}$ will be a reversed edge for $V_1, \dots, V_k$, which implies that $u, v \in V_j$ for some $j \in [k]$. Thus, in G' , the distance between u, v is $2 \cdot d = O(\log^2 n) \cdot \widehat{\text{dist}}_G(u, v)$. Since we repeat the LDD for $O(\log n)$ independent times, one of them will satisfy this inequality w.h.p. Applying union bound over all pairs of vertices gives the final result. □

Lemma B.40. *For any $h_{\text{cov}} > 0$, $s > 1$, and $\varepsilon \in (0, 1)$, and any directed graph $G = (V, E)$ with edge lengths ℓ , there is a randomized algorithm that computes a neighborhood cover with respect to $\widehat{\text{dist}}_G$, together with separation witnesses, with*

- *covering radius h_{cov} ,*
- *separation factor s ,*
- *diameter bound*

$$h_{\text{diam}} = \frac{1}{\varepsilon} \cdot O(s \log^2 n)^{O(1/\varepsilon)} \cdot h_{\text{cov}},$$

- *width*

$$\omega = n^{O(\varepsilon)} \cdot \log n.$$

The algorithm runs in $\hat{O}(m)$ time given that both $h_{\text{diam}}/h_{\text{cov}}$ and ω are in the order of $n^{o(1)}$.

We need the following lemma for undirected graphs.

Lemma B.70 (Theorem 5.32 of [HHT24]). *For any $h_{\text{cov}} > 0$, $s > 1$, and $\varepsilon \in (0, 1)$, and any undirected graph $G = (V, E)$ with edge lengths ℓ , there is an algorithm that computes a neighborhood cover with respect to dist_G with covering radius h_{cov} , separation factor s , diameter*

$$h_{\text{diam}} = \frac{1}{\varepsilon} \cdot O(s)^{O(1/\varepsilon)} \cdot h_{\text{cov}},$$

and width

$$\omega = n^{O(\varepsilon)} \cdot \log n.$$

The algorithm runs in $m \cdot h_{\text{diam}} \cdot n^{O(\varepsilon)}$ time.

Proof of Lemma B.40. Let $\alpha = O(\log^2 n)$ denote the distortion in Lemma B.69, so that with high probability

$$\widehat{\text{dist}}_G(u, v) \leq \text{dist}_{G'}(u, v) \leq \alpha \cdot \widehat{\text{dist}}_G(u, v) \quad \text{for all } u, v \in V.$$

Condition on this event.

Algorithm. We construct the desired neighborhood cover in three steps.

1. Apply Lemma B.69 to G and obtain an undirected graph G' .
2. Apply the hopset construction from [Coh00] to G' . This yields a set H of $n^{1+o(1)}$ edges such that:
 - every hopset edge $(x, y) \in H$ has length $\text{dist}_{G'}(x, y)$, and hence

$$\text{dist}_{G' \cup H}(u, v) = \text{dist}_{G'}(u, v) \quad \text{for all } u, v \in V;$$

- for every pair $u, v \in V$, there exists a path in $G' \cup H$ from u to v using at most $h = n^{o(1)}$ edges and having length at most $1.1 \cdot \text{dist}_{G'}(u, v)$.

3. Define

$$\lambda := \frac{\alpha h_{\text{cov}}}{h}.$$

Let $\widehat{G}$ be the graph on vertex set V and edge set $E(G') \cup H$ in which every edge e of original length $w(e)$ is assigned the new length

$$\widehat{\ell}(e) := \left\lceil \frac{w(e)}{\lambda} \right\rceil.$$

Since all edge lengths in $\widehat{G}$ are positive integers, Lemma B.70 applies to $\widehat{G}$.

We invoke Lemma B.70 on $\widehat{G}$ with covering radius $10h$ and separation factor $40\alpha s$. Let

$$\widehat{\mathcal{N}} = \{\widehat{\mathcal{S}}_1, \dots, \widehat{\mathcal{S}}_\omega\}$$

denote the resulting neighborhood cover. For each $i \in [\omega]$, let $\widehat{h}_i$ be the diameter parameter of the clustering $\widehat{\mathcal{S}}_i$. Thus

$$10h \leq \widehat{h}_i \leq \widehat{h}_{\text{diam}} \quad \text{for all } i \in [\omega],$$

where

$$\widehat{h}_{\text{diam}} = \frac{1}{\varepsilon} \cdot O(\alpha s)^{O(1/\varepsilon)} \cdot h.$$

For each clustering $\widehat{\mathcal{S}}_i$, we output the same partition of vertices, now viewed as a clustering in the metric $\widehat{\text{dist}}_G$, and assign it diameter

$$h_i := \lambda \widehat{h}_i.$$

For each cluster $S \in \widehat{\mathcal{S}}_i$, define its witness set by

$$S' := \text{ball}_{\widehat{G}}(S, 10\alpha s \widehat{h}_i).$$

We claim that the resulting family is the desired neighborhood cover in G .

Correctness analysis. We verify the covering property, the diameter bound, the separation guarantee, and the existence of separation witnesses.

Covering radius. Fix any vertex $u \in V$ and let $v \in \text{ball}_{\widehat{\text{dist}}_G}(u, h_{\text{cov}})$. Then

$$\text{dist}_{G'}(u, v) \leq \alpha h_{\text{cov}}.$$

By the hopset guarantee, there exists a path in $G' \cup H$ from u to v with at most h edges and total length at most $1.1\alpha h_{\text{cov}}$. After scaling and rounding, the length of this path in $\widehat{G}$ is at most

$$\frac{1.1\alpha h_{\text{cov}}}{\lambda} + h = 1.1h + h < 10h.$$

Hence

$$\text{ball}_{\widehat{\text{dist}}_G}(u, h_{\text{cov}}) \subseteq \text{ball}_{\widehat{G}}(u, 10h).$$

Since $\widehat{\mathcal{N}}$ is a neighborhood cover in $\widehat{G}$ with covering radius $10h$, there exists a cluster in some $\widehat{\mathcal{S}}_i$ containing $\text{ball}_{\widehat{G}}(u, 10h)$, and therefore also containing $\text{ball}_{\widehat{\text{dist}}_G}(u, h_{\text{cov}})$. Thus the output has covering radius h_{cov} with respect to $\widehat{\text{dist}}_G$.

Diameter bound. Fix any clustering $\widehat{\mathcal{S}}_i$ and any two vertices x, y in the same cluster of $\widehat{\mathcal{S}}_i$. Since that clustering has diameter $\widehat{h}_i$ in $\widehat{G}$,

$$\text{dist}_{\widehat{G}}(x, y) \leq \widehat{h}_i.$$

By construction of $\widehat{G}$, every edge e satisfies

$$w(e) \leq \lambda \cdot \widehat{\ell}(e).$$

Therefore every path of length L in $\widehat{G}$ corresponds to a path in $G' \cup H$ of length at most λL , and hence

$$\text{dist}_{G'}(x, y) = \text{dist}_{G' \cup H}(x, y) \leq \lambda \cdot \text{dist}_{\widehat{G}}(x, y) \leq \lambda \widehat{h}_i = h_i.$$

Using $\widehat{\text{dist}}_G(x, y) \leq \text{dist}_{G'}(x, y)$, we get

$$\widehat{\text{dist}}_G(x, y) \leq h_i.$$

Thus the i th output clustering has diameter h_i .

Moreover,

$$h_i \leq \lambda \widehat{h}_{\text{diam}} = \frac{\alpha h_{\text{cov}}}{h} \cdot \frac{1}{\varepsilon} \cdot O(\alpha s)^{O(1/\varepsilon)} \cdot h = \frac{1}{\varepsilon} \cdot O(\alpha s)^{O(1/\varepsilon)} \cdot \alpha h_{\text{cov}}.$$

Since $\alpha = O(\log^2 n)$, this is

$$h_i \leq h_{\text{diam}} := \frac{1}{\varepsilon} \cdot O(s \log^2 n)^{O(1/\varepsilon)} \cdot h_{\text{cov}}.$$

Hence the output has diameter bound h_{diam} .

Separation factor. Fix a clustering $\widehat{\mathcal{S}}_i$, and let S, T be two distinct clusters in it. We show that

$$\widehat{\text{dist}}_G(S, T) \geq s h_i.$$

Suppose for contradiction that there exist $x \in S$ and $y \in T$ such that

$$\widehat{\text{dist}}_G(x, y) < s h_i = s \lambda \widehat{h}_i.$$

Then

$$\text{dist}_{G'}(x, y) \leq \alpha \widehat{\text{dist}}_G(x, y) < \alpha s \lambda \widehat{h}_i.$$

By the hopset guarantee, there exists a path in $G' \cup H$ from x to y using at most h edges and having total length at most

$$1.1\alpha s \lambda \widehat{h}_i.$$

After scaling and rounding, this yields

$$\text{dist}_{\widehat{G}}(x, y) \leq 1.1\alpha s \widehat{h}_i + h.$$

Since $\widehat{h}_i \geq 10h$, we have

$$1.1\alpha s \widehat{h}_i + h \leq \left(1.1 + \frac{1}{10}\right) \alpha s \widehat{h}_i < 40\alpha s \widehat{h}_i,$$

contradicting the fact that distinct clusters in $\widehat{\mathcal{S}}_i$ are at distance at least

$$40\alpha s \widehat{h}_i$$

in $\widehat{G}$. Therefore

$$\widehat{\text{dist}}_G(S, T) \geq s h_i,$$

and the output clustering has separation factor s .

Separation witnesses. Fix a clustering $\widehat{\mathcal{S}}_i$, and recall that for each cluster $S \in \widehat{\mathcal{S}}_i$ we defined

$$S' := \text{ball}_{\widehat{G}}(S, 10\alpha s \widehat{h}_i).$$

We first show that these witness sets are pairwise disjoint. Indeed, if $S \neq T$ and some vertex belonged to both S' and T' , then by the triangle inequality in $\widehat{G}$ we would have

$$\text{dist}_{\widehat{G}}(S, T) \leq 20\alpha s \widehat{h}_i,$$

contradicting the separation guarantee

$$\text{dist}_{\widehat{G}}(S, T) \geq 40\alpha s \widehat{h}_i.$$

Now let x be any vertex with

$$\widehat{\text{dist}}_G(x, S) \leq \frac{s h_i}{2} = \frac{s \lambda \widehat{h}_i}{2}.$$

Then for some $y \in S$,

$$\widehat{\text{dist}}_G(x, y) \leq \frac{s \lambda \widehat{h}_i}{2}, \quad \text{hence} \quad \text{dist}_{G'}(x, y) \leq \frac{\alpha s \lambda \widehat{h}_i}{2}.$$

Applying the hopset guarantee once more, there exists a path from x to y in $G' \cup H$ using at most h edges and having total length at most

$$1.1 \cdot \frac{\alpha s \lambda \widehat{h}_i}{2}.$$

After scaling and rounding, this gives

$$\text{dist}_{\widehat{G}}(x, S) \leq 0.55\alpha s \widehat{h}_i + h.$$

Since $\widehat{h}_i \geq 10h$, we obtain

$$0.55\alpha s \widehat{h}_i + h < 10\alpha s \widehat{h}_i,$$

and therefore $x \in S'$. Thus

$$\text{ball}_{\widehat{\text{dist}}_G}\left(S, \frac{s h_i}{2}\right) \subseteq S'.$$

So the sets $\{S' : S \in \widehat{\mathcal{S}}_i\}$ form a valid separation witness for the i th clustering.

This proves that the output is a neighborhood cover with respect to $\widehat{\text{dist}}_G$ with the claimed parameters, together with separation witnesses.

Running time analysis. The graph $\widehat{G}$ has

$$|E(\widehat{G})| = |E(G')| + |H| = m + n^{1+o(1)} = \hat{O}(m)$$

edges.

The work of the call to [Lemma B.70](#) on $\widehat{G}$ is

$$|E(\widehat{G})| \cdot \widehat{h}_{\text{diam}} \cdot n^{O(\varepsilon)}.$$

Since

$$\widehat{h}_{\text{diam}} = \frac{h}{\alpha h_{\text{cov}}} \cdot h_{\text{diam}},$$

the assumption that both $h_{\text{diam}}/h_{\text{cov}}$ and $\omega = n^{O(\varepsilon)} \log n$ are of order $n^{o(1)}$ implies that

$$\widehat{h}_{\text{diam}} = n^{o(1)} \quad \text{and} \quad n^{O(\varepsilon)} = n^{o(1)}.$$

Therefore the work of the FatLDD step is

$$\widehat{O}(m) \cdot n^{o(1)} \cdot n^{o(1)} = \widehat{O}(m).$$

The preprocessing steps [Lemma B.69](#) and the hopset construction also take $\widehat{O}(m)$ time. Hence the total running time is $\widehat{O}(m)$. $\square$

B.9.3 The Cutmatch Algorithm

In this section we will prove [Lemma B.43](#).

We first prove two lemmas about using DAG projection in the way of length-bounded flow.

At first sight, it is not obvious how to use the dual part of the projection algorithm together with an oracle for $\mathcal{O}_{\kappa, \lambda}^{\text{PDMCF}}$ on DAGs. The following lemma is a convenient bridge.

Lemma B.71. *There is an algorithm that takes as input a directed graph G with positive polynomially bounded integer edge capacities and lengths, and an s - t feasible demand D on G , and in time proportional to one call to $\mathcal{O}_{\kappa, \lambda}^{\text{PDMCF}}$ on G , outputs*

- a flow f routing D with congestion at most κ and average cost $h = \text{cost}(f)/\text{val}(f)$, and
- a h/λ -length moving cut C with size at most $\text{val}(f)$ such that $\text{dist}_{G-C}(s, t) \geq h/\lambda$.

Proof. Run $\mathcal{O}_{\kappa, \lambda}^{\text{PDMCF}}$ on (G, D) and let $(f, \phi, \tilde{\ell})$ be the output. By definition, f routes D with congestion at most κ ; set $h = \text{cost}(f)/\text{val}(f)$.

Let $S = \sum_{e \in E} U(e) \tilde{\ell}(e)$, and let $\widetilde{G}$ be the graph obtained from G by replacing each edge length $\ell(e)$ by $\ell(e) + \tilde{\ell}(e)$. Since $(f, \phi, \tilde{\ell})$ is a (κ, λ) -approximate PD-MCF, we have

$$\text{cost}(f) \leq \lambda(\text{val}(f) \cdot (\phi(t) - \phi(s)) - S),$$

so

$$\frac{h}{\lambda} \leq (\phi(t) - \phi(s)) - \frac{S}{\text{val}(f)}.$$

By dual feasibility, $\phi(v) - \phi(u) \leq \ell(e) + \tilde{\ell}(e)$ for every edge $e = (u, v)$, hence $\phi(t) - \phi(s) \leq \text{dist}_{\widetilde{G}}(s, t)$. Therefore

$$\frac{h}{\lambda} \leq \text{dist}_{\widetilde{G}}(s, t) - \frac{S}{\text{val}(f)}.$$

Let $L := \text{dist}_{\widetilde{G}}(s, t)$ and define $C(e) := \tilde{\ell}(e)/L$ for every edge e . Viewed as an L -length moving cut, $G - C$ is exactly $\widetilde{G}$, so $\text{dist}_{G-C}(s, t) = L$. Also,

$$|C| = \sum_{e \in E} U(e) C(e) = \frac{S}{L} \leq \text{val}(f),$$

where the last inequality follows from $h/\lambda \leq L - S/\text{val}(f)$.

It remains to show that the same cut values also form a (h/λ) -length moving cut with $\text{dist}_{G-C}(s, t) \geq h/\lambda$.

Since all edge lengths are positive and $h/\lambda \leq L$, for every s - t path P we have

$$\ell_G(P) + \frac{h/\lambda}{L} \tilde{\ell}(P) \geq \frac{h/\lambda}{L} (\ell_G(P) + \tilde{\ell}(P)).$$

The right-hand side is at least $(h/\lambda) \cdot \text{dist}_{\tilde{G}}(s, t)/L = h/\lambda$. Hence every s - t path in the graph with length function $\ell_G + (h/\lambda) \cdot C$ has length at least h/λ , i.e. $\text{dist}_{G-C}(s, t) \geq h/\lambda$. $\square$

Now we can combine this lemma with the projection algorithm.

Corollary B.72. *There is an algorithm that takes as input a directed graph G with positive polynomially bounded integer edge capacities and lengths, an s - t feasible demand D on G , and a graph G' associated with a h -length (κ', λ') -projection algorithm, and in time proportional to one call to $\mathcal{O}_{\kappa, \lambda}^{\text{PDMCF}}$ on a graph of size $O(|G'|)$ and one call to the projection algorithm, outputs*

- a flow f routing D with congestion at most $\kappa\kappa'$ and average cost $a = \text{cost}(f)/\text{val}(f)$, and
- a $\min(h, a/(\lambda\lambda'))$ -length moving cut C with size at most $\text{val}(f)$ such that

$$\text{dist}_{G-C}(s, t) \geq \min\left\{h, \frac{a}{\lambda\lambda'}\right\}.$$

Proof. Let $D' := \hat{\pi}^{-1}(D)$. Since D is an s - t demand, D' is a s^{first} - t^{last} demand on G' with the same value.

Apply [Lemma B.71](#) to (G', D') using the oracle $\mathcal{O}_{\kappa, \lambda}^{\text{PDMCF}}$. We obtain a flow $\hat{f}$ routing D' with congestion at most κ and average cost $a = \text{cost}(\hat{f})/\text{val}(\hat{f})$, together with an (a/λ) -length moving cut C' on G' such that $|C'| \leq \text{val}(\hat{f})$ and

$$\text{dist}_{G'-C'}(s^{\text{first}}, t^{\text{last}}) \geq \frac{a}{\lambda}.$$

We first project the flow. Since $\text{Dem}(\hat{f}) = D' = \hat{\pi}^{-1}(D)$, the primal part of the projection algorithm returns a flow f on G routing D , with the same cost as $\hat{f}$ and congestion at most $\kappa'\kappa$. As $\text{val}(f) = \text{val}(\hat{f})$, its average cost is also a .

For the cut, let $L' := \min\{a/\lambda, h\lambda'\}$. Since all edge lengths in G' are positive and $L' \leq a/\lambda$, the same cut values C' may also be viewed as an L' -length moving cut, and the same scaling argument as in [Lemma B.71](#) gives

$$\text{dist}_{G'-C'}(s^{\text{first}}, t^{\text{last}}) \geq L'.$$

Now apply the dual part of the projection algorithm to this L' -length moving cut. We obtain a moving cut C on G with length $L'/\lambda' = \min\{h, a/(\lambda\lambda')\}$ and size $|C| = |C'| \leq \text{val}(\hat{f}) = \text{val}(f)$.

It remains to prove the distance lower bound. Suppose for contradiction that

$$\text{dist}_{G-C}(s, t) < \frac{L'}{\lambda'}.$$

Since $L'/\lambda' \leq h$, the guarantee of the h -length projection algorithm applies to the pair (s, t) , giving

$$\text{dist}_{G'-C'}(s^{\text{first}}, t^{\text{last}}) \leq \lambda' \cdot \text{dist}_{G-C}(s, t) < L',$$

contradicting $\text{dist}_{G'-C'}(s^{\text{first}}, t^{\text{last}}) \geq L'$. Therefore

$$\text{dist}_{G-C}(s, t) \geq \frac{L'}{\lambda'} = \min\left\{h, \frac{a}{\lambda\lambda'}\right\}.$$

This proves the corollary. $\square$

We also need the following helpful lemma.

Lemma B.73. *Given a directed graph G with positive integral edge capacities and lengths, an h -length single-commodity demand D , and parameters $\phi \in (0, 0.1)$ and $s > 1$, there is an algorithm making one call to $\mathcal{O}_{hs, \kappa', \lambda'}^{\text{Proj}}$ and one call to $\mathcal{O}_{\kappa, \lambda}^{\text{PDMCF}}$, each on a graph of size $O(|G|)$, that either*

- outputs an implicit flow-path decomposition of an hs -length flow routing some demand $D' \preceq D$ with congestion at most $\kappa\kappa'$ such that

$$|D'| \geq \frac{\phi}{6} \cdot |D|,$$

or

- outputs a $\frac{hs}{4\lambda\lambda'}$ -length moving cut C that is $(h, \frac{s}{4\lambda\lambda'})$ -length ϕ -sparse with respect to A_D and has size at least

$$\frac{\phi}{3} \cdot |D|.$$

Proof. Let $\Delta := |D|$. Construct an auxiliary graph H from G by adding a super-source s and a super-sink t , where:

- for every vertex v with $D(v) > 0$, we add an edge (s, v) of capacity $D(v)$ and length 0;
- for every vertex v with $D(v) < 0$, we add an edge (v, t) of capacity $-D(v)$ and length 0.

Clearly, H has size $O(|G|)$.

Now consider the s - t demand of value $\phi\Delta/3$ in H , and apply [Corollary B.72](#). This returns

- a flow f routing value $\phi\Delta/3$ with congestion at most $\kappa\kappa'$ and average cost

$$a := \frac{\text{cost}(f)}{\text{val}(f)},$$

and

- a moving cut $\widehat{C}$ of length

$$h' := \min\left\{hs, \frac{a}{\lambda\lambda'}\right\}$$

and size at most

$$|\widehat{C}| \leq \text{val}(f) = \frac{\phi\Delta}{3},$$

such that

$$\text{dist}_{H-\widehat{C}}(s, t) \geq h'.$$

We distinguish two cases.

Flow case: $a \leq hs/2$. Apply [Theorem B.9](#) to f , obtaining a decomposition

$$\{(L_i, \lambda_i, s_i, t_i)\}_{i \in [k]}.$$

Keep only those triples with $L_i \leq hs$, and output the resulting collection

$$\{(\lambda_i, s_i, t_i)\}_{i \in [k']}.$$

This is an implicit flow-path decomposition of an hs -length flow.

It remains to lower bound its value. If the retained flow had value less than $\text{val}(f)/2$, then more than $\text{val}(f)/2$ units of flow would be routed on paths of length strictly larger than hs . This would imply

$$\text{cost}(f) > \frac{\text{val}(f)}{2} \cdot hs,$$

contradicting the assumption $a = \text{cost}(f)/\text{val}(f) \leq hs/2$. Therefore, the retained flow has value at least

$$\frac{1}{2} \text{val}(f) = \frac{\phi\Delta}{6}.$$

Hence it routes some demand $D' \preceq D$ of value at least $\phi|D|/6$, as desired.

Cut case: $a > hs/2$. In this case,

$$h' = \min\left\{hs, \frac{a}{\lambda\lambda'}\right\} \geq \frac{hs}{2\lambda\lambda'}.$$

Let C^* be the restriction of $\widehat{C}$ to the edges of G , multiplied by 2. We interpret C^* as a $\frac{hs}{4\lambda\lambda'}$ -length

moving cut on G . Thus, for every $e \in E(G)$,

$$\ell_{C^*}(e) = \frac{hs}{4\lambda\lambda'} \cdot C^*(e) = \frac{hs}{2\lambda\lambda'} \cdot \widehat{C}(e).$$

Call a vertex $u \in V(G)$ *heavy* if the auxiliary edge incident to u in H receives cut value more than $1/4$ under $\widehat{C}$, and *light* otherwise. Since

$$|\widehat{C}| \leq \frac{\phi\Delta}{3},$$

the total demand on heavy vertices is at most

$$4|\widehat{C}| \leq \frac{4\phi\Delta}{3} < \frac{\Delta}{6},$$

where the last inequality uses $\phi < 0.1$. Hence the total demand on light vertices is at least

$$\frac{5\Delta}{6}.$$

Now let $u, v \in V(G)$ be light vertices with $D(u) > 0$ and $D(v) < 0$. Since the auxiliary edges (s, u) and (v, t) each contribute less than $h'/4$ in $H - \widehat{C}$, we must have

$$\text{dist}_{H-\widehat{C}}(u, v) \geq \text{dist}_{H-\widehat{C}}(s, t) - \frac{h'}{2} \geq \frac{h'}{2}.$$

We claim that

$$\text{dist}_{G-C^*}(u, v) \geq \frac{hs}{4\lambda\lambda'}.$$

Indeed, for any u -to- v path P in G ,

$$\ell_{G-C^*}(P) = \ell_G(P) + \frac{hs}{2\lambda\lambda'} \widehat{C}(P).$$

Since $\frac{hs}{2\lambda\lambda'} \leq h'$ and $\ell_G(P) \geq 0$, we have

$$\ell_G(P) + \frac{hs}{2\lambda\lambda'} \widehat{C}(P) \geq \frac{hs}{2\lambda\lambda'h'} (\ell_G(P) + h' \widehat{C}(P)).$$

Taking the minimum over all such paths P , we obtain

$$\text{dist}_{G-C^*}(u, v) \geq \frac{hs}{2\lambda\lambda'h'} \cdot \text{dist}_{H-\widehat{C}}(u, v) \geq \frac{hs}{2\lambda\lambda'h'} \cdot \frac{h'}{2} = \frac{hs}{4\lambda\lambda'}.$$

Therefore every light positive vertex and every light negative vertex are $\frac{hs}{4\lambda\lambda'}$ -separated by C^* . Since the total demand on light vertices is at least $5\Delta/6$, it follows that C^* is $(h, \frac{s}{4\lambda\lambda'})$ -length ϕ -sparse with respect to A_D after padding if necessary.

Finally, note that

$$|C^*| = 2|\widehat{C}| \leq \frac{2\phi\Delta}{3}.$$

If $|C^*| < \phi\Delta/3$, we pad C^* until its size becomes exactly $\phi\Delta/3$. Since padding only increases edge lengths, the separation property is preserved. Moreover, after padding the cut still has size at most $2\phi\Delta/3$, while it separates at least $5\Delta/6$ units of demand, so its sparsity remains at most

$$\frac{(2\phi\Delta/3)}{(5\Delta/6)} = \frac{4}{5}\phi < \phi.$$

Thus the resulting cut is still $(h, \frac{s}{4\lambda\lambda'})$ -length ϕ -sparse with respect to A_D , and has size at least $\phi|D|/3$. $\square$

Now we are ready to prove [Lemma B.43](#).

Lemma B.43. *Given a directed graph G with positive integral edge capacities and lengths, an h -length single-commodity demand D , and parameters $\phi, \varepsilon \in (0, 0.1)$ and $s > 1$, there is an algorithm making $O(\frac{\log(1/\varepsilon)}{\phi})$ -efficient oracle calls to $\mathcal{O}_{hs, \kappa', \lambda'}^{\text{Proj}}$ and to $\mathcal{O}_{\kappa, \lambda}^{\text{PDMCF}}$, that either*

- *outputs an implicit flow-path decomposition of an hs -length flow routing some demand $D' \preceq D$ with congestion*

$$O\left(\frac{\kappa\kappa' \log(1/\varepsilon)}{\phi}\right),$$

such that

$$|D'| \geq (1 - \varepsilon) \cdot |D|,$$

or

- *outputs a $\frac{hs}{4\lambda\lambda'}$ -length moving cut C that is $(h, \frac{s}{4\lambda\lambda'})$ -length ϕ -sparse with respect to A_D and has size at least*

$$\frac{\phi\varepsilon}{3} \cdot |D|.$$

Proof. Let $D^{(0)} := D$, and let $\Delta^{(0)} := |D|$. We repeatedly apply [Lemma B.73](#) to the current residual demand.

At iteration r , suppose the current residual demand is $D^{(r)}$. If

$$|D^{(r)}| < \varepsilon|D|,$$

then we stop and output the sum of all previously returned flows. Otherwise, apply [Lemma B.73](#) to $D^{(r)}$.

If the lemma returns a cut, then this cut has size at least

$$\frac{\phi}{3} |D^{(r)}| \geq \frac{\phi\varepsilon}{3} |D|,$$

and we output it.

Otherwise, the lemma returns an implicit flow-path decomposition of an hs -length flow routing some demand $E^{(r)} \preceq D^{(r)}$ with

$$|E^{(r)}| \geq \frac{\phi}{6} |D^{(r)}|.$$

We then update the residual demand by

$$D^{(r+1)} := D^{(r)} - E^{(r)}.$$

Hence

$$|D^{(r+1)}| \leq \left(1 - \frac{\phi}{6}\right) |D^{(r)}|.$$

By induction,

$$|D^{(r)}| \leq \left(1 - \frac{\phi}{6}\right)^r |D|.$$

Therefore, after

$$T = O\left(\frac{\log(1/\varepsilon)}{\phi}\right)$$

iterations, we must have

$$|D^{(T)}| < \varepsilon|D|.$$

If the algorithm terminates in this way, then the sum of all returned flows routes a demand $D' \preceq D$ with

$$|D'| \geq (1 - \varepsilon)|D|.$$

Each iteration contributes congestion at most $\kappa\kappa'$, so the total congestion is at most

$$T \cdot \kappa\kappa' = O\left(\frac{\kappa\kappa' \log(1/\varepsilon)}{\phi}\right).$$

Since each constituent flow is hs -length bounded, their sum is also hs -length bounded, and concatenating all implicit flow-path decompositions gives the required output.

This proves the lemma. □

B.10 Path Cover with Bounded Width

In this section, we will prove [Lemma B.12](#).

Lemma B.12. *There exists a deterministic algorithm that, given a directed graph $G = (V, E)$ with non-negative edge weights, a diameter parameter d , a parameter $\varepsilon \in (\log \log n / \log n, 1)$ and a stretch parameter $q \in \mathbb{N}_+$, constructs a qd -path cover H of G such that*

1. *every SCC of H has strong diameter $d \cdot \exp(O(1/\varepsilon))$,*
 2. *the width is at most $q \cdot n^{O(\varepsilon)}$,*
- in $q \cdot m^{1+o(\varepsilon)}$ time.*

Throughout the section, we write $\rho = 2^{1/\varepsilon}$ for convenience.

B.10.1 The Algorithm

The algorithm is recursive. A recursive call carries a positive integer weight on every vertex.

Definition B.74 (Loaded recursive instance). A *loaded instance* is a pair (A, μ) where $A \subseteq V$ and

$$\mu : A \rightarrow \{1, \rho, \rho^2, \dots\}.$$

For every $S \subseteq A$, define

$$\mu(S) := \sum_{v \in S} \mu(v).$$

For $S \subseteq A$, define the reweighting operator

$$(\rho \upharpoonright_S \mu)(v) := \begin{cases} \rho \cdot \mu(v), & v \in S, \\ \mu(v), & v \notin S. \end{cases}$$

The recursive procedure is denoted by

$$\text{PATHCOVER}(A, \mu),$$

and the initial call is

$$\text{PATHCOVER}(V, \mathbf{1}),$$

where $\mathbf{1}(v) = 1$ for every $v \in V$.

Base case. If $|A| = 1$, return $G[A]$ with the trivial projection map and representative.

Recursive step. Fix a loaded instance (A, μ) with $|A| > 1$, and choose an arbitrary root vertex $u \in A$.

We run a forward Dijkstra search in $G[A]$ from u and a backward Dijkstra search in $G[A]$ toward u . For $i \geq 0$, define

$$B_i^{\text{out}} := \text{Ball}_{G[A]}^{\text{out}}(u, i \cdot d), \quad B_i^{\text{in}} := \text{Ball}_{G[A]}^{\text{in}}(u, i \cdot d).$$

For either direction, let i be the smallest positive integer such that

$$\mu(B_i) \leq (1 + \tau)\mu(B_{i-1}),$$

where

$$\tau := \frac{1}{1000\rho^3}.$$

Let i^{out} and i^{in} be the forward and backward stopping indices. Define

$$\begin{aligned} B^{\text{out}} &= \text{Ball}_{G[A]}^{\text{out}}(u, i^{\text{out}} \cdot d), & \bar{B}^{\text{out}} &= A - \text{Ball}_{G[A]}^{\text{out}}(u, (i^{\text{out}} - 1) \cdot d), \\ B^{\text{in}} &= \text{Ball}_{G[A]}^{\text{in}}(u, i^{\text{in}} \cdot d), & \bar{B}^{\text{in}} &= A - \text{Ball}_{G[A]}^{\text{in}}(u, (i^{\text{in}} - 1) \cdot d). \end{aligned}$$

Also define the forward and backward rings

$$\begin{aligned} R^{\text{out}} &:= B^{\text{out}} \cap \bar{B}^{\text{out}} = \text{Ball}_{G[A]}^{\text{out}}(u, i^{\text{out}} \cdot d) \setminus \text{Ball}_{G[A]}^{\text{out}}(u, (i^{\text{out}} - 1) \cdot d), \\ R^{\text{in}} &:= B^{\text{in}} \cap \bar{B}^{\text{in}} = \text{Ball}_{G[A]}^{\text{in}}(u, i^{\text{in}} \cdot d) \setminus \text{Ball}_{G[A]}^{\text{in}}(u, (i^{\text{in}} - 1) \cdot d). \end{aligned}$$

Simultaneous forward and backward ball growing: For running time analysis, we do the ball growing forward and backward simultaneously using Dijkstra's algorithm. Specifically, we interleave Dijkstra's algorithm for the forward and backward ball growing in the following way: let A contain all the nodes v that have been extracted from the heap (i.e., have been relaxed and contribute $\deg(v)$ to the running time) in the forward Dijkstra's algorithm, and B be the backward version, if $\mu(A) \geq \mu(B)$ then we continue the forward Dijkstra's algorithm (extract the next node and do relaxation, update A), otherwise we continue the backward Dijkstra's algorithm. If one of them finds a thin layer, both procedures stop.

Without loss of generality, assume the forward search stops first. Fix

$$\gamma := \frac{1}{100\rho}.$$

We distinguish two cases.

1. **Case 1:** $\mu(B^{\text{out}}) \leq (1 - \gamma)\mu(A)$.

Recurse on the two overlapping pieces, and multiply the weights on the forward ring in both recursive children:

$$H^{\text{out}} \leftarrow \text{PATHCOVER}(B^{\text{out}}, (\rho \uparrow_{R^{\text{out}}} \mu)|_{B^{\text{out}}}), \quad \bar{H}^{\text{out}} \leftarrow \text{PATHCOVER}(\bar{B}^{\text{out}}, (\rho \uparrow_{R^{\text{out}}} \mu)|_{\bar{B}^{\text{out}}}).$$

Let G' be the layered projection induced from $(\bar{H}^{\text{out}}, H^{\text{out}})$. Return G' .

2. **Case 2:** $\mu(B^{\text{out}}) > (1 - \gamma)\mu(A)$.

Continue the backward search until it reaches its own stopping index i^{in} .

Let T^{out} be the shortest-path tree in $G[B^{\text{out}}]$ rooted at u , and let T^{in} be the reversed shortest-path tree in $G[B^{\text{in}}]$ rooted at u .

Let

$$M := B^{\text{out}} \cap B^{\text{in}}.$$

Let $\widetilde{T^{\text{out}}}$ be the subtree of T^{out} containing all vertices that can reach M in T^{out} , and let $\widetilde{T^{\text{in}}}$ be the subtree of T^{in} containing all vertices that can be reached from M in T^{in} . Define

$$H^{\text{mid}} := G[V(\widetilde{T^{\text{out}}}) \cup V(\widetilde{T^{\text{in}}})].$$

As before, H^{mid} is a subgraph of G , so the projection map and representatives are defined trivially.

Now recurse on the two recursive pieces, multiplying *all* weights in each child by ρ :

$$\tilde{H}^{\text{out}} \leftarrow \text{PATHCOVER}(\bar{B}^{\text{out}} \cap B^{\text{in}}, (\rho\mu)|_{\bar{B}^{\text{out}} \cap B^{\text{in}}}), \quad \bar{H}^{\text{in}} \leftarrow \text{PATHCOVER}(\bar{B}^{\text{in}}, (\rho\mu)|_{\bar{B}^{\text{in}}}),$$

where

$$(\rho\mu)(v) := \rho\mu(v) \quad \text{for all } v \in A.$$

Let G' be the layered projection induced from $(\tilde{H}^{\text{out}}, H^{\text{mid}}, \bar{H}^{\text{in}})$. Return G' .

B.10.2 The Global Potential

We show that a global ℓ_2^2 potential over the currently active recursive calls never increases.

Definition B.75 (Active frontier and potential). At any moment during the recursion, let $\mathcal{F}$ be the set of loaded instances that have been created but not yet expanded (i.e., all the leaves of the current recursion

tree). Define

$$\Phi(\mathcal{F}) := \sum_{(A, \mu) \in \mathcal{F}} \mu(A)^2.$$

Initially $\mathcal{F} = \{(V, \mathbf{1})\}$, so

$$\Phi(\mathcal{F}) = n^2.$$

Finally, $\mathcal{F}$ becomes all the leaves of the final recursion tree. Our goal is to show that in the end, we still have $\Phi(\mathcal{F}) \leq n^2$.

We now show that every recursive expansion never increases Φ .

Lemma B.76 (Thin rings). *We have*

$$\mu(R^{\text{out}}) \leq \tau \mu(B^{\text{out}}), \quad \mu(R^{\text{in}}) \leq \tau \mu(B^{\text{in}}).$$

Proof. By definition of the stopping index,

$$\mu(B^{\text{out}}) \leq (1 + \tau) \mu(B^{\text{out}} \setminus R^{\text{out}}).$$

Hence

$$\mu(R^{\text{out}}) = \mu(B^{\text{out}}) - \mu(B^{\text{out}} \setminus R^{\text{out}}) \leq \tau \mu(B^{\text{out}} \setminus R^{\text{out}}) \leq \tau \mu(B^{\text{out}}).$$

The proof for R^{in} is identical. □

Lemma B.77 (Case 1: the ring is tiny compared to the smaller side). *Assume Case 1. Then*

$$\mu(R^{\text{out}}) \leq \frac{1}{10\rho^2} \cdot \min\{\mu(B^{\text{out}}), \mu(\bar{B}^{\text{out}})\}.$$

Proof. If $\mu(B^{\text{out}}) \leq \mu(\bar{B}^{\text{out}})$, then by [Lemma B.76](#),

$$\mu(R^{\text{out}}) \leq \tau \mu(B^{\text{out}}) \leq \frac{1}{10\rho^2} \mu(B^{\text{out}}),$$

since $\tau \leq 1/(10\rho^2)$.

Now assume $\mu(\bar{B}^{\text{out}}) < \mu(B^{\text{out}})$. Let

$$Y := A \setminus B^{\text{out}}.$$

Then $\bar{B}^{\text{out}} = Y \cup R^{\text{out}}$, and since we are in Case 1,

$$\mu(Y) = \mu(A) - \mu(B^{\text{out}}) \geq \gamma \mu(A).$$

Also, by [Lemma B.76](#),

$$\mu(R^{\text{out}}) \leq \tau \mu(B^{\text{out}}) \leq \tau \mu(A) \leq \frac{\tau}{\gamma} \mu(Y) \leq \frac{1}{10\rho^2} \mu(Y) \leq \frac{1}{10\rho^2} \mu(\bar{B}^{\text{out}}),$$

because $\tau/\gamma = 1/(10\rho^2)$. □

Lemma B.78 (Case 2: both complements are small). *Assume Case 2. Then*

$$\mu(\bar{B}^{\text{out}}) \leq 2\gamma \mu(A), \quad \mu(\bar{B}^{\text{in}}) \leq 2\gamma \mu(A).$$

Proof. Since we are in Case 2,

$$\mu(B^{\text{out}}) > (1 - \gamma) \mu(A).$$

Because the searches are interleaved one virtual access at a time and the forward search stops first, by the time the forward search reaches i^{out} , the backward search has already performed at least as many virtual accesses. Therefore the backward search has already settled weighted mass at least $\mu(B^{\text{out}})$. Continuing it

until its own stopping index can only increase that mass. Hence

$$\mu(B^{\text{in}}) \geq \mu(B^{\text{out}}) > (1 - \gamma)\mu(A).$$

Now

$$\bar{B}^{\text{out}} = (A \setminus B^{\text{out}}) \cup R^{\text{out}},$$

so by Case 2 and [Lemma B.76](#),

$$\mu(\bar{B}^{\text{out}}) \leq \gamma\mu(A) + \tau\mu(B^{\text{out}}) \leq (\gamma + \tau)\mu(A) \leq 2\gamma\mu(A).$$

The proof for $\bar{B}^{\text{in}}$ is identical. □

Lemma B.79 (The potential never increases). *Every recursive expansion never increases $\Phi(\mathcal{F})$. Consequently,*

$$\Phi(\mathcal{F}) \leq n^2 \quad \text{throughout the entire recursion.}$$

Proof. Fix an active instance (A, μ) and write

$$s := \mu(A).$$

Case 1. Write A as a disjoint union

$$A = X \dot{\cup} R^{\text{out}} \dot{\cup} Y,$$

where

$$X := \text{Ball}_{G[A]}^{\text{out}}(u, (i^{\text{out}} - 1) \cdot d), \quad Y := A \setminus B^{\text{out}}.$$

Let

$$x := \mu(X), \quad r := \mu(R^{\text{out}}), \quad y := \mu(Y).$$

The parent mass is

$$s = x + r + y.$$

The two child masses are

$$s_1 = x + \rho r, \quad s_2 = y + \rho r.$$

Assume without loss of generality that $x \leq y$, i.e., B^{out} is the smaller side. By [Lemma B.77](#),

$$r \leq \frac{x + r}{10\rho^2},$$

which implies

$$x \geq (10\rho^2 - 1)r.$$

Now

$$s_1^2 + s_2^2 - s^2 = -2xy + 2(\rho - 1)r(x + y) + (2\rho^2 - 1)r^2.$$

Since $y \geq x$, this is at most

$$-2x^2 + 4(\rho - 1)rx + (2\rho^2 - 1)r^2.$$

Also $x \geq (10\rho^2 - 1)r \geq 4\rho r$, so

$$4(\rho - 1)rx \leq x^2.$$

Therefore

$$s_1^2 + s_2^2 - s^2 \leq -x^2 + (2\rho^2 - 1)r^2 < 0.$$

Case 2. By [Lemma B.78](#),

$$\mu(\bar{B}^{\text{out}} \cap B^{\text{in}}) \leq \mu(\bar{B}^{\text{out}}) \leq 2\gamma s, \quad \mu(\bar{B}^{\text{in}}) \leq 2\gamma s.$$

After multiplying all weights in these recursive children by ρ , their masses satisfy

$$s_1 \leq \rho \cdot 2\gamma s, \quad s_2 \leq \rho \cdot 2\gamma s.$$

Since $\gamma = 1/(100\rho)$, we obtain

$$s_1, s_2 \leq \frac{s}{50}.$$

Hence

$$s_1^2 + s_2^2 \leq 2 \left(\frac{s}{50} \right)^2 < s^2.$$

In both cases the total potential of the active frontier does not increase. Initially the potential is n^2 , so the claim follows. $\square$

Corollary B.80. *For every active recursive instance (A, μ) ,*

$$\mu(A) \leq n.$$

Proof. $\mu(A)^2$ is one term of $\Phi(\mathcal{F}) \leq n^2$. $\square$

B.10.3 Correctness: Path Covering

The path-covering property is proved the same as in the original proof in [HJS25], because the recursive domains are still

$$(\bar{B}^{\text{out}}, B^{\text{out}}) \quad \text{or} \quad (\bar{B}^{\text{out}} \cap B^{\text{in}}, H^{\text{mid}}, \bar{B}^{\text{in}}),$$

and the final graph is still obtained by the same layered-projection mechanism. In this section, we only mark several changes to the proof that is affected by our definition of thin layer, which is defined by the μ mass and several parameter changes.

Lemma B.81 (Number of layers). *Both i^{out} and i^{in} are at most $O(\rho^3 \log n)$.*

Proof. For every $i < i^{\text{out}}$, the forward stopping rule fails, so

$$\mu(\text{Ball}_{G[A]}^{\text{out}}(u, i \cdot d)) > (1 + \tau) \mu(\text{Ball}_{G[A]}^{\text{out}}(u, (i - 1) \cdot d)).$$

Multiplying over all $i < i^{\text{out}}$ gives

$$\mu(B^{\text{out}}) > (1 + \tau)^{i^{\text{out}} - 1} \mu(\{u\}).$$

Since $\mu(\{u\}) \geq 1$ and **Corollary B.80** gives $\mu(B^{\text{out}}) \leq n$, we get

$$(1 + \tau)^{i^{\text{out}} - 1} \leq n,$$

and hence

$$i^{\text{out}} = O\left(\frac{\log n}{\tau}\right) = O(\rho^3 \log n).$$

The proof for i^{in} is identical. $\square$

Lemma B.82 (Diameter bound). *Every SCC of the graph returned by $\text{PATHCOVER}(A, \mu)$ has diameter*

$$O(\rho^3 \log n) \cdot d.$$

Proof. The proof is identical to the original proof in [HJS25] of the diameter bound, replacing the old bound on $i^{\text{out}}, i^{\text{in}}$ by **Lemma B.81**. $\square$

Lemma B.83 (Path Covering). *For every loaded instance (A, μ) , the graph returned by $\text{PATHCOVER}(A, \mu)$ is a d -path cover of $G[A]$. Moreover, every path p in $G[A]$ of length at most d has a lift p' .*

Proof. This is identical to the original path-covering proof in [HJS25]. $\square$

B.10.4 Width, Size and Running time

We now use the potential to bound the width.

Lemma B.84 (Width bound). *Let H be the final graph returned by $\text{PATHCOVER}(V, 1)$. Then the width is at most $n^{O(\varepsilon)}$.*

Proof. Fix a vertex $v \in V$ and consider the rooted tree T_v of all copies of v created by the recursion.

In Case 1, a copy of v can create at most two recursive children. In Case 2, it can create at most two recursive children, plus possibly one additional terminal copy inside H^{mid} .

Let $L(t)$ be the maximum number of final copies obtainable from a current copy that can still be reweighted at most t more times. Then

$$L(0) = 1, \quad L(t) \leq 2L(t-1) + 1.$$

Hence $L(t) \leq 2^{t+1}$.

Every time a reweight happens on v , the weight of v 's copy is multiplied by ρ . Thus, after t such events, that active copy has weight at least ρ^t . On the other hand, by [Corollary B.80](#), every active recursive instance has total mass at most n , so no active copy can ever have weight larger than n . Hence $\rho^t \leq n$ and $t = O(\log_\rho n)$.

Given that $\rho = \exp(1/\varepsilon)$, the number of final copies of v is at most $n^{O(\varepsilon)}$. $\square$

Lemma B.85 (Size bound). *Let H be the final graph returned by $\text{PATHCOVER}(V, 1)$. Then $|H| = m^{1+O(\varepsilon)}$.*

Proof. Since the path cover is induced, which means $(u', v') \in E(H)$ implies $(\pi(u), \pi(v)) \in E(G)$, according to [Lemma B.84](#), each node can have at most $n^{O(\varepsilon)}$ copies, so the number of copies for each edge is at most $n^{O(\varepsilon)}$ and the lemma follows. $\square$

Lemma B.86. *The algorithm takes time $m^{1+O(\varepsilon)}$.*

Proof. The running time of Dijkstra and the modification of the graph can be charged into $\sum_{v' \in V(H)} \deg_G(\pi(v'))$, according to [Lemma B.84](#), the running time is at most $m^{1+O(\varepsilon)}$. $\square$

B.10.5 Amplifying the covered length by concatenation

We now show how to cover paths of length qd for an arbitrary integer $q \geq 1$ while keeping the SCC diameter at $O(d \cdot \rho^3 \log n)$.

Definition B.87 (Consecutive q -chain of a path cover). Let H be a projection onto G . For an integer $q \geq 1$, define $\text{Chain}_q(H \rightarrow G)$ as follows.

Take q disjoint copies

$$H_1, H_2, \dots, H_q$$

of H , preserving the same projection map on every layer. For every $j \in \{1, \dots, q-1\}$ and every two vertices

$$x \in V(H_j), \quad y \in V(H_{j+1}),$$

if

$$(\pi(x), \pi(y)) \in E(G)$$

then add the edge (x, y) with weight $w(\pi(x), \pi(y))$.

Lemma B.88 (Amplifying the covered length). *Suppose H is a d -path cover of G with representatives, and every SCC of H has diameter at most λd . Then for every integer $q \geq 1$, the graph*

$$G^{(q)} := \text{Chain}_q(H \rightarrow G)$$

satisfies:

1. $G^{(q)}$ covers every path in G of length at most qd ,
2. every SCC of $G^{(q)}$ has diameter at most λd ,
- 3.

$$\text{width}(G^{(q)}) \leq q \cdot \text{width}(H),$$

4.

$$|E(G^{(q)})| \leq q \cdot |E(H)| + (q-1) \sum_{x \in V(H)} \deg_G(\pi(x)) = O(q \cdot |E(H)|).$$

Proof. Fix any path p of length at most qd in G . Because all edge weights are non-negative, we can greedily partition p into at most q consecutive subpaths

$$p = p_1 \oplus p_2 \oplus \cdots \oplus p_t, \quad t \leq q,$$

such that each p_i has length at most d .

For every $i \in [t]$, since H is a d -path cover, p_i has a lift p'_i inside layer H_i . Now the last vertex of p'_i projects to the last vertex of p_i , and the first vertex of p'_{i+1} is the representative in layer H_{i+1} of the first vertex of p_{i+1} . Since the last edge of p_i and the first edge of p_{i+1} are consecutive on the original path p , the definition of $\text{Chain}_q(H \rightarrow G)$ adds the necessary connector edge from $\text{End}(p'_i)$ to $\text{Start}(p'_{i+1})$. Thus

$$p'_1 \oplus p'_2 \oplus \cdots \oplus p'_t$$

is a lift of p in $G^{(q)}$.

All inter-layer edges go strictly forward from layer j to layer $j+1$, so no SCC can use two different layers. Hence every SCC of $G^{(q)}$ is contained in a single copy of H and still has diameter at most λd .

The width bound is immediate because every original vertex has at most $\text{width}(H)$ copies per layer and there are q layers.

For the size bound, each layer contributes $|E(H)|$ internal edges. Between two consecutive layers, every vertex copy x adds at most $\deg_G(\pi(x))$ outgoing connector edges, because the head is always the unique representative in the next layer. Summing over the $q-1$ consecutive pairs of layers gives the stated bound. $\square$

Paper C

Deterministic Negative-Weight Shortest Paths in Nearly Linear Time via Path Covers

Bernhard Haeupler, Yonggang Jiang, Thatchaphol Saranurak

Published at ACM Symposium on Theory of Computing (STOC) 2026.

Full version available at <https://arxiv.org/abs/2511.08551>.

Abstract

We present the first *deterministic* nearly-linear time algorithm for single-source shortest paths with negative edge weights on directed graphs: given a directed graph G with n vertices, m edges whose weights are integers in $\{-W, \dots, W\}$, our algorithm either computes all distances from a source s or reports a negative cycle in $\tilde{O}(m) \cdot \log(nW)$ time.

All known near-linear time algorithms for this problem have been inherently randomized [BNW22; BCF23; Fis+25; LM24], as they crucially rely on *low-diameter decompositions*.

To overcome this barrier, we introduce a new structural primitive for directed graphs called the *path cover*. This plays a role analogous to *neighborhood covers* in undirected graphs [Awe+98], which have long been central to derandomizing algorithms that use low-diameter decomposition in the undirected setting. We believe that path covers will serve as a fundamental tool for the design of future deterministic algorithms on directed graphs.

C.1 Introduction

In the negative weight single-source shortest paths (SSSP) problem, given an m -edge n -vertex directed weighted graph $G = (V, E, w)$ with *possibly negative* edge weight $w(e) \in \{-W, \dots, W-1, W\}$ for each $e \in E$ and a source vertex $s \in V$, the goal is to either compute the distance from s to every other vertex or find a negative-weight cycle.

When weights are non-negative, the textbook Dijkstra algorithm takes near-linear $O(m + n \log n)$ time. Another textbook Bellman-Ford algorithm can handle negative weights, but requires much slower $O(mn)$ time. This contrast motivates the search for near-linear time algorithms since 1980s [Gab85; GT89; Gol95; Coh+17; Bra+20a; AMV20; Bra+21].

In 2022, two algorithmic breakthroughs happened. Chen et al. [Che+25] gave a randomized algorithm with $m^{1+o(1)} \log W$ time that solves an even more general minimum cost flow problem. Their approach is based on convex optimization and dynamic data structures and was later improved to be deterministic [Bra+23; Che+24; Van+24]. However, the dynamic data structures required in this approach are complicated and the $m^{o(1)}$ factor in time seems hard to avoid.

Independently, in the same year, Bernstein, Nanongkai, and Wulff-Nilsen [BNW22] showed an algorithm with $O(m \log^8(n) \log(W))$ time, which is faster, combinatorial, and bypasses complicated dynamic data structures. The follow-up work [BCF23; Ash+24a; LM24; Fis+25] subsequently sped up the time to $O((m+n \log \log n) \log^2(n) \log(nW))$. Unfortunately, algorithms in this line of work are inherently randomized.

Until now, there is no deterministic negative weight SSSP algorithm with near-linear $O(m \cdot \text{polylog}(nW))$ time.

Our result. We give the first deterministic near-linear algorithm, improving upon both lines of work above.

Theorem C.1 (Deterministic SSSP). *There is a deterministic algorithm solving the negative weight single-source shortest path problem in $O(m \log^8 n \log(nW))$ time.*

To prove [Theorem C.1](#), we introduce a novel graph structure called *path cover*, which we believe will find further applications beyond [Theorem C.1](#) itself. We do not optimize the polylogarithmic factor in order to present the clearest version of this new concept. A more relaxed version of path covers can speed up our algorithm to $O(m \log^5 n \log(nW))$ time. Below, we first define the path covering problem to motivate path covers.

The Path Covering Problem. We say a path p is a d -path if its length is at most d , and a subgraph $H \subseteq G$ covers p if $p \subseteq E(H)$. A graph H is d -clustered if every strongly-connected component of H has diameter at most d . The negative-weight SSSP problem can be reduced to the following problems.

Problem C.2 (Covering d -Paths with Clustered Subgraphs). Fix a graph G with *non-negative* weights and $d \geq 0$. An adversary chooses an *unknown target* set P of paths of length at most d in G where $|P| = O(n)$. We must choose $\tilde{O}(d)$ -clustered subgraphs $G'_1, \dots, G'_z \subseteq G$ where $z = O(\log n)$ such that each path $p \in P$ is covered by some G'_i .

The frameworks of [Ash+24a] and [Fis+25] showed that a deterministic near-linear time algorithm for [Problem C.2](#) implies another one for negative-weight SSSP with $m^{1+o(1)} \log(nW)$ time and $O(m \cdot \text{polylog}(nW))$ time, respectively.

Previous Solutions: Either Randomized or Undirected. [Problem C.2](#) admits a randomized algorithm with near-linear time via the probabilistic *low-diameter decomposition* (LDD), a powerful primitive [Bar96a; MPX13] extended to directed graphs by [BNW22]. For any d , an LDD algorithm samples a $\tilde{O}(d)$ -clustered subgraph $G' \subseteq G$ that covers each d -path with constant probability. Thus, $z = O(\log n)$ samples $G'_1, \dots, G'_z$ will cover $|P| = \text{poly}(n)$ unknown target paths with high probability, solving [Problem C.2](#).

All prior negative-weight SSSP algorithms with near-linear time [BNW22; BCF23; Ash+24a; LM24; Fis+25] employ probabilistic LDDs, and, hence, are inherently randomized because of it.

For deterministic algorithms, [Problem C.2](#) is much more difficult. Observe that we must cover *all* exponentially many d -paths, as any uncovered path can be picked by the adversary into the target set P . Thus, even the *existence* of the solution is highly unclear.

Yet, in undirected graphs, a deterministic near-linear time algorithm follows directly from efficient constructions of *neighborhood covers* [Awe+98]. This very strong property of neighborhood covers has been used for developing many deterministic algorithms in undirected graphs [Awe+98; GP19; Chu21; CZ23; HLS24; CP25]. For completeness, we define LDD and neighborhood covers, and explain how they give the solution of [Problem C.2](#) in either the randomized or undirected settings in [Section C.6](#).

This raises a natural question: is there a *deterministic* algorithm on *directed* graphs for [Problem C.2](#)?

A Barrier for Subgraphs: No Deterministic Algorithm on Directed Graphs. Unfortunately, unlike undirected graphs, we show that [Problem C.2](#) *cannot* be solved deterministically on directed graphs. In fact, even the *total size* of the $\tilde{O}(d)$ -clustered subgraphs must be $\tilde{\Omega}(m\sqrt{m})$ in the worst case. Thus, it takes $\tilde{\Omega}(m\sqrt{m})$ time to even write down these subgraphs.

Theorem C.3 (Barrier for Subgraphs). *For every $m, \lambda \geq 1$, there exists an $O(m)$ -edge directed graph G and d such that every collection of $d\lambda$ -clustered graphs $G'_1, \dots, G'_z \subseteq G$ covering all d -paths in G must have total size $\sum_i |E(G'_i)| = \Omega(m\sqrt{m/\lambda})$.*

See [Section C.7](#) for the proof. This barrier seems bad because the only two main tools mentioned above, LDDs and neighborhood covers, naturally only give a collection of $\tilde{O}(d)$ -clustered *subgraphs* of G and, hence, must suffer from the barrier of [Theorem C.3](#).

Bypassing the Barrier via Projection. We bypass the barrier in [Theorem C.3](#) using a *projection*, instead of subgraphs. A *projection* to G is a graph G' whose vertices of G' are copies of vertices in G , and edges of G' may connect the copy u' to v' only if the original vertex u is connected to v in G . More formally, there is a mapping $\pi : V(G') \rightarrow V(G)$ where, if $(u', v') \in E(G')$, then $(u = \pi(u'), v = \pi(v')) \in E(G)$.¹ Intuitively, a projection is formed by “merging” subgraphs together.

Observe that every path $p' = (v'_1, \dots, v'_t)$ in G' is mapped to a path $p = (v_1, \dots, v_t)$ in G where $v_i = \pi(v'_i)$. Here, we say that p is *covered* by G' and p' is a *lift* of p . If a projection G' covers all d -paths in G , then G' is a *d -path cover* of G .

As a toy example to illustrate the power of projection, consider a directed unweighted cycle G on $[n]$. Observe that a path $G' = (1, \dots, n, 1, \dots, n)$ is an n -path cover of G , which is a 0-clustered graph, i.e., a DAG. In contrast, any collection of 0-clustered subgraphs $G'_1, \dots, G'_z \subseteq G$ cannot cover any n -path of G . Otherwise, it would introduce a cycle in G'_i . See [Figure C.1](#) for another example where subgraphs require larger total size than a projection.

As our main technical contribution, we show how projection can bypass the barrier from [Theorem C.3](#) for subgraphs: while some graph requires $\tilde{O}(d)$ -clustered subgraphs of total size $\tilde{\Omega}(m\sqrt{m})$ to cover all d -paths, we show that every graph G admits a $\tilde{O}(d)$ -clustered projection G' to G of linear size that covers all d -paths.

Theorem C.4 (Covering d -Paths with Clustered Projection). *Given a graph G with non-negative weights and $d \geq 0$, there exists a projection G' to G where G' is a $O(d \log^6 n)$ -clustered graph of size $|E(G')| \leq (1 + \frac{1}{\log n})|E(G)|$ that covers all d -paths of G . Moreover, G' can be deterministically computed in $O(m \log^4 n)$ time.*

In other words, G' is a d -path cover which is a $\tilde{O}(d)$ -clustered graph of size $(1 + \frac{1}{\log n})|E(G)|$. Importantly, this size bound is even stronger than that of the known randomized LDD-based approach, which requires the total size of $O(|E(G)| \log n)$.

Finally, we obtain our SSSP algorithm [Theorem C.1](#) by showing that the reduction of [Ash+24a] from negative-weight SSSP to [Problem C.2](#) extends from subgraphs to projections. By replacing the randomized

¹That is, π is a graph homomorphism.

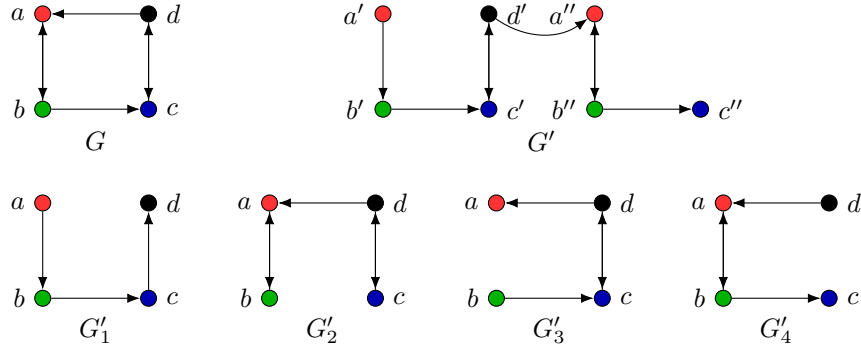


Figure C.1: At the top right, G' is a 1-clustered *projection* to G that covers all 3-paths in G . At the second row, G'_1, G'_2, G'_3, G'_4 are 1-clustered *subgraphs* of G' that cover all 3-paths in G .

LDD with the deterministic projection from [Theorem C.4](#), we simultaneously derandomize their algorithm and speed up the time from $m^{1+o(1)}$ to near-linear.

This speed-up crucially exploits our size bound of $(1 + \frac{1}{\log n})|E(G)|$. Thus, when the algorithm makes $O(\log n)$ -depth recursion, the total size blow-up is at most a constant factor as $(1 + \frac{1}{\log n})^{O(\log n)}|E(G)| = O(|E(G)|)$. In contrast, the LDD-based approach gives subgraphs of total size $O(|E(G)| \log n)$. The $O(\log n)$ blow-up per level causes their $m^{1+o(1)}$ running time in [\[Ash+24a\]](#).²

Stronger Property: Anchored Path Cover. Path covers can be viewed as a deterministic counterpart of LDD in directed graphs, similar to how neighborhood cover is a deterministic counterpart of LDD in undirected graphs. Since neighborhood covers have been a powerful tool in undirected graphs, we believe [Theorem C.1](#) is only the first of many applications of path covers.

Related work. In the setting of *undirected* graphs, the idea of simplifying a graph G via another graph G' , that makes copies of the vertices of the original graph G and connects them together, appeared in the *clan embedding* of [\[FL21\]](#) and the *copy tree embedding* of [\[HHZ22\]](#).

Independent work. In an independent work, Li [\[Li25\]](#) also obtains a near-linear time deterministic algorithm for negative-weight shortest paths. He defines a new variant of *padded decomposition* which can be of independent interest. In constructing our path cover, we perform a similar graph decomposition.

C.2 Overview

This section sketches the *existence* of the d -path cover G' of G in [Theorem C.4](#).

We start with some standard notation: For every vertex $v \in V$ let $\deg_G^{\text{out}}(v)$ be the out-degree of v , i.e., the number of edges leaving v . For a vertex set $S \subseteq V$ let $\deg_G^{\text{out}}(S) = \sum_{v \in S} \deg_G^{\text{out}}(v)$ be the number of edges starting in S . Lastly, for any vertex $u \in V$, define

$$\text{Ball}_G^{\text{out}}(u, r) = \{v \in V \mid \text{dist}_G(u, v) \leq r\}$$

to be the ball centered at u with radius r .

Construction. Our algorithm is recursive.

If the diameter of G is one strongly connected component with diameter $O(d \log^3 n)$, then one can trivially return the whole graph and terminate the recursion.

²The projection from [Theorem C.4](#) can derandomize [\[Fis+25\]](#), too. Here, a more relaxed size bound of $|E(G')| = O(|E(G)| \log n)$ is sufficient but their framework additionally requires that every SCC of G' contains no duplicated copies of vertices. Although our projection G' *does* satisfy this property, the overall argument is more complicated, and we do not want to impose such restrictions on the definition of path cover. So, we choose to present a derandomization of [\[Ash+24a\]](#) in this paper.

Otherwise, the algorithm finds a good partition by picking an arbitrary node u and growing a ball from it by computing $\text{Ball}_G^{\text{out}}(u, i \cdot d)$ for $i = 1, 2, 3, \dots$ until we find a “thin layer” i^* such that

$$\deg_G^{\text{out}}(\text{Ball}_G^{\text{out}}(u, i^* \cdot d)) \leq \left(1 + \frac{1}{\log^2 n}\right) \cdot \deg_G^{\text{out}}(\text{Ball}_G^{\text{out}}(u, (i^* - 1) \cdot d)). \quad (\text{C.1})$$

Since the ball cannot grow multiplicatively too often it is clear that $i^* = O(\log^3 n)$. The algorithm works separately and recursively on both

$$B^{\text{out}} := \text{Ball}_G^{\text{out}}(u, i^* \cdot d) \quad \text{and} \quad \bar{B}^{\text{out}} := V \setminus \text{Ball}_G^{\text{out}}(u, (i^* - 1) \cdot d).$$

Note that B^{out} and $\bar{B}^{\text{out}}$ only *overlap* at the “ring” around u from radius $(i^* - 1) \cdot d$ to $i^* \cdot d$.

We focus here on the interesting case that the ball grown from u is not more than a constant fraction of the graph, i.e., $\deg_G^{\text{out}}(B^{\text{out}}) \leq 0.9m$, our proof shows that otherwise there is a way to carve out a large ball and still make progress. Note that by running Dijkstra’s algorithm locally from u the time needed to grow the ball B^{out} is $\tilde{O}(\deg_G^{\text{out}}(B^{\text{out}}))$ and therefore essentially proportional to its size and independent of the size of $\bar{B}^{\text{out}}$. This makes the recursion efficient. Let H^{out} and $\bar{H}^{\text{out}}$ be d -path covers of the induced subgraphs $G[B^{\text{out}}]$ and $G[\bar{B}^{\text{out}}]$, respectively, which are constructed *recursively*.

For technical reasons, we strengthen the property of H^{out} (and likewise for $\bar{H}^{\text{out}}$): not only do we require each d -path in $G[B^{\text{out}}]$ to have a lift p' in H^{out} , we require a *one-to-one* mapping between the start vertices of p and p' . In other words, each vertex v in $G[B^{\text{out}}]$ has a *representative* vertex v' in H^{out} such that if a d -path in G starts at v , then its lift in G' starts at v' .

We build the d -path cover of G (denoted G') by concatenating $\bar{H}^{\text{out}}$ and H^{out} as follows: for every $(u, v) \in E$ with $u \in \bar{B}^{\text{out}}$ and $v \in V \setminus \bar{B}^{\text{out}}$, we add edges from all copies of the vertex u in $\bar{H}^{\text{out}}$ to the *representative* of the vertex v in H^{out} . Having a representative for each vertex avoids having to add prohibitively many edges from all copies of u to all copies of v . Defining the representatives of vertices in G' throughout the recursion is straightforward once the path-covering property in the next paragraph is clear.

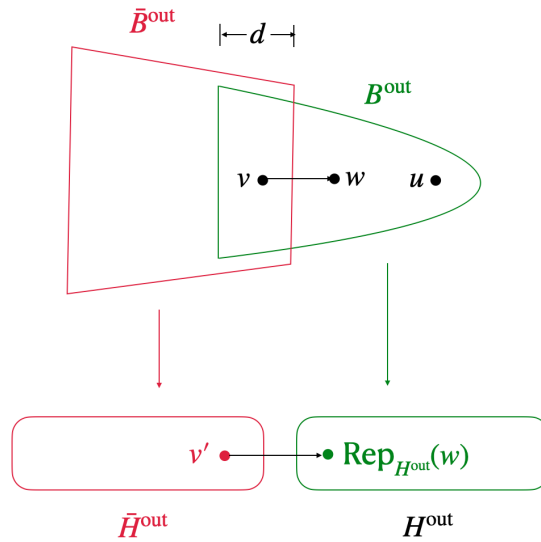


Figure C.2: An illustration of the construction. A path entering $V - \bar{B}^{\text{out}}$ (for example a path starting at w) will not leave B^{out} again because of the length d layer in the middle, thus covered by H^{out} .

$\tilde{O}(d)$ -clustered graph. Notice that if $\bar{H}^{\text{out}}, H^{\text{out}}$ are $\tilde{O}(d)$ -clustered graphs, then G' is a $\tilde{O}(d)$ -clustered graph because edges can only be from $\bar{H}^{\text{out}}$ to H^{out} .

Path Covering. We show that G' indeed covers all d -paths in G .

We consider the case where a path p starts at $v_p \in \bar{B}^{\text{out}}$ and leaves $\bar{B}^{\text{out}}$ at some point (i.e., enters $V \setminus \bar{B}^{\text{out}}$). Other situations (completely inside $\bar{B}^{\text{out}}$ or starting in $V \setminus \bar{B}^{\text{out}}$) can be handled similarly. Let v'_p be the first vertex on p that is not in $\bar{B}^{\text{out}}$. Let p_1 be the subpath of p before v'_p (excluding v'_p) and let p_2 be the remainder of p (starting at v'_p).

Notice that

$$v'_p \in V \setminus \bar{B}^{\text{out}} = V \setminus (V \setminus \text{Ball}_G^{\text{out}}(u, (i^* - 1) \cdot d)) = \text{Ball}_G^{\text{out}}(u, (i^* - 1) \cdot d).$$

This means that every path starting at v'_p with length at most d (in particular, p_2) is contained in

$$\text{Ball}_G^{\text{out}}(u, (i^* - 1) \cdot d + d) = \text{Ball}_G^{\text{out}}(u, i^* \cdot d) = B^{\text{out}}.$$

Therefore p_1 lies completely inside $G[\bar{B}^{\text{out}}]$ and p_2 lies completely inside $G[B^{\text{out}}]$, so $\bar{H}^{\text{out}}$ covers p_1 with a lift p'_1 and H^{out} covers p_2 with a lift p'_2 .

We then concatenate p'_1 and p'_2 using the edge from the last vertex of p'_1 to the first vertex of p'_2 (which is the representative of v'_p in H^{out}). The concatenated path maps to p .

Size of the path cover. We first show that the size of G' satisfies

$$|E(G')| \leq \sum_{v' \in V(G')} \deg_G^{\text{out}}(\pi(v')).$$

Let (u', v') be an edge concatenating $\bar{H}^{\text{out}}$ and H^{out} , where $u' \in V(\bar{H}^{\text{out}})$ and $v' \in V(H^{\text{out}})$ are copies of $u \in \bar{B}^{\text{out}}$ and $v \in V \setminus \bar{B}^{\text{out}}$, respectively, so that $(u, v) \in E$ and v' is the representative of v in H^{out} . In this case, we say that (u, v) *contributes to* u' . Notice that (i) (u, v) can contribute to u' at most once among all edges concatenating $\bar{H}^{\text{out}}$ and H^{out} , because we add only one edge from u' to the representative of v in H^{out} (which is why we define representatives); and (ii) (u, v) cannot contribute to u' again in the recursive construction of $\bar{H}^{\text{out}}$, because (u, v) is not an edge of $G[\bar{B}^{\text{out}}]$. Combining these two facts, every vertex u' is contributed to by at most $\deg_G^{\text{out}}(\pi(u'))$ edges, which establishes the claimed inequality.

It remains to bound $\sum_{v' \in V(G')} \deg_G^{\text{out}}(\pi(v'))$. Intuitively, Equation (C.1) guarantees that each recursion layer increases the total degree by at most a factor of $(1 + 1/\log^2 n)$ times the smaller side which is sufficient to yield an overall bound of $(1 + 1/\log n) \cdot m$ on the final total degree (we defer the calculation to the technical section). Since we have $|E(G')| \leq \sum_{v' \in V(G')} \deg_G^{\text{out}}(\pi(v'))$, we get the desired size bound

$$|E(G')| \leq (1 + 1/\log n) \cdot m$$

C.3 Preliminaries

We use the following notations, $[a] = \{1, 2, \dots, a\}$, $\tilde{O}(f) = f \cdot \text{polylog}(n)$, where we always use n, m to denote the number of nodes and edges of the input graph. For a function $\pi : V \rightarrow V'$, we write $\pi^{-1}(v') = \{v \in V \mid \pi(v) = v'\}$. All algorithms in this paper are deterministic.

Graph. All graphs of this paper, unless specified, are directed graphs with edge weights, denoted by $G = (V, E, w)$. We assume edge weights are *integers* within the range $[-W, W]$. We use standard definitions of graph terminologies for directed graphs which can be found in textbooks, including paths, distances, strongly connected components (SCCs), negative cycles, shortest-path trees, etc. For a cycle or path H , we use $w(H)$ to denote the sum of weights of edges on the cycle or path, and $|H|$ to denote the number of edges on the cycle or path (note that a path can have repeated edges; in this case, both the weight and the number of such an edge are counted multiple times). We use $w_{\geq 0}$ to denote the weight truncated at 0, i.e., $w_{\geq 0}(e) = \max\{0, w(e)\}$ for every $e \in E$. Similarly, we write $G_{\geq 0} = (V, E, w_{\geq 0})$.

In many cases of this paper, a path is not a *simple path*, so a path can have repeated vertices (but usually a shortest path is a simple path). We use $\text{Start}(p)$ and $\text{End}(p)$ to denote the starting vertex and ending vertex of a directed path p . We use $p_1 \oplus p_2$ to denote the concatenation of two paths: if $p_1 = (v_1, \dots, v_z)$

and $p_2 = (u_1, \dots, u_{z'})$, then $p_1 \oplus p_2 = (v_1, \dots, v_z, u_1, \dots, u_{z'})$. Notice that $p_1 \oplus p_2$ is a path only if (v_z, u_1) is an edge.

We use $V(G)$ and $E(G)$ to denote the vertex set and edge set of a graph G . We use w_G to stress that w_G is the weight function for G . We write $\deg_G^{\text{out}}(u)$ to denote the number of edges (u, v) for $v \in V$, and $\deg_G^{\text{in}}(v)$ to denote the number of edges (u, v) for $u \in V$. We write $\deg_G(v) = \deg_G^{\text{out}}(v) + \deg_G^{\text{in}}(v)$. We write $\deg_G^{\text{out}}(A) = \sum_{v \in A} \deg_G^{\text{out}}(v)$, $\deg_G^{\text{in}}(A) = \sum_{v \in A} \deg_G^{\text{in}}(v)$, and $\deg_G(A) = \sum_{v \in A} \deg_G(v)$.

Graph distance. We use $\text{dist}_G(s, t)$ to denote the distance from s to t . When G is clear from the context, we simply write $\text{dist}(s, t)$. We write

$$\text{Ball}_G^{\text{out}}(s, d) = \{v \in V(G) \mid \text{dist}_G(s, v) \leq d\} \quad \text{and} \quad \text{Ball}_G^{\text{in}}(s, d) = \{v \in V(G) \mid \text{dist}_G(v, s) \leq d\}.$$

We allow distance to be $+\infty$ (in the case of not connected) or $-\infty$ (in the case of reaching a negative cycle) so that every distance is well-defined.

Given a directed graph $G = (V, E, w)$ and a vertex set $A \subseteq V$, the *diameter* of A is defined as $\max_{s, t \in A} \text{dist}_{G[A]}(s, t)$ where $G[A]$ is the induced subgraph of G on A . The diameter of A is also the diameter of $G[A]$.

The single-source shortest path problem (SSSP). Given a directed graph $G = (V, E)$ with integer (possibly negative) edge weights and a source $s \in V$, either output $\text{dist}_G(s, v)$ for every $v \in V$, or output a negative cycle of G . It is folklore that we can get a shortest-path tree from a distance function, and vice versa.

Potential adjustment. Given a *potential* $\phi : V \rightarrow \mathbb{Z}$ and a graph $G = (V, E, w)$, we define an adjusted weight $w_\phi(u, v) = w(u, v) + \phi(u) - \phi(v)$. We define the adjusted graph $G_\phi = (V, E, w_\phi)$.

The following lemma shows the power of potential adjustment. The lemma is folklore and was already used in [Joh77].

Lemma C.5 ([Joh77]). *Let G be a graph and $\phi : V \rightarrow \mathbb{Z}$ be a potential function. If G has no negative cycle, a subgraph is a shortest-path tree of G iff it is a shortest-path tree of G_ϕ . More precisely, let $d_G^s(u)$ be the distance in G from s to u , then*

$$d_{G_\phi}^s(u) = d_G^s(u) + \phi(s) - \phi(u).$$

C.4 Path Cover via a Clustered Graph

In this section, we present a method for covering every bounded-length path in a directed graph using a clustered graph. We begin with the necessary definitions. All graphs in this section have non-negative edge weights.

C.4.1 Definitions and the Main Theorem

We first formally define graph projections introduced in the introduction.

Definition C.6 (Graph Projections). Let $G = (V, E, w)$ be a graph. A *projection onto G* is a graph $G' = (V', E', w')$ together with a projection map $\pi = \pi_{G'}$ where π is a *weight-preserving graph homomorphism*: for every $(x, y) \in E(G')$, we have $(\pi(x), \pi(y)) \in E(G)$ and $w'(x, y) = w(\pi(x), \pi(y))$.

When we refer to a projection G' , we implicitly assume its map $\pi_{G'}$.

Definition C.7 (Path Covering). Given a path $p = (v_1, \dots, v_z)$ in G , we say that a projection (G', π) onto G *covers* p if there exists a path $p' = (v'_1, \dots, v'_z)$ in G' such that $\pi(v'_i) = v_i$ for every $i \in [z]$. We call p' the *lift* of p . The projection is *d-path-covering* if every path in G of length at most d is covered.

Our goal is to cover every simple path of length at most d by a projection whose SCCs have small diameter. For this purpose, we define clustered graphs as follows.

Definition C.8 (Clustered Graphs). We say that a directed graph G' is a d -clustered graph if every strongly connected component (SCC) of G' has diameter at most d .

Definition C.9 (Path Cover). A d -path cover of a directed graph $G = (V, E, w)$ with diameter slack λ is a d -path-covering projection G' where G' is a $\lambda \cdot d$ -clustered graph.

We will prove the following theorem in this section.

Theorem C.10 (Path Cover via Clustered DAGs). *There exists a deterministic algorithm that, given a directed graph $G = (V, E, w)$ with non-negative edge weights, a diameter parameter d , and a slack parameter $\lambda \geq 10000 \log^4 n$, constructs a d -path cover G' of G with diameter slack λ and size at most*

$$\left(1 + \frac{100 \log^2 n}{\sqrt{\lambda}}\right) \cdot m,$$

in $O(\sqrt{\lambda} \cdot m \log n)$ time.

Moreover, we have

$$\sum_{v \in V(G')} \deg_G(\pi(v)) \leq \left(1 + \frac{100 \log^2 n}{\sqrt{\lambda}}\right) \cdot m.$$

To get [Theorem C.4](#), we only need to set $\lambda = 10000 \log^6 n$.

Remark C.11. The running time can be improved to $O(m \log^2 n)$, independent of λ , if we relax the structure of the path cover so that we only guarantee *weak diameter w.r.t. G* . More precisely, for each strongly connected component U in G' , $\max_{u, v \in U} \text{dist}_G(\pi(u), \pi(v)) \leq d\lambda$. This weaker guarantee is sufficient for our negative-weight shortest path algorithms, and we can improve the running time from $O(m \log^8 n \log(nW))$ to $O(m \log^5 n \log(nW))$ in this way. However, we choose to construct the path cover with a strong diameter guarantee to ease the understanding of this new concept.

C.4.2 The Algorithm

In this section, we describe the algorithm for [Theorem C.10](#). Let $G = (V, E, w)$ be the input directed graph and d be the input diameter parameter.

The following definition is useful when describing algorithms. Let G' be a projection onto G . We say a vertex $v \in V(G)$ is *present* (in the projection) if $\pi^{-1}(v) \neq \emptyset$. We say the projection has *representatives* if every present vertex $v \in V(G)$ has a unique representative $\text{rep}_{G'}(v) \in V(G')$ with $\pi_{G'}(\text{rep}_{G'}(v)) = v$. In this case, we say $\text{rep}_{G'}(v)$ is a *representative*.³

Definition C.12 (Layered Projection). Let G_i be projections onto G with representatives for $i = 1, \dots, z$. A *layered projection* $G' = \text{Layer}((G_1, \dots, G_z) \rightarrow G)$ induced from $(G_1, \dots, G_z)$ onto G with representatives is constructed as follows. First, initialize G' as a disjoint union of $G_1, \dots, G_z$ with the same projection map on vertices⁴. For every $v \in V(G)$, suppose v is present in G_i where i is the smallest such index, select the representative $\text{rep}_{G'}(v)$ to be $\text{rep}_{G_i}(v)$. Then, for every two vertices $u' \in V(G_i)$ and $v' \in V(G_j)$, if $i < j$, $(\pi_{G_i}(u'), \pi_{G_j}(v')) \in E$ and v' is a representative in G' , we add the edge (u', v') to G' with weight $w(\pi_{G_i}(u'), \pi_{G_j}(v'))$.

The algorithm is recursive. For convenience, we use $\text{PATHCOVER}(A)$ to denote the subroutine that takes a vertex set $A \subseteq V$ as input and returns a d -path cover of $G[A]$ with representatives. It suffices to call $\text{PATHCOVER}(V)$ to finish [Theorem C.10](#).

³The representative in our case is basically the first appearance of the copy of v in G' . First is defined on the SCC order of G' (we will have the guarantee that each SCC is a subgraph of G , which does not have repeated vertices).

⁴When G_i and G_j share the same vertex v , we make different copies of v in G' .

We define the following parameters:

$$\varepsilon = \frac{1}{\sqrt{\lambda}}, \quad \varepsilon' = \frac{9 \log n}{\lambda}.$$

Base case. When $|A| = 1$, we simply return $G[A]$ with the trivial projection map and representative.

Recursive step. Pick an arbitrary vertex u in A . We execute the following two procedures in an interleaved fashion: we perform one memory access of each procedure in turn, alternating between them, until one of them terminates. As soon as one procedure stops, the entire process halts.

1. **(Forward growing)** Run Dijkstra's algorithm to find the smallest $i^{\text{out}} \in \mathbb{N}_{>0}$ such that

$$\deg_G(\text{Ball}_{G[A]}^{\text{out}}(u, i^{\text{out}} \cdot d)) \leq (1 + \varepsilon') \cdot \deg_G(\text{Ball}_{G[A]}^{\text{out}}(u, (i^{\text{out}} - 1) \cdot d)).$$

Such an i^{out} must exist because for sufficiently large i , we have $\text{Ball}_{G[A]}^{\text{out}}(u, i \cdot d) = G[A]$.

To be precise, Dijkstra's algorithm fixes the distances from u to other nodes in increasing order. We run it only up to the point where the distance exceeds $i \cdot d$, in order to compute $\text{Ball}_{G[A]}^{\text{out}}(u, i \cdot d)$. This can take sublinear time because the algorithm stops once the desired index i^{out} is found.

For technical reasons, we assume that when Dijkstra's algorithm fixes the distance of a node v , it performs $\deg_G(v)$ memory accesses, i.e., it looks up all adjacent edges of v in G (but only performs $\deg_{G[A]}^{\text{out}}(v)$ relaxations). This setting is purely for the convenience of analysis.

2. **(Backward growing)** Identical to the forward growing procedure, but with all occurrences of 'out' replaced by 'in'.

For simplicity, we use the following notation. Notice that the algorithm only knows $B^{\text{out}}, \bar{B}^{\text{out}}$ but does not know $B^{\text{in}}, \bar{B}^{\text{in}}$ if forward growing stops earlier than backward growing, and vice versa:

$$\begin{aligned} B^{\text{out}} &= \text{Ball}_{G[A]}^{\text{out}}(u, i^{\text{out}} \cdot d), & \bar{B}^{\text{out}} &= A - \text{Ball}_{G[A]}^{\text{out}}(u, (i^{\text{out}} - 1) \cdot d), \\ B^{\text{in}} &= \text{Ball}_{G[A]}^{\text{in}}(u, i^{\text{in}} \cdot d), & \bar{B}^{\text{in}} &= A - \text{Ball}_{G[A]}^{\text{in}}(u, (i^{\text{in}} - 1) \cdot d). \end{aligned}$$

Without loss of generality, assume forward growing stops earlier than backward growing (the other case is handled symmetrically). Consider two cases.

1. **Case 1:** $\deg_G(B^{\text{out}}) < (1 - \varepsilon) \deg_G(A)$. In this case, let

$$H^{\text{out}} \leftarrow \text{PATHCOVER}(B^{\text{out}}), \quad \bar{H}^{\text{out}} \leftarrow \text{PATHCOVER}(\bar{B}^{\text{out}}).$$

Let G' be the layered projection induced from $(\bar{H}^{\text{out}}, H^{\text{out}})$ (Definition C.12). Return G' as the output of $\text{PATHCOVER}(A)$.

2. **Case 2:** $\deg_G(B^{\text{out}}) \geq (1 - \varepsilon) \deg_G(A)$. Continue the backward growing until it stops. Now the algorithm also knows $B^{\text{in}}, \bar{B}^{\text{in}}$. Let T^{out} be the shortest-path tree in $G[B^{\text{out}}]$ with root u , and let T^{in} be the reversed shortest-path tree in $G[B^{\text{in}}]$ with root u (i.e., T^{in} contains the shortest paths from every node in B^{in} to u).

Let

$$M \leftarrow B^{\text{out}} \cap B^{\text{in}}.$$

Let $\tilde{T}^{\text{out}}$ be the subtree of T^{out} that contains all nodes in $V(T^{\text{out}})$ that can reach M in T^{out} . Similarly, define $\tilde{T}^{\text{in}}$ to be the subtree of T^{in} that contains all nodes in $V(T^{\text{in}})$ that can be reached from M in T^{in} .

Define

$$H^{\text{mid}} \leftarrow G[V(\tilde{T}^{\text{out}}) \cup V(\tilde{T}^{\text{in}})].$$

Notice that H^{mid} is a subgraph of G , so a projection map and representatives can be defined trivially.

Make the following recursive calls:

$$\bar{H}^{\text{in}} \leftarrow \text{PATHCOVER}(\bar{B}^{\text{in}}), \quad \tilde{H}^{\text{out}} \leftarrow \text{PATHCOVER}(\bar{B}^{\text{out}} \cap B^{\text{in}}).$$

Let G' be the layered projection induced from $(\tilde{H}^{\text{out}}, H^{\text{mid}}, \bar{H}^{\text{in}})$. Return G' as the output of $\text{PATHCOVER}(A)$.

For a graph illustration, see [Figure C.2](#) which shows Case 1. Case 2 can be viewed as after applying Case 1 to get $(H^{\text{in}}, \bar{H}^{\text{in}})$, we recursively build H^{in} by $(\tilde{H}^{\text{out}}, H^{\text{mid}})$.

C.4.3 Correctness

We prove the correctness of the algorithm described in [Section C.4.2](#). Specifically, $\text{PATHCOVER}(A)$ returns a d -path cover of $G[A]$. We proceed by induction on $|A|$.

Base case. Suppose $|A| = 1$. The only SCC has diameter 0. Thus, $G[A]$ is a d -clustered graph. Moreover, the trivial projection map certifies that all paths are covered.

Induction step. Assume PATHCOVER is correct for every input size smaller than $|A|$. We first show the easy part: G' is a $\lambda \cdot d$ -clustered graph. We need the following lemma.

Lemma C.13. *Both i^{out} and i^{in} are at most $\lambda/4$.*

Proof. For every $i < i^{\text{out}}$, by the definition of i^{out} ,

$$\deg_G(\text{Ball}_{G[A]}^{\text{out}}(u, i \cdot d)) > (1 + \varepsilon') \cdot \deg_G(\text{Ball}_{G[A]}^{\text{out}}(u, (i-1) \cdot d)).$$

Combining these inequalities for every $i < i^{\text{out}}$ gives

$$m \geq \deg_G(\text{Ball}_{G[A]}^{\text{out}}(u, i^{\text{out}} \cdot d)) > (1 + \varepsilon')^{i^{\text{out}}-1} = \left(1 + \frac{9 \log n}{\lambda}\right)^{i^{\text{out}}-1}.$$

Since $\lambda \geq 10000 \log^4 n$, we obtain $i^{\text{out}} \leq \lambda/4$. The same proof holds symmetrically for i^{in} . $\square$

Bounding diameter. To show G' is a $\lambda \cdot d$ -clustered graph, let C be an SCC of G' ; we must show $\text{diam}(C) \leq \lambda \cdot d$. If the algorithm goes to Case 1, then G' is the layered projection induced from $(\bar{H}^{\text{out}}, H^{\text{out}})$. By [Definition C.12](#), there are only edges from $\bar{H}^{\text{out}}$ to H^{out} but not backwards, so C is either completely inside $\bar{H}^{\text{out}}$ or inside H^{out} ; in either case, C has diameter at most $\lambda \cdot d$ by the induction hypothesis. If the algorithm goes to Case 2, then G' is the layered projection induced from $(\tilde{H}^{\text{out}}, H^{\text{mid}}, \bar{H}^{\text{in}})$. Similarly, if C is completely inside $\tilde{H}^{\text{out}}$ or $\bar{H}^{\text{in}}$, then C has diameter at most $\lambda \cdot d$ by the induction hypothesis. If C is inside H^{mid} , we show that H^{mid} is strongly connected and has diameter at most $\lambda \cdot d$. Recall

$$H^{\text{mid}} \leftarrow G[V(\tilde{T}^{\text{out}}) \cup V(\tilde{T}^{\text{in}})],$$

where $\tilde{T}^{\text{out}}$ is the subtree of T^{out} whose every node can reach $W = B^{\text{out}} \cap B^{\text{in}}$, and $\tilde{T}^{\text{in}}$ is the subtree of T^{in} whose every node can be reached from W . Recall also

$$B^{\text{out}} = \text{Ball}_{G[A]}^{\text{out}}(u, i^{\text{out}} \cdot d), \quad B^{\text{in}} = \text{Ball}_{G[A]}^{\text{in}}(u, i^{\text{in}} \cdot d),$$

and $T^{\text{out}}, T^{\text{in}}$ are shortest-path trees in $G[B^{\text{out}}]$ and $G[B^{\text{in}}]$. By [Lemma C.13](#), each shortest-path tree has depth at most $(\lambda/4) \cdot d$. Thus, every node in $\tilde{T}^{\text{out}}$ can reach W with distance at most $(\lambda/4) \cdot d$ in $\tilde{T}^{\text{out}}$; every node in $\tilde{T}^{\text{in}}$ can be reached from W with distance at most $(\lambda/4) \cdot d$ in $\tilde{T}^{\text{in}}$. Similarly, u can reach every node in $\tilde{T}^{\text{out}}$ within distance at most $(\lambda/4) \cdot d$ in $\tilde{T}^{\text{out}}$, and u can be reached from every node in $\tilde{T}^{\text{in}}$ within distance at most $(\lambda/4) \cdot d$ in $\tilde{T}^{\text{in}}$. We conclude that the diameter of H^{mid} is at most $\lambda \cdot d$.

Covering all length- d paths. Next we show that G' is a d -path cover of $G[A]$. Let p be an arbitrary path of $G[A]$ with length at most d . We prove that G' covers p (see [Definition C.7](#)). Moreover, we prove a stronger statement: there exists a lift p' of p such that $\text{Start}(p') = \text{rep}(\text{Start}(p))$.

Since G' depends on whether the algorithm goes to Case 1 or Case 2, we prove the two cases separately.

Induction step (Case 1). Here G' is the layered projection induced from $(\bar{H}^{\text{out}}, H^{\text{out}})$ where

- $\bar{H}^{\text{out}}$ is a path cover of $G[\bar{B}^{\text{out}}]$, and
- H^{out} is a path cover of $G[B^{\text{out}}]$.

If p is completely contained in $G[\bar{B}^{\text{out}}]$, then by the induction hypothesis, p is covered by G' . Moreover, the lift p' of p has $\text{Start}(p') = \text{rep}_{\bar{H}^{\text{out}}}(\text{Start}(p))$. By [Definition C.12](#), $\text{rep}_{\bar{H}^{\text{out}}}(\text{Start}(p)) = \text{rep}_{G'}(\text{Start}(p))$ since $\bar{H}^{\text{out}}$ precedes H^{out} . Thus, $\text{Start}(p') = \text{rep}_{G'}(\text{Start}(p))$.

Now assume p is not completely contained in $G[\bar{B}^{\text{out}}]$, so there exists a vertex on p that lies in $A - \bar{B}^{\text{out}}$. Let a be the first such vertex on p .

Let p_1 be the subpath of p before vertex a (excluding a ; possibly empty), and let p_2 be the subpath of p after vertex a (including a). By definition, p_1 is contained in $G[\bar{B}^{\text{out}}]$, so by induction, p_1 is covered by $\bar{H}^{\text{out}}$.

We now show that p_2 is contained in $G[B^{\text{out}}]$. Recall

$$A - \bar{B}^{\text{out}} = \text{Ball}_{G[A]}^{\text{out}}(u, (i^{\text{out}} - 1) \cdot d), \quad B^{\text{out}} = \text{Ball}_{G[A]}^{\text{out}}(u, i^{\text{out}} \cdot d).$$

Thus, if $a \in A - \bar{B}^{\text{out}}$, any path of length at most d starting at a is contained in B^{out} .

Hence p_1 is covered by $\bar{H}^{\text{out}}$ and p_2 is covered by H^{out} , with lifts p'_1, p'_2 respectively. We have $\pi(\text{End}(p'_1)) = \text{End}(p_1)$ and $\text{Start}(p'_2) = \text{rep}_{H^{\text{out}}}(\text{Start}(p_2))$. Since $\text{Start}(p_2) = a \in A - \bar{B}^{\text{out}}$, a is not present in $\bar{H}^{\text{out}}$, so $\text{rep}_{G'}(a) = \text{rep}_{H^{\text{out}}}(a)$ and $\text{Start}(p'_2) = \text{rep}_{G'}(a)$. By [Definition C.12](#), there is an edge $(\text{End}(p'_1), \text{Start}(p'_2))$ in G' , so $p'_1 \oplus p'_2$ is a path in G' and is a lift of $p = p_1 \oplus p_2$.

If p'_1 is non-empty, then $\text{Start}(p'_1) = \text{rep}_{\bar{H}^{\text{out}}}(\text{Start}(p_1))$ by induction, hence $\text{Start}(p'_1 \oplus p'_2) = \text{rep}_{G'}(\text{Start}(p))$ since $\bar{H}^{\text{out}}$ precedes H^{out} in G' . Otherwise, p'_1 is empty, so p starts at $a \in V - \bar{B}^{\text{out}}$. In this case, since a is not present in $\bar{H}^{\text{out}}$, we must have $\text{rep}_{G'}(a) = \text{rep}_{H^{\text{out}}}(a) = \text{Start}(p'_2)$, where the second equality is by induction.

Induction step (Case 2). Here G' is the layered projection induced from $(\tilde{H}^{\text{out}}, H^{\text{mid}}, \bar{H}^{\text{in}})$, where

- $\tilde{H}^{\text{out}}$ is a path cover of $G[\bar{B}^{\text{out}} \cap B^{\text{in}}]$,
- $H^{\text{mid}} \supseteq G[B^{\text{out}} \cap B^{\text{in}}]$,
- $\bar{H}^{\text{in}}$ is a path cover of $G[\bar{B}^{\text{in}}]$.

If p is completely contained in $G[\bar{B}^{\text{out}} \cap B^{\text{in}}]$, then p is covered by G' . Moreover, since $\tilde{H}^{\text{out}}$ precedes $H^{\text{mid}}, \bar{H}^{\text{in}}$, we have $\text{rep}_{G'}(\text{Start}(p)) = \text{rep}_{\tilde{H}^{\text{out}}}(\text{Start}(p))$, which equals the starting vertex of the lift of p in G' .

Otherwise, there is a vertex of p in $A - (\bar{B}^{\text{out}} \cap B^{\text{in}})$; let a be the first such vertex. Let p_1 be the subpath of p before a (excluding a ; possibly empty), and let p_2 be the subpath of p starting at a . Then $p_1 \subseteq G[\bar{B}^{\text{out}} \cap B^{\text{in}}]$, so p_1 is covered by $\tilde{H}^{\text{out}}$ with lift p'_1 . If p_1 is non-empty, then $\text{rep}_{G'}(\text{Start}(p)) = \text{rep}_{\tilde{H}^{\text{out}}}(\text{Start}(p)) = \text{Start}(p'_1)$ by induction.

We first show that p_2 is completely contained in $V(H^{\text{mid}}) \cup \bar{B}^{\text{in}}$. Recall that

$$\begin{aligned} A - (\bar{B}^{\text{out}} \cap B^{\text{in}}) &\subseteq (A - \bar{B}^{\text{out}}) \cup (A - B^{\text{in}}) = \text{Ball}_{G[A]}^{\text{out}}(u, (i^{\text{out}} - 1) \cdot d) \cup (A - \text{Ball}_{G[A]}^{\text{in}}(u, i^{\text{in}} \cdot d)), \\ V(H^{\text{mid}}) \cup \bar{B}^{\text{in}} &\supseteq (B^{\text{out}} \cap B^{\text{in}}) \cup \bar{B}^{\text{in}} \supseteq \text{Ball}_{G[A]}^{\text{out}}(u, i^{\text{out}} \cdot d) \cup (A - \text{Ball}_{G[A]}^{\text{in}}(u, (i^{\text{in}} - 1) \cdot d)). \end{aligned}$$

Hence, if $a \in A - (\bar{B}^{\text{out}} \cap B^{\text{in}})$ and p_2 has length at most d , it is contained in $V(H^{\text{mid}}) \cup \bar{B}^{\text{in}}$. We now argue within $V(H^{\text{mid}}) \cup \bar{B}^{\text{in}}$.

If p_2 is completely contained in $V(H^{\text{mid}})$, then p_2 is covered by H^{mid} trivially (as H^{mid} is an induced subgraph of G) with a lift p'_2 . Moreover, $\text{rep}_{G'}(a) = \text{rep}_{H^{\text{mid}}}(a) = a = \text{Start}(p'_2)$ since a is not present in $\tilde{H}^{\text{out}}$.

Otherwise, p_2 has a vertex in $A - (\bar{B}^{\text{out}} \cap B^{\text{in}}) - V(H^{\text{mid}})$; let b be the first such vertex. Let $p_{2,1}$ be the subpath of p_2 before b (excluding b ; possibly empty) and let $p_{2,2}$ be the subpath of p_2 from b onward. Then $p_{2,1} \subseteq V(H^{\text{mid}})$ and hence is covered by H^{mid} with lift $p'_{2,1}$. If $p_{2,1}$ is non-empty, then as above $\text{rep}_{G'}(a) = \text{Start}(p'_{2,1})$.

We prove that $p_{2,2} \subseteq G[\bar{B}^{\text{in}}]$, so that it is covered by $\bar{H}^{\text{in}}$ with lift $p'_{2,2}$. Note

$$\begin{aligned} A - (\bar{B}^{\text{out}} \cap B^{\text{in}}) - V(H^{\text{mid}}) &\subseteq A - (\bar{B}^{\text{out}} \cap B^{\text{in}}) - (B^{\text{out}} \cap B^{\text{in}}) = A - B^{\text{in}} = A - \text{Ball}_{G[A]}^{\text{in}}(u, i^{\text{in}} \cdot d), \\ \bar{B}^{\text{in}} &= A - \text{Ball}_{G[A]}^{\text{in}}(u, (i^{\text{in}} - 1) \cdot d). \end{aligned}$$

Since $b = \text{Start}(p_{2,2}) \in A - \text{Ball}_{G[A]}^{\text{in}}(u, i^{\text{in}} \cdot d)$ and $p_{2,2}$ has length at most d , $p_{2,2}$ is contained in $\bar{B}^{\text{in}}$ by definition (otherwise it would enter $\text{Ball}_{G[A]}^{\text{in}}(u, (i^{\text{in}} - 1) \cdot d)$, contradicting $b \in A - \text{Ball}_{G[A]}^{\text{in}}(u, i^{\text{in}} \cdot d)$).

Moreover, b is not present in $\tilde{H}^{\text{out}}$ or H^{mid} , so $\text{rep}_{G'}(b) = \text{rep}_{\bar{H}^{\text{in}}}(b) = \text{Start}(p'_{2,2})$ by induction.

Now p_1 is covered by $\tilde{H}^{\text{out}}$ with lift p'_1 , $p_{2,1}$ is covered by H^{mid} with lift $p'_{2,1}$, and $p_{2,2}$ is covered by $\bar{H}^{\text{in}}$ with lift $p'_{2,2}$. Assuming all are non-empty (the empty cases are analogous), we have $\pi(\text{End}(p'_1)) = \text{End}(p_1)$, $\text{Start}(p'_{2,1}) = \text{rep}_{G'}(a)$ and $\pi(\text{End}(p'_{2,1})) = \text{End}(p_{2,1})$, and $\text{Start}(p'_{2,2}) = \text{rep}_{G'}(b)$. By [Definition C.12](#), edges $(\text{End}(p'_1), \text{Start}(p'_{2,1}))$ and $(\text{End}(p'_{2,1}), \text{Start}(p'_{2,2}))$ are in G' , so $p'_1 \oplus p'_{2,1} \oplus p'_{2,2}$ is a path in G' and is a lift of $p = p_1 \oplus p_{2,1} \oplus p_{2,2}$.

C.4.4 Size and Time Complexity

The algorithm is recursive. We first prove the following lemma.

Lemma C.14. *Suppose $G' \leftarrow \text{PATHCOVER}(A)$. The size of G' is at most $\sum_{v \in V(G')} \deg_{G[A]}(\pi(v))$.*

Proof. By induction on $|A|$. When $|A| = 1$, there are no edges in G' , so the lemma is trivial. Suppose $|A| > 1$. The algorithm returns different G' based on two cases.

Case 1. The size of G' contains two parts: (1) the sizes of $\bar{H}^{\text{out}}$, H^{out} , which, by induction, are at most

$$\sum_{v \in V(\bar{H}^{\text{out}})} \deg_{G[\bar{B}^{\text{out}}]}(\pi(v)) + \sum_{v \in V(H^{\text{out}})} \deg_{G[B^{\text{out}}]}(\pi(v)).$$

(2) The edges added by [Definition C.12](#). For every $v \in V(\bar{H}^{\text{out}})$, the number of edges added from v equals $\deg_{G[A]}(\pi(v)) - \deg_{G[\bar{B}^{\text{out}}]}(\pi(v))$, because only vertices in $A - \bar{B}^{\text{out}}$ can have representatives in H^{out} ; all other vertices are present in $\bar{H}^{\text{out}}$. Summing over $\sum_{v \in V(\bar{H}^{\text{out}})} (\deg_{G[A]}(\pi(v)) - \deg_{G[\bar{B}^{\text{out}}]}(\pi(v)))$ and $\sum_{v \in V(\bar{H}^{\text{out}})} \deg_{G[\bar{B}^{\text{out}}]}(\pi(v)) + \sum_{v \in V(H^{\text{out}})} \deg_{G[B^{\text{out}}]}(\pi(v))$ gives the bound $\sum_{v \in V(G')} \deg_{G[A]}(\pi(v))$.

Case 2. (i) *The $\tilde{H}^{\text{out}}$ -term.* All edges with both endpoints in $\tilde{H}^{\text{out}}$ lie inside the induced subgraph $G[\bar{B}^{\text{out}} \cap B^{\text{in}}]$. Thus

$$|E(\tilde{H}^{\text{out}})| \leq \sum_{v \in V(\tilde{H}^{\text{out}})} \deg_{G[\bar{B}^{\text{out}} \cap B^{\text{in}}]}(\pi(v)).$$

Any additional edge incident to $v \in V(\tilde{H}^{\text{out}})$ that goes to $H^{\text{mid}} \cup \bar{H}^{\text{in}}$ must use a representative outside $\bar{B}^{\text{out}} \cap B^{\text{in}}$, so the number of such outgoing edges is at most

$$\sum_{v \in V(\tilde{H}^{\text{out}})} \left(\deg_{G[A]}(\pi(v)) - \deg_{G[\bar{B}^{\text{out}} \cap B^{\text{in}}]}(\pi(v)) \right).$$

Consequently, the total number of edges charged to $\tilde{H}^{\text{out}}$ is at most

$$\sum_{v \in V(\tilde{H}^{\text{out}})} \deg_{G[A]}(\pi(v)).$$

(ii) *The H^{mid} -term.* Edges internal to H^{mid} are contained in the induced subgraph on $V(H^{\text{mid}})$, so

$$|E(H^{\text{mid}})| \leq \sum_{v \in V(H^{\text{mid}})} \deg_{G[V(H^{\text{mid}})]}(\pi(v)).$$

Edges from H^{mid} to $\bar{H}^{\text{in}}$ are added only from the H^{mid} -side, and for each $v \in V(H^{\text{mid}})$ there are at most

$$\deg_{G[A]}(\pi(v)) - \deg_{G[V(H^{\text{mid}})]}(\pi(v))$$

such edges. Hence the total number of edges charged to H^{mid} is at most

$$\sum_{v \in V(H^{\text{mid}})} \deg_{G[A]}(\pi(v)).$$

(iii) *The $\bar{H}^{\text{in}}$ -term.* By the induction hypothesis applied within $\bar{H}^{\text{in}}$, the edges with both endpoints in $\bar{H}^{\text{in}}$ are at most

$$|E(\bar{H}^{\text{in}})| \leq \sum_{v \in V(\bar{H}^{\text{in}})} \deg_{G[A]}(\pi(v)).$$

Inter-piece edges incident to $\bar{H}^{\text{in}}$ are *not* charged here (they were already charged to their tails in $\bar{H}^{\text{out}}$ or H^{mid}), so there is no double counting.

Summing the contributions from (i)–(iii) yields

$$\begin{aligned} |E(G')| &\leq \sum_{v \in V(\bar{H}^{\text{out}})} \deg_{G[A]}(\pi(v)) + \sum_{v \in V(H^{\text{mid}})} \deg_{G[A]}(\pi(v)) + \sum_{v \in V(\bar{H}^{\text{in}})} \deg_{G[A]}(\pi(v)) \\ &= \sum_{v \in V(G')} \deg_{G[A]}(\pi(v)), \end{aligned}$$

which completes the proof. □

Thus, it remains to bound $\sum_{v \in V(G')} \deg_G(\pi(v))$, which we prove next.

Lemma C.15. *Suppose $G' \leftarrow \text{PATHCOVER}(A)$. Then*

$$\sum_{v \in V(G')} \deg_G(\pi(v)) \leq \left(1 + 9(\log(\deg_G(A))) \cdot \frac{\varepsilon'}{\varepsilon}\right) \deg_G(A).$$

The running time of $\text{PATHCOVER}(A)$ is at most $\log(\deg_G(A)) \cdot O(\deg_G(A)/\varepsilon)$.

Proof. By induction on $|A|$. For $|A| = 1$, PATHCOVER returns $G[A]$, and the claim is trivial.

The algorithm first runs two procedures (forward and backward growing) in lockstep, one edge access at a time, and stops as soon as one stops. The time between two edge accesses is at most $O(\log n)$ using a standard heap. Assume forward growing stops earlier.

The forward procedure runs Dijkstra from u until it finds i^{out} , costing $O(\log n \cdot \deg_G(B^{\text{out}}))$.

We analyze the two cases.

Case 1: $\deg_G(B^{\text{out}}) < (1-\varepsilon) \deg_G(A)$. Here $V(G') = V(H^{\text{out}}) \cup V(\bar{H}^{\text{out}})$ with $H^{\text{out}} = \text{PATHCOVER}(B^{\text{out}})$ and $\bar{H}^{\text{out}} = \text{PATHCOVER}(\bar{B}^{\text{out}})$. By induction,

$$\begin{aligned} \sum_{v \in V(G')} \deg_G(\pi(v)) &\leq \left(1 + \frac{9\varepsilon' \log|\deg_G(B^{\text{out}})|}{\varepsilon}\right) \deg_G(B^{\text{out}}) + \left(1 + \frac{9\varepsilon' \log|\deg_G(\bar{B}^{\text{out}})|}{\varepsilon}\right) \deg_G(\bar{B}^{\text{out}}) \\ &\leq \left(1 + \frac{9\varepsilon'(\log(1-\varepsilon) + \log|\deg_G(A)|)}{\varepsilon}\right) (1 + \varepsilon') \deg_G(A - \bar{B}^{\text{out}}) \\ &\quad + \left(1 + \frac{9\varepsilon' \log|\deg_G(\bar{B}^{\text{out}})|}{\varepsilon}\right) \deg_G(\bar{B}^{\text{out}}) \\ &\leq \left(1 + \frac{9\varepsilon' \log|\deg_G(A)|}{\varepsilon}\right) \deg_G(A). \end{aligned}$$

The second inequality uses $\deg_G(B^{\text{out}}) < (1-\varepsilon) \deg_G(A)$, $\lambda \geq 1000 \log^2 n$, and the definitions of $B^{\text{out}}, \bar{B}^{\text{out}}$. The third inequality uses

$$\left(1 + \frac{9\varepsilon'(\log(1-\varepsilon) + \log|\deg_G(A)|)}{\varepsilon}\right) \leq 2, \quad \frac{\log(1-\varepsilon)}{\varepsilon} \leq -0.4,$$

after plugging in $\varepsilon', \varepsilon$.

The running time consists of two recursive calls and building the layered projection. The latter is bounded by $\sum_{v \in V(G')} \deg_G(\pi(v))$, since every edge we add corresponds to some $v' \in V(G')$ and an edge $(\pi(v'), v)$ in G . The total complexity is

$$\begin{aligned} O(\log n \cdot \deg_G(B^{\text{out}})) + \frac{\log(\deg_G(A)) + \log(1-\varepsilon)}{\varepsilon} \cdot O(\log n \cdot (\deg_G(B^{\text{out}}) + \deg_G(\bar{B}^{\text{out}}))) + O(\deg_G(A)) \\ \leq \log(\deg_G(A)) \cdot O(\log n \cdot \deg_G(A)/\varepsilon), \end{aligned}$$

where the first term is for finding i^{out} (or i^{in}), the second for recursion by induction, and the last for constructing G' , which is $\leq 2 \deg_G(A)$ since $\lambda > 10000 \log^4 n$.

Case 2: $\deg_G(B^{\text{out}}) \geq (1-\varepsilon) \deg_G(A)$. Now $V(G') = V(\tilde{H}^{\text{out}}) \cup V(H^{\text{mid}}) \cup V(\bar{H}^{\text{in}})$ with $\tilde{H}^{\text{out}} = \text{PATHCOVER}(\bar{B}^{\text{out}} \cap B^{\text{in}})$, $H^{\text{mid}} = G[B^{\text{out}} \cap B^{\text{in}}]$, and $\bar{H}^{\text{in}} = \text{PATHCOVER}(\bar{B}^{\text{in}})$. We need:

Lemma C.16. $\deg_G(\bar{B}^{\text{out}}), \deg_G(\bar{B}^{\text{in}}) \leq 2\varepsilon \deg_G(A)$.

Proof. Since forward growing stops earlier than backward growing, the backward procedure accesses at least as many edges before stopping. As each accessed edge contributes to $\deg_G(B^{\text{in}})$, we have $\deg_G(B^{\text{out}}), \deg_G(B^{\text{in}}) \geq (1-\varepsilon) \deg_G(A)$. By the definitions of $\bar{B}^{\text{out}}, \bar{B}^{\text{in}}$,

$$\deg_G(\bar{B}^{\text{out}}), \deg_G(\bar{B}^{\text{in}}) \leq \varepsilon \deg_G(A) + \varepsilon' \deg_G(A) \leq 2\varepsilon \deg_G(A).$$

□

By induction,

$$\begin{aligned}
\sum_{v \in V(G')} \deg_G(\pi(v)) &\leq \left(1 + \frac{9\varepsilon' \log |\deg_G(\bar{B}^{\text{out}} \cap B^{\text{in}})|}{\varepsilon}\right) \deg_G(\bar{B}^{\text{out}} \cap B^{\text{in}}) \\
&\quad + \left(1 + \frac{9\varepsilon' \log |\deg_G(\bar{B}^{\text{in}})|}{\varepsilon}\right) \deg_G(\bar{B}^{\text{in}}) \\
&\quad + \deg_G(A) \\
&\leq \left(1 + \frac{9\varepsilon' (\log |\deg_G(A)| + \log(2\varepsilon))}{\varepsilon}\right) \cdot 2\varepsilon \deg_G(A) \\
&\quad + \left(1 + \frac{9\varepsilon' (\log |\deg_G(A)| + \log(2\varepsilon))}{\varepsilon}\right) \cdot 2\varepsilon \deg_G(A) + \deg_G(A) \\
&\leq \deg_G(A) + 9\varepsilon \deg_G(A) \leq \left(1 + \frac{9\varepsilon' \log |\deg_G(A)|}{\varepsilon}\right) \deg_G(A),
\end{aligned}$$

using [Lemma C.16](#) and the definitions of B^{out} , $\bar{B}^{\text{out}}$, then plugging in $\varepsilon, \varepsilon'$.

The running time here includes two global SSSP computations costing $O(\deg_G(A) \log n)$, plus recursion time bounded by $(\log(1 - \varepsilon) + \log(\deg_G(A))) \cdot O(\log n \cdot \deg_G(A)/\varepsilon)$. Thus, the total is $\log(\deg_G(A)) \cdot O(\log n \cdot \deg_G(A)/\varepsilon)$. $\square$

C.5 Deterministic Negative Weight SSSP

In [Section C.5.1](#), we show how to solve the *restricted SSSP* problem (defined below). Bringmann, Cassis, and Fischer [[BCF23](#)] showed a black-box reduction from solving SSSP on general graphs to restricted SSSP. Although the main statements in [[BCF23](#)] are randomized (see Sections 4 and 5 in [[BCF23](#)]), the reduction to a restricted graph is deterministic.

Definition C.17 ([[BCF23](#)]). A *restricted graph* is a directed graph $G = (V, E, w)$ satisfying

- the edge weights lie in the set $\{-1, 0, 1, \dots, n\}$,⁵
- for every cycle C , we have $w(C)/|C| \geq 1$,
- there is a vertex $s \in V$ with an edge of weight 0 from s to every other vertex.

The problem *restricted SSSP* is: given a restricted graph, find a shortest-path tree from s .

Sections 4 and 5 of [[BCF23](#)] prove the following reduction.

Lemma C.18 (Sections 4 and 5 of [[BCF23](#)]). *If there is a deterministic algorithm for restricted SSSP with running time $T_{\text{RSSP}}(m, n)$, then there is a deterministic algorithm for general SSSP with running time $O(T_{\text{RSSP}}(m, n) \cdot \log(Wn))$.*

In [Section C.5.1](#), we prove the next lemma.

Lemma C.19. *There is a deterministic algorithm solving restricted SSSP in $O(m \log^8 n)$ time.*

Combining [Lemmas C.18](#) and [C.19](#) yields [Theorem C.1](#).

C.5.1 An Algorithm for Restricted SSSP

We now describe the algorithm for restricted SSSP. The input is a directed graph $G = (V, E, w)$ and a source s satisfying [Definition C.17](#). The algorithm outputs a shortest-path tree of G from s .

⁵In the original definition of [[BCF23](#)], there is no upper bound n on the edge weights. We can assume an upper bound n because any edge of weight $> n$ cannot appear in the shortest-path tree from s : otherwise that path would have weight at least 1 (the most negative edge has weight at least -1 and a simple path has at most $n - 1$ edges), contradicting that s has a 0-weight edge to every vertex. This assumption also appears in [[Fis+25](#)].

Before stating the algorithm, we recall two standard subroutines: (1) computing distances when all negative edges must cross different SCCs, and (2) computing distances when the number of negative edges on every shortest path is small (by alternating Dijkstra and Bellman–Ford).

Lemma C.20. *Given a directed graph $G = (V, E, w)$ where $w(u, v) \geq 0$ for every edge (u, v) with u, v in the same SCC of G , there is a deterministic $O(m \log n)$ -time algorithm for SSSP on G .*

Proof. Let the SCCs of G be $(V_1, V_2, \dots, V_z)$ in a topological order of the condensation DAG. Define a potential function ϕ by

$$\forall i \in [z], \forall v \in V_i : \phi(v) = -W \cdot i.$$

If $(u, v) \in E$ has u, v in the same SCC, then $w_\phi(u, v) = w(u, v) \geq 0$. Otherwise, if $u \in V_i$ and $v \in V_j$ with $i < j$, then

$$w_\phi(u, v) = w(u, v) + \phi(u) - \phi(v) = w(u, v) + W(j - i) \geq w(u, v) + W \geq 0.$$

Hence all edges of G_ϕ have nonnegative weight, so Dijkstra's algorithm computes SSSP on G_ϕ in $O(m \log n)$ time. By Lemma C.5, this yields a shortest-path tree for G as well. $\square$

Lemma C.21 (SSSP with Few Negative Edges). *There is a deterministic algorithm which, given a directed graph $G = (V, E, w)$ with no negative cycle, a source s , and an integer k such that every s -to- v shortest path uses at most k negative edges, computes $\text{dist}_G(s, \cdot)$ in $O(km \log n)$ time.*

Proof. Build a $(k+1)$ -layer graph G' as follows. For each $i \in [k+1]$, let G_i be a copy of $G_{\geq 0}$ (the subgraph of nonnegative-weight edges), and let u_i denote the copy of $u \in V$ in layer i . For every negative edge $(u, v) \in E$ with $w(u, v) < 0$ and every $i \in [k]$, add the interlayer edge (u_i, v_{i+1}) with weight $w(u, v)$. Run Dijkstra's algorithm from s_1 in G' , and return

$$\min_{i \in [k+1]} \text{dist}_{G'}(s_1, u_i) \quad \text{as } \text{dist}_G(s, u).$$

Running time. We have $|V(G')| = (k+1)|V|$ and

$$|E(G')| = (k+1)|E_{\geq 0}| + k|E_{< 0}| \leq O(km).$$

Thus Dijkstra on G' takes $O(km \log |V(G')|) = O(km \log n)$ time since $|V(G')| \leq (k+1)n$.

Correctness. First, fix $i \in [k+1]$ and let P' be any $s_1 \rightarrow u_i$ path in G' . Projecting layers and replacing each interlayer edge by its original negative edge yields an $s \rightarrow u$ walk in G of weight at most $w(P')$. Hence $\text{dist}_{G'}(s_1, u_i) \geq \text{dist}_G(s, u)$.

Conversely, let p be a shortest $s \rightarrow u$ path in G . By assumption, p uses at most k negative edges. Deleting these negative edges decomposes p as

$$p = p_1 \oplus p_2 \oplus \dots \oplus p_z,$$

with $z \leq k+1$ and each p_j consisting only of nonnegative edges. For each $j \in [z]$, let p'_j be the corresponding path in layer G_j (same vertices, same weight). For each negative edge between p_j and p_{j+1} , use the interlayer edge from layer j to $j+1$. Concatenating $p'_1, \dots, p'_z$ with these interlayer edges yields an $s_1 \rightarrow u_z$ path in G' of weight exactly $w(p) = \text{dist}_G(s, u)$. Therefore

$$\min_i \text{dist}_{G'}(s_1, u_i) \leq \text{dist}_G(s, u).$$

Combining both directions gives $\text{dist}_G(s, u) = \min_i \text{dist}_{G'}(s_1, u_i)$. $\square$

Our algorithm is recursive. We describe an algorithm $\text{kSSSP}(H, s, k)$ that solves a restricted SSSP instance (H, s) under the promise that every shortest s -to- v path in H uses at most k negative edges. It suffices to call $\text{kSSSP}(H, s, n)$ to prove Lemma C.19.

Base case. When $k = O(\log^6 n)$, apply [Lemma C.21](#) to (H, s, k) to obtain all distances from s in H . For the remainder, assume $k = \omega(\log^6 n)$.

Building a path cover. Compute a d_{cov} -path cover $G'_{\geq 0}$ of $H_{\geq 0}$ with at most $(1 + 1/\log n) \cdot |E(H)|$ edges and diameter slack λ such that

$$d_{\text{diam}} \leq \lambda \cdot d_{\text{cov}} = k/2.$$

By [Theorem C.10](#), it suffices to set

$$\lambda = 10000 \log^6 n \quad \text{and} \quad d_{\text{cov}} = \frac{k}{20000 \log^6 n},$$

and invoke [Theorem C.10](#) to obtain such a $G'_{\geq 0}$.

Recursive call. From $G'_{\geq 0}$, form G' by restoring each 0-weight edge back to its original weight in H : for every $(u', v') \in E(G'_{\geq 0})$ with $w(\pi(u'), \pi(v')) < 0$, set $w_{G'}(u', v') := w(\pi(u'), \pi(v'))$.

Define G'_{SCC} from G' by:

1. computing the SCC decomposition of G' and deleting all edges whose endpoints lie in different SCCs, and
2. adding a super-source s' and an edge of weight 0 from s' to every vertex of G' .

Call $\kappa\text{SSSP}(G'_{\text{SCC}}, s', k/2)$ to obtain $d_{s'}(\cdot) = \text{dist}_{G'_{\text{SCC}}}(s', \cdot)$.

Computing distances for H . Construct G'' as follows.

1. Make $x = 2\lambda$ copies of G' , denoted $G'_1, \dots, G'_x$. For $u' \in V(G')$, let $u'_i \in V(G'_i)$ be its copy.
2. For every $1 \leq i < x$, each $(u, v) \in E(H)$, and each copy $u'_i \in \pi^{-1}(u)$ in G'_i and the representative $v'_{i+1} \in \pi^{-1}(v)$ of v in G'_{i+1} , add the edge (u'_i, v'_{i+1}) with weight $w(u, v)$.
3. Add a source s'' and, for every vertex u'_i with $\pi(u') = s$, add the 0-weight edge (s'', u'_i) .

Define a potential ϕ on G'' by

$$\phi(u'_i) = d_{s'}(u') \quad \text{for } u'_i \neq s'', \quad \phi(s'') = 0.$$

We will show that G''_ϕ satisfies the premise of [Lemma C.20](#). Invoke [Lemma C.20](#) on G''_ϕ to compute $d_{s''} : V(G'') \rightarrow \mathbb{Z}$. For every $u \in V(H)$, output

$$d_s(u) = \min\{d_{s''}(u'_i) + \phi(u'_i) \mid u'_i \in V(G''), \pi(u') = u\}.$$

C.5.2 Correctness

Lemma C.22. $\kappa\text{SSSP}(H, s, k)$ correctly outputs $\text{dist}_H(s, \cdot)$ provided that

- (H, s) is a restricted SSSP instance, and
- every shortest path from s in H uses at most k negative edges.

We proceed by induction on k . The base case $k = O(\log^6 n)$ follows directly from [Lemma C.21](#). For the induction step, assume $k = \omega(\log^6 n)$.

Validity of oracle calls. We first verify the two oracle calls: the recursive call $\kappa\text{SSSP}(G'_{\text{SCC}}, s', k/2)$ and the call to [Lemma C.20](#).

Lemma C.23. (G'_{SCC}, s') is a restricted SSSP instance.

Proof. We verify the three bullets in [Definition C.17](#). Since $G'_{\geq 0}$ is a projection onto $H_{\geq 0}$ ([Definition C.6](#)) and G' restores weights only by replacing some 0-weights with the original weights in H , G' is a projection

onto H . As $G'_{\text{SCC}} - \{s'\}$ is a subgraph of G' , it is also a projection onto H ; hence all edge weights lie in $\{-1, 0, 1, \dots, n\}$ (edges adjacent to s' have weight 0).

Let C be any cycle in G'_{SCC} . Since s' has no incoming edges, C lies in $G'_{\text{SCC}} - \{s'\}$, which projects to a (possibly non-simple) cycle $\pi(C)$ in H . By [Definition C.17](#), $w(\pi(C))/|\pi(C)| \geq 1$, hence $w(C)/|C| \geq 1$. The last bullet holds by construction of s' . $\square$

Lemma C.24. *Every shortest path from s' in G'_{SCC} uses at most $k/2$ negative edges.*

Proof. Suppose, for contradiction, that a shortest s' -to- v' path p in G'_{SCC} uses $> k/2$ negative edges. Let p' be p with its first vertex (the source s') removed; then p' still uses $> k/2$ negative edges and lies inside a single SCC of G' by construction of G'_{SCC} . Also $w_{G'_{\text{SCC}}}(p') \leq 0$, as otherwise the direct 0-edge from s' to $\text{End}(p')$ would give a strictly shorter path, contradicting minimality of p .

Write $\text{Start}(p') = u'$ and $\text{End}(p') = v'$. Since each SCC of $G'_{\geq 0}$ has (strong) diameter at most $d_{\text{diam}} \leq k/2$, there exists in $H_{\geq 0}$ a path $\tilde{p}$ from $\pi(v')$ to $\pi(u')$ of length at most $k/2$. The walk $\pi(p') \oplus \tilde{p}$ forms a (possibly non-simple) cycle in H , so by [Definition C.17](#),

$$w(\pi(p')) + w(\tilde{p}) \geq |\pi(p')| + |\tilde{p}|.$$

Using $w(\pi(p')) = w_{G'_{\text{SCC}}}(p') \leq 0$ and $w(\tilde{p}) \leq w_{\geq 0}(\tilde{p}) \leq k/2$, the number of *all* edges in p' is

$$|p'| = |\pi(p')| \leq k/2,$$

contradicting that p' has more than $k/2$ negative edges. $\square$

Lemma C.25. *For every edge $(u'', v'') \in E(G''_{\phi})$ with u'', v'' in the same SCC of G''_{ϕ} , we have $w_{G''_{\phi}}(u'', v'') \geq 0$.*

Proof. The source s'' has no incoming edges, so assume $u'', v'' \neq s''$. By construction of G'' , edges go only from G'_i to G'_j with $j \geq i$, hence if u'', v'' lie in the same SCC of G''_{ϕ} , they must belong to the same copy G'_i and correspond to $u', v' \in V(G')$ lying in the same SCC of G' . It suffices to show

$$w_{G'}(u', v') + \phi(u') - \phi(v') \geq 0.$$

By definition, $\phi(z') = \text{dist}_{G'_{\text{SCC}}}(s', z')$ for $z' \in V(G')$. The triangle inequality in G'_{SCC} gives

$$w_{G'_{\text{SCC}}}(u', v') + \text{dist}_{G'_{\text{SCC}}}(s', u') - \text{dist}_{G'_{\text{SCC}}}(s', v') \geq 0.$$

Since $w_{G'_{\text{SCC}}}(u', v') = w_{G'}(u', v')$, we obtain $w_{G'}(u', v') + \phi(u') - \phi(v') \geq 0$, as desired. $\square$

Correctness of the returned distances.

Lemma C.26. $d_s(u) \geq \text{dist}_H(s, u)$ for all $u \in V(H)$.

Proof. By [Lemma C.20](#) and [Lemma C.25](#), $d_{s''}(z)$ equals $\text{dist}_{G''_{\phi}}(s'', z)$ for all $z \in V(G'')$. By [Lemma C.5](#),

$$d_{s''}(u'_i) + \phi(u'_i) = \text{dist}_{G''_{\phi}}(s'', u'_i).$$

Any s'' -to- u'_i path in G'' maps to an s -to- u path in H of the same weight by replacing each u'_i with $\pi(u'_i)$; the initial edge (s'', u'_i) has weight 0 and ensures the path starts at s . Taking minima over copies yields $d_s(u) \geq \text{dist}_H(s, u)$. $\square$

Lemma C.27. $d_s(u) \leq \text{dist}_H(s, u)$ for all $u \in V(H)$.

Proof. Let p be a shortest s -to- u path in H . Since every edge weight is at least -1 , we have the number of negative edges in p is at most k (by assumption) and hence $w_{\geq 0}(p) \leq k$; otherwise $w(p) \geq -k + w_{\geq 0}(p) > 0$, contradicting the 0-edges from s .

Recall $x = 2\lambda$ and $d_{\text{cov}} = k/(2\lambda)$. Partition p into at most x subpaths $p = p_1 \oplus \dots \oplus p_x$ with $w_{\geq 0}(p_i) \leq d_{\text{cov}}$. Each p_i is covered by $G'_{\geq 0}$, hence also by G' (weights restored). For each i , let p'_i be a lift in G' , and let p''_i be the copy of p'_i in G'_i . By construction of G'' , there is an edge from $\text{End}(p''_i)$ to $\text{Start}(p''_{i+1})$ with weight equal to $w(\text{End}(p_i), \text{Start}(p_{i+1}))$. Thus $p'' = p''_1 \oplus \dots \oplus p''_x$ is an s'' -to- u'_x path in G'' with $w_{G''}(p'') = w(p)$. Therefore,

$$\min_{\pi(u')=u} \text{dist}_{G''}(s'', u'_i) \leq \text{dist}_H(s, u).$$

Using $d_{s''}(u'_i) + \phi(u'_i) = \text{dist}_{G''}(s'', u'_i)$ finishes the proof. $\square$

C.5.3 Time Complexity

Let $T(m, k)$ denote the running time of $\text{kSSSP}(H, s, k)$. We show $T(m, n) = O(m \log^8 n)$ by writing a recurrence for $T(m, k)$.

- Building the path cover for $H_{\geq 0}$ takes $O(m \log^4 n)$ time by [Theorem C.10](#) (since $\sqrt{\lambda} = 100 \log^3 n$).
- The recursive call requires an SCC decomposition in $O(m)$ time and costs

$$T\left(\left(1 + \frac{1}{\log n}\right)m, \frac{k}{2}\right).$$

- For computing distances, we build G'' . Each vertex u'_i has out-degree $O(\deg_H(\pi(u')))$, and there are $x = O(\log^6 n)$ copies, hence

$$|E(G'')| \leq O(\log^6 n) \cdot \sum_{u' \in V(G')} \deg_H(\pi(u')) = O(m \log^6 n)$$

by [Theorem C.10](#). Therefore [Lemma C.20](#) on G'' costs $O(m \log^7 n)$ time.

We obtain the recurrence

$$T(m, k) \leq T\left(\left(1 + \frac{1}{\log n}\right)m, \frac{k}{2}\right) + O(m \log^7 n),$$

with base case $T(m, O(\log^6 n)) = O(m \log^7 n)$ by [Lemma C.21](#). Solving gives $T(m, n) = O(m \log^8 n)$.

C.6 Randomized or Undirected Solutions for [Problem C.2](#)

Low Diameter Decomposition: Randomized Solution. Let G be a directed graph G with non-negative weight and $d' \geq 0$. A *low-diameter decomposition* (LDD) is an edge set $E' \subseteq E$ such that $G \setminus E'$ is a $\tilde{O}(d')$ -clustered graph and, for each $e \in E$, $\Pr[e \in E'] \leq \frac{w(e)}{d'} \log n$. Bernstein et al. [[BNW22](#)] showed how to sample E' in near-linear time.

Observe that, by sampling $z = O(\log n)$ copies of LDD E'_i of diameter parameter $d' = 2d \log n$ and set $G'_i = G \setminus E'_i$, each length- d path p is covered by some G'_i with probability at least $1 - 1/n^{10}$. This is because, for each i ,

$$\Pr[p \cap E'_i \neq \emptyset] \leq \sum_{e \in p} \Pr[e \in E'_i] \leq \sum_{e \in p} \frac{w(e)}{d'} \log n \leq \frac{w(p)}{d'} \log n \leq 1/2.$$

So, there is i where $p \subseteq G \setminus E'_i$ with probability $1 - 1/2^z \geq 1 - 1/n^{10}$. Thus, every path in the unknown target set is covered with probability at least $1 - 1/n^8$ by union bound.

Neighborhood Cover: Undirected Solution. A d -clustering $\mathcal{C}$ consists of vertex-disjoint subsets called *clusters*, each of which has diameter at most d . For any vertex v , let $B_v(v, d) = \{w \mid \text{dist}_G(v, w) \leq d\}$.

A d -neighborhood cover $\mathcal{N}_d$ of G is a collection of $z = O(\log n)$ many $O(d \log n)$ -clustering $\mathcal{C}_1, \dots, \mathcal{C}_z$ such that, for every vertex v , $B_v(v, d') \subseteq S$ for some cluster S in some clustering $\mathcal{C}_i$. For any d , Awerbuch et al. [[Awe+98](#)] showed how to construct $\mathcal{N}_d$ deterministically in near-linear time.

Observe that $\mathcal{N}_d$ gives the solution to [Problem C.2](#). Indeed, for each $i \leq z$, let $G'_i = \bigcup_{S \in \mathcal{C}_i} G[S]$. Consider any length- d path p in G starting from vertex u . We have that $p \subseteq B_v(v, d) \subseteq G'_i$ for some i .

C.7 A Barrier for Covering Paths via Subgraphs

In this section, we prove [Theorem C.3](#).

Theorem C.3 (Barrier for Subgraphs). *For every $m, \lambda \geq 1$, there exists an $O(m)$ -edge directed graph G and d such that every collection of $d\lambda$ -clustered graphs $G'_1, \dots, G'_z \subseteq G$ covering all d -paths in G must have total size $\sum_i |E(G'_i)| = \Omega(m\sqrt{m/\lambda})$.*

Fix $m, \lambda \geq 1$. We will build a directed graph G with $O(m)$ edges and a distance parameter d .

Step 1: choose parameters. Let

$$L := \lfloor \sqrt{m/\lambda} \rfloor \quad \text{and} \quad d := 2 + 3L.$$

So $L = \Theta(\sqrt{m/\lambda})$ and $d = \Theta(\sqrt{m/\lambda})$.

We will also use one more parameter

$$R := 2d\lambda.$$

Note that $R > d\lambda$ by construction.

Step 2: ladder gadget. Create vertices $a_1, a_2, \dots, a_{L+1}$. For each layer $j = 1, \dots, L$ do:

- create R new vertices $w_{j,1}, \dots, w_{j,R}$,
- add the directed cycle

$$w_{j,1} \rightarrow w_{j,2} \rightarrow \dots \rightarrow w_{j,R} \rightarrow w_{j,1},$$

- add edges $a_j \rightarrow w_{j,r}$ for every $r \in [R]$,
- add edges $w_{j,r} \rightarrow a_{j+1}$ for every $r \in [R]$.

Thus, to “go through” layer j you can do

$$a_j \rightarrow w_{j,r} \rightarrow w_{j,r+1} \rightarrow a_{j+1},$$

which is 3 edges.

The number of edges in one layer is

$$R \text{ (cycle)} + R \text{ (entries)} + R \text{ (exits)} = 3R = 3 \cdot 2d\lambda = 6d\lambda.$$

Since there are L layers, the total number of edges in the ladder is

$$6d\lambda \cdot L = 6(2 + 3L)\lambda L = O(\lambda L^2) = O(\lambda \cdot (m/\lambda)) = O(m).$$

Key property of the ladder. Let H be any subgraph of G whose SCCs all have diameter at most $d\lambda$. Consider a fixed layer j . If H contained all R edges of the directed cycle of layer j , then that SCC would have directed diameter $R - 1 \geq d\lambda$, contradicting that H is $d\lambda$ -clustered. Hence: For every $d\lambda$ -clustered subgraph H and every layer $j \in [L]$, H omits at least one edge of that layer’s cycle.

Step 3: star in front of the ladder. Add a special vertex r (the center of the star) and add the edge $r \rightarrow a_1$. Then add

$$M := \lfloor m/20 \rfloor$$

new vertices $s_1, \dots, s_M$ and add star edges

$$s_t \rightarrow r \quad \text{for } t = 1, \dots, M.$$

This adds $M + 1 = O(m)$ edges. Since the ladder already cost $O(m)$ edges, the whole graph G has $O(m)$ edges.

Step 4: the d -paths we care about. Consider a path that:

1. starts with some star edge $s_t \rightarrow r$,
2. then takes $r \rightarrow a_1$,
3. then for each layer $j = 1, \dots, L$ takes a 3-edge traversal

$$a_j \rightarrow w_{j,r_j} \rightarrow w_{j,r_{j+1}} \rightarrow a_{j+1}.$$

This path has length

$$1 \text{ (star)} + 1 \text{ (to } a_1) + 3L = 2 + 3L = d.$$

So these are exactly d -paths in G .

Step 5: diagonalization for one star edge. Let $\{G'_1, \dots, G'_z\}$ be any collection of $d\lambda$ -clustered subgraphs of G that covers all d -paths.

Fix one star edge $e_t := s_t \rightarrow r$. Define

$$I_t := \{i \in [z] : e_t \in E(G'_i)\}.$$

We claim that

$$|I_t| \geq L. \tag{C.2}$$

Suppose toward a contradiction that $|I_t| = q < L$. For each $i \in I_t$, for every layer j it omits at least one cycle edge in that layer. Since $q < L$, we can pick for every $i \in I_t$ a *distinct* layer $\ell(i) \in [L]$ and let f_i be one cycle edge in layer $\ell(i)$ that G'_i omits.

Now we build a d -path P as in Step 4, but with the following extra choice: in layer $\ell(i)$ we force the traversal to *use* exactly the edge f_i , and in all other layers we traverse arbitrarily. This is a valid d -path because every layer allows us to pick any two consecutive cycle vertices.

By construction:

- If $i \notin I_t$, then G'_i does not contain the first edge e_t , so $P \not\subseteq G'_i$.
- If $i \in I_t$, then G'_i does not contain f_i (by choice), and P uses f_i , so again $P \not\subseteq G'_i$.

Thus P is a d -path that is not contained in any G'_i , contradicting the assumption that the family covers all d -paths. Therefore (C.2) holds.

Step 6: counting. For each of the $M = \Theta(m)$ star edges e_t we have found at least L subgraphs that contain it. So the total number of *edge-subgraph incidences* is at least

$$\sum_{t=1}^M |I_t| \geq M \cdot L.$$

But

$$\sum_{i=1}^z |E(G'_i)|$$

counts all edge-subgraph incidences (each time an edge appears in some G'_i it is counted once). Hence

$$\sum_{i=1}^z |E(G'_i)| \geq M \cdot L.$$

Finally, $M = \Theta(m)$ and $L = \Theta(\sqrt{m/\lambda})$, so

$$\sum_{i=1}^z |E(G'_i)| \geq \Omega(m) \cdot \Omega(\sqrt{m/\lambda}) = \Omega(m\sqrt{m/\lambda}),$$

which is exactly the desired lower bound.

This completes the proof.

Paper D

Global vs. s-t Vertex Connectivity Beyond Sequential: Almost-Perfect Reductions & Near-Optimal Separations

Joakim Blikstad, Yonggang Jiang, Sagnik Mukhopadhyay,
Sorrachai Yingchareonthawornchai

Published at ACM Symposium on Theory of Computing (STOC) 2025.

Full version available at <https://arxiv.org/abs/2503.20366>.

Abstract

A recent breakthrough by [LNPSY STOC'21] showed that solving s-t vertex connectivity is sufficient (up to polylogarithmic factors) to solve (global) vertex connectivity in the sequential model. This raises a natural question: what is the relationship between s-t and global vertex connectivity in other computational models? In this paper, we demonstrate that the connection between global and s-t variants behaves very differently across computational models:

- In parallel and distributed models, we obtain almost tight reductions from global to s-t vertex connectivity. In PRAM, this leads to a $n^{\omega+o(1)}$ -work and $n^{o(1)}$ -depth algorithm for vertex connectivity, improving over the 35-year-old $\tilde{O}(n^{\omega+1})$ -work $O(\log^2 n)$ -depth algorithm by [LLW FOCS'86], where ω is the matrix multiplication exponent and n is the number of vertices. In CONGEST, the reduction implies the first sublinear-round (when the diameter is moderately small) vertex connectivity algorithm. This answers an open question in [JM STOC'23].
- In contrast, we show that global vertex connectivity is strictly harder than s-t vertex connectivity in the two-party communication setting, requiring $\tilde{\Theta}(n^{1.5})$ bits of communication. The s-t variant was known to be solvable in $\tilde{O}(n)$ communication [BvdBEMN FOCS'22]. Our results resolve open problems raised by [MN STOC'20, BvdBEMN FOCS'22, AS SOSA'23].

At the heart of our results is a new graph decomposition framework we call *common-neighborhood clustering*, which can be applied in multiple models. Finally, we observe that global vertex connectivity cannot be solved without using s-t vertex connectivity, by proving a s-t to global reduction in dense graphs, in the PRAM and communication models.

D.1 Introduction

The problem of finding (global) vertex connectivity of a simple undirected graph aims to find the minimum number of vertices whose removal disconnects the graph (or the graph becomes a singleton). This problem, along with a very closely related problem of (global) edge connectivity (which asks for the minimum number of edges to be removed to disconnect the graph), are classic and fundamental graph problems that have drawn the attention of researchers for the last fifty years [Kle69; Pod73; ET75; Eve75; Gal80; EH84; Mat87; Bec+82; LLW88; CT91; NI92; CR94; Hen97; HRG00; Gab06; CGK14b; SY22]. Both problems can be easily solved by n^2 calls to their s-t variants¹, which is in addition given two vertices s, t and asking the minimum number of vertices (or edges) whose removal disconnects s and t . This simple idea is not efficient for two reasons. Firstly, s-t vertex (or edge) connectivity is known to be equivalent to max flow with unit vertex capacities (or edge capacities), which is a hard problem.² Besides, n^2 calls add an even larger polynomial factor to their s-t variants running time. Thus, researchers focus on two questions: is there an algorithm to circumvent their s-t variants? Can we reduce the number of calls to their s-t variants?

For edge connectivity, the answer to the first question is affirmative. In the conventional unit-cost RAM (i.e., sequential) model, a long line of work spanning over many decades in the last century was concluded by a near-linear time algorithm [Kar00], which does not require s-t edge connectivity computation, but instead uses a technique called ‘tree-packing’. This technique has been proven to be versatile enough to lead to near-optimal algorithms in various computational models [MN20; LMN21; Dor+21]. One can view these developments through the lens of the *cross-paradigm algorithm* design—a relatively new research direction where the general theme is to come up with schematic algorithms that can be implemented in several computational models without much trouble [MN20; Bli+22; Roz+22b; Nan24]. The second question was also answered affirmatively recently by the technique ‘isolating cuts’ [LP20], which reduces the number of calls to polylogarithmic.

For the vertex connectivity problem, in the sequential model, the second question was solved very recently [Li+21] which reduces the number of calls to polylogarithmic. However, unlike the tree packing framework for edge connectivity which has been successfully implemented in various models, the reduction in [Li+21] is highly sequential and suffers from several bottlenecks which make it only suitable for the sequential model (we discuss more about these bottlenecks in Section D.2). Moreover, most the known algorithms for vertex connectivity [Eve75; Bec+82; LLW88; CT91; CR94; HRG00; Gab06; Li+21] This is in contrast with the progress on edge connectivity, which is known to be easier to handle than its s-t variant. Thus, a central question this paper aims to answer is:

*Is solving s-t vertex connectivity necessary and/or sufficient for solving
(global) vertex connectivity in various computational models?*

In this paper, we show a trifecta of such connections:

Sufficiency: We show that s-t vertex connectivity is *sufficient* for vertex connectivity on *parallel and distributed models* by providing an almost-perfect reduction that uses only $n^{o(1)}$ calls. This reduction is built on a new framework using *common-neighborhood clustering*, which is of independent interest.

Insufficiency: In contrast, we show that s-t vertex connectivity is *not sufficient* for vertex connectivity in the *communication model*, by showing a near-optimal lower bound separating the complexity of s-t vertex connectivity and vertex connectivity. We also provide a matching upper bound, thus settling the communication complexity of vertex connectivity.

Necessity: Unlike the case for edge-connectivity, we observe that s-t vertex connectivity is indeed *necessary* for vertex connectivity, at least in dense graphs. This follows from a straightforward reduction from s-t to global vertex connectivity in dense graphs. The reduction works in the PRAM and communication

¹Throughout this paper, we use n to denote the number of vertices and m to denote the number of edges of the input graph.

²While there now exist almost linear time maximum flow algorithms in the sequential setting [Che+25], these algorithms are far from simple. Moreover, in other models of computation it remains largely open how efficiently one can solve (even unit-vertex-capacitated) maximum flow.

models, but when applied in the CONGEST model it adds additional communication edges. To the best of our knowledge, this simple reduction is novel. This observation implies that any algorithm developed for global vertex connectivity can be adapted to solve s-t vertex connectivity on dense graphs (and hence also unit-vertex-capacitated maximum flow). This explains the lack of vertex connectivity algorithms that work without calling its s-t variant, unlike what is the case for edge connectivity.

D.1.1 Our Results: Reductions and Separations

Parallel and Distributed Computing. We use the standard PRAM and CONGEST models for parallel and distributed computing respectively (see Section D.3.2 for the detailed definitions). Our first results are almost perfect reduction algorithms in both models.

Theorem D.1. *In PRAM model, if s-t vertex connectivity can be solved in $W(m, n)$ work³ and $D(m, n)$ depth where $W(m, n)$ is superadditive⁴ on m , then vertex connectivity can be solved in $W(m, n) \cdot n^{o(1)}$ work and $D(m, n) \cdot n^{o(1)}$ depth.*

Theorem D.2. *In CONGEST model, if S-T vertex connectivity⁵ can be solved in $R(m, n, D)$ rounds,⁶ then vertex connectivity can be solved in $R(m, n, D) \cdot n^{o(1)}$ rounds.*

Our main technical contribution for showing Theorems D.1 and D.2 is a new algorithmic framework based on graph decomposition for vertex connectivity. We introduce a graph property called *common-neighborhood* and show that if the graph satisfies this property, then the reduction can be handled in PRAM and CONGEST. For the general graph, we develop an efficient algorithmic framework that decomposes the graph into *common-neighborhood* clusters. The decomposition algorithm, as well as its efficient implementation in both PRAM and CONGEST⁷, might be of independent interest.

Our approach is similar to recent advances in graph algorithms, where problems are first solved on graphs with “nice properties” and then extended to general graphs through decomposition techniques (for example, low-diameter decomposition [FRT04; ABN08; Coh+14; BNW22], expander decomposition [ST04; She13; RST14], length-constraint expander decomposition [HRG22; Hae+24]). Certain graph decompositions are also proven to be highly parallelizable [MPX13; CS19; CS20; Ash+24a]; this is also the case for our common-neighborhood clustering.

Two-Party Communication. Given the positive results in the sequential, PRAM, and CONGEST models, one might expect a perfect reduction from vertex connectivity to its s-t variant in other models as well. Perhaps surprisingly, we show that this is not always the case. In particular in the classic *two-party communication* model, we show a lower bound separating the communication complexity of vertex connectivity and s-t vertex connectivity by $\tilde{\Theta}(\sqrt{n})$.⁸ We also show a matching upper bound, thus resolving the communication complexity of vertex connectivity up to polylogarithmic factors.

Theorem D.3. *The two-party communication complexity of vertex connectivity is $\tilde{\Theta}(n^{1.5})$.*

It is known from previous work [Bli+22] that the communication complexity of s-t vertex connectivity is $\tilde{\Theta}(n)$. Thus, Theorem D.3 provides a strict separation of these two problems. Previously, a lower bound of $\Omega(n^2)$ is only known to hold on multigraph [AS23].

³We assume all the complexity functions in this paper are reasonably smooth in the sense that $W(Cm, Cn) = \text{poly}(C) \cdot W(m, n)$.

⁴ $W(m, n)$ is superadditive on m means $W(m_1 + m_2, n) \geq W(m_1, n) + W(m_2, n)$. For example, $W(m, n) = m\sqrt{n}$ is superadditive, while $W(m, n) = n^\omega$ would not be superadditive.

⁵S-T vertex connectivity is given two disjoint vertex sets S, T and asking the minimum vertex cut separating S and T . In other models we can simply contract S and T into single vertices. However, in CONGEST, this is not allowed as it would change the underlying communication network. This variant does not incur new challenges as commonly both s-t or S-T vertex connectivity are solved by maximum bipartite matching.

⁶ D is the diameter of the input network.

⁷More precisely, our algorithm works in the recently introduced *minor-aggregation model* [Roz+22b] in $n^{o(1)}$ rounds.

⁸Throughout this paper, we use $\tilde{O}(\cdot)$ to hide poly log n factors, and $\hat{O}(\cdot)$ to hide $n^{o(1)}$ factors.

In contrast, we note that for *edge-connectivity*, seemingly the s-t variant is harder than the global variant: it is known that the global variant admits an $\tilde{O}(n)$ communication protocol [MN20], while it remains an open question if this is also the case for the s-t edge-connectivity.

D.1.2 Our Results: Implications

Implications in PRAM and CONGEST. By folklore reductions, s-t vertex connectivity is known to be equivalent to *maximum bipartite matching* (BMM). By combining with our reduction Theorems, we directly get the following results.

Theorem D.4. *In PRAM model, vertex connectivity can be solved in $mn^{0.5+o(1)}$ work and $n^{0.5+o(1)}$ depth.*

Theorem D.5. *In CONGEST model, vertex connectivity can be solved in $n^{o(1)} \cdot (n^{3/8}D^{1/2} + n^{1/2}D + n^{7/10}D^{7/10})$ rounds, which is sublinear as long as $D = n^{3/7-\varepsilon}$ for some constant $\varepsilon > 0$.*

The superadditive requirement in Theorem D.1 might prevent us from reduction to BMM algorithm in matrix multiplication work and depth [Lov79; KUW86; MVV87]. However, although not directly implied by Theorem D.1, our reduction framework is versatile enough to obtain vertex connectivity in matrix multiplication work and depth as well.

Theorem D.6. *In PRAM model, vertex connectivity can be solved in $n^{\omega+o(1)}$ work and $n^{o(1)}$ depth, where n^ω is the optimal work for matrix multiplication in $n^{o(1)}$ depth.*

Remark D.7. The $n^{\omega+o(1)}$ work and $n^{o(1)}$ depth can be improved to $\tilde{O}(n^\omega)$ work and $\tilde{O}(1)$ depth if we apply a common-neighborhood cover algorithm with $\tilde{O}(n^2)$ work and $\tilde{O}(1)$ depth instead of $\tilde{O}(m)$ work and $n^{o(1)}$ depth. However, for consistency throughout the paper across different models, we do not optimize this here.

Previous Work. Parallel vertex connectivity was mostly studied for small κ (the vertex connectivity of the input graph) [CT91; CKT93; GH20], which have at least $\Omega(n)$ depths for large κ . Despite the long-known fact that BMM can be solved in matrix multiplication work and depth [Lov79; KUW86; MVV87], the only known sublinear depth algorithm for vertex connectivity in the general case is by reduction to matrix multiplication [LLW88] in $\tilde{O}(n^{\omega+1})$ work and $O(\log^2 n)$ depth. Our result Theorem D.6 improves this 35-year-old result in the subpolynomial depth regime to almost matrix multiplication time. Moreover, in the sublinear depth regime, our result Theorem D.4 is faster than matrix multiplication time on moderately sparse graphs.

In the CONGEST model, vertex connectivity was either studied for small κ [Thu97; PT11; Par20; PP22; JM23], or by $O(\log n)$ approximation [CGK14b]. For exact computation and large κ , all of the mentioned results cost at least $\Omega(n)$ rounds, even for a graph with constant diameter. Our result Theorem D.5 gives the first sublinear round algorithm for exact large vertex connectivity as long as D is moderately small.

Implications for Streaming. We use the standard graph streaming model (formally defined in Section D.3.2). It is a well-known fact that an efficient streaming protocol yields an efficient communication protocol. Thus, our communication lower bound in Theorem D.3 immediately implies the following streaming lower bound.

Theorem D.8 (Streaming lower bound). *Any $P(n)$ -pass randomized streaming algorithm of vertex connectivity needs $\Omega(n^{1.5}/P(n))$ space.*

Our algorithmic ideas can be used in the streaming model as well and give the following reduction theorem. In the randomized streaming model, we say a reduction from vertex connectivity to maximum bipartite matching is an (α, β) -reduction if vertex connectivity can be solved in $\alpha S(n)$ space and $\beta P(n)$ passes as long as maximum bipartite matching can be solved in $S(n)$ space and $P(n)$ passes.

Theorem D.9 (Streaming upper bound). *In the randomized streaming model, there is a $(\tilde{O}(n^{0.5}), O(1))$ -reduction from vertex connectivity to maximum bipartite matching.*

A natural question is whether the reduction of [Theorem D.9](#) is optimal or not. Although we cannot show a lower bound, there is a simple observation that any better reduction gives a strong streaming lower bound for maximum bipartite matching.

Corollary D.10. *If an (α, β) -reduction from vertex connectivity to maximum bipartite matching exists for $\alpha \cdot \beta < n^{0.5-\epsilon}$, then in the semi-streaming model, maximum bipartite matching cannot be solved in $o(n^\epsilon)$ passes.*

The implied BMM lower bound of [Corollary D.10](#) would be a breakthrough in semi-streaming lower bound [[AG11](#); [AR20b](#); [Che+21](#)]. Hence, if we are to believe the conjecture that maximum bipartite matching can be solved in $\tilde{O}(n)$ space and $n^{o(1)}$ passes, then according to [Corollary D.10](#), the reduction of [Theorem D.9](#) can not be improved by a polynomial factor. Alternatively, [Corollary D.10](#) can be viewed as a new potential way to obtain polynomial pass semi-streaming lower bound for bipartite matching through developing better reductions for vertex connectivity.

At last, by combining [Theorem D.9](#) with the current progress on BMM [[Ass+22](#)], we get the following result.

Corollary D.11. *There is a randomized streaming algorithm for vertex connectivity with $\tilde{O}(n^{1.5})$ space and $n^{0.75+o(1)}$ passes.*

D.2 Technical Overview

Given an undirected graph $G = (V, E)$, a partition (L, S, R) of V (where L, S, R are all non-empty) is a *vertex cut* (or simply *cut* in this work) if there are no edges between L and R . The size of the cut (L, S, R) is defined as $|S|$, and we say S is a *separator* or more precisely, (s, t) -separator for any $s \in L, t \in R$. We say a cut (L, S, R) is a *t -sink cut* if $t \in R$. Throughout this paper, we will focus on solving the following single-sink version of vertex connectivity.

Definition D.12 (SSVC (single-sink vertex connectivity)). Given an undirected graph $G = (V, E)$ and a vertex $t \in V$, outputs a minimum t -sink cut.

There is a simple randomized reduction from the general vertex connectivity problem to SSVC (see [Lemma D.89](#)), and fixing a sink makes the algorithm easier to state. So we focus on the single-sink version.

Another useful observation is that to solve SSVC, it suffices to solve the following Min-Neighbor problem. Given a vertex $u \in V$, define $N_G(u)$ (or $N(u)$ when the context is clear) to be the set of neighboring vertices of u , i.e., vertices that share an edge with u . Let $N[u] := N(u) \cup \{u\}$. For a vertex set $V' \subseteq V$, define $N(V') = (\cup_{u \in V'} N(u)) - V'$ and $N[V'] = N(V') \cup V'$.

Definition D.13 (Min-Neighbor). Given an undirected graph $G = (V, E)$ and a vertex set $V' \subseteq V$, find $L \subseteq V'$ such that $|N_G(L)|$ is minimized.

If we have an algorithm for Min-Neighbor, to solve SSVC, we set $V' = V - N[t]$ and call the Min-Neighbor algorithm to get $L \subseteq V'$ such that $|N(L)|$ is minimized. One can verify that $(L, N(L), V - L - N(L))$ is a valid t -sink cut for any $L \subseteq V'$. Moreover, every minimum t -sink cut (L, S, R) satisfies that $L \subseteq V'$ and $S = N(L)$ (otherwise we can set $S = N(L)$ which decrease the size of the cut).

We can also show the reversed reductions by simple arguments, so Min-Neighbor is, essentially, equivalent to vertex connectivity. Throughout this overview, we focus on solving Min-Neighbor.

Bottlenecks of [Li+21] beyond the sequential model. We first discuss the bottlenecks of implementing the sequential algorithm of [Li+21] in PRAM, CONGEST, and two-party communication models.

1. The algorithm is inherently sequential in the sense that it uses a breadth-first-search with unbounded depth as a subroutine. This makes it hard to be implemented in situations where parallelism is a requirement, such as PRAM and CONGEST.
2. Very informally, the algorithm reduces a vertex connectivity instance on a graph G with n vertices and m edges to a bipartite maximum matching instance on a graph G' with $\tilde{O}(m)$ edges and $\tilde{O}(n^2)$ vertices. In other words, when n is the parameter, the reduction is not efficient. In many situations, n is the parameter for measuring the complexity, such as CONGEST and communication, or in the matrix multiplication work of PRAM.

As mentioned in the introduction, we develop a new framework based on common-neighborhood clustering to sidestep these bottlenecks. The organization of this overview is as follows.

1. In [Section D.2.1](#), we define the common-neighborhood property of a graph, and show how to reduce Min-Neighbor to s-t vertex connectivity preserving the number of edges (but not vertices) in a *common-neighborhood cluster* (defined in [Section D.2.1](#)). The algorithm is described in PRAM model for the sake of simplicity.
2. In [Section D.2.2](#), we show our decomposition algorithm for common-neighborhood clustering in PRAM, which can also be implemented efficiently in various models like CONGEST and communication. Common-neighborhood clustering is a way to reduce solving Min-Neighbor on a general graph to solving Min-Neighbor on common-neighborhood clusters. Combined with [Section D.2.1](#), this proves [Theorem D.1](#) (not preserving the number of vertices is fine here because we assumed $W(m, n)$ is superadditive on m in [Theorem D.1](#)).
3. For the CONGEST model, to prove [Theorem D.2](#), we do not have the guarantee that $R(m, n, D)$ is superadditive. Thus, it suffers from the second bottleneck of [\[Li+21\]](#). We show that it can be resolved nearly perfectly in [Section D.2.3](#) and prove [Theorem D.2](#).
4. To prove that vertex-connectivity can be solved in matrix multiplication work in PRAM ([Theorem D.6](#)), only preserving the number of edges does not suffice. We show how to leverage our framework to circumvent this bottleneck in [Section D.2.4](#).
5. [Section D.2.5](#) gives an overview of our results in the two-party communication setting ([Theorem D.3](#)). We show that the number of vertices cannot be perfectly preserved by proving a near-optimal lower bound for vertex connectivity separating it from s-t vertex connectivity. The near-optimality of the lower bound is shown by an upper bound.

D.2.1 Building the Framework: Common-neighborhood Cluster

Recall that the input to Min-Neighbor is an undirected graph $G = (V, E)$ and a vertex set $V' \subseteq V$. Throughout this overview, we always use L^* to denote an arbitrary one of the minimizers of $\min_{L' \subseteq V'} |N(L')|$. Wlog., we will assume $G[L^*]$ is connected, otherwise we can take a connected component of $G[L^*]$ denoted as L' , we have $N(L') \subseteq N(L^*)$ so we can use L' as the minimizer instead.

Our goal is to find L^* , and let us assume that we know its size.⁹ Now we are ready to define common-neighborhood clusters. For two sets A, B , define $A \Delta B = (A - B) \cup (B - A)$ as their *symmetric difference*.

Definition D.14. Given an undirected graph $G = (V, E)$ and a vertex set $C \subseteq V$, we say C has *neighborhood difference* d if for any $u, v \in C$, $|N(u) \Delta N(v)| \leq d$.

We say C is a *common-neighborhood cluster* if C has neighborhood difference at most $\hat{O}(|L^*|)$. In this section, we will assume V' is a common-neighborhood cluster and show a PRAM algorithm solving Min-Neighbor. We will use the following property of a common-neighborhood cluster. The formal statement and proofs are in [Claim D.42](#).

⁹This assumption can be easily removed by guessing the size of L^* by powers of 2 which only adds a logarithmic factor to the complexity.

Lemma D.15 (Informal version of [Claim D.42](#)). *If the input vertex set V' is a common-neighborhood cluster, then one of the following two cases happens.*

1. $|N_G(L^*) \cap V'| = \hat{O}(|L^*|)$.
2. $G[V']$ is almost a clique. For this overview, assume V' is indeed a clique for convenience.

Now we show the rough idea of how to find L^* in each case. The formal proofs are in [Section D.5](#).

Case 1: $|N(L^*) \cap V'| = \hat{O}(|L^*|)$. In this case, we can employ the *isolating cuts* technique [[Li+21](#)], that, given an independent set T , finds the minimum vertex cut separating exactly one node in T from the other nodes in T . For our case, we let T be an independent sample by including each node $v \in V'$ in T with probability $1/\hat{O}(|L^*|)$. This means that with probability $\hat{\Omega}(1)$, T has exactly one node in L^* and no nodes in $N(L^*)$.

Case 2: $G[V']$ is a clique. In this case, we are trying to solve **Min-Neighbor** given that (i) V' is a clique, and (ii) every two nodes in V' has their neighbors in G differ by at most $\hat{O}(|L^*|)$ nodes. We call this problem **MinNeighbor-NearClique-CommonNeighborhood**, or **MinNNCC** for short.

For a vertex $s \in V'$, define the s -source **Min-Neighbor** problem as finding $\arg \min_{s \in L' \subseteq V'} N(L')$. Note that it can be solved in one s-t vertex connectivity call: add a super node t connecting to $N(V')$ and call s-t vertex connectivity. Given this, the algorithm for dealing with Case 2 contains two steps.

1. Sample $|V'|/|L^*|$ nodes uniformly at random so that one of them is in L^* .
2. For every sample node, solve s -source **Min-Neighbor** on a *sparsified graph* (which we will define later) with $|V'| \cdot |L^*|$ edges.

The cumulative number of edges for calling s-t vertex connectivity is $|V'|^2$, which is proportional to the number of edges in G (since V' is a clique). Thus, the reduction preserves the total number of edges. Moreover, if the work function $W(m, n)$ for solving s-t connectivity is superadditive on m , then the work of solving **MinNNCC** can be written as $O(W(|V'| \cdot |L^*|, n)) \cdot |V'|/|L^*| \leq O(W(|V'| \cdot |L^*| \cdot (|V'|/|L^*|), n)) = O(W(m, n))$.

The remaining question is how to run s -source **Min-Neighbor** on a *sparsified graph with $|V'| \cdot |L^*|$ edges*. This is because we can delete all edges from $V' - \{s\}$ to neighbors of s , and deleting them will not change the set $N(L')$ for any set $L' \ni s$. Recall that V' has neighborhood difference at most $\hat{O}(|L^*|)$, and hence $u \in V'$ can have at most $\hat{O}(|L^*|)$ remaining edges outside $N(s)$. See [Figure D.1](#) for an illustration.

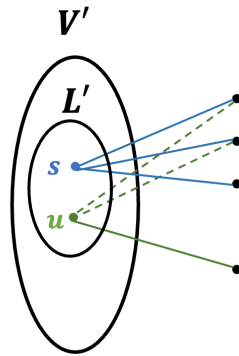


Figure D.1: All the dashed edges (edges from $V' - \{s\}$ to neighbors of s) are deleted. For every L' with $s \in L'$, $N(L')$ does not change.

Remark D.16. The proof of [Claim D.42](#) and the idea of sparsifying the instance can be viewed as an adaption from the proof in [[Li+21](#)], so we do not claim novelty here. Our novelty lies in (i) introducing the graph decomposition for common-neighborhood clustering, and showing how it can be implemented in

various models, (ii) circumvent the bottleneck of not preserving the number of vertices, as we will illustrate in the following sections.

Building the Framework. To summarize, once we have a common-neighborhood cluster V' , the problem reduces to one of (i) finding a minimum isolating cut, or (ii) solving MinNNCC. If V' is not a common-neighborhood cluster, the natural idea is to decompose it into common-neighborhood clusters, which we call *common-neighborhood clustering* defined as follows.

Definition D.17 (Common-neighborhood clustering). Given $G = (V, E)$ and $V' \subseteq V$, output a set of vertex sets $\mathcal{C}$ such that

1. each vertex $u \in V'$ is contained in at most $\hat{O}(1)$ clusters in $\mathcal{C}$,
2. for any $C \in \mathcal{C}$, C is a common-neighborhood cluster and $C \subseteq V'$,
3. $L^* \subseteq C$ for some $C \in \mathcal{C}$ with probability at least $1/n^{o(1)}$.

Our framework can be thus summarized as follows.

Step 1: Compute a *common-neighborhood clustering* denoted as $\mathcal{C}$.

Step 2: For every $C \in \mathcal{C}$, solve *minimum isolating cut* and MinNNCC to get non-empty $L_C \subseteq C$ minimizing $N(L_C)$.

Step 3: Output the minimum $N(L_C)$ among all $C \in \mathcal{C}$.

D.2.2 A Decomposition Algorithm for Common-neighborhood Clustering

In this section, we show a PRAM algorithm for common-neighborhood clustering which can be implemented in different models. One crucial property of L^* that we state here is that L^* has neighborhood difference at most $2|L^*|$ —we defer the formal proof of this statement to [Lemma D.26](#).

A proof of existence. We first give a simple, but inefficient, algorithm that shows the existence of common-neighborhood clustering. Define the neighborhood-difference graph $G' = (V', E')$ as follows: for any pair of vertices $(u, v) \in V' \times V'$, $(u, v) \in E'$ if and only if $|N_G(u) \Delta N_G(v)| \leq 2|L^*|$.

We also need the notion of the sparse neighborhood cover that is studied in the literature [\[Awe+98\]](#). We define it as follows.

Definition D.18 (Sparse neighborhood cover (SNC)). Given an undirected graph $G = (V, E)$, the *sparse neighborhood cover* of G is a set of vertex sets $\mathcal{C}$ (called clusters) such that

1. (sparse) each vertex $v \in V$ is contained in at most $\tilde{O}(1)$ clusters in $\mathcal{C}$,
2. (low diameter) the diameter of $G[C]$ for any $C \subseteq \mathcal{C}$ is at most $O(\log n)$,
3. (cover) for every $v \in V$, there exists a cluster $C \subseteq \mathcal{C}$ such that $N_G[v] \subseteq C$.

It is also proved in [\[Awe+98\]](#) that SNC can be constructed efficiently. We claim that applying SNC on G' gives us a common-neighborhood clustering. Let us verify each property of common-neighborhood clustering one by one.

1. Each vertex $u \in V'$ is contained in at most $\tilde{O}(1)$ clusters in $\mathcal{C}$ according to the (*sparse*) condition.
2. For any $C \in \mathcal{C}$ and any $u, v \in C$, there exists a path from u to v in G' with length $O(\log n)$ according to *low diameter*, denoted by $(u, v_1, v_2, \dots, v)$. Notice that an edge $(u, v_1) \in E'$ means we can delete and add $2|L^*|$ nodes from $N_G(u)$ to get $N_G(v_1)$. We can repeat this argument along the path, which shows that $|N_G(u) \Delta N_G(v)| = O(|L^*| \log n)$. This proves that C has neighborhood difference at most $O(|L^*| \log n)$, i.e., it is a common-neighborhood cluster.
3. L^* has neighborhood difference at most $2|L^*|$ according to [Lemma D.26](#), L^* must be a clique in G' according to the definition of G' (actually here we do not require $G[L^*]$ to be connected, this additional property is to assist a faster algorithm which we will explain later). Thus, take an arbitrary $v \in L^*$, and note $L^* \subseteq N_{G'}[v]$, which means L^* is contained in some cluster due to property (*cover*).

This completes the existential proof. However, construction G' is not efficient and runs in $O(n^3)$ time in a trivial way. $\square$

Speeding up the construction of G' : approximating the neighborhood differences. One reason for the slow computation of G' is that in order to decide if an edge (u, v) exists in G' or not, we need to go over the neighbors of both u and v , which in the worst case use time n , resulting in a total time $O(n^3)$. This is unavoidable if we want to know the exact size of $|N_G(u) \Delta N_G(v)|$. However, we will show that an approximate size of $|N_G(u) \Delta N_G(v)|$ suffices.

We can build G' in the following way: (i) if $(u, v) \in E'$, then $|N_G(u) \Delta N_G(v)| \leq 3|L^*|$, (ii) if $(u, v) \notin E'$, then $|N_G(u) \Delta N_G(v)| > 2|L^*|$. Such G' can be constructed by only knowing an approximate value of $|N_G(u) \Delta N_G(v)|$ for any $u, v \in V$. One can verify that applying SNC on this G' does not make too much difference and can still give us a common-neighborhood clustering. Moreover, computing an approximate value of $|N_G(u) \Delta N_G(v)|$ only requires $\tilde{O}(1)$ time by standard sampling and sketching techniques. This is proved in [Corollary D.30](#).

Assumption D.19. *In the rest of this overview, we use $\text{apd}(u, v)$ to denote a 1.1-approximation value of $|N_G(u) \Delta N_G(v)|$, which can be found using [Corollary D.30](#).*

This means that G' can be constructed in time $\tilde{O}(n^2)$. However, this is still slow if our goal is $\tilde{O}(m)$ work. More severely, computing G' is not a ‘localized’ procedure since it requires two very far away nodes to decide whether they have an edge in G' or not, posing a challenge for distributed computing. Thus, our algorithm will not compute G' , or a SNC of G' , but instead involves a completely new idea.

A fast algorithm with large depth. Now we describe an algorithm that has a large depth but in $\tilde{O}(m)$ work in PRAM. Given that L^* has neighborhood difference at most $2|L^*|$ and $G[L^*]$ is connected, if we know a vertex $u \in L^*$, we can construct a common-neighborhood cluster C with $L^* \subseteq C$ in the following way: build a BFS tree on G with root u , such that a vertex v is included in the BFS tree only if $\text{apd}(u, v) \leq 3|L^*|$. Let us denote the vertices in this BFS tree as $B'[3|L^*|](u)$. It can be equivalently defined as follows.

Definition D.20. Define $B'[\ell](u)$ as the largest possible set such that (i) $G[B'[\ell](u)]$ is connected, (ii) for any $v \in B'[\ell](u)$ we have $\text{apd}(u, v) \leq \ell$.

In this way, we can guarantee that the neighborhood difference of C is at most $O(|L^*|)$, $L^* \subseteq C$ and the construction time of C is proportional to the number of adjacent edges of C .

Forgetting what L^* is. In general, we do not know a vertex in L^* . A natural idea is to sample vertices with probability $1/|L^*|$ and run the above algorithm for each sampled node and add all the resulting clusters into $\mathcal{C}$. In this way, with good probability one of the clusters in $\mathcal{C}$ will contain L^* . However, it is easy to see that the (*sparse*) property does not hold, i.e., each vertex could appear in as many as $n/|L^*|$ clusters (this also results in a large work). The reason for the large overlap is that clusters can intersect with each other. A naive way to reduce the overlap would be to delete every cluster that intersects with others. Unfortunately, this does not work because the cluster that contains L^* could be deleted by this procedure. Hence, we set our goal as follows: sample some vertices u and build the cluster $B'[\hat{O}(|L^*|)](u)$, such that (i) at least one sampled vertex is in L^* , and (ii) the cluster containing L^* does not intersect other clusters. If these conditions are satisfied, we can let $\mathcal{C}$ be all the clusters that do not intersect with others, and we will be done. The work is also guaranteed to be $\tilde{O}(m)$ since if two BFS trees meet together at one node, they can stop growing and they are both excluded from being in $\mathcal{C}$.

To achieve the aforementioned non-overlap requirement (ii), an ideal situation would be the following. There are two vertex sets A, B with $L^* \subseteq A \subseteq B$ such that:

1. the size of B is roughly equal to the size of A ,
2. B has neighborhood difference at most $\hat{O}(|L^*|)$, and
3. Any vertex outside B has large neighborhood difference from the vertices in A (e.g., at least $10d_A$ where the neighborhood difference of A is d_A).

If we have such A, B , then we can sample with probability $1/|B|$ such that, with good probability, one of the sampled nodes will be in A while all other sampled nodes are outside B . Now we can build $B'[2d_A](u)$ for each sampled node u and A will be contained in one of the clusters, and the cluster containing A will not overlap with other clusters.

Now we discuss how to prove the existence of such A, B . For a vertex $v \in L^*$, $L^* \subseteq B'[3|L^*|](v)$. In what follows we assume v is an arbitrary vertex in L^* . If we can find a parameter $r > 2$ such that $|B'[99r|L^*|](v)| \leq n^{o(1)}|B'[r|L^*|](v)|$ holds, then we can set $A = B'[r|L^*|](v)$ and $B = B'[99r|L^*|](v)$ and all the three conditions for A and B hold. Our goal is to find the smallest such r starting from $r = 2$ and increasing r to $99r$ iteratively. Notice that each repetition increases the size of $B'[r|L^*|](v)$ by a factor of $n^{o(1)}$ but only increases r by a factor of 99, by setting the parameter correct we can make sure $r = \hat{O}(1)$ in the end. This proves the existence of A and B . As long as A, B exists, we can guess the size of B by powers of 2, so that we can sample with probability $1/|B|$ as the algorithm suggests.

A side remark is that the algorithm only guarantees $L^* \subseteq C \in \mathcal{C}$ with probability $1/n^{o(1)}$. This probability can be boosted by repeating the algorithm $n^{o(1)}$ and add all the found clusters to $\mathcal{C}$.

Reducing the depth by graph shrinking. Unfortunately, the above idea still requires doing BFS with unbounded depth, which is not suitable for both parallel and distributed models. The main bottleneck is about finding $B'[\ell](u)$ for many different u and some parameter ℓ . If $B'[\ell](u)$ contains less than $n^{o(1)}$ nodes, then it can be done in $n^{o(1)}$ depth since the BFS can include at most $n^{o(1)}$ nodes. Thus, let us assume $B'[\ell](u)$ contains much more than $n^{o(1)}$ nodes.

The idea is to shrink the graph size by a factor of $n^{o(1)}$ while not losing too much information about $B'[\ell](u)$. We sample vertices with probability $1/n^{o(1)}$ and denote the sampled vertex set as X . With good probability we have (i) $X \cap B'[\ell](u) \neq \emptyset$, (ii) size of X decrease the size of V by a factor of $1/n^{o(1)}$. We decompose the graph by growing vertex disjoint BFS trees simultaneously from every vertex $v \in X$, such that a vertex w is included in the BFS tree of v only if $\text{apd}(v, w) \leq 3\ell$. After that, we contract each BFS tree and call them *super nodes*. The graph size decreases by a factor of $1/n^{o(1)}$. Moreover, all vertices in $B'[\ell](u)$ are preserved in some super nodes since X contains at least one node $v \in B'[\ell](u)$ and all nodes in $B'[\ell](u)$ are eligible to join the BFS tree of v .

We can repeat the above operation as long as the number of super nodes intersecting $B'[\ell](u)$ is at least $n^{o(1)}$: sample super nodes with probability $1/n^{o(1)}$ to make sure one sampled super node intersects $B'[\ell](u)$; from each super node v grows a BFS tree only including another supernode w such that there exist original nodes $v' \in v, w' \in w$ with $\text{apd}(v, w) \leq 3\ell$; contract each BFS tree. Notice that the neighborhood difference of each super node grows by a constant factor in each iteration.

We repeat until the number of super nodes intersecting $B'[\ell](u)$ becomes $n^{o(1)}$, then we can find all of them by a simple BFS of depth $n^{o(1)}$. The only problem is that a super node might contain extra nodes besides nodes in $B'[\ell](u)$. This is fine due to the following reason: in each iteration the neighborhood difference of each super node grows by a constant factor, while the number of nodes of the graph shrink by a factor of $1/n^{o(1)}$; by setting the parameter correctly, the final neighborhood difference of each super node is at most $n^{o(1)}\ell$. Thus, all the nodes in the super nodes intersecting $B'[\ell](u)$ are contained in $B'[n^{o(1)}\ell](u)$. This $n^{o(1)}$ factor is not too large and a slight change in the algorithm can handle it.

The algorithm and analysis are described in details in [Section D.4](#).

D.2.3 Distributed Algorithms

Recall the framework for our CONGEST algorithms: we first apply common-neighborhood cluster described in [Section D.2.2](#) so that we can assume V' is a common-neighborhood cluster as defined in [Section D.2.1](#), then we use the observations in [Section D.2.1](#) to reduce the problem to (i) minimum isolating cut, and (ii) MinNNCC. Both problems can be reduced to s-t vertex connectivity while preserving the total number of edges m when doing the reduction, as explained in [Section D.2.1](#).

Now, we describe the key obstacles to implementing distributed CONGEST and outline our solution to overcome the technical obstacles.

Bottlenecks of Proving **Theorem D.2**: Minimum Isolating Cut

The following isolating cuts lemma can almost handle our case.

Lemma D.21 (Lemma 4.2 of [Li+21]). *Given a graph $G = (V, E)$ and an independent set $T \subseteq V$ of size at least 2, there is an algorithm that outputs for each $v \in T$ a $(v, T - \{v\})$ -min-separator C_v . The algorithm makes calls to s - t vertex mincut on graphs with $O(m)$ total number of vertices and $O(m)$ total number of edges and takes $\tilde{O}(m)$ additional time.*

This version of the isolating cuts lemma works perfectly fine for sequential settings as they focus on bounding the total number of edges. Their algorithm can also be easily parallelized. However, the number of vertices can be as large as $O(m)$ which is not suitable for distributed computing.

To handle this situation, we prove a more efficient version of the isolating cuts lemma that also guarantees $\tilde{O}(n)$ total number of vertices and $O(m)$ total number of edges.

Lemma D.22. *Given a graph $G = (V, E)$ and an independent set $T \subseteq V$ of size at least 2, there is an algorithm that outputs for each $v \in T$ a $(v, T - \{v\})$ -min-separator C_v . The algorithm makes calls to s - t vertex mincut on graphs with $\tilde{O}(n)$ total number of vertices and $O(m)$ total number of edges and takes $\tilde{O}(m)$ additional time.*

To elaborate for minimum isolating cut, by following the proof of **Lemma D.21** (formally described in Lemma 4.2 of [Li+21]), we remove a set of vertices X from the graph so that each connected components C of the remaining graph contains at most one terminal $t \in T$. For each connected component C with one terminal $t \in T$ inside, we try to find $\min_{s \in L' \subseteq C} N_G(L')$. Notice that this minimization problem only depends on all the edges adjacent to C , which are non-overlapping for different C , so the total number of edges is bounded by m . However, it depends on all vertices in $N_G[C]$, which could intersect a lot for different C .

To obtain the improvement as in **Lemma D.22**, we briefly explain how to deal with this issue. For every C where we wish to find $\min_{s \in L' \subseteq C} N_G(L')$, we prove that it suffices to keep $\tilde{O}(|C|)$ vertices in $N_G(C)$ and deleting the others, while preserving $\min_{s \in L' \subseteq C} N_G(L')$. This is done by finding a maximum bipartite matching between C and $N_G(C)$. After that, the total number of vertices is upper bounded by $\sum_C \tilde{O}(|C|) = \tilde{O}(n)$. Given **Lemma D.22**, we can easily obtain PRAM and CONGEST algorithms where the number of vertices involved in the isolating cuts lemma is $\tilde{O}(n)$. We refer to **Section D.6** for details.

Bottlenecks of Proving **Theorem D.2**: Solving MinNNCC

It is easy to see that the algorithm described in **Section D.2.1** for MinNNCC does not preserve the number of edges. In fact, it contains $|V'|/|L^*|$ many instances, where each instance contains $|V'| \cdot |L^*|$ edges and n vertices.

Now we briefly explain how this is solved in CONGEST. We will use the fact that V' is a clique. For each instance with $|V'|/|L^*|$, we map those edges uniformly at random into the clique $G[V']$, and we solve the instance by an oracle call to s - t vertex connectivity where communication happens only on the mapped edges. Since there are in total at most $\tilde{O}(|V'|^2)$ edges, and $G[V']$ is a clique, in expectation each edge is only included in $\tilde{O}(1)$ many instances.

Random mapping of the communication inside the clique. Here we elaborate on the random mapping procedure. Our solution is to offload the communication of each instance to the cliques by a simple random load-balancing strategy.

Given each instance G'_s , which is the sparsified graph with $\tilde{O}(|V'| \cdot |L^*|)$ edges as described in **Section D.2.1**, we define a random (public) function $f_s : N(V') \rightarrow V'$ that maps a node $v \in N(V')$ to a random node in V' . For each $u \in N(V')$, our goal is to have $f(u) \in V'$ simulate u (in other words, $f(u)$ acts as a proxy for u), i.e., $f(u)$ acts as if it is node u . In order for $f(u)$ to simulate u , $f(u)$ must learn the following information:

- The algorithm description of node u for executing s - t vertex mincut algorithm in G'_s . This can be done by having u send its code to $f(u)$.

- $f(u)$ learns every node $N_G(u) \cap V'$. This can be done by letting every neighbor $v \in N_G(u) \cap C$ send its id to $f(u)$.

In this case, $f(u)$ can act as if it is node u . So, any communication between $v \in V'$ to $u \in N(V')$ can be moved to the same communication between $v \in V'$ and $f(u) \in V'$. Therefore, the communication between every edge (v, u) where $v \in C, u \in N(V')$ will map to a $(v, f(u))$ where u is a random node. Using the fact of graph G'_s that every node $v \in C - \{s\}$ has at most $\hat{O}(|L^*|)$ edges to $N(V')$, we conclude that every node $v \in C - \{s\}$ in the instance G'_s communicates using random $\hat{O}(|L^*|)$ edges adjacent to V' .

In summary, fix a node $v \in V'$, and the algorithm will communicate from v to $\tilde{O}(|L^*|)$ random vertices in V' for each instance G'_s . Since every $\tilde{O}(\frac{|V'|}{|L^*|})$ instance will use $\tilde{O}(|L^*|)$ random neighbors from v to in V' , and v has degree $|V'| - 1$ inside V' , in expectation, the load of each edge in V' incident to v is $\tilde{O}(1)$. This means every edge e in V' will have $\tilde{O}(1)$ number of instances G'_s use e to communicate. We can now run all instances with $\tilde{O}(1)$ congestion. We present the full algorithm in [Section D.7](#).

Remark D.23. If the cluster V' is not a clique, but close enough to a clique, then we simulate a clique communication with $O(\log n)$ factor overhead in communication, before running the clique algorithm. We formalize the algorithm in [Section D.7.2](#).

D.2.4 Parallel Algorithm in Matrix Multiplication Work

Recall that in the last section, we explain how to solve minimum isolating cut and MinNNCC by reductions to s-t vertex connectivity while preserving both the number of edges and vertices, in CONGEST model. For the case of minimum isolating cut, one can easily check that the idea for preserving the number of vertices described in the last section also works in the PRAM model, and thus we can solve s-t vertex connectivity using a reduction to bipartite matching which can be solved in $O(n^\omega)$ work.

We now focus on solving the case of MinNNCC. We will use the same local graph $G_{V'}$. Recall that the algorithm that sparsifies $G_{V'}$ into G'_s can result in $\hat{O}(m) = O(n^2)$ vertices in total, which is not suitable for our purpose.

In this section, we explain how to solve MinNNCC in matrix multiplication time by utilizing *convex embedding* introduced in [\[LLW88\]](#). The details can be found at [Section D.8](#). Throughout this section, we let $n_0 = |N_G[V']| + 1$ be the number of vertices in $G_{V'}$. The goal is to solve MinNNCC in $\tilde{O}(n_0^\omega)$ work (and polylog depth). For simplicity, we assume that $|N(V')| \leq |V'|$. Later we show how to remove this assumption.

Let $\mathbb{F}$ be a finite field. For $k \geq 0$, the space $\mathbb{F}^k$ is a k -dimensional linear space over $\mathbb{F}$. Let $X = \{x_1, \dots, x_n\}$ be a finite set of points within $\mathbb{F}^k$. The *affine hull* of X is defined as

$$\text{aff}(X) = \left\{ \sum_{i=1}^k c_i x_i : x_i \in X \text{ and } \sum_{i=1}^k c_i = 1 \right\}.$$

The rank of X , represented as $\text{rank}(X)$, is one plus the dimension of $\text{aff}(X)$. Specifically, if $\mathbb{F} = \mathbb{R}$, we refer to the *convex hull* of X , denoted $\text{conv}(X)$. For any sets V, W , any function $f : V \rightarrow W$, and any subset $U \subseteq V$, we denote $f(U) := \{f(u) : u \in U\}$.

Definition D.24 (Convex X -embedding [\[LLW88\]](#)). For any $X \subset V$, a convex X -embedding of a graph $G = (V, E)$ is a function $f : V \rightarrow \mathbb{R}^{|X|-1}$ such that for each $v \in V \setminus X$, $f(v) \in \text{conv}(f(N_G(v)))$.

We defer the details of construction to [Section D.8](#). For now, let us assume that we are given $N_{G_{V'}}(t)$ -convex embedding f_t in graph $G_{V'}$ such that for all $x \in V'$, $\kappa_{G_{V'}}(s, t) = \text{rank}(f_t(N_{G_{V'}}(x)))$. The task now is to compute the rank of $f_t(N_{G_{V'}}(x))$, which can be done in $\tilde{O}(n_0^\omega)$ time. In total, it would take $\tilde{O}(\frac{|V'|}{|L^*|} \cdot n_0^\omega)$ work to compute $\kappa_{G_{V'}}(s, t)$ for all instances of s -source Min-Neighbor via this approach.

Here is the speedup we can exploit: Since V' has neighborhood difference at most $\hat{O}(|L^*|)$, and thus for any $s, s' \in V'$, $|N(s) \Delta N(s')| \leq \hat{O}(|L^*|)$. Therefore, $\text{rank}(f_t(N_{G_{V'}}(s')))$ can be obtained from *low-rank* updates of $\text{rank}(f_t(N_{G_{V'}}(s)))$ given we preprocess the matrix representing $f_t(N_{G_{V'}}(s))$. Here, low-rank updates correspond to changing $\hat{O}(|L^*|)$ columns of the matrix representing $f_t(N_{G_{V'}}(s))$ since

$$|N(s) \Delta N(s')| \leq \hat{O}(|L^*|).$$

In terms of low-rank updates, we can use a *dynamic matrix data structure* to support this operation (see [Lemma D.72](#) for the formal definition). Namely, we can preprocess the matrix M_s representing $f_t(N_{G_{V'}}(s))$ in $\tilde{O}(n_0^\omega)$ time so that given $s' \in V'$, we can decide if $\text{rank}(f_t(N_{G_{V'}}(s'))) \geq k$ in $T_{\text{MM}}(|L^*|, n_0, |L^*|)$ time where $T_{\text{MM}}(n, k, r)$ be the number of field operations needed to multiply an $n \times k$ matrix with a $k \times r$ matrix in $O(\log^2 n)$ depth.

To summarize, using the matrix data structure, the total work to compute $\kappa_{G_{V'}}(s, t)$ for $\tilde{O}(\frac{|V'|}{|L^*|})$ instances of s -source Min-Neighbor is at most

$$\tilde{O}(n_0^\omega) + \tilde{O}\left(\frac{|V'|}{|L^*|}\right) \cdot T_{\text{MM}}(|L^*|, n_0, |L^*|) \leq \tilde{O}(n_0^\omega) = \tilde{O}(|V'|^\omega).$$

D.2.5 Communication Complexity

When implementing our framework in the communication model, common-neighborhood clustering (as described in [Section D.2.3](#)) works well. The problem we face is solving MinNNCC: the best s-t vertex connectivity algorithm [\[Bli+22\]](#) takes n as the parameter, but in the worst case we need to run n sparse instances, each with n vertices. One might hope that there is a way to solve this issue with an almost-perfect reduction just like we did in CONGEST. However, we proved that this is not true by showing an $\Omega(n^{1.5})$ lower bound.

An $\Omega(n^{1.5})$ lower bound. We will reduce the following problem to vertex connectivity. Alice and Bob will get $\sqrt{n}$ instances of *subset-tribes*, each containing $\sqrt{n}$ instances of *subset* problems with length $\sqrt{n}$. The inputs to Alice and Bob are denoted as $A_1^{(i)}, A_2^{(i)}, \dots, A_{\sqrt{n}}^{(i)} \subseteq [\sqrt{n}]$ and $B_1^{(i)}, B_2^{(i)}, \dots, B_{\sqrt{n}}^{(i)} \subseteq [\sqrt{n}]$ from $i = 1$ to $i = \sqrt{n}$, where it is guaranteed that $|A_j^{(i)}| = \sqrt{n}/2$ for any i, j . They want to determine whether there exists i such that $B_j^{(i)} \subseteq A_j^{(i)}$ for any j . This problem has randomized communication lower bound $\Omega(n^{1.5})$ (see [Lemma D.85](#)). Now we show how to reduce this problem to vertex connectivity.

We let $G = (V_1 \cup V_2 \dots \cup V_{\sqrt{n}} \cup U, E)$ where V_i is a clique with $\sqrt{n}$ vertices, and U is a clique with n vertices. We connect V_i to U based on $A_1^{(i)}, \dots, A_{\sqrt{n}}^{(i)}$ and $B_1^{(i)}, \dots, B_{\sqrt{n}}^{(i)}$, such that $N(V_i)$ has size n iff $B_j^{(i)} \subseteq A_j^{(i)}$ for any j , while making sure that any other cut has size at least $n + 1$. The way of connecting V_i to U is as follows: split U into $\sqrt{n}$ pieces $U_1, \dots, U_{\sqrt{n}}$, and connect the j -th vertex in V_i (denoted by $v_j^{(i)}$) to U_j according to $B_j^{(i)}$; connect the j -th vertex in V_i to U_k for any $k \neq j$ according to $A_k^{(i)}$. See [Figure D.2](#) for an illustration. The details of this construction and proofs are in [Section D.9.2](#).

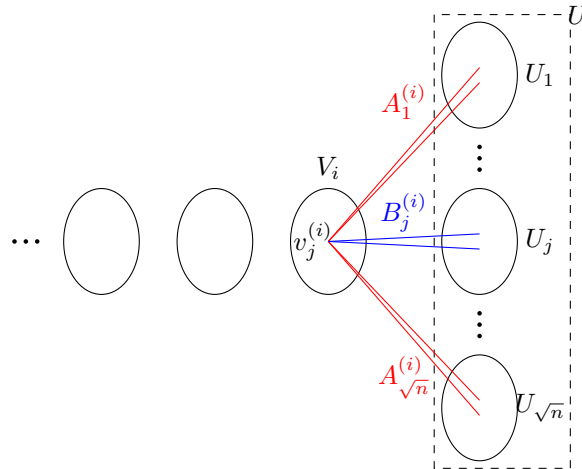


Figure D.2: Each ellipsoid (and the set U) represents a clique. Every node in V_i for $i = 1$ to $\sqrt{n}$ has edges to U as illustrated in the graph.

A simple nearly tight upper bound. Now we describe a simple way to solve Min-Neighbor when V' is a common-neighborhood cluster in $\tilde{O}(n^{1.5})$ communication. We break it into two cases based on the size of L^* .

When $|L^*| > n^{0.5}$, we can sample $\tilde{O}(\sqrt{n})$ nodes and one of them will hit L^* . Recall that if we know a node in L^* , we can find L^* in one s-t vertex connectivity call. Thus, in this case, the communication complexity is $\tilde{O}(n^{1.5})$.

If $|L^*| < n^{0.5}$, for a common-neighborhood cluster V' , we can recover all edges adjacent to V' by using $O(n^{1.5})$ communications in the following way: choose an arbitrary vertex $u \in V'$, Alice and Bob learn the set $N_G(u)$ by $O(n)$ communication, then for every $v \in V' - \{u\}$, Alice and Bob can learn $N_G(v)$ by $O(n^{0.5})$ communication since $|N(u) \triangle N(v)| \leq n^{0.5}$. Knowing all edges adjacent to V' suffices to solve MinNNCC.

Remark D.25. The algorithm for communication can be easily implemented in the streaming model. Moreover, there is an even simpler communication and streaming protocol in $\tilde{O}(n^{1.5})$ total communication that does not require common-neighborhood clustering. For completeness, we present it in [Section D.9](#).

D.3 Preliminaries

D.3.1 Terminologies

Basic graph terminologies. An (undirected) graph is denoted by $G = (V, E)$ where V is the vertex (or node) set and $E \subseteq \{\{x, y\} \mid x, y \in V, x \neq y\}$ is the edge set¹⁰. Following the convention, we abuse the notations a bit and will also use (x, y) to denote an edge, which is a set $\{x, y\}$. x, y are the *endpoints* of the edge (x, y) . $N_G(u)$ is the set of neighbors of u defined by $\{v \in V \mid \{u, v\} \in E\}$. We omit G in the subscript if G is clear from the context. We also define $N[u] = N(u) \cup \{u\}$. The *degree* of a vertex u is defined as $|N(u)|$. The *minimum degree* of G is defined as $\min_{v \in V} |N(v)|$. Define $E(A, B) = \{\{x, y\} \in E \mid x \in A, y \in B\}$.

For a vertex set V' , $G[V']$ is called the induced subgraph of G on V' defined by $G[V'] = (V', \{\{u, v\} \in E \mid u, v \in V'\})$.

Throughout the paper, we use n to denote $|V|$ and m to denote $|E|$. We write $\tilde{O}(f) = O(f \cdot \log^c n)$ for some constant c , and $\hat{O}(f) = f \cdot n^{o(1)}$. It is important that n represents $|V|$ here, which is roughly the input size.

Set terminologies. We define $[z] = \{1, 2, \dots, z\}$. For two sets A, B , we use $A - B$ or $A \setminus B$ to denote $\{a \in A \mid a \notin B\}$. We define $A \triangle B$ as the symmetric difference $(A - B) \cup (B - A)$. We say $(A_1, \dots, A_z)$ is a partition of A if $\cup_{i \in [z]} A_i = A$ and $A_i \cap A_j = \emptyset$ for any $i \neq j$.

Vertex cut. A *vertex cut* (or simply *cut* in this paper) is a partition of V denoted by (L, S, R) where (1) $L \neq \emptyset, R \neq \emptyset, E(L, R) = \emptyset$ or (2) $|S| \geq n - 1$. The size of a vertex cut (L, S, R) is defined as $|S|$. The *minimum vertex cut* refers to the vertex cut with the smallest size. For a vertex $s \in V$, The *minimum t -sink vertex cut* is defined as a vertex cut (L, S, R) with $t \in R$ which minimizes $|S|$.

We say a vertex cut (L, S, R) as a (u, v) -vertex cut if $u \in L, v \in R$. For two vertex sets $A, B \subseteq V$, we say (L, S, R) is an (A, B) -vertex cut if $A \subseteq L, B \subseteq R$. In this case, we say S is a (u, v) -separator or (A, B) -separator. S is called a *separator* (or a *vertex cut*) if S is a (u, v) -separator for some $u, v \in V$. We use $\kappa_G(u, v)$ to denote the size of the smallest u, v -separator in G , and $\kappa(G)$ to denote the smallest separator in G . When there are no (u, v) -separator or no separator in G , we define $\kappa_G(u, v)$ or $\kappa(G)$ as $n - 1$. In this case, we say any $n - 1$ nodes is a separator.

For a vertex set $X \subseteq V$, we say X has *neighborhood difference* d if for any $u, v \in X$ we have $|N_G(u) \triangle N_G(v)| \leq d$. The following lemma is crucial to our algorithm.

Lemma D.26. *Suppose (L, S, R) is a minimum vertex cut, then the neighborhood difference of L is at most $2|L|$.*

¹⁰All graphs in this paper are simple, i.e., without multi-edges and self-loops, as written in the definition.

Proof. Since S is the minimum vertex cut, we have $|N_G(u)| \geq |S|$ for any $u \in V$, otherwise $N_G(u)$ is a smaller vertex cut. Moreover, for $u, v \in L$, $N_G(u) \cup N_G(v) \subseteq L \cup S$, otherwise S is not a vertex cut. Thus, $|N_G(u) \triangle N_G(v)| = 2|N_G(u) \cup N_G(v)| - |N_G(u)| - |N_G(v)| \leq 2(|S| + |L|) - 2|S| = 2|L|$. $\square$

D.3.2 Computational Models

PRAM model. In PRAM model, we have a set of processors and a shared memory space. Each processor can independently read and write on the shared memory space or do other unit operations. The input is given initially on the shared memory space, and the processors are required to jointly compute a specific problem given the input. The complexity is measured by *work* and *depth*, where work is measured by the **total** amount of unit operations performed by all the processes, and depth is measured by the time consumed before the output is generated.

CONGEST model. In the distributed CONGEST model, we have an initial graph $G = (V, E)$ of n nodes with diameter D where each node has a unique ID and unlimited computational power. Initially, each node only knows its neighbors. The algorithm runs in synchronous rounds. For each round, each node can exchange $O(\log n)$ bits of information to its neighbors. The goal is to design an algorithm that minimizes the number of rounds needed to determine the output, for example, the vertex connectivity of G .

Two-party communication model. In the two-party communication model (or simply communication model), the edge set E is arbitrarily partitioned into two sets $E = E_1 \cup E_2$ where $E_1 \cap E_2 = \emptyset$. Two players Alice and Bob both know the vertex set V , and Alice knows the set E_1 , and Bob knows the set E_2 . Their goal is to exchange as few bits as possible to find some intrinsic property of the graph $G = (V, E)$.

Graph streaming model. In the graph streaming model (or simply streaming model), a graph $G = (V, E)$ is given to the algorithm as an arbitrarily ordered stream of edges. The algorithm has limited space and can read this stream in sequential passes. The goal is to minimize both the space usage and the number of passes. *Semi-streaming* refers to the case when the space is restricted to $\tilde{O}(n)$.

Remark D.27 (public and private randomness). In the two-party communication model and CONGEST model, *public randomness* refers to the case when every party (or node) can access the same random bits, *private randomness*, in contrast, does not allow parties (or nodes) to share the randomness. According to Newman's Theorem [New91], to simulate public randomness using private randomness, parties only need to share an additional $O(\log n)$ bits of message (which can be broadcasted in CONGEST model). Thus, in this paper, we assume access to public randomness when devising algorithms in communication or CONGEST model.

D.3.3 Problem Definitions

We define the problems considered in this paper as follows. We say a (randomized) algorithm solves a problem if it outputs the correct answer with high probability (or simply w.h.p.), i.e., with probability at least $1 - \frac{1}{n^c}$ for an arbitrarily large constant c . All algorithms in this paper are randomized except with explicit clarification.

Vertex connectivity (VC). Given an undirected graph $G = (V, E)$, output a minimum vertex cut.

Single sink undirected vertex connectivity (SSVC). Given a undirected graph $G = (V, E)$ and a sink vertex $t \in V$, output a minimum t -sink vertex cut.

s-t vertex connectivity (s-t VC). Given an undirected graph $G = (V, E)$ and $s, t \in V$, output a minimum (s, t) -vertex cut.

S,T-vertex connectivity (S-T VC). Given an undirected graph $G = (V, E)$, and two vertex sets $A, B \subseteq V$, compute the minimum (A, B) -vertex cut.

Subgraph problems in CONGEST. For subgraph S-T vertex connectivity (or other subgraph problems) in CONGEST, we are given a network along with a subgraph H where every node knows its adjacent edges in H , and we want to solve the problem on H while other edges of G are only used for communication purposes.

Bipartite maximum matching (BMM). Given a bipartite graph (A, B, E) (which is defined as $E \subseteq A \times B$ where $A \cup B$ is the vertex set), output the maximum matching (a matching is defined as a set of edges with mutually disjoint endpoints).

Bipartite minimum vertex cover (BMC). Given a bipartite graph (A, B, E) , output the minimum vertex cover (a vertex cover is defined as a set of vertices such that every edge has at least one endpoint in this set).

D.3.4 Sketching

We will use the standard linear sketching algorithms from [CF14].

Theorem D.28 (Section 3 in [CF14]). *For any numbers n and s , there is an algorithm that preprocesses in $\tilde{O}(s)$ work and $\tilde{O}(1)$ depth and then, given any vector $v \in \{-1, 0, 1\}^n$, return a sketch $sk_s(v) \in \mathbb{Z}^{\tilde{O}(s)}$ in $\tilde{O}(\|v\|_2^2)$ ¹¹ work and $\tilde{O}(1)$ depth and guarantees the following w.h.p. (as long as the number of recovery operations is $\text{poly}(n)$).*

- If $\|v\|_2^2 \leq s$, then we can recover v from $sk_s(v)$ in $\tilde{O}(s)$ time and $\tilde{O}(1)$ depth. (More specifically, we obtain all non-zero entries of v together with their indices).
- Otherwise, if $\|v\|_2^2 > s$, then the algorithm returns $\perp$.

Moreover, the sketch is linear, i.e. $sk_s(u + v) = sk_s(v) + sk_s(u)$ for any $u, v \in \mathbb{Z}^n$.

Remark D.29. The algorithm in [CF14] can be naturally implemented in the streaming model, in the sense that v is given in a stream, where each time a non-zero entry of v is given, and the algorithm needs to compute the sketching using $\tilde{O}(s)$ space in total after all entries of v is revealed. An easy way to see this fact is from the linearity of sketching: we can view v as a summation of vectors where each vector only contains one non-zero entry, and once a non-zero entry is revealed, we calculate the corresponding sketching for this vector and add it to the final sketching. This observation also implies the parallel complexity, where we can compute the sketching for each entry separately, and in parallel add them together.

The following corollary is from Theorem D.28 combined with sampling.

Corollary D.30. *For any number n , there is an algorithm that preprocesses in $\tilde{O}(n)$ time and then, given any vector $v \in \{-1, 0, 1\}^n$, return a sketch $sk_{\approx}(v) \in \mathbb{R}^{\tilde{O}(1)}$ in $\tilde{O}(\|v\|_2^2)$ time such that whp. (as long as the number of recovery operations is $\text{poly}(n)$) we can recover an approximate size of $\|v\|_2^2$ (a value between $0.9\|v\|_2^2$ and $1.1\|v\|_2^2$) in $\tilde{O}(1)$ work. Moreover, the sketch is linear, i.e. $sk_{\approx}(u + v) = sk_{\approx}(v) + sk_{\approx}(u)$ for any $u, v \in \mathbb{R}^n$.*

Proof. For any $i \in [\log n]$, we sample 2^i coordinates denoted by I_i . Given any vector $v \in \mathbb{R}^n$, we write v_{I_i} as the vector restricted to I_i . For every i we use Theorem D.28 with $n = |I_i|$, $s = \log^2 n$ to compute $sk_s(v_{I_i})$. The $sk_s(v_{I_i})$ for all i comprises our $sk_{\approx}(v)$.

Given $sk_{\approx}(v)$, we look at the first i such that when recovering from $sk_s(v_{I_i})$ the algorithm returns a value larger than $100 \log n$ and not $\perp$. Such an i must exist since we set $s = \log^2 n$: when i is large enough so that the expected values of I_i hitting a non-zero entry of v is roughly $500 \log n$, according to the Chernoff bound, it will return a value larger than $100 \log n$. Moreover, this value provides a good approximation of $\|v\|_2^2$ according to Chernoff bound. $\square$

¹¹Notice that $\|v\|_2^2$ is basically the number of non-zero entries of v .

When using [Corollary D.30](#) and [Theorem D.28](#), we sometimes will write $sk_s(S)$ where $S \subseteq V$ is a vertex set instead of a vector in $\{0, 1\}^n$. In this case, we consider S as a vector in $\{0, 1\}^n$ by treating each coordinate of the vector as a vertex in V , and each coordinate is 1 if and only if the corresponding vertex is in S .

D.4 Parallel and Distributed Common-Neighborhood Clustering

We first give some definitions of terminologies used in the algorithm. The algorithm will compute $sk_{\approx}(N_G(u))$ for any $u \in V$ in the beginning. During the algorithm, we say two nodes $u, v \in V$ are d -close if the approximate size recovered from $sk_{\approx}(N_G(u)) - sk_{\approx}(N_G(v))$ (see [Corollary D.30](#)) is at most d . We say Y is adjacent to X if $Y \cap X = \emptyset$ and there is an edge connecting a vertex in Y and a vertex in X .

Algorithm 1: $\mathcal{C} \leftarrow \text{COMNEICLUSTERING}(G, \ell)$

Data: An undirected graph $G = (V, E)$, an integer ℓ .

Result: A set of clusters $\mathcal{C} \subseteq 2^V$ satisfying [Lemma D.31](#)

```

1  $\mathcal{C} \leftarrow \emptyset$ ;
2 Compute sketchings  $sk_{\approx}(N_G(u))$  for any  $u \in V$ ;
3 foreach  $i = 0, 1, \dots, \log^{0.1} n, j = 0, 1, \dots, \log n$  do
4   Let  $d = 2^{i\sqrt{\log n}\ell}, s \leftarrow 2^j$ ;
5    $\mathcal{P} \leftarrow \{\{u\} \mid u \in V\}$ ;
6   Mark each vertex  $u \in V$  as center independently at random with probability  $1/s$ ;
7   foreach  $t_1 = 0, 1, \dots, \log^{0.3} n$  do
8      $\mathcal{P}' \leftarrow \mathcal{P}$ ;
9     Assign each center  $u \in V$  a vertex set  $S_u = C \in \mathcal{P}'$  where  $u \in C$ , delete  $C$  from  $\mathcal{P}'$ ;
10    foreach  $t_2 = 0, 1, \dots, 2^{\log^{0.9} n}$  do
11      foreach Center  $u \in V$  and every  $C \in \mathcal{P}'$  that is adjacent to  $S_u$  do
12        [ Pick an arbitrary  $v \in C$ , if  $u, v$  are  $9^{t_1}d$ -close, let  $S_u \leftarrow S_u \cup C, \mathcal{P}' \leftarrow \mathcal{P}' - C$ ;
13      Add  $S_u$  to  $\mathcal{C}$  for any center  $u \in V$ ;
14      Mark each vertex set in  $\mathcal{P}$  as center with probability  $1/2^{\log^{0.8} n}$ ;
15      Assign each center  $C \in \mathcal{P}$  a vertex set  $S_C = C$  and delete  $C$  from  $\mathcal{P}$ ;
16      For each center  $C \in \mathcal{P}$ , pick an arbitrary node  $c \in C$  denoted by  $u_C$ ;
17      foreach  $t_2 = 0, 1, \dots, 2^{\log^{0.9} n}$  do
18        foreach Center  $C \in V$  and every  $X \in \mathcal{P}$  that is adjacent to  $S_C$  do
19          [ Pick an arbitrary  $v \in X$ , if  $u_C, v$  are  $9^{t_2}d$ -close,  $S_C \leftarrow S_C \cup X, \mathcal{P} \leftarrow \mathcal{P} - X$ ;
20      Set  $\mathcal{P} \leftarrow \{S_C \mid C \text{ is a center}\}$ ;
21 return  $\mathcal{C}$ 

```

Lemma D.31 (common neighborhood clustering). *There exists a randomized algorithm ([Algorithm 1](#)) that given an undirected graph G and an integer ℓ , return $\mathcal{C}$ satisfying the following properties*

1. (*sparse*) for every $v \in V$, v is included in at most $\tilde{O}(1)$ clusters in $\mathcal{C}$,
2. (*common neighborhood*) any $C \in \mathcal{C}$ has neighborhood difference $\ell \cdot 2^{\log^{0.7} n}$ with high probability,
3. (*cover*) for every $L \subseteq V$ such that $G[L]$ is connected and L has neighborhood difference ℓ , we have $\Pr[\exists C \in \mathcal{C}, L \subseteq C] \geq \frac{1}{n^{o(1)}}$.

The algorithm can be implemented in $\tilde{O}(m)$ work and $\hat{O}(1)$ depth in PRAM model, or $\hat{O}(\sqrt{n} + D)$ rounds in CONGEST model.

Proof of [Lemma D.31](#) correctness. (**Sparse**) we need to following claim.

Claim D.32. *Throughout the algorithm, $\mathcal{P}$ only contains disjoint vertex sets.*

Proof. $\mathcal{P}$ is generated in [Algorithm 1](#), which contains S_C for each center C , where S_C is updated in [Algorithm 1](#). Notice that if X joins S_C , then it is deleted from the original $\mathcal{P}$ and cannot join other S_C , thus, as long as the original $\mathcal{P}$ contains vertex disjoint sets, the new $\mathcal{P}$ also contains vertex disjoint sets. Initially $\mathcal{P}$ contains all singleton vertices (see [Algorithm 1](#)), so the claim holds. $\square$

$\mathcal{C}$ is only updated in [Algorithm 1](#), where S_u is generated in [Algorithm 1](#). If C is included in S_u then it is deleted from $\mathcal{P}'$. So S_u are disjoint vertex sets for centers $u \in V$ as long as $\mathcal{P}'$ contains vertex disjoint sets, which is true from [Claim D.32](#). Therefore, every execution of [Algorithm 1](#) contributes at most 1 cluster for each node $v \in V$. There are $\tilde{O}(1)$ loops of [Algorithm 1](#), which proves the (sparse) property.

(Common neighborhood) We need the following lemma which bounds the neighborhood difference of every set in $\mathcal{P}$.

Lemma D.33. *At line [Algorithm 1](#), it always holds that for any $C \in \mathcal{P}$, C has neighborhood difference $9^{t_1}d/3$ w.h.p.*

Proof. We will prove it by induction. When $t_2 = 0$, $\mathcal{P} = \{\{u\} \mid u \in V\}$ and the lemma trivially holds. $\mathcal{P}$ will be changed at [Algorithm 1](#) to the next iteration, thus, we only need to prove that for any center C , S_C has neighborhood difference $9^{t_1+1}d/3$, given that the neighborhood difference of $\mathcal{P}$ is bounded by $9^{t_1}d/3$.

Initially in [Algorithm 1](#) we have the neighborhood difference of S_C being at most $9^{t_1}d/3$. Notice that in [Algorithm 1](#), we only add vertices in X to S_C if there is $v \in X$ and $u_C \in C$ that are $9^{t_1}d$ -close. In other words, for every $u, v \in S_C$, there exists u_1 (in the same X as u), and v_1 (in the same X as v) such that both u, u_1 and v_1, v has neighborhood symmetric difference at most $9^{t_1}d/3$ by induction hypothesis, and u_1, u_C and v_1, u_C has symmetric difference at most $1.1 \cdot 9^{t_1}d$ w.h.p. according to the definition of $9^{t_1}d$ -close. Thus, we have

$$|N(u) \Delta N(v)| \leq 2 \cdot (9^{t_1}d/3) + 2 \cdot 1.1 \cdot 9^{t_1}d \leq 9^{t_1+1}d/3$$

This completes the induction step. $\square$

Notice that $\mathcal{C}$ is only updated in [Algorithm 1](#) where S_u is updated in [Algorithm 1](#), which initially contains a vertex set in $\mathcal{P}$ and later will only includes vertex sets C in $\mathcal{P}'$ (also in $\mathcal{P}$) such that there exists $v \in C$ which is $9^{t_1}d$ -close to u . Therefore, for any $u, v \in S_u$, there exists u_1 (in the same C as u), v_1 (in the same C as v) such that both u, u_1 , v_1, v has neighborhood symmetric difference at most $9^{t_1}d/3$ by [Lemma D.33](#), and u_1, u and v_1, u has neighborhood symmetric difference at most $1.1 \cdot 9^{t_1}d$ w.h.p. according to the definition of $9^{t_1}d$ -close. Thus, we have

$$|N(u) \Delta N(v)| \leq 2 \cdot (9^{t_1}d/3) + 2 \cdot 1.1 \cdot 9^{t_1}d \leq 2^{O(\log^{0.1} n)} \cdot d = 2^{\log^{0.7} n} \ell$$

This completes the proof of property (common neighborhood).

(cover) let $L \subseteq V$ be a set such that $G[L]$ is connected and has neighborhood difference ℓ . We will prove that there exists $C \in \mathcal{C}$ such that $L \subseteq C$ with probability at least $1/n^{o(1)}$. For an integer d , define $B'_d(L)$ as the largest L' such that (i) for any $u \in L'$, there exists a $v \in L$ such that $|N_G(u) \Delta N_G(v)| \leq d$, (ii) $G[L']$ is connected. In other words, $B'_d(L)$ can be found by doing BFS from L while only including vertices that has neighborhood symmetric difference at most d . Let i be the smallest natural number such that $2^{\log^{0.9} n} \cdot |B'[2^i \sqrt{\log n} \ell/4](L)| \geq |B'[2^{(i+1)} \sqrt{\log n} \ell/4](L)|$. Notice that if the inequality does not hold, then $|B'[2^{(i+1)} \sqrt{\log n} \ell/4](L)|$ is $2^{\log^{0.9} n}$ multiplicative larger than $|B'[2^i \sqrt{\log n} \ell/4](L)|$. Since the total number of nodes is n , i cannot be larger than $\log^{0.1} n$. Thus, in [Algorithm 1](#), there exists a loop such that

1. i satisfies $2^{\log^{0.9} n} \cdot |B'[2^i \sqrt{\log n} \ell/4](L)| \geq |B'[2^{(i+1)} \sqrt{\log n} \ell/4](L)|$, notice that $B'[2^i \sqrt{\log n} \ell/4](L)$ has neighborhood difference at most d ,
2. s is roughly equal to the size of $B'[2^{(i+1)} \sqrt{\log n} \ell/4](L)$ (with multiplicative error 2), this is because we set $s = 2^j$ where j is ranging from 0 to $\log n$.

We will prove that in this loop, we will add a C to $\mathcal{C}$ such that $L \subseteq C$ with probability at least $1/n^{o(1)}$. Let us write $A = B'[2^i \sqrt{\log n} \ell/4](L)$ and $B = B'[2^{(i+1)} \sqrt{\log n} \ell/4](L)$. It is clear that $|B| \leq s \leq 2|B|$ and $2^{\log^{0.9} n} \cdot |A| \geq |B|$.

According to the two properties, at [Algorithm 1](#), the following two events happen simultaneously with probability at least $1/n^{o(1)}$ (i) there is exactly one center inside $B'[2^i\sqrt{\log n}\ell/4](L)$, (ii) every node in $B'[2^{(i+1)}\sqrt{\log n}\ell/4](L) - B'[2^i\sqrt{\log n}\ell/4](L)$ is not a center. In what follows we will assume the two events.

We first show that if $\mathcal{P}$ do not split A too much and covers A , then we can add a C with $L \subseteq C$ at [Algorithm 1](#).

Lemma D.34. *If $|\{C \in \mathcal{P} \mid C \cap A \neq \emptyset\}| \leq 2^{\log^{0.9} n}$ and $A \subseteq \cup_{C \in \mathcal{P}} C$, let u be the only center inside A , then $A \subseteq S_u$ at [Algorithm 1](#).*

Proof. Let $C \in \mathcal{P}$ be a set with $C \cap A \neq \emptyset$. We first show that C can never be included in S_v for $v \neq u$. If on the contrary there is a center $v \neq u$ such that C joins S_v at [Algorithm 1](#), we will prove a contradiction. We first prove that for any $w \in S_v$, w and v are $9^{t_1}d/3 + 9^{t_1}d$ close. This is because $w \in C' \in \mathcal{P}$ for some C' which has neighborhood difference $9^{t_1}d/3$ according to [Lemma D.33](#), and there exists $w' \in C'$ which is $9^{t_1}d$ -close to v according to [Algorithm 1](#). Since $C \cap A \neq \emptyset$, let $a \in C \cap A$ be an arbitrary vertex. Now we have every vertex in S_v is $2(9^{t_1}d/3 + 9^{t_1}d)$ -close to a . Moreover, $G[S_v]$ is connected because at [Algorithm 1](#) S_v only includes connected subgraphs that is adjacent to it. Remember the definition $A = B'[2^i\sqrt{\log n}\ell/4](L)$, which implies that any node in S_v is $\left(2(9^{t_1}d/3 + 9^{t_1}d) + 2^i\sqrt{\log n}\ell/4\right)$ -close to a vertex in L . Notice that $t_1 \leq \log^{0.3} n$ and $d = 2^i\sqrt{\log n}\ell/4$, so $\left(2(9^{t_1}d/3 + 9^{t_1}d) + 2^i\sqrt{\log n}\ell/4\right) \leq 2^{(i+1)}\sqrt{\log n}\ell/4$, which contradicts the fact that $v \notin B = B'[2^{(i+1)}\sqrt{\log n}\ell/4](L)$.

Now we show that C can be added to S_u at [Algorithm 1](#) as long as C is adjacent to the current S_u . Let $v \in C$ be an arbitrary vertex, let $a \in C \cap A$, according to [Lemma D.33](#), a and v are $9^{t_1}d/3$ -close. According to the definition of A , a and u are $\left(2 \cdot 2^i\sqrt{\log n}\ell/4 + \ell\right)$ -close. Thus v and u are $9^{t_1}d$ -close.

Since we have $A \subseteq \cup_{C \in \mathcal{P}} C$ and $G[A]$ is connected, every time [Algorithm 1](#) is executed, if $A \not\subseteq S_u$ yet, since $u \in S_u$, S_u must be adjacent to at least one $C \in \mathcal{P}$ with $C \cap A \neq \emptyset$, and it is added to S_u . After $2^{\log^{0.9} n}$ rounds, we have $A \subseteq S_u$. $\square$

It remains to prove that at some point we will have $|\{C \in \mathcal{P} \mid C \cap A \neq \emptyset\}| \leq 2^{\log^{0.9} n}$ and $A \subseteq \cup_{C \in \mathcal{P}} C$. Let us inspect how $\mathcal{P}$ changes.

Initially $\mathcal{P} = \{\{u\} \mid u \in V\}$ and it holds that $A \subseteq \cup_{C \in \mathcal{P}} C$. If $|A| \leq 2^{\log^{0.9} n}$ then we are done. Otherwise, we can use the following lemma.

Lemma D.35. *If $|\{C \in \mathcal{P} \mid C \cap A \neq \emptyset\}| > 2^{\log^{0.9} n}$ and $A \subseteq \cup_{C \in \mathcal{P}} C$, then at [Algorithm 1](#), we have $A \subseteq \cup_{C \in \mathcal{P}} C$ w.h.p.*

Proof. For any $C \in \mathcal{P}$ with $C \cap A \neq \emptyset$, we will prove that there exists a center $C' \in \mathcal{P}$ such that C is added to $S_{C'}$ at [Algorithm 1](#). Combined with the fact that $A \subseteq \cup_{C \in \mathcal{P}} C$ initially, the lemma is proved.

Let $(C_0, C_1, \dots, C_{2^{\log^{0.9} n}})$ be a sequence of different vertex sets in $\mathcal{P}$ such that $C_0 = C$ and for any $i > 0$, C_i is adjacent to C_{i-1} and $C_i \cap A \neq \emptyset$. Since $|\{C \in \mathcal{P} \mid C \cap A \neq \emptyset\}| > 2^{\log^{0.9} n}$, such a sequence must exist. At [Algorithm 1](#), we mark each element in $\{C \in \mathcal{P} \mid C \cap A \neq \emptyset\}$ as center independently with probability $1/2^{\log^{0.8} n}$, thus, w.h.p. there exists i such that C_i is marked as center. Let I be the smallest index such that C_I is inside some $S_{C'}$ for some center C' . We will prove that I decreases by at least 1 at each iteration of [Algorithm 1](#) in the next paragraph. If this is true, I will be decreased to 0 in the end and the lemma is proven.

Since C_{I-1} is adjacent to C_I , suppose $C_I \subseteq S_{C'}$ for some center C' , we only need to prove that when we pick an arbitrary $v \in C_{I-1}$, $u_{C'}$ (which is an arbitrary vertex in C') and v are $9^{t_1}d$ -close. Notice that both C_{I-1} and C' intersect A by at least one vertex, let the vertices be a, b separately. v, a are $9^{t_1}d/3$ -close, $b, u_{C'}$ are $9^{t_1}d/3$ -close according to [Lemma D.33](#); a, b are d -close according to the definition of A . Thus, as long as $t_1 > 0$, $u_{C'}$ and v are $9^{t_1}d$ -close. For the case of $t_1 = 0$, every vertex set of $\mathcal{P}$ only contain 1 vertex which means v, a and $b, u_{C'}$ are identical, so $v, u_{C'}$ are d -close. Thus, C_{I-1} will be added to some $S_{C'}$, which means I is decreased by at least 1. $\square$

We also need the following lemma which makes sure $|\{C \in \mathcal{P} \mid C \cap A \neq \emptyset\}| > 2^{\log^{0.9} n}$ will not always stay large.

Claim D.36. *At the last loop of Algorithm 1, $\mathcal{P}$ becomes empty w.h.p.*

Proof. In Algorithm 1, each element becomes a center with probability $1/2^{\log^{0.8} n}$, which is the only way that element can contribute to one element to $\mathcal{P}$ in the next loop according to Algorithm 1. Thus, we can think of it as each element “surviving” at each loop with probability $1/2^{\log^{0.8} n}$, and after $\log^{0.3} n$ loops it vanishes w.h.p. $\square$

Now start with a $\mathcal{P}$ with $A \subseteq \cup_{C \in \mathcal{P}} C$, either we already have $|\{C \in \mathcal{P} \mid C \cap A \neq \emptyset\}| \leq 2^{\log^{0.9} n}$ which according to Lemma D.34 we are done; or we have $|\{C \in \mathcal{P} \mid C \cap A \neq \emptyset\}| > 2^{\log^{0.9} n}$, which according to Lemma D.35, we still have $A \subseteq \cup_{C \in \mathcal{P}} C$ in the next iteration and we can repeat this argument. According to Claim D.36, there must exist a point when this argument ends, which gives us $|\{C \in \mathcal{P} \mid C \cap A \neq \emptyset\}| \leq 2^{\log^{0.9} n}$ and $A \subseteq \cup_{C \in \mathcal{P}} C$. $\square$

Now we talk about the parallel and distributed implementation of Algorithm 1. Most of the steps can be trivially implemented.

Lemma D.37. *Algorithm 1 can be implemented in PRAM in $\tilde{O}(m)$ work and $\hat{O}(1)$ depth, and in CONGEST in $\hat{O}(\sqrt{n} + D)$ rounds.*

Proof of PRAM. Computing sketchings uses $\tilde{O}(m)$ work and $\tilde{O}(1)$ depths according to Theorem D.28. The loop for i, j, t_1 only contributes $\tilde{O}(1)$ to work and depth, so let us focus on one loop fixing i, j, t_1 . Generating $\mathcal{P}$ and marking centers can be trivially implemented. The non-trivial part is the loop for t_2 , we implement it by contracting each vertex set in $\mathcal{P}$, and running a BFS like procedure with root on every center, where the BFS only includes a vertex set C if it satisfies the condition in Algorithm 1. BFS trees have depth bound $2^{\log^{0.9} n}$, each check for whether or not to include a vertex set into the BFS tree can be done in $\tilde{O}(1)$ work according to Theorem D.28; when different BFS trees meet together, the shared node joins a BFS tree arbitrarily, this does not affect the analysis, and makes sure that the total work is $\tilde{O}(m)$ and the total depth is $\hat{O}(1)$. After finding S_u for each center $u \in V$, we again mark centers in $\mathcal{P}$, and try to implement Algorithm 1. This step is also done by contracting each vertex set in $\mathcal{P}$ and start BFS from every center which uses Algorithm 1 to check whether or not to include a vertex set into a BFS tree. The total work is $\tilde{O}(m)$ and the total depth is $\tilde{O}(1)$. $\square$

Proof of CONGEST. For any $u \in V$, we use shared randomness (see Remark D.27) to locally compute $sk_{\approx}(N_G(u))$ based on the information $N_G(u)$ which is known by u initially. The loop for i, j, t_1 only contributes $\tilde{O}(1)$ to rounds, so let us focus on one loop fixing i, j, t_1 .

$\mathcal{P}$ is maintained as follows: each vertex set in $\mathcal{P}$ gets a unique ID which is known by every node in that vertex set. Initially, $\mathcal{P} = \{\{u\} \mid u \in V\}$, and the ID is the ID for the specific node. Each vertex becomes a center independently, which can be done without communication. The non-trivial part is implementing Algorithm 1, which requires a center node u to first find out S_u (which is connected in G), and look at all $C \in \mathcal{P}$ that are adjacent to S_u , then verifies whether or not to add them to S_u . We achieve this by well-known techniques for CONGEST: if a vertex set contains at most $\sqrt{n}$ vertices, we use broadcasting inside this vertex set to spread and gather information in $\sqrt{n}$ rounds; otherwise we use the whole network to do this in D rounds. As long as all the vertex sets are disjoint, the total dilation is $O(D + \sqrt{n})$ and congest is $O(\sqrt{n})$ since there can be at most $\sqrt{n}$ sets with more than $\sqrt{n}$ nodes. We maintain the size of S_u to determine which strategy to use. Now for each center u , it broadcast to all nodes in S_u about $sk_{\approx}(N_G(u))$, and for every $C \in \mathcal{P}'$, it chooses a representative node v_C and let all nodes in C know $sk_{\approx}(N_G(v_C))$. If there is an edge (v_1, v_2) where $v_1 \in S_u$ and $v_2 \in C \in \mathcal{P}$ (certifying that C is adjacent to S_u), then v_1 and v_2 can communicate and decide if C should be added to S_u or not. After that, for each $C \in \mathcal{P}'$, it chooses an arbitrary S_u to join if there are many (this can be decided by broadcasting inside C), i.e., they mark their ID to be identical to the ID for S_u . This is repeated $2^{\log^{0.9} n}$ times, which in total contributes to $\hat{O}(\sqrt{n} + D)$ rounds. Another non-trivial part is Algorithm 1, which is handled identically as Algorithm 1. $\square$

Remark D.38. The implementation in CONGEST can be viewed as $\hat{O}(1)$ rounds in *minor-aggregation model* (see [Roz+22b]). For any network, one round of minor-aggregation can be simulated by $O(\sqrt{n} + D)$ rounds in CONGEST, which implies our Lemma D.37. For a more efficient network like a planner graph, it can be simulated faster ($\tilde{O}(D)$ rounds). It is a convenient framework for recent works of universal optimality (defined for weighted graphs). However, we are working on unweighted graphs and it is not clear if maximum bipartite matching exists a universally optimal algorithm or not, we do not explore the definition here.

D.5 A Schematic Reduction for Vertex Connectivity

In this section, we show a general framework for solving SSVc, which can be implemented in different models, and prove its correctness. It essentially uses common-neighborhood clustering to reduce the problem to solving *isolating cuts* and solving min-neighbor in a *near-clique common-neighborhood* graph, which we define as follows. For convenience, we write the second problem MinNNCC.

Definition D.39 (Isolating Cuts Problem). In the isolating cuts problem, we are given an undirected graph $G = (V, E)$, vertex sets $C, T \subseteq V$ where $T \subseteq C$ is an independent set inside C , and the goal is to solve the following minimization problem.

$$\min_{L \subseteq C} |N_G(L)| \quad \text{s.t.} \quad |L \cap T| = 1 \quad \text{and} \quad N_G(L) \cap T = \emptyset.$$

The algorithm returns a minimizer L^* along with its neighbors' size $|N_G(L^*)|$.

Definition D.40 (Minimum Neighbor in a Near-Clique Cluster (MinNNCC)). In this problem, we are given an undirected graph $G = (V, E)$, a vertex set $C \subseteq V$ where $N_G[C] \neq V$, and an integer ℓ . The goal is to find a non-empty vertex set $L \subseteq C$ minimizing $|N_G(L)|$ and also return $|N_G(L)|$. The inputs have the following guarantees:

1. (correct estimate). There exists $L \subseteq C$ which is a minimizer of $\min_{L' \subseteq C: L' \neq \emptyset} |N_G(L')|$ such that

$$|L| \leq \ell \leq 2|L|.$$

2. (near clique). For any $u \in C$,

$$|C - N_G(u)| \leq 2^{\log^{0.8} n} \cdot \ell, \tag{D.1}$$

3. (common neighborhood). the cluster C has neighborhood difference $2^{\log^{0.7} n} \cdot \ell$. That is, for all $u, v \in C$,

$$|N_G(u) \Delta N_G(v)| \leq 2^{\log^{0.7} n} \cdot \ell \tag{D.2}$$

If the input guarantees are unsatisfied, the algorithm returns an arbitrary $L \subseteq C$ and $|N_G(L)|$.

D.5.1 The Reduction

Let us denote the algorithm solving isolating cuts by $\text{ISOLATINGCUTS}(G, C, T)$ and the algorithm solving MinNNCC by $\text{MinNNCC}(G, C, \ell)$.

The Schematic Algorithm. The framework for solving SSVc is presented in Algorithm 2.

The following lemma shows the correctness of Algorithm 2 based on the correctness of ISOLATINGCUTS and MinNNCC.

Lemma D.41 (Correctness of the framework). *Given that ISOLATINGCUTS and MinNNCC correctly output according to Definition D.39 and Definition D.40, Algorithm 2 returns a valid vertex cut, which is a minimum t -sink vertex cut with probability at least $1/n^{o(1)}$.*

Algorithm 2: $S \leftarrow \text{SSVC}(G, t)$ **Data:** An undirected graph $G = (V, E)$, a vertex t .**Result:** A minimum t -sink vertex cut (L, S, R) .

```

1 foreach  $i = 0, 1, \dots, \log n$  do
2    $\ell \leftarrow 2^i$ ;
3    $\mathcal{C} \leftarrow \text{COMNEICLUSTERING}(G, 2\ell)$ ;
4   Let  $C \leftarrow C - N_G[t]$  for every  $C \in \mathcal{C}$ ;
5   foreach  $C \in \mathcal{C}$  do
6     Let  $T$  include each node in  $C$  independently at random with probability  $\frac{1}{\ell \cdot 2^{\log^{0.8} n}}$ ;
7     Let  $G'$  be a subgraph of  $G$  only containing edges adjacent to  $C$ ;
8      $L_{C,1}, s_{C,1} \leftarrow \text{ISOLATINGCUTS}(G', C, T)$ ;
9      $L_{C,2}, s_{C,2} \leftarrow \text{MinNNCC}(G', C, \ell)$ ;
10  Return  $N_G(L_{C,i})$  where  $s_{C,i}$  is minimized among all  $C \in \mathcal{C}$  and  $i \in \{1, 2\}$ ;

```

Proof. Let (L, S, R) be one of the minimum t -sink vertex cut such that $G[L]$ is connected (if it is not, take one connected component of $G[L]$ as L which will not increase the cut size). There must exist a loop such that $|L| \leq \ell \leq 2|L|$. Notice that $N_G(L_{C,i})$ is always a valid t -sink vertex cut for any C and i since $L_{C,i} \subseteq V - N_G[t]$, so we only need to prove that in that loop, either $N_G(L_{C,1})$ or $N_G(L_{C,2})$ is a minimum t -sink vertex cut for some $C \in \mathcal{C}$.

According to [Lemma D.26](#) and [Lemma D.31](#), with probability at least $1/n^{o(1)}$ there exists $C \in \mathcal{C}$ such that $L \subseteq C$ in [Algorithm 2](#). Let us focus on such C .

We will prove that there are only two possible cases for C . Let $d = \ell \cdot 2^{\log^{0.7} n}$.

Claim D.42. *If C has neighborhood difference d and $L \subseteq C$, then either $|N(L) \cap C| \leq 2d$, or $|C - N[L]| \leq 3d$.*

Proof. Let us assume $|N(L) \cap C| > 2d$ and we will prove $|C - N[L]| \leq 3d$.

For an arbitrary $x \in L$, we have $N(x) \cap (C - N[L]) = \emptyset$. Since every node $v \in C$ satisfies $|N(x) \Delta N(v)| \leq d$, we have $|N(v) \cap (C - N[L])| \leq d$. Thus, the total number of edges from $C \cap N(L)$ to $C - N[L]$ is at most $|C \cap N(L)| \cdot d$.

Moreover, for an arbitrary $x \in L$, we have $|N(x)| \geq |N(L)|$ ($|N(L)|$ is minimum) and $N(x) \subseteq L \cup N(L)$, which implies $|(L \cup N(L)) - N(x)| \leq |L| \leq \ell$. This gives $|N(L) - N(x)| \leq \ell$. Thus, every node $v \in C$ satisfies $|N(L) - N(v)| \leq \ell + d$, which implies $|N(v) \cap N(L) \cap C| \geq |N(L) \cap C| - (\ell + d) \geq |N(L) \cap C|/3$. The last inequality is due to $|N(L) \cap C| > 2d$ and $d = \ell \cdot 2^{\log^{0.7} n}$. Therefore, the number of edges from $C - N[L]$ to $C \cap N(L)$ is at least $|C - N[L]| \cdot |N(L) \cap C|/3$.

Combining the two inequalities from the last two paragraphs gives $|C - N[L]| \leq 3d$. $\square$

In the first case when $|N(L) \cap C| \leq 2d$, [Algorithm 2](#) will make $|T \cap L| = 1$ and $|T \cap S| = 0$ with probability $1/n^{o(1)}$, to make sure T is an independent set we simply delete vertices in T that is adjacent to another vertex in T , which according to [Definition D.39](#) makes $N(L_{C,1})$ a minimum t -sink cut; in the second case when $|C - N[L]| \leq 3d$, according to [Definition D.40](#) $N(L_{C,2})$ is a minimum t -sink cut. $\square$

D.5.2 Implementation in CONGEST and PRAM models

As applications of the reduction, we present algorithms that solve ISOLATINGCUTS and MinNNCC in PRAM and CONGEST models in [Sections D.6](#) and [D.7](#), which are summarized as in the following two lemmas.

Lemma D.43. *Given a ISOLATINGCUTS instance (G, C, T) where the input graph has m edges and n vertices, there is an algorithm solving it correctly with high probability in*

1. **PRAM model**, the algorithm can be implemented to run in work $\tilde{O}(W(m, |C|))$ and depth $\tilde{O}(D(m, |C|))$ where $W(m, n)$ and $D(m, n)$ are the work and depth of s - t vertex connectivity.
2. **CONGEST model**, the algorithm can be implemented to run in $\tilde{O}(R(m, n, D))$ rounds where $R(m, n, D)$ is the round complexity of subgraph S - T vertex connectivity and D is the diameter of G ; furthermore,

the algorithm only communicates via the set of edges that is incident to C .

Lemma D.44. *Given a MinNNCC instance (G, C, ℓ) (Definition D.40), it can be solved with high probability in*

1. PRAM model in $\tilde{O}(W(m, n))$ work and $\tilde{O}(D(m, n))$ depth where $W(m, n)$ and $D(m, n)$ are the work and depth of s - t vertex connectivity, satisfying that $W(m, n)$ is superadditive on m (meaning $W(m_1 + m_2, n) \geq W(m_1, n) + W(m_2, n)$ for every m_1, m_2, n),
2. CONGEST model in $\tilde{O}(R(m, n, O(1)))$ rounds where $R(m, n, D)$ is the round complexity of subgraph S - T vertex connectivity; furthermore, the algorithm only communicates via the set of edges that is incident to C .

To solve ISOLATINGCUTS and MinNNCC problems, in addition to sparsification techniques from [Li+21], we introduce novel vertex sparsification lemmas that leverages the structure of the common-neighborhood property. We discuss them in details in Section D.5.3.

With these two lemmas, we are ready to prove our main theorem about solving SSVC.

Lemma D.45. *If, in PRAM model, the s - t vertex connectivity problem can be solved in $W(m, n)$ work and $D(m, n)$ depth where $W(m, n)$ is superadditive on m , then SSVC can be solved in $W(m, n) \cdot n^{o(1)}$ work and $D(m, n) \cdot n^{o(1)}$ depth.*

Proof. The correctness of Algorithm 2 is proved by Lemma D.41. Notice that to boost the correct probability to w.h.p., we repeat the algorithm $\hat{O}(1)$ times and choose the smallest vertex cut as the output, which only increases the work and depth by at most a $\hat{O}(1)$ factor. Now we calculate the total work and depth for one run of Algorithm 2.

There are $O(\log n)$ iterations of the outer loop of i . Common-neighborhood clustering uses $\hat{O}(m)$ work and $\hat{O}(1)$ depth according to Lemma D.37. For every $C \in \mathcal{C}$, we solve the problems ISOLATINGCUTS and MinNNCC simultaneously. Write E_C as the edge set of edges in G adjacent to C . Each one of them on C cost work $\tilde{O}(W(|E_C|, n))$ and depth $\tilde{O}(D(|E_C|, n))$ according to Lemma D.43 and Lemma D.44. Notice that each edge can be contained in at most $\tilde{O}(1)$ different C according to Lemma D.31. Thus, the total work is $\sum_{C \in \mathcal{C}} \tilde{O}(W(|E_C|, n)) \leq \tilde{O}(W(\sum_{C \in \mathcal{C}} |E_C|, n)) = \tilde{O}(W(m, n))$ where the inequality is due to W is superadditive on m , and the depth is $\tilde{O}(D(m, n))$ since they are run simultaneously. $\square$

Lemma D.46. *If, in CONGEST model, the subgraph S - T vertex connectivity problem can be solved in $R(m, n, D)$ rounds, then SSVC can be solved in $R(m, n, D) \cdot n^{o(1)}$ depth.*

Proof. The correctness of Algorithm 2 is proved by Lemma D.41. Notice that to boost the correct probability to w.h.p., we repeat the algorithm $\hat{O}(1)$ times and choose the smallest vertex cut as the output (the size of each vertex cut is also output according to Definitions D.39 and D.40), which only increases the work and depth by at most a $\hat{O}(1)$ factor. Now we calculate the total rounds for one run of Algorithm 2.

There are $O(\log n)$ iterations of the outer loop of i . For each iteration, common-neighborhood clustering uses $\hat{O}(\sqrt{n} + D)$ rounds according to Lemma D.37. For every $C \in \mathcal{C}$, we solve the problems MinNNCC simultaneously. Notice that each edge can be adjacent to at most $\tilde{O}(1)$ different C according to Lemma D.31. Thus, each edge is involved in at most $\tilde{O}(1)$ different algorithms of ISOLATINGCUTS and MinNNCC, which results in total congestion of $\tilde{O}(R(m, n, O(1)))$ according to Lemma D.43 and Lemma D.44. So the total number of rounds caused by MinNNCC is at most $\tilde{O}(R(m, n, O(1)))$.

Now we explain how to implement ISOLATINGCUTS for every cluster $C \in \mathcal{C}$ by one call to Lemma D.43. We first explain why we can not do similar things to MinNNCC: this is because the round complexity stated in Lemma D.43 depends on the diameter of the subgraph G' which only includes edges incident to C , and the diameter could be much larger than the diameter of G . We solve this issue in the following way. For each $C \in \mathcal{C}$, we first apply Lemma D.50 with G, C, T to get $C' \subseteq N_G(C)$ and K so that it suffices to run ISOLATINGCUTS on $G'[C \cup C']$ as they preserve all isolating cuts up to a fixed number K . We also have $|C'| = \hat{O}(|C|)$ since $|T| = \hat{O}(|C|/\ell)$ w.h.p. Then we construct a virtual graph with $\hat{O}(n)$ vertices in the following way: the virtual graph contains $|\mathcal{C}|$ many disconnected connected components, where each

component corresponds to a cluster $C \in \mathcal{C}$, which we call G_C , where G_C contains the vertex set $C \cup C'$ and all edges adjacent to C in G . Now we call ISOLATINGCUTS on this virtual graph with the input vertex C defined in [Definition D.39](#) set to be $\cup_{C \in \mathcal{C}} C$ and the input vertex set T set to be the union of sample T in the algorithm. To summarize, we call ISOLATINGCUTS($G_{vir}, \cup_{C \in \mathcal{C}} C, \cup_{C \in \mathcal{C}} T_C$) where T_C is the sampled set for C .

We first show how to use the original communication network to simulate the virtual graph: for each vertex $u \in V$, u will simulate all the nodes in the virtual graph which is a duplication of u . Consider an edge adjacent to u which is (u, v) , we should prove that (u, v) is not duplicated too many times into the virtual graph so that the congestion of simulating the virtual graph is bounded. To see this, notice that an edge can be included in G_C for a cluster $C \in \mathcal{C}$ only if (u, v) is adjacent to C . This can happen at most $\tilde{O}(1)$ times according to [Lemma D.31](#). Thus, we can simulate ISOLATINGCUTS algorithm on the virtual graph using the original graph with a congestion increased by a factor of $\tilde{O}(1)$. The total round complexity is $R(\tilde{O}(m), \tilde{O}(n), \tilde{O}(D)) = \tilde{O}(R(m, n, D))$.

Then, we show that ISOLATINGCUTS($G_{vir}, \cup_{C \in \mathcal{C}} C, \cup_{C \in \mathcal{C}} T_C$) on the virtual graph gives the answer to ISOLATINGCUTS(G', C, T_C) for all independent instances on C . Firstly, every minimum isolating cut $N_{G_{vir}}(L)$ with $|L \cap \cup_{C \in \mathcal{C}} T_C| = 1$ cannot cross different connected components in G_{vir} because otherwise we can restrict L into a single connected component, which cannot increase the value of $N_{G_{vir}}(L)$. Thus, the value returned by ISOLATINGCUTS($G_{vir}, \cup_{C \in \mathcal{C}} C, \cup_{C \in \mathcal{C}} T_C$) is at least the size of ISOLATINGCUTS(G', C, T_C) for some $C \in \mathcal{C}$. Moreover, every isolating cut $N_{G'}(L)$ with $L \subseteq C$ and $|L \cap T_C| = 1$ is definitely a valid isolating cut for G_{vir} . Thus, the value returned by ISOLATINGCUTS($G_{vir}, \cup_{C \in \mathcal{C}} C, \cup_{C \in \mathcal{C}} T_C$) is exactly what we want. $\square$

D.5.3 Cluster Sparsification

A *bipartite matching* of a bipartite graph (A, B, E) (where A, B forms a partition of the vertex set and $E \subseteq A \times B$) is an edge set M such that any two edges do not share an endpoint. For convenience, we define $M_A = \{u \mid u \in A, \exists v \in B, (u, v) \in M\}$, and M_B similarly. By the definition, $|M_A| = |M_B| = |M|$. Define $M(u) = v$ where $u \in M_A, v \in B, (u, v) \in M$, and $M(U) = \{M(u) \mid u \in U\}$ where $U \subseteq M_A$. Similarly, we can define $M^{-1}(v) = u$ and $M^{-1}(U) = \{M^{-1}(v) \mid v \in U\}$ for $v \in M_B$ and $U \subseteq M_B$. A *vertex cover* of this bipartite graph is a set of vertices $C \subseteq A \cup B$ such that every edge in E is adjacent to at least one vertex in C .

Lemma D.47. *Let $G = (V, E)$ be an undirected graph and $s, t \in V$. Let $Z = N_G(s) \cap N_G(t)$, and $H := G - Z$. Then, $\kappa_G(s, t) = \kappa_H(s, t) + |Z|$. Furthermore, If S is an (s, t) -min-separator in H , then $Z \cup S$ is an (s, t) -min-separator in G .*

Proof. This follows because every vertex $v \in N_G(s) \cap N_G(t)$ belongs to every (s, t) -separator in G . $\square$

Lemma D.48. *Given a bipartite graph $G = (A, B, E)$ where $|A| \leq |B|$, there is an $\tilde{O}(m)$ -work $O(\log n)$ -depth algorithm that outputs a vertex set $D \subseteq B$ of size $O(|A| \log n)$ such that there is a maximum matching M where $M_B \subseteq D$.*

Proof. This set can be computed by $O(\log n)$ computations of maximal bipartite matching, see Appendix B of [\[Ass+22\]](#). $\square$

Lemma D.49 (Cluster Boundary Sparsification Lemma (Singleton Version)). *Let $G = (V, E)$ be an undirected graph and $s, t \in V$ satisfying $N_G[s] \cap N_G[t] = \emptyset$. Define $B = N_G(t)$ and $A = V - N_G[t]$ and let M be a maximum matching of $(A, B, E_G(A, B))$. Define $G' = G[V - (B - M_B)]$ as the graph G after removing neighbors of t that are unmatched by the maximum matching M . Then, we have $\kappa_G(s, t) = \kappa_{G'}(s, t)$. In particular, every minimum (s, t) -separator in G' is a minimum (s, t) -separator in G .*

Proof. We first prove that $\kappa_G(s, t) \geq \kappa_{G'}(s, t)$. Let S be a minimum (s, t) -separator in G . So, every (s, t) -path in G passes a vertex in S . Since G' is a subgraph of G , every (s, t) -path in G' is an (s, t) -path in G , and thus it passes a vertex in S . Thus, $\kappa_G(s, t) = |S| \geq \kappa_{G'}(s, t)$.

It remains to prove $\kappa_G(s, t) \leq \kappa_{G'}(s, t)$. We argue that there is a maximum (s, t) vertex-disjoint paths of size $\kappa_G(s, t)$ in G' . There exists $\kappa = \kappa_G(s, t)$ internal vertex disjoint paths $p_1, \dots, p_\kappa$ in G . Let b_i be the vertex that p_i intersect B the first time, and let $B' = \{b_i \mid i \in [\kappa]\}$. There exists a matching M' of size κ such that $M'_B = B'$ (by matching b_i to the previous node of b_i on path p_i). This means the set $A' := M'_A$ contains all the previous nodes of b_i on p_i . To prove $\kappa_G(s, t) \leq \kappa_{G'}(s, t)$, it suffices to prove that there exists a matching M'' of size κ such that $M''_A = A'$ and $M''_B \subseteq M_B$ (in which case there exists κ internal vertex disjoint path connecting s, t in G').

Now we show the existence of such a matching M'' . Let C be a minimum vertex cover of $(A, B, E_G(A, B))$. By König's theorem, $|C| = |M|$. Let $C_A = C \cap A$ and $C_B = C \cap B$. By definitions, $C_A \subseteq M_A, C_B \subseteq M_B$. Each edge in M has exactly one endpoint in C_A , or one endpoint in C_B (because every edge in M must have one endpoint in C while $|C| = |M|$). Let $C'_A = C_A \cap A'$ and $C'_B = C_B \cap B'$. We define M'' in the following way: for every $u \in C'_A$, define $M''(u) = M(u)$; for every $u \in A' - C'_A$, define $M''(u) = M'(u)$. We first prove that $M''_B \subseteq M_B$. For any $u \in C'_A$, trivially we have $M''(u) = M(u) \in M_B$. For any $u \in A' - C'_A$, since C is a vertex cover and $(u, M'(u))$ is an edge where $u \notin C_A$, we have $M'(u) \in C_B \subseteq M_B$. Thus, $M''_B \subseteq M_B$. Now we prove that M'' is a valid matching, i.e., for any $u \in C'_A, v \in A' - C'_A$, we need to prove $M(u) \neq M'(v)$. Since $u \in C'_A$, we have $M(u) \notin C_B$. Moreover, we have $M'(v) \in C_B$, thus, $M(u) \neq M'(v)$. $\square$

Lemma D.50 (Cluster Boundary Sparsification Lemma (Batched Version)). *Given $G = (V, E)$, $C \subseteq V$ and ℓ satisfying the common-neighborhood property $|N_G(u) \triangle N_G(v)| = \hat{O}(\ell)$ for every $u, v \in C$, and in addition given $X \subseteq C$ with $|X| = \hat{O}(|C|/\ell)$. There exists PRAM and CONGEST algorithm that outputs $C' \subseteq N_G(C)$ and an integer K such that*

1. $|C'| = \tilde{O}(|C|)$,
2. for every $x \in X$, define $G' = G[C \cup C']$, we have $\min_{L' \subseteq C: x \in L'} |N_G(L')| = \min_{L' \subseteq C: x \in L'} |N_{G'}(L')| + K$ for a fixed number K .
3. for every $x \in X$, define $G' = G[C \cup C']$, we have $\min_{L' \subseteq C: x \in L', |L' \cap X|=1} |N_G(L')| = \min_{L' \subseteq C: x \in L', |L' \cap X|=1} |N_{G'}(L')| + K$ for a fixed number K .

The algorithm runs in $\tilde{O}(m)$ work and $\tilde{O}(1)$ depth in PRAM where m is the number of edges in G , and $\tilde{O}(1)$ rounds in CONGEST.

Proof. If X is empty, return $C' = \emptyset$, and the lemma is vacuously true; so let us assume X is non-empty.

Notice that if the maximum degree of nodes in C becomes much larger than $\hat{O}(|C|)$, then we can delete all vertices in $(\cap_{x \in X} N_G(x)) \cap N_G(C)$, which decrease the value of $|N_G(L')|$ by $K = |(\cap_{x \in X} N_G(x)) \cap N_G(C)|$ for every L' satisfying $x \in L'$ for some $x \in X$. This set $(\cap_{x \in X} N_G(x)) \cap N_G(C)$ can be found in $O(m)$ work and $O(1)$ depth in PRAM, and in 1 round in CONGEST, after deleting which can reduce the maximum degree of nodes in C to $\hat{O}(|C|)$ because of the common-neighborhood property: in order for a vertex $v \in N_G(x)$ to not be deleted, either $v \in C$, or there exists $x' \in X$ such that $v \in N_G(x') \triangle N_G(x)$, which can happen at most $\hat{O}(\ell)$ times for each x' . Thus, the degree is at most $\hat{O}(|C| + |X| \cdot \ell) = \hat{O}(|C|)$. In the following proof we will assume the maximum degree of nodes in C is at most $\hat{O}(|C|)$, so the total number of edges adjacent to C is $\hat{O}(|C|^2)$.

We first show that for a specific x , in order for $\min_{L' \subseteq C: x \in L'} |N_G(L')| = \min_{L' \subseteq C: x \in L'} |N_{G'}(L')|$ to be true, it suffices to let C' contain (i) $N_G(x) \cap N_G(C)$, (ii) the (right) endpoints of the edges in an arbitrary maximum bipartite matching between C and $N_G(C) - N_G(x)$. To see this, we need Lemma D.49, in which we will set $s = x$, t be a new super node connecting to every vertex in $N_G(C)$ and G be the graph after adding the super node t and deleting all common neighbors of s and t . It is easy to notice that by Lemma D.49, preserving the vertices in an arbitrary maximum bipartite matching between C and $N_G(C) - N_G(x)$, along with $N_G(x) \cap N_G(C)$ suffices to preserve the value of $\min_{L' \subseteq C: x \in L'} |N(L')|$.

Thus, the problem becomes finding C' such that for every $x \in X$, it preserves at least one maximum bipartite matching between C and $N_G(C) - N_G(x)$, and $C' \supseteq N_G(x) \cap N_G(C)$. Let H be the bipartite subgraph of G only containing edges between C and $N_G(C)$. We will use the folklore vertex size reduction for BMM, which can be found in Appendix B of [Ass+22]. A critical lemma can be summarized as follows.

Lemma D.51 (Lemma 14 of [Ass+22]). *For a bipartite graph $G = (V, E)$, and a vertex cover $\tilde{V} \subseteq V$ (i.e., for every $(u, v) \in E$, either $u \in \tilde{V}$ or $v \in \tilde{V}$), let M be an arbitrary maximal bipartite matching between $\tilde{V}$ and $V - \tilde{V}$ and $V(M)$ be the vertex set of its endpoints. If the maximum bipartite matching of $G[\tilde{V}]$ has size F and the maximum bipartite matching of G has size F^* , then the maximum bipartite matching of $G[\tilde{V} \cup V(M)]$ has size at least $\frac{1}{3}(F^* - F) + F$.*

Thus, for a specific $x \in X$, it suffices to set $C' = \emptyset$ initially, then repeatedly finding an arbitrary maximal bipartite matching between $C \cup C'$ and $N_G(C) - N_G(x) - C'$, after which add the endpoints of the maximal matching to C' . Notice that C is a vertex cover for H . According to Lemma D.51, after $O(\log n)$ iterations, we are guaranteed that $H[C \cup C']$ contains the same size of maximum bipartite matching as $H[C \cup (N_G(C) - N_G(x))]$. In the end we add $N_G(x) \cap N_G(C)$ to C' .

However, we need to construct C' for all $x \in X$, which will increase the size of C' to $|X| \cdot |C|$, far from our goal of $\tilde{O}(|C|)$. To solve this, we let $\tilde{C} = \bigcup_{x \in X} (N_G(x) \cap N_G(C))$. Notice that $|\tilde{C}| = \hat{O}(|X| \cdot \ell + |C|) = \hat{O}(|C|)$ according to the common neighborhood property and the maximum degree bound. Then, we repeatedly find an arbitrary maximal bipartite matching between C and $N_G(C) - \tilde{C}$, add the endpoints to $\tilde{C}$ and repeat for $O(\log n)$ times. A critical observation is that, in each iteration, the edges between C and $N_G(C) - \tilde{C}$ are the same as edges between $C \cup \tilde{C}$ and $(N_G(C) - N_G(x)) - (\tilde{C} - N_G(x))$ for every x because $N_G(x) \cap N_G(C) \subseteq \tilde{C}$ and $N_G(C)$ is an independent set in H . Thus, in the end $H[C \cup (\tilde{C} - N_G(x))]$ contains the same size of maximum bipartite matching as $H[C \cup (N_G(C) - N_G(x))]$, according to Lemma D.51. In the end we set $C' = \tilde{C}$ which certainly contains $N_G(x) \cap N_G(C)$ for every $x \in X$, so C' preserves $\min_{L' \subseteq C: x \in L'} |N_{G'}(L')|$ for every $x \in X$. Since there are $O(\log n)$ iterations where each maximal bipartite matching has size at most $|C|$, we have $|C'| = \hat{O}(|C|)$. The algorithm can be implemented in $\tilde{O}(m)$ work and $\tilde{O}(1)$ depth because maximal bipartite matching can be solved in $\tilde{O}(m)$ work and $\tilde{O}(1)$ depth. Moreover, in CONGEST, maximal bipartite matching can be solved in $\tilde{O}(1)$ rounds.

In order to also make sure $\min_{L' \subseteq C: x \in L', |L' \cap X|=1} |N_G(L')| = \min_{L' \subseteq C: x \in L', |L' \cap X|=1} |N_{G'}(L')| + K$, we the same algorithm while setting $C \leftarrow C - X$. In this way, one maximum matching between $C - X$ and $N_G(C) - N_G(x)$ is preserved for every $x \in X$. According to Lemma D.49, the value of $\min_{L' \subseteq C: x \in L', |L' \cap X|=1} |N_G(L')|$ is preserved for every $x \in X$. □

D.6 Isolating Cuts Lemma for Parallel and Distributed Algorithms (Proof of Lemma D.43)

In this section we will focus on proving the following lemma.

Lemma D.52 (Parallel and Distributed Isolating Cuts Lemma). *Given a graph $G = (V, E)$ and an independent set $T \subseteq V$ of size at least 2, there is an algorithm that outputs for each $v \in T$ a $(v, T - \{v\})$ -min-separator C_v . The (sequential) algorithm makes calls to s - t vertex mincut on graphs with $\tilde{O}(n)$ total number of vertices and $O(m)$ total number of edges and takes $\tilde{O}(m)$ additional time.*

- In the PRAM model, the algorithm can be implemented to run in work $\tilde{O}(W(m, n))$ and depth $\tilde{O}(D(m, n))$ where $W(m, n)$ and $D(m, n)$ are the work and depth of s - t vertex connectivity.
- In the distributed CONGEST model, the algorithm can be implemented to run in $\tilde{O}(R(m, n, D))$ rounds where $R(m, n, D)$ is the round complexity of subgraph S - T vertex connectivity and D is the diameter of G .

We first show that Lemma D.43 directly follows from Lemma D.52.

Proof of Lemma D.52. Given a graph G , a vertex set C and a vertex set T , we first use Lemma D.50 to construct $C' \subseteq N_G[C]$ such that $G[C \cup C']$ preserves the minimum isolating cut. Write $G' = G[C \cup C']$. Then for each $v \in N_{G'}(C)$, we add a new node v' and connect v' to v by an edge, add v' to T , let the resulting terminal set be T' . In the distributed network, v' is simulated by v . Now we run the algorithm

described in [Lemma D.52](#) on G' with T' . It is easy to see that this solves [Lemma D.43](#). Moreover, G' only has $\hat{O}(C)$ many vertices according to [Lemma D.50](#). Thus, the work and depth for PRAM are $W(\tilde{m}, |C|)$ and $D(\tilde{m}, |C|)$. $\square$

We prove the parallel version of the isolating cut lemma [\[Li+21\]](#). Previously, the isolating cut lemma [\[Li+21\]](#) can only guarantee $O(m)$ total number of vertices which is suboptimal for parallel implementation. Before we prove [Lemma D.52](#), we first review an algorithm for the isolating cut lemma [\[Li+21\]](#) and then we state the refinement.

Algorithm [\[Li+21\]](#) The inputs consist of the input graph $G = (V, E)$ and an independent set $T \subseteq V$ of size at least 2.

1. Encode each $t \in T$ as a binary string of length $\lceil \log_2 |T| \rceil$. For each $i \leq \lceil \log_2 |T| \rceil$, define $A_i := \{t \in T : i^{\text{th}} \text{ bit of } t = 0\}$ and $B_i := T - A_i$. Compute $F_i := (A_i, B_i)$ -vertex mincut.
2. For each $s \in T$, compute a connected component containing s , denoted by U_s , in $G - \bigcup_i F_i$.
3. For each $s \in T$, define G'_s as $G[N_G[U_s]]$ followed by (1) adding an additional vertex t and all edges from t to $N_G(U_s)$, and (2) remove all edges inside $N_G(U_s)$. Then, compute a minimum (s, t) -separator denoted by C_s in G'_s .

We summarize the correctness into the following lemma.

Lemma D.53 ([\[Li+21\]](#)). *For each $s \in T$, C_s is an $(s, T - \{s\})$ -min-separator in G .*

This construction yields $O(m)$ total number of vertices because the boundary of U_s for all $s \in T$ can be large.

Refinement. Our refinement is at step 3: before computing the minimum (s, t) -separator in G'_s , we further sparsify G'_s , while preserving the minimum (s, t) -separator, so that the total number of vertices is $\tilde{O}(|U_s|)$.

Lemma D.54 (Cluster Sparsification). *Let G'_s be given as an input, and denote $Z_s := N_{G'_s}(s) \cap N_{G'_s}(t)$ as the common neighbors between s and t in G'_s . There is an $\tilde{O}(\text{vol}_G(U_s))$ -time algorithm that returns a vertex set $D \subseteq N_G(U_s) - Z_s$ satisfying the following property: Let G''_s be the graph G'_s after removing $Z_s \cup (N_G(U_s) - D)$.*

- $\kappa_{G'_s}(s, t) = \kappa_{G''_s}(s, t) + |Z_s|$,
- If W is an (s, t) -min-separator in G''_s , then $W \cup Z_s$ is an (s, t) -min-separator in G'_s , and
- $|V(G''_s)| = O(|U_s| \log n)$.

In the PRAM model, G''_s can be constructed in $\tilde{O}(\text{vol}_G(U_s))$ -work and $\tilde{O}(1)$ -depth.

Proof. We are given the graph G'_s , and the goal is to compute a graph G''_s with the properties described in the statement. We describe the construction of G''_s in two steps.

(R1) Let $Z := N_{G'_s}(s) \cap N_{G'_s}(t)$ from G'_s .

(R2) Remove all vertices in $N_G(U_s) - (Z \cup D)$ where D is computed from the following.

- (a) Define the bipartite graph (A, B, E') where $A := U_s$, $B := N_G(U_s) - Z$ and $E' = E_G(A, B)$.
- (b) If $|A| \leq |B|$, apply [Lemma D.48](#) on the bipartite graph (A, B, E') to obtain the vertex set $D \subseteq B$. Otherwise, $D = N_G(U_s) - Z$.

Analysis In terms of work, it takes $O(m)$ work to construct the bipartite graph instance (A, B, E') . In step (R2), we compute the set D in $\tilde{O}(m)$ work and $O(\log n)$ depth [Lemma D.49](#). All these steps can be implemented in $\tilde{O}(1)$ depth. We now argue the size. In step (R2), if $|A| > |B|$, then we are done. Otherwise, the set D has size $O(|A| \log n)$ by [Lemma D.48](#). Therefore, $|V(G''_s)| \leq |U_s| + 2|D| = O(|U_s| \log n)$. For correctness, the fact that $\kappa_{G'_s}(s, t) = \kappa_{G''_s}(s, t) + |Z|$ follows from [Lemmas D.47](#) to [D.49](#). $\square$

Therefore, [Lemma D.52](#) can be shown as follows. We apply the same algorithm in the first two steps, but we slightly modify Step 3 as follows. In Step 3, for each $s \in T$, we replace the graph G'_s with G''_s

produced by [Lemma D.52](#), and return $Z_s \cup C'_s$ where Z_s is the common neighbors between s and t as stated in [Lemma D.54](#), and C'_s is a minimum (s, t) -separator C'_s on G'_s . The total number of edges over all $s \in T$ is $O(m)$ as before because we only remove edges from G'_s . Now, the total number of vertices is

$$\sum_{s \in T} |V(G'_s)| \leq O\left(\sum_{s \in T} |U_s| \log n\right) \leq \tilde{O}(n).$$

It is easy to see that we can implement in the PRAM model in total $\tilde{O}(m)$ work and $\tilde{O}(1)$ depth outside of the calls to s-t vertex mincut. Note that (A_i, B_i) -vertex mincut can be reduced to s-t vertex mincut by contracting A_i into s and B_i into t .

Implementation in distributed CONGEST setting. The algorithm is similar to PRAM except that contracting a vertex set in Step 3 is a global operation. To deal with this issue, we modify G'_s into a *cut-equivalent* graph by adding new terminal sets along $N_G(U_s)$. For any vertex sets $A, B \subseteq V$, $\kappa_G(A, B)$ is the size of an (A, B) -vertex mincut.

Lemma D.55. *Given G'_s , define H_s as follows. First, remove t from G'_s . Then, for each vertex $u \in N(U_s)$, add a new vertex y along with an edge (u, y) . Denote Y as the set of new vertices. Then, every (s, t) -cut in G'_s is an (s, Y) -cut in H_s and vice versa. In particular, $\kappa_{G'_s}(s, t) = \kappa_{H_s}(s, Y)$.*

Proof. If S is an (s, t) -separator in G'_s , then denote a vertex cut (L, S, R) in G'_s where $s \in L, t \in R$. Since $N_{G'_s}(t) \subseteq S \cup R$, every vertex in Y in H_s does not have an edge to L . Therefore, $(L, S, (R \cup Y) - \{t\})$ is a vertex cut in H_s . If S is an (s, Y) -separator in H_s , then the same separator must be an (s, t) -separator in G'_s by contracting all Y into t . $\square$

Note that the set Y can be simulated inside the incident vertices. Now, we can state the cluster sparsification lemma for the distributed version as follows. The proof is similar to [Lemma D.54](#).

Lemma D.56. *Given H_s , denote $Z_s := N_{H_s}(s) \cap N_{H_s}(Y)$ as the common neighbors between s and Y . There is an $\tilde{O}(1)$ -round algorithm that returns a vertex set $D \subseteq N_G(U_s) - Z_s$ satisfying the following property. Define H'_s as the graph H_s after removing $Z_s \cup (N_G(U_s) - D)$ and isolated vertices in Y , and denote Y' to be the remaining nodes in Y . Then, $\kappa_{H_s}(s, Y) = \kappa_{H'_s}(s, Y') + |Z|$. If W is an (s, Y') -mincut in H'_s , then $W \cup Z$ is an (s, Y) -mincut in H_s . Furthermore, $|V(H'_s)| = O(|U_s|)$.*

Finally, observe that each graph H'_s alone can have a larger diameter than the original graph. To handle this case, instead of running each graph separately, we will combine all H'_s instances and compute (s, Y) -mincut over all $s \in T$ and over all the union of H'_s . Running on all instances the combined graph will preserve the diameter. We are ready to state the distributed algorithm in CONGEST model.

Algorithm (CONGEST). The inputs consist of graph $G = (V, E)$, an independent set $T \subseteq V$ of size at least 2.

1. Encode each $t \in T$ as a binary string of length $\lceil \log_2 |T| \rceil$. For each $i \leq \lceil \log_2 |T| \rceil$, define $A_i := \{t \in T : i^{\text{th}} \text{ bit of } t = 0\}$ and $B_i := T - A_i$. Compute $F_i := (A_i, B_i)$ -vertex mincut.
2. For each $s \in T$, compute a connected component containing s , denoted by U_s , in $G - \bigcup_i F_i$.
3. For each $s \in T$, construct H_s and then compute H'_s using [Lemma D.56](#). Let Z_s be the common neighbors in the lemma.
4. Compute an (s, Y') -min-separator W in H'_s for all $s \in T$ on the union of H'_s overall $s \in T$.
5. For each $s \in T$, by [Lemma D.56](#), $Z_s \cup W_s$ is an (s, Y) -mincut in H_s .

Running Time. The first two steps can be implemented in $O(R(m, n, D) \cdot \log |T|)$ rounds ([Lemma D.88](#)). Step 3 can be done in $\tilde{O}(1)$ rounds by [Lemma D.56](#). For each $s \in T$, let m_s and n_s be the number of edges and number of vertices of H'_s . We can run Step 4 all in parallel on the union of H'_s as a single graph, which can be done in $O(R(\sum_{s \in T} m_s, \sum_{s \in T} n_s, D)) = \tilde{O}(R(m, n, D))$ rounds.

Correctness. Steps 1 and 2 are identical to the sequential algorithm. We now focus on Step 3. Fix a vertex $s \in T$. We prove that the algorithm returns an (s, t) -min-separator in G'_s . By [Lemma D.107](#), an (s, Y) -min-separator in H_s is also a (s, t) -min-separator in G'_s . By [Lemma D.56](#), an (s, Y') -min-separator in H'_s is also an (s, Y) -min-separator in H_s , and we are done.

D.7 Parallel and Distributed Algorithms for MinNNCC (Proof of [Lemma D.44](#))

D.7.1 PRAM Algorithm

This section proves part 1 of [Lemma D.44](#). We are given the MinNNCC instance (G, C, ℓ) ([Definition D.40](#)). The PRAM algorithm goes as follows. We first sample $\hat{O}(|C|/\ell)$ random vertices in C , denoted by X . Given X, C and ℓ , we apply the batched version of the cluster boundary sparsification lemma ([Lemma D.50](#)) to replace the boundary $N_G(C)$ to C' where $|C'| = \tilde{O}(|C'|)$ without changing the minimizer. For each $x \in X$, we compute (x, t) -min separator in the cluster after sparsification using similar reduction rules in [\[Li+21\]](#).

Algorithm. The inputs consist of (G, C, ℓ) .

1. As preprocessing, we first apply [Theorem D.28](#) to compute $sk_z(N(u))$ for every $u \in C$ where $z = 2^{\log^{0.8} n} \ell = \hat{O}(\ell)$.
2. Let X be a set of $\hat{O}(|C|/\ell)$ random vertices sampled in C .
3. Given G, C, ℓ, X , apply the batched version of the cluster boundary sparsification lemma ([Lemma D.50](#)) to obtain $C' \subseteq N_G(C)$ and an integer K .
4. For each $x \in X$, construct the following graph $H_x = (V_x, E_x)$ where

$$\begin{aligned} V_x &:= C \sqcup C'_x \sqcup \{t\}, \\ E_x &:= \bigcup_{v \in C} E_G(v, V_x - (N_G(x) \cap C)) \cup \{(v, t) : v \in C'_x\}, \text{ and} \\ C'_x &:= C' - Z_x \text{ where } Z_x := N_G(x) \cap C'. \end{aligned}$$

5. Let $x^* := \arg \min_{x \in X} \left\{ \kappa_{H_x}(x, t) + |Z_x| + K \right\}$
6. Let (L, S, R) be a vertex mincut in H_{x^*} such that $x^* \in L$ and $t \in R$.
7. **Return** L and $|S|$.

Before we present the analysis, we establish the following sparsification lemma at Step 4.

Lemma D.57. *Let $C' \subseteq N_G(C)$ be the sparsified boundary of C defined in Step 3. For each $x \in X$ where X is defined in Step 2, the graph H_x defined in Step 4 can be constructed in $\hat{O}(|C| \cdot \ell)$ work and polylog depth and H_x satisfies the following properties:*

1. For every vertex $v \in C$, $\deg_{H_x}(v, V_x - (N_G(x) \cap C)) \leq O(2^{\log^{0.7} n} \cdot \ell)$
2. $\kappa_{H_x}(x, t) + |Z_x| \geq \min_{L' \subseteq C: L' \neq \emptyset} |N_G(L')|$.
3. If $x \in L^*$, then $\kappa_{H_x}(x, t) + |Z_x| = \min_{L' \subseteq C: L' \neq \emptyset} |N_G(L')|$.

Proof. We focus on the fast construction of H_x . By the preprocessing step, we have an access to the oracle that can list all the elements in the symmetric difference $N(s) \triangle N(s')$ for any pair (s, s') up to $\hat{O}(\ell)$ elements in $\hat{O}(\ell)$ time. By the property of C' , $|N_{H_x}(C)| \leq \tilde{O}(|C|)$. Hence, the set Z_x can be computed quickly. For each vertex $v \in C - \{x\}$, we list all the neighbors of v that are not the neighbors of x in $\hat{O}(\ell)$ work using the oracle, and we can do it all in parallel. Given the batch version of the cluster boundary sparsification lemma, the correctness proof is similar to [\[Li+21\]](#). $\square$

We are ready to analyze the algorithm.

Analysis. Since we sample $\hat{O}(|C|/\ell)$ many vertices X , there is $x \in X$ such that $x \in L^*$ with high probability. Therefore, the correctness follows from [Lemma D.57](#). For the running time, the algorithm is easily parallelizable. We bound the total work needed to complete this operation. The sketching can be done in almost linear work [Theorem D.28](#). By [Lemma D.57](#), the total work is

$$\sum_{x \in X} \hat{O}(W(|C|\ell, |C|)) \leq \hat{O}(W(|C|^2, |C|)) \leq \hat{O}(W(m, n)).$$

D.7.2 CONGEST Algorithm

In this section, we prove part 2 of [Lemma D.44](#). We are given the MinNNCC instance (G, C, ℓ) ([Definition D.40](#)).

High-level algorithm. At high level, we sample $\tilde{O}(|C|/\ell)$ many vertices in C . With high probability, there is a node $x \in L \subseteq C$. Fix x . We construct a graph H_x such that the minimizer of $\min_{L' \subseteq C: L' \neq \emptyset} |N_G(L')|$ can be computed by calling maximum bipartite matching in H_x . However, this may cause too much congestion when we run all instances of H_x for all $x \in X$. To reduce congestion, we *map* its communication of $\mathcal{A}$ on H_x to a random graph H'_x so that the congestion due to running $\mathcal{A}$ is *load-balanced* in that with high probability, the maximum overlap (in terms of the number of instances on a pair $(u, v) \in C \times C$) of H'_x instances overall $x \in X$ is at most $\hat{O}(1)$.

Step 1. Graph Sparsification. Given $x \in C$, we construct a graph H_x as in the following lemma.

Lemma D.58. *Given $x \in C$, we construct graph $H_x = (V_x, E_x)$ in $O(1)$ rounds where*

$$\begin{aligned} V_x &:= C \sqcup C'_x \sqcup Y_x, \text{ and} \\ E_x &:= \bigcup_{v \in C} E_G(v, V_x - (N_G(x) \cap C)) \sqcup M_x, \\ M_x &:= \text{the perfect matching between } C'_x \text{ and } Y_x \\ C'_x &:= N_G(C) - Z_x \text{ where } Z_x := N_G(x) \cap N_G(C). \end{aligned}$$

The graph H_x satisfies the following properties:

1. For every vertex $v \in C$, $\deg_{H_x}(v, V_x - (N_G(x) \cap C)) \leq O(2^{\log^{0.7} n} \cdot \ell)$
2. $\kappa_{H_x}(x, Y_x) + |Z_x| \geq \min_{L' \subseteq C: L' \neq \emptyset} |N_G(L')|$.
3. If $x \in L^*$, then $\kappa_{H_x}(x, Y_x) + |Z_x| = \min_{L' \subseteq C: L' \neq \emptyset} |N_G(L')|$.

The proof is based on the reduction rules in [\[Li+21\]](#). For completeness, we prove [Lemma D.58](#) in [Section D.12](#).

Step 2. Mapping the communication. We would like to run $\mathcal{A}$ on H_x . There are two issues. First, the graph H_x is a subgraph of G so the diameter can be large. Second, the same edge in H_x can be used for multiple instances of $x \in X$. To deal with the first issue, we reduce the diameter of H_x by adding a star to H_x .

Definition D.59 (H_x^*). We define $H_x^* = (V(H_x) \cup \{s^*\}, E(H_x) \cup E^*)$ where we add (s^*, v) edge to E^* for all $v \in V(H_x)$.

Observe that the diameter of H_x^* is $O(1)$, and adding a star to H_x does not change the minimizer of $\min_{L' \subseteq C: L' \neq \emptyset} |N_G(L')|$. We assume s^* exists for now; we describe how to simulate the star node s^* later, which can be done efficiently since C is nearly a clique.

To deal with the second issue, we *simulate an algorithm $\mathcal{A}$ on H_x^* by mapping the communication of $\mathcal{A}$ to a random graph*.

Definition D.60 (Mapping the communication of $\mathcal{A}$ on H^*). Let $f_x : C \cup C'_x \cup \{s^*\} \rightarrow C$ be a function. Given $H_x^* = (C \cup C'_x \cup \{s^*\}, E_x \cup E^*)$, define the graph $H'_x := (C, E'_x)$ where $(u, v) \in E_x \cup E^*$ if and only if $(f_x(u), f_x(v))_{(u, v)} \in E'_x$ (with possibly parallel edges, and self-loop). We say that the communication of $\mathcal{A}$

on H_x^* is mapped to H'_x via f_x if every node $v \in C$ knows the neighbor sets of its preimage, i.e., v knows $N_{H_x}(w)$ for all $w \in f^{-1}(v)$.

If we can set up such a mapping of the communication, then the algorithm $\mathcal{A}$ on H_x^* can be *simulated* entirely on H'_x .

Lemma D.61 (Simulation). *If the communication of $\mathcal{A}$ on H_x^* is mapped to H'_x via f_x then there is an algorithm $\mathcal{A}'$ running on H'_x such that every round the message exchange (u, v) in H_x^* by $\mathcal{A}$ is equivalent to exchanging the same message between $(f_x(u), f_x(v))_{(u,v)}$ in H'_x by $\mathcal{A}'$.*

Furthermore, if f_x is a random function, then H'_x is a random graph, and thus running multiple instances of $\mathcal{A}$ on H_x^* (using many x 's) can be better load-balanced by simulating on multiple instances of H'_x .

Lemma D.62. *Given $x \in C$ and the description of algorithm $\mathcal{A}$ running on H_x^* , we can map the communication of $\mathcal{A}$ to the graph H'_x (Definition D.60) via a random function f_x where, in H'_x , every vertex $v \in C$ has $\widehat{O}(\ell)$ random neighbors. The mapping can be constructed using $O(1)$ rounds of communication in H'_x .*

We prove Lemmas D.61 and D.62 in Section D.7.2.

Step 3. Virtual Clique in C . Notice that the algorithm $\mathcal{A}'$ assumes the communication on every edge inside H'_x , which may not exist in the original network G_x . To run $\mathcal{A}'$ on H'_x , we can simulate clique communication inside C .

Definition D.63. An t -virtual clique network in $C \subseteq V$ of a graph $G = (V, E)$ is a protocol that simulates the clique communication in C in that any message exchange between $u, v \in C \cup \{s^*\}$ can be done by at most t rounds in G .

Lemma D.64. *Given a cluster $C \subseteq V$ in a graph $G = (V, E)$, we can construct a t -virtual clique network $\mathcal{V}$ in $\tilde{O}(1)$ rounds. With high probability, $t \leq O(\log n)$.*

We prove Lemma D.64 in Section D.7.2.

We are now ready to state the main CONGEST algorithm.

Algorithm (CONGEST). The input is a graph $G = (V, E)$ along with a cluster $C \subseteq V$ and an estimate ℓ .

1. Let $\mathcal{V}$ be the virtual clique network on C constructed from Lemma D.64.
2. Sample $100 \cdot \frac{|C|}{\ell} \cdot \log n$ random vertices in C into X .
3. For each $x \in X$,
 - (a) Construct a graph H_x using Lemma D.58. Let $\mathcal{A}$ be the algorithm that computes minimum (x, Y_x) -separator in H_x .
 - (b) Apply Lemma D.62 to map the communication of $\mathcal{A}$ to H'_x via f_x . Let $\mathcal{A}'$ be the algorithm running on H'_x obtained from Lemma D.61.
 - (c) Run $\mathcal{A}'$ on H'_x using the virtual clique network $\mathcal{V}$.
 - (d) Obtain the output of $\mathcal{A}$ on H_x^* from the output of $\mathcal{A}'$ in H'_x using f_x .
4. Return the smallest cut and its size found so far.

Analysis. In step 2, there exists $x \in X$ such that $x \in L$ with high probability since $\ell = \Theta(|L|)$. Therefore, Lemma D.58 implies that a minimum (x, Y_x) -separator of H_x plus Z_x , and thus we obtain the minimizer of $\min_{L' \subseteq C: L' \neq \emptyset} |N_G(L')|$ at the end of step 4. In terms of round complexity, observe that the bottleneck is to run $\mathcal{A}'$ running on a H'_x as described in Lemma D.62. Fix a vertex $v \in C$, we show in expectation there are at most $O(\log n)$ instances of $\mathcal{A}'$ on every edge from v to C . This follows because there are $\tilde{O}(\frac{|C|}{\ell})$ independent instances of $\mathcal{A}'$ running on v where each instance corresponds to random $\widehat{O}(\ell)$ edges inside C . Therefore, in expectation, each edge has $\widehat{O}(1)$ congestion due to running $\mathcal{A}'$.

Mapping the Communication

In this section, we prove [Lemmas D.61](#) and [D.62](#). We start with the proof of [Lemma D.62](#). The goal is to map the communication of $\mathcal{A}$ on H_x^* to H'_x using a function f_x defined as follows.

Definition D.65. Given $x \in C$, we define $f_x : C \cup C'_x \cup \{s^*\} \rightarrow C$ where

$$f_x(v) = \begin{cases} v & \text{if } v \in N_G[x] \cap C, \\ \text{a random node in } C & \text{otherwise.} \end{cases}$$

Note that f_x is a public random function. So every node can access f_x . To establish a mapping ([Definition D.60](#)), we describe an algorithm to broadcast neighborhoods.

Algorithm. For all $u \in C \cup C'_x - (N_G[x] \cap C)$, every node $v \in N_{H_x^*}(u)$ sends its id and u to $f(u)$ so that $f(u)$ receives (v, u) to learn that v is a neighbor of u .

Complexity. It is convenient to fix a node v and describe the set of vertices to which v sends its id. By design, every v sends its id to the image $f(w)$ of every node $w \in N(v) - (N_G[x] \cap C)$ where $f(w)$ is a random node in C . The distribution can be described as a random graph where every node $u \in C - \{x\}$ has $|N(v) - N_G[x]| \leq O(\ell \log n)$ random neighbors in C . The inequality follows from [Lemma D.58](#).

Therefore, [Lemma D.62](#) follows from f_x and this algorithm. We next prove [Lemma D.61](#). After the mapping above, we have established the following property.

Property 1: For all $u \in C$, and for all $w \in f_x^{-1}(u)$, u learned the set of $N_{H_x}(w)$ of w .

Property 1 means that for every $u \in C \cup C'_x \cup \{s^*\}$ in H_x^* , $f_x(u)$ can act as a surrogate of u since $f_x(u)$ knows all the neighbors of u . That is, we can simulate any algorithm running on $H_x^* = (C \cup C'_x \cup \{s^*\}, E_x \cup E^*)$ by running on another network $H'_x := (C, E'_x)$ where an edge $(u, v) \in E_x$ if and only if $(f_x(u), f_x(v))_{(u,v)} \in E'_x$ to E'_x where we allow parallel edges and self-loops indexed by the pair (u, v) in H'_x .

We are ready to prove [Lemma D.61](#).

Proof of [Lemma D.61](#). Given $\mathcal{A}$, we define another algorithm $\mathcal{A}'$ that simulates the execution of $\mathcal{A}$ as follows:

- For each round in $\mathcal{A}$,
 - if u sends/receives a message m to v in $\mathcal{A}$ then $f_x(u)$ sends/receives the message m to $f_x(v)$ in $\mathcal{A}'$ using the edge $(f_x(u), f_x(v))_{(u,v)}$.

Note that every node v knows its preimage $f_x^{-1}(v)$ since f_x is a public (random) function.

□

Virtual Clique Network

We prove [Lemma D.64](#) in this section. We start with basic facts about C being a near-clique. For all $u \in C$, let $NN_C(u) := C - N_G[u]$ be the non-neighbors of u in C .

We assume that

$$|C| \geq 100 \cdot 2^{\log^{0.8} n} \cdot \ell. \quad (\text{D.3})$$

Otherwise, we sample $n^{o(1)}$ nodes inside C and run bipartite matching trivially on each of them and we are done.

Claim D.66. For all $u \in C$,

$$|N_G(u) \cap C| \geq \Omega(|NN_C(u)|). \quad (\text{D.4})$$

For all $u, v \in C$,

$$|N_G(u) \cap N_G(v) \cap C| \geq \Omega(|C|). \quad (\text{D.5})$$

Proof. We prove Equation (D.4).

$$|N_G(u) \cap C| = |C| - |NN_C(u)| - 1 \stackrel{(D.1)}{\geq} |C| - \ell \cdot 2^{\log^{0.8} n} - 1 \stackrel{(D.3)}{\geq} 99 \cdot 2^{\log^{0.8} n} \cdot \ell - 1 \stackrel{(D.1)}{\geq} \Omega(|NN_C(u)|).$$

We prove Equation (D.5).

$$|N_G(u) \cap N_G(v) \cap C| \geq |C| - |NN_C(u)| - |NN_C(v)| - 1 \stackrel{(D.1)}{\geq} |C| - 2 \cdot 2^{\log^{0.8} n} \cdot \ell - 1 \stackrel{(D.3)}{\geq} \Omega(|C|).$$

□

Preprocessing. For each $u, v \in C$ such that u and v are non-neighbors in G , u and v agree on a random node $r_{u,v}$ from the common neighbors in C , i.e., $(N_G(u) \cap N_G(v)) \cap C$. This step can be implemented as follows.

- For all $u \in C$, and for all $v \in C - N_G[u]$ where $u < v$ (the id of u is less than the id of v),
 1. u sends a message requesting to connect to v to $\Theta(\log n)$ random neighbors $N_G(u) \cap C$.
 2. An intermediate node $w \in N_G(u)$, upon receiving the request from u , sends to v if $v \in N_G(w) \cap C$.
 3. If node v receives the request of u from the intermediate nodes, then v selects one node w arbitrarily and replies to w so w sends the confirmation back to u .

Correctness. It is enough to prove that the preprocessing is *successful*; that is for every pair of non-neighbors u, v in C , u and v agree on the node $r_{u,v} \in N_G(u) \cap N_G(v) \cap C$ as an intermediate node. By design, this happens if and only if, during the preprocessing, one of u 's messages requesting to v was sent to $N_G(u) \cap N_G(v) \cap C$. Fix a node $u \in C$, the probability that a random message its request is sent to $N_G(u) \cap N_G(v) \cap C$ is $\frac{|N_G(u) \cap N_G(v) \cap C|}{|C|} \stackrel{(D.5)}{\geq} \Omega(1)$. Therefore, after $\Theta(\log n)$ trials and by the union bounds, the setup is successful with high probability.

Clique Simulation. We simulate a message from u to v in clique for all $u, v \in C$ as follows. If there is an edge (u, v) , then we send the same message directly. Otherwise, u sends the message to $r_{u,v}$ for which $r_{u,v}$ forwards to v . This step can be implemented in CONGEST easily.

Complexity of the Simulation. We prove that each clique round can be implemented in $O(\log n)$ rounds of clique simulation. For each u, v where $u < v$ and $v \notin N_G(u) \cap C$, we define a 2-path (a path of length 2), denoted by $P_{u,v} := (u, r_{u,v}, v)$. Let $\mathcal{P} := \{P_{u,v} : u < v, v \notin N_G(u) \cap C\}$. For any edge $e \in E_G(C, C)$, the *congestion*, denoted by $c(e)$, is the number of paths $P_{u,v}$ that contains $e \in E_G(C, C)$. We show that the maximum congestion $\max_e c(e)$ over all edges is $O(\log n)$ with high probability.

Fix an edge $e = (x, y) \in E_G(C, C)$. If a 2-path P contains e , then e is either the first or the second edge of P . Let $\Gamma_1(x, y) := \{w \in C - N_G[x] : \exists P_{x,w} = (x, r_{x,w}, w) \in \mathcal{P}, r_{x,w} = y\}$ be the set of all 2-paths in $\mathcal{P}$ that contains (x, y) as the first edge. Similarly let $\Gamma_2(x, y)$ be the set of all 2-paths in $\mathcal{P}$ that contains (x, y) as the second edge. By definitions, $c(e) = |\Gamma_1(x, y)| + |\Gamma_2(x, y)|$.

Next, we show that $|\Gamma_1(x, y)| \leq O(\log n)$ and $|\Gamma_2(x, y)| \leq O(\log n)$ with high probability. We focus on bounding the size of $\Gamma_1(x, y)$. By definition of $\Gamma_1(x, y)$, (x, y) belongs to the first edge of a path $P_{x,w} = (x, r_{x,w}, w) \in \mathcal{P}$ if w is a non-neighbor of x where x and w agree on $r_{x,w} = y$ as an intermediate node. Thus, $|\Gamma_1(x, y)|$ is the number of non-neighbors w of x whose intermediate node is selected to be y . Let $I[r_{x,w} = y]$ be an indicator random variable. We have

$$|\Gamma_1(x, y)| = \sum_{w \in NN_C(x)} I[r_{x,w} = y]$$

For all non-neighbors $w \in NN(x)$, since $r_{x,w}$ is a random node in $N_G(x) \cap N_G(w) \cap C$, the probability

that $r_{x,w} = y$ is

$$\begin{aligned}\mathbb{P}(r_{x,w} = y) &= \begin{cases} 0 & \text{if } y \notin N_G(w) \cap C, \\ \frac{1}{|N(x) \cap N(w) \cap C|} & \text{otherwise} \end{cases} \\ &\leq \frac{1}{|N(x) \cap N(w) \cap C|} \\ &\leq O\left(\frac{1}{|N(x) \cap C|}\right).\end{aligned}$$

The last inequality follows since

$$|N(x) \cap N(w) \cap C| \stackrel{(D.5)}{\geq} \Omega(|C|) \geq \Omega(|N(x) \cap C|).$$

Therefore, in expectation,

$$\begin{aligned}\mathbb{E}[|\Gamma_1(x, y)|] &= \sum_{w \in NN_C(x)} \mathbb{P}[r_{x,w} = y] \\ &\leq |NN_C(x)| \cdot O\left(\frac{1}{|N(x) \cap C|}\right) \\ &\stackrel{(D.4)}{\leq} O(1).\end{aligned}$$

Similarly, we can also prove that $\mathbb{E}[|\Gamma_2(x, y)|] \leq O(1)$. By Chernoff bound, we conclude that $c(x, y) \leq O(\log n)$ with high probability.

Round Complexity. The round complexity for the preprocessing is $O(\log n)$ factor larger than one iteration of clique simulation.

D.8 Parallel Implementation in Matrix Multiplication Work

In this section, we prove the main lemma for solving SSVC in PRAM model in matrix multiplication work and subpolynomial depth.

Lemma D.67. *There is a randomized PRAM algorithm that solves SSVC in $n^{\omega+o(1)}$ work and $n^{o(1)}$ depth.*

Lemma D.67 can be proved by the following two implementations of ISOLATINGCUTS and MinNNCC.

Lemma D.68. *There is a randomized PRAM algorithm that solves an MinNNCC instance (G, C, ℓ) (Definition D.40) in $\tilde{O}(|C|^\omega)$ work and $\tilde{O}(1)$ depth.*

Lemma D.69. *There is a randomized PRAM algorithm that solves an ISOLATINGCUTS instance (G, C, T) (Definition D.39) in $\tilde{O}(|C|^\omega)$ work and $\tilde{O}(1)$ depth.*

Proof. We use the algorithm [Lov79] that solves maximum bipartite matching in matrix multiplication time and work. We can turn that algorithm into an s-t vertex connectivity algorithm by using Lemma D.88 (it reduces s-t vertex connectivity to minimum vertex cover, which can be found by first finding a maximum bipartite matching, then run reachability). Now s-t vertex connectivity can be solved in $\tilde{O}(n^\omega)$ work and $\tilde{O}(1)$ depth; we can plug it in Lemma D.43 to finish the proof. $\square$

We prove Lemma D.68 in Section D.8.1. We are now ready to prove Lemma D.67.

Proof of Lemma D.67. We use Algorithm 2. The correctness is implied by Lemma D.41. So, we only need to calculate the complexity. We run the algorithm $\hat{O}(1)$ times to boost the probability of succeeding and take the minimum cut as output.

There are $\log n$ outer loops. Inside each loop, for each C , the work for running ISOLATINGCUTS and MinNNCC is $|C|^\omega$ and the depth is $\tilde{O}(1)$. We run the algorithm for all $C \in \mathcal{C}$ simultaneously so the depth

does not change. The total work is $\tilde{O}(\sum_{C \in \mathcal{C}} |C|^\omega) = \tilde{O}(n^\omega)$ because $\sum_{C \in \mathcal{C}} |C| = \tilde{O}(n)$ according to [Lemma D.31](#). $\square$

D.8.1 Proof of [Lemma D.68](#)

This section is devoted to proving [Lemma D.68](#). Recall the notations introduced in [Section D.5](#). We are given the MinNNCC instance (G, C, ℓ) ([Definition D.40](#)). In this section, we describe a sequential algorithm for simplicity, but these algorithms can be easily parallelized. We denote $T_{\text{MM}}(n, k, r)$ as the number of field operations needed to multiply an $n \times k$ matrix with a $k \times r$ matrix.

Lemma D.70 ([\[Chr+20\]](#)).

$$T_{\text{MM}}(n, k, r) = O(T_{\text{MM}}(k, n, r)) = O(T_{\text{MM}}(n, r, k)).$$

Matrix Data Structures. We will use the matrix data structure such that the $\text{rank}(M)$ can be maintained quickly under low-rank updates.

Definition D.71. Let n, a be natural numbers where $a < n$. A matrix $V \in \mathbb{R}^{n \times a}$ is a *selector matrix* if V is a binary matrix, every column has exactly one 1's and every row has at most one 1's.

Selector matrices can be used to construct a larger matrix. To illustrate, let

$$\mathbf{u}_j = \begin{bmatrix} u_{j1} \\ u_{j2} \\ \vdots \\ u_{jn} \end{bmatrix} \quad \text{and} \quad \mathbf{u}_k = \begin{bmatrix} u_{k1} \\ u_{k2} \\ \vdots \\ u_{kn} \end{bmatrix}$$

where $\mathbf{u}_j$ will define the values in the j -th column and $\mathbf{u}_k$ will define the values in the k -th column. and

$$\mathbf{v}_j = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \quad \text{and} \quad \mathbf{v}_k = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

where the 1 in $\mathbf{v}_j$ is in the j -th position and the 1 in $\mathbf{v}_k$ is in the k -th position.

Define a selector matrix $V = \begin{bmatrix} \mathbf{v}_j & \mathbf{v}_k \end{bmatrix}$. Also, define $U = \begin{bmatrix} \mathbf{u}_j & \mathbf{u}_k \end{bmatrix}$. We obtain a larger matrix by

$$UV^T = \begin{bmatrix} \mathbf{u}_j & \mathbf{u}_k \end{bmatrix} \begin{bmatrix} \mathbf{v}_j^T \\ \mathbf{v}_k^T \end{bmatrix} = \begin{bmatrix} 0 & \cdots & u_{j1} & \cdots & u_{k1} & \cdots & 0 \\ 0 & \cdots & u_{j2} & \cdots & u_{k2} & \cdots & 0 \\ \vdots & & \vdots & & \vdots & & \vdots \\ 0 & \cdots & u_{jn} & \cdots & u_{kn} & \cdots & 0 \end{bmatrix}.$$

We will represent column updates of M of the form $M + UV^T$ where V is a selector matrix.

Lemma D.72. *There is a data structure $\mathcal{M}$ that supports the following operations.*

- $\mathcal{M}.\text{CONSTRUCT}(M, k)$ where $M \in \mathbb{R}^{n \times n}$: Preprocess and construct the data structure using M and integer $k \leq n$ in $O(n^\omega)$ time. With high probability, $\mathcal{M}$ correctly operates ISRANKGTK defined next.
- $\mathcal{M}.\text{ISRANKGTK}(U, V)$ where $U, V \in \mathbb{R}^{n \times a}$, $a \leq n$, and V is a selector matrix: decide if $\text{rank}(M + UV^T) \geq k$ in $O(T_{\text{MM}}(n, a, a))$ time.

We will use low-rank update techniques as in [\[BNS19; San07\]](#), but their proofs are described using sequential updates. For our purpose, we prove the batch version and perform the updates in one shot.

Proof. We explain the construction first.

$\mathcal{M}.\text{CONSTRUCT}(M, k)$. We are given $M \in \mathbb{R}^{n \times n}$ and an integer $k \leq n$ for preprocessing. The data structure constructs the following matrix:

$$f(M) = \begin{bmatrix} M & X & 0 \\ Y & 0 & I \\ 0 & I & I^{(q)} \end{bmatrix} \quad (\text{D.6})$$

where $I^{(q)}$ is an n -by- n binary matrix whose first q diagonal entries are 1, X and Y are independent and uniformly random matrices where each entry is randomly chosen from some field extension of size $\Omega(n^2)$. Sankowski ([San07], Lemma 4.1, Theorem 4.1) showed that the matrix $f(M)$ is full rank w.h.p. if and only if $\text{rank}(M) \geq n - q$.

By setting $q = n - k$, we can decide if $f(M)$ has rank $\geq k$ by checking if $f(M)$ is full rank, which can be checked by computing its determinant. We fix $q = n - k$ from now on.

In preparation for the query operation where we will be given two matrices $U, V \in \mathbb{R}^{n \times a}$. We can decide if $\text{rank}(f(M + UV^\top)) \geq k$ by computing the determinant of $f(M + UV^\top)$. That is,

Lemma D.73. *Given $U, V \in \mathbb{R}^{n \times a}$ where $a \leq n$, $\text{rank}(f(M + UV^\top)) \geq k$ if and only if $f(M + UV^\top)$ is full rank with high probability.*

We also precompute $\det(f(M))$ and $f(M)^{-1}$ in $O(n^\omega)$ time. For simplicity, we assume that $f(M)$ is invertible. This assumption can be removed; see Theorem C.10 of [San07].

$\mathcal{M}_k.\text{ISRANKGTK}(U, V)$. We are given where $U, V \in \mathbb{R}^{n \times a}$, $k, a < n$, and V is a selector matrix. By Lemma D.73, it is enough to compute the determinant $\det(f(M + UV^\top))$. To do so, we use the matrix determinant lemma.

Lemma D.74 (Matrix Determinant Lemma). *Let $A \in \mathbb{R}^{n \times n}$ be an invertible matrix and $U, V \in \mathbb{R}^{n \times a}$.*

$$\det(A + UV^\top) = \det(A) \det(I + V^\top A^{-1}U). \quad (\text{D.7})$$

Let U_0 be a matrix and $V_0^\top$ be a selector matrix such that

$$f(M) + U_0 V_0^\top = f(M + UV^\top). \quad (\text{D.8})$$

Therefore, we can compute $\det(f(M + UV^\top))$ as follows.

$$\begin{aligned} \det(f(M + UV^\top)) &\stackrel{(\text{D.8})}{=} \det(f(M) + U_0 V_0^\top) \\ &\stackrel{(\text{D.7})}{=} \det(f(M)) \det(I_a + V_0^\top (f(M))^{-1} U_0). \end{aligned}$$

Since we precomputed $\det(f(M))$ during the preprocessing, it remains to compute

$$\det(I + V_0^\top (f(M))^{-1} U_0),$$

which can be computed in time

$$O(n \cdot a + T_{\text{MM}}(a, n, a) + a^\omega) = O(T_{\text{MM}}(a, n, a)).$$

Indeed, we break down the computation as follows.

1. Compute $Q_0 := V_0^\top f(M)^{-1}$ in $O(n \cdot a)$ time. Since V_0 is a selector matrix, the matrix $V_0^\top f(M)^{-1}$ can be obtained by reading corresponding rows $f(M)^{-1}$ selected by the 1's entries in $V_0^\top$ in $O(n \cdot a)$ time. The matrix $(f(M))^{-1}$ was stored at the preprocessing step.
2. Compute $Q_1 := Q_0 U_0$ in $O(T_{\text{MM}}(a, n, a))$ time.
3. Compute $Q_2 := I_a + Q_1$ in $O(a^2)$ time.
4. Compute $\det(Q_2) = \det(I_a + V_0^\top (f(M))^{-1} U_0)$ in $O(a^\omega)$ time.

This concludes the query operation in $O(T_{\text{MM}}(a, n, a))$ time. $\square$

Convex Embedding. Let $\mathbb{F}$ be a finite field. For $k \geq 0$, the space $\mathbb{F}^k$ is a k -dimensional linear space over $\mathbb{F}$. Let $X = \{x_1, \dots, x_n\}$ be a finite set of points within $\mathbb{F}^k$. The *affine hull* of X is defined as

$$\text{aff}(X) = \left\{ \sum_{i=1}^k c_i x_i : x_i \in X \text{ and } \sum_{i=1}^k c_i = 1 \right\}.$$

The rank of X , represented as $\text{rank}(X)$, is one plus the dimension of $\text{aff}(X)$. Specifically, if $\mathbb{F} = \mathbb{R}$, we refer to the *convex hull* of X , denoted $\text{conv}(X)$. For any sets V, W , any function $f : V \rightarrow W$, and any subset $U \subseteq V$, we denote $f(U) := \{f(u) : u \in U\}$.

Definition D.75 (Convex X -embedding [LLW88]). For any $X \subset V$, a convex X -embedding of a graph $G = (V, E)$ is a function $f : V \rightarrow \mathbb{R}^{|X|-1}$ such that for each $v \in V \setminus X$, $f(v) \in \text{conv}(f(N_G(v)))$.

For efficient construction, [LLW88; CR94] define convex-embedding over finite field $\mathbb{F}$. In particular, they construct the convex X -embedding over the field of integers modulo a prime p , $\mathbb{Z}_p$ by fixing a random prime number $p \in [n^5, n^6]$, and choosing a random nonzero coefficient function $c : E \rightarrow (\mathbb{Z}_p \setminus \{0\})$ on edges. This construction yields a function $f : V \rightarrow (\mathbb{Z}_p^{|X|-1})$ called *random modular X -embedding*.

Lemma D.76 (Theorem 4.13 of [LLW88]). Let t be a fixed node in a graph $G = (V, E)$. A random modular $N(t)$ -embedding f can be constructed in $\tilde{O}(n^\omega)$ time. With high probability, for every node $s \in V - N[r]$, we have $\kappa_G(s, t) = \text{rank}(f(N(s)))$.

Using matrix data structure (Lemma D.72), we can construct a fast connectivity data structure as follows.

Lemma D.77. Let t be a fixed node in a graph $G = (V, E)$. Suppose we have an oracle that can list all the elements in the symmetric difference $N(s) \Delta N(s')$ for any pair (s, s') up to a elements in $\tilde{O}(a)$ time.

- A random modular $N(t)$ -embedding f can be constructed in $\tilde{O}(n^\omega)$ time.
- With additional $\tilde{O}(n^\omega)$ preprocessing time on another fixed node $s \in V - N[t]$ and parameter k , we have the following with high probability: for all $s' \in V - N[t]$, we can decide if $\kappa_G(s', t) \geq k$ in $O(T_{\text{MM}}(n, a, a))$ time where $a = |N(s) \Delta N(s')|$.

Proof. Let f be a random modular $N_G(t)$ -embedding f , which can be constructed in $\tilde{O}(n^\omega)$ using Lemma D.76. We can represent $f(V)$ as a matrix $M_V := [\dots f(v) \dots]$ where each column corresponds to a vector $f(v)$. For all $U \subseteq V$, we denote M_U as a submatrix of M_V when we restrict the columns of M_V to those in $f(U)$.

We will use the matrix data structure $\mathcal{M}$ in Lemma D.72. We first use $M := M_{N(x)}$ where x is a fixed node and k is a parameter for the construction of $\mathcal{M}$ in $O(n^\omega)$ time. Next, we are given query $s' \in V - N[t]$, where $|N(s) \Delta N(s')| = a' \leq a$. Let $W = N(s')$, and so we can write $M_W := M + UV^T$ where $U \in \mathbb{R}^{k \times a'}$ corresponds to the column differences in the embedding, and $V \in \mathbb{R}^{k \times a'}$ is a selector matrix. Therefore, we can check if $\text{rank}(M_W) \geq k$ by calling $\mathcal{M}.\text{ISRANKGTK}(U, V)$, which can be done in $O(T_{\text{MM}}(k, a, a))$ time by Lemma D.72. $\square$

We are ready to present the PRAM algorithm and prove Lemma D.68.

PRAM Algorithm. We are given $G = (V, E)$, $C \subseteq V$ and ℓ as the inputs to MinNNCC satisfying the guarantees described in Definition D.40.

1. Use Theorem D.28 to compute $sk_z(N(u))$ for every $u \in C$ where $z = 2^{\log^{0.8} n \ell}$
2. Let X be a set of $\hat{O}(|C|/\ell)$ random vertices sampled in C .
3. Apply Lemma D.50 using G, C, ℓ and X as parameters to obtain $C' \subseteq N_G(C)$. Let K be the fixed number in the lemma statement.
4. Let $G' = G[C \cup C']$.
5. Let G'' be G' after adding a sink t and the set of edges from t to every node in C' .

6. Binary search on k until we obtain $k = \min_{x \in X} \kappa_{G''}(x, t)$ on the following problem: decide if $\min_{x \in X} \kappa_{G''}(x, t) \geq k$.
 - (a) Given the guess value k , apply [Lemma D.77](#) using t as the fixed node, and the sketching sk_z as an oracle that can list $N_G(s) \triangle N_G(s')$ for any pair (s, s') up to $\hat{O}(\ell)$ elements in $\hat{O}(\ell)$ work.
 - i. Construct a random modular $N_{G''}(t)$ -embedding f . Let $s \in C$ be the other fixed node and use parameter k for the preprocessing.
 - ii. For all $x \in X$, decide if $\kappa_{G''}(x, t) \geq k$ using [Lemma D.77](#).
7. returns the corresponding minimum cut.

Correctness. Let L^* be the minimizer of $\min_{L' \subseteq C: L' \neq \emptyset} |N_G(L')|$. By [Lemma D.50](#), we know that G'' have the same minimizer as in G for all $x \in X$. That is, for all $x \in X$, $\min_{L' \subseteq C: x \in L'} |N_G(L')| = \min_{L' \subseteq C: x \in L'} |N_{G'}(L')| + K$. Since we sample $\hat{O}(|C|/\ell)$ random vertices into X , there is $x \in L^*$ with high probability. Therefore, by the correctness of [Lemma D.77](#), we also obtain the correct minimizer with high probability.

Complexity. The preprocessing steps and sparsification can be done in $\hat{O}(m)$ work. The main bottleneck is to binary search on k in step 6. In this step, the running time follows from [Lemma D.77](#). That is,

$$\tilde{O}(|C|^\omega) + \hat{O}\left(\frac{|C|}{\ell}\right) \cdot \tilde{O}(T_{\text{MM}}(n, \ell, \ell)) = \hat{O}(|C|^\omega).$$

The first term on LHS follows from the preprocessing time since $|V(G'')| = \tilde{O}(|C|)$ by [Lemma D.50](#). The second term on LHS follows from $|X|$ calls of $\kappa_G(s', s) \geq k$ queries of [Lemma D.77](#).

D.9 Communication and Streaming

D.9.1 A Simple Communication and Streaming algorithm

In this section, we will provide a simple communication algorithm for SSVC, and its direct implementation in streaming model, i.e., the following two theorems.

Lemma D.78. *There is a randomized communication protocol solving SSVC in $\tilde{O}(n^{1.5})$.*

Lemma D.79. *If there is a randomized semi-streaming algorithm solving Bipartite Maximum Matching (BMM) in $P(n)$ passes, then there is a randomized streaming algorithm solving SSVC in $\tilde{O}(n^{1.5})$ space and $O(P(n))$ passes.*

We first give a schematic algorithm as described in [Algorithm 3](#), and we will show its correctness, and how to implement it in the communication and streaming setting. According to [Remark D.27](#), we assume public randomness.

The following lemma shows the correctness of [Algorithm 3](#).

Lemma D.80. *At the end of [Algorithm 3](#), S^* is the minimum t -sink vertex cut in G .*

Proof. Suppose (L, S, R) is a minimum t -sink vertex cut, suppose $|L| \geq \sqrt{n}/3$, then w.h.p. $P \cap L \neq \emptyset$, thus, S^* has size at most $|S|$ according to [Algorithm 3](#). In order to prove $|S^*| = |S|$, we first need to prove $\kappa_u \geq \kappa_G(u, t)$ for any u . Let us first prove some helpful claims.

For any arbitrary $u \in L$, let us consider the loop for u in [Algorithm 3](#).

Claim D.81. *W.h.p., for any $v_1, v_2 \in V$ such that $v_1 \in V'$, $(v_1, v_2) \in E$, $(v_1, v_2) \notin E'$ (which we say (v_1, v_2) is missing) iff. one of the following events happen: (i) for any $v \in V'$, we have $(v, v_2) \in E$, (ii) $(u, v_2) \in E$, $v_1 \neq u$.*

Algorithm 3: Schematic Algorithm for SSVc

Data: An undirected graph $G = (V, E)$ and a vertex $t \in V$.
Result: A minimum t -sink vertex cut (L, S, R) .

- 1 Let $s = \lceil \sqrt{n} \rceil$ in **Theorem D.28**, calculate $sk_v = sk_s(N_G(v))$ for any $v \in V$;
- 2 Let $A = V - N_G(t)$;
// Suppose (L, S, R) is a minimum t -sink vertex cut.
// Phase 1: $|L| \geq \sqrt{n}/3$
- 3 Sample $\lceil 100\sqrt{n} \log n \rceil$ nodes $P \subseteq A$ uniformly at random *// w.h.p. $P \cap L \neq \emptyset$*
- 4 $S^* \leftarrow V$;
- 5 **foreach** $v \in P$ **do**
- 6 Find the minimum (v, t) vertex cut in G denoted by S_v ;
- 7 Let $S^* = S_v$ if $|S_v| < |S^*|$;
- 8 *// Phase 2: $|L| < \sqrt{n}/3$*
- 8 **foreach** $u \in A$ **do**
- 9 Maintain a virtual graph $G' = (V, E')$ and a set $V' \subseteq A$. Initially $E' = \emptyset, V' = \emptyset$;
- 10 **foreach** $v \in A$ **do**
- 11 Let $X = N_G(u) - N_G(v)$ and $Y = N_G(v) - N_G(u)$, which is calculated by the recovery algorithm of **Theorem D.28**;
- 12 **if** the recovery algorithm does not return $\perp$ **then**
- 13 Add v to V' ;
- 14 Add (u, x) to E' for any $x \in X$;
- 15 Add (v, x) to E' for any $x \in Y$;
- 16 Among all L with $u \in L \subseteq V'$, find one with the smallest $|N_{G'}(L)|$ denoted as L_u ;
- 17 $\kappa_u \leftarrow |N_{G'}(L_u)| + |N_G(u)| - |N_{G'}(u)|$ *// w.h.p. $\kappa_u = |N_G(L_u)|$*
- 18 If there exists u such that $\kappa_u < |S^*|$, let $S^* = N_G(L_u)$ where κ_u is the smallest;
- 19 **return** S^*

Proof. “ $\Leftarrow$ ” **Algorithm 3** only add edges to x where $u, v \in V'$ differs by the neighborhood relationship to x , so (i) implies (v_1, v_2) is missing; if $v_1 \neq u$, then (v_1, v_2) will only be added to E' when $(u, v_2) \notin E$, so (ii) implies (v_1, v_2) is missing. “ $\Rightarrow$ ”. If both (i),(ii) do not happen, then either (i) $v_1 = u$, there exists $v \in V'$ such that $(v, v_2) \notin E$, then v_2 is in the difference between A_u, A_v , so (u, v_2) is not missing, or (ii) $v_1 \neq u$, then $(u, v_2) \notin E$, so v_2 is in the difference between A_u, A_{v_1} , (u, v_2) is not missing. $\square$

Claim D.82. *W.h.p., for any $L \subseteq V'$, $N_G(L) - N_{G'}(L) = N_G(u) - N_{G'}(u)$.*

Proof. Notice that for any $u \in L \subseteq V'$, a vertex $v \in N_G(L) - N_{G'}(L)$ iff. $\forall w \in V', (w, v) \in E$. Proof: “ $\Rightarrow$ ”, if $v \in N_G(L) - N_{G'}(L)$, then (u, v) is missing, which means (i) happens. “ $\Leftarrow$ ”, it implies (w, v) is missing for any $w \in V'$.

Based on the claim above, we prove $N_G(L) - N_{G'}(L) = N_G(u) - N_{G'}(u)$ as follows. “ $\subseteq$ ” if $w \in N_G(L) - N_{G'}(L)$, then $\forall w \in V', (w, v) \in E$, so $w \in N_G(u)$, and $w \notin N_{G'}(u)$ (otherwise $w \in N_{G'}(L)$, which leads to $w \notin N_G(L) - N_{G'}(L)$). “ $\supseteq$ ” if $w \notin N_G(L) - N_{G'}(L)$, then there exists $v \in V' - L$ such that $(v, w) \notin E$, then (u, w) is not missing and $w \notin N_G(u) - N_{G'}(u)$. So we have $|N_G(L)| = |N_{G'}(L_u)| + \deg_G(u) - \deg_{G'}(u)$. C_u correctly computes the size of $N_G(L_u)$. $\square$

Now we are ready to prove that for any $u \in A$ we have $\kappa_u \geq \kappa_G(u, t)$. $\kappa_u = N_{G'}(L_u) + N_G(u) - N_{G'}(u) = N_G(L_u)$ according to **Claim D.82**, thus, $\kappa_u \geq \kappa_G(u, t)$.

Now suppose $|L| < \sqrt{n}/3$. We have $|S^*| \geq |S|$ according to the fact that $\kappa_u \geq \kappa_G(u, t)$ for any u . Suppose $u \in L$. Consider the loop for u in **Algorithm 3**. According to **Lemma D.26** and the fact that V' includes all nodes v with $|N_G(u) \triangle N_G(v)| \leq \lceil \sqrt{n} \rceil$ w.h.p., we have $L \subseteq V'$. Since $N_{G'}(L_u)$ and $N_G(L_u)$ differs by a fixed value according to **Claim D.82**, $\kappa_u \leq |N_G(L)|$. Thus, $|S^*| = |S|$. $\square$

Now we prove the complexity and our main lemmas of this section.

Proof of Lemma D.79. The correctness is implied by Lemma D.80. Now we show the complexity.

Before Phase 1. $\mathbf{V}$ can be sampled without accessing the stream. The sketches can be computed in $\tilde{O}(n\sqrt{n})$ space and 1 pass according to Remark D.29. A can be computed with $O(n)$ space and 1 pass.

Phase 1. P is sampled without accessing the stream. We simulate the (S, T) -vertex connectivity algorithm on the stream for any $u \in P$. We do all of them in parallel, using $\tilde{O}(n|P|)$ space and $P(n)$ passes. Remember that $\tilde{O}(n|P|) = \tilde{O}(n\sqrt{n})$.

Phase 2. All calculations of Phase 2 are done without accessing the stream, and they will use $\tilde{O}(n^{1.5})$ space since the number of edges added to G' for each loop in Algorithm 3 is at most $\sqrt{n}$.

In summary, we use $\tilde{O}(n^{1.5})$ space and $O(P(n))$ passes. □

Proof of Lemma D.78. We use the folklore simulation of a streaming algorithm using a communication protocol. Alice and Bob simulate the streaming algorithm in the previous proof on the data stream where the stream first goes through all of Alice's edges, and then Bob's edges. The memory space of the streaming algorithm (which is $\tilde{O}(n^{1.5})$) is sent to Bob, and Bob continues simulating the streaming algorithm.

We can simulate the algorithm before Phase 1 and Phase 2 using the technique above, which requires $\tilde{O}(n^{1.5})$ communication. □

D.9.2 Lower Bounds

In this section, we will prove a communication lower bound for VC.

Lemma D.83. *In the two-party communication model, any randomized algorithm solving VC needs $\Omega(n^{1.5})$ bits of communication.*

It directly implies a streaming lower bound by standard reduction from communication to streaming.

Corollary D.84. *In the graph streaming model, any randomized algorithm solving VC in $P(n)$ pass needs $\Omega(n^{1.5}/P(n))$ space.*

Proof. As in the last section, we try to simulate any streaming algorithm in the following way: the stream first goes through all of Alice's edges, and then Bob's edges. The memory space of the streaming algorithm is sent to Bob after Alice's edges are all presented, and Bob continues simulating the streaming algorithm. Thus, solving VC in $P(n)$ pass needs $o(n^{1.5}/P(n))$ space implies a communication algorithm with $o(n^{1.5})$ communication. □

The following communication problem is where we will reduce VC from. For convenience, suppose $\sqrt{n}$ is an even integer.

$\sqrt{n}$ -OR $\sqrt{n}$ -AND subset problem. Alice is given $\sqrt{n}$ batches, the i -th batch contains $\sqrt{n}$ sets $A_1^{(i)}, A_2^{(i)}, \dots, A_{\sqrt{n}}^{(i)} \subseteq [\sqrt{n}]$, each with size exactly $\sqrt{n}/2$. Bob is given $\sqrt{n}$ batches, the i -th batch contains $\sqrt{n}$ sets $B_1^{(i)}, B_2^{(i)}, \dots, B_{\sqrt{n}}^{(i)} \subseteq [\sqrt{n}]$. Their goal is to determine whether there exists $i \in [\sqrt{n}]$ such that for any $j \in [\sqrt{n}]$, $B_j^{(i)} \subseteq A_j^{(i)}$.

The following lemma shows the randomized communication complexity of the problem above.

Lemma D.85. *There exists $k = \Theta(\sqrt{n})$ such that any randomized protocol solving $\sqrt{n}$ -OR $\sqrt{n}$ -AND k -subset problem requires $\Omega(n^{1.5})$ communication.*

Proof. We will reduce the following problem to $\sqrt{n}$ -OR $\sqrt{n}$ -AND k -subset. Alice is given $\sqrt{n}$ sets $X_1, \dots, X_{\sqrt{n}} \subseteq [n/2]$, Bob is given $Y_1, \dots, Y_{\sqrt{n}} \subseteq [n/2]$, and they want to determine whether or not there exists i such that $X_i \cap Y_i \neq \emptyset$. This problem has $\Omega(n^{1.5})$ lower bound [JKS03; HJ13]. Now we construct a $\sqrt{n}$ -OR

$\sqrt{n}$ -AND k -subset instance as follows. Firstly, define $X'_i = [n/2] - X_i$. Then, we split X'_i into $\sqrt{n}$ blocks and fit each block into $\lceil \sqrt{n}/2 \rceil$, i.e., define the j -th block $X_j^{(i)} = \{k \mid k \in [\sqrt{n}/2], (j-1)(\sqrt{n}/2) + k \in X'_i\}$. We do the same thing to Y_i , let $Y_j^{(i)} = \{k \mid k \in [\sqrt{n}/2], (j-1)(\sqrt{n}/2) + k \in Y_i\}$. Define $A_j^{(i)} = X_j^{(i)} \cup \{\sqrt{n}/2 + k \mid 1 \leq k \leq \sqrt{n}/2 - |X_j^{(i)}|\}$. Similarly, define $B_j^{(i)} = Y_j^{(i)} \cup \{\sqrt{n}/2 + k \mid 1 \leq k \leq \sqrt{n}/2 - |Y_j^{(i)}|\}$. Roughly speaking, we do padding to make sure $|A^{(i)}| = \sqrt{n}/2$, while make sure $Y_j^{(i)} \subseteq X_j^{(i)}$ iff. $B_j^{(i)} \subseteq A_j^{(i)}$. Notice that " $Y_j^{(i)} \subseteq X_j^{(i)}$ for all j " iff. " $Y_i \subseteq X_i$ " iff. " $X_i \cap Y_i = \emptyset$ ". Thus, the $\sqrt{n}$ -OR $\sqrt{n}$ -AND subset problem on $A_j^{(i)}, B_j^{(i)}$ output "yes" iff. the OR of set-disjointness problem on X_i, Y_i outputs "yes".

□

Now we are ready to prove the lower bound for VC.

Proof. Given a $\sqrt{n}$ -OR $\sqrt{n}$ -AND subset instance $A_j^{(i)}, B_j^{(i)}$ for $i, j \in [\sqrt{n}]$, define graph $G = (V, E)$ as follows. We will assume $\sqrt{n} \geq 10$.

$$\begin{aligned}
 V &= U \cup \left(\bigcup_{i \in [\sqrt{n}]} V^{(i)} \right) \\
 U &= \{u_j^{(i)} \mid i, j \in [\sqrt{n}]\} \\
 V^{(i)} &= \{v_j^{(i)} \mid j \in [\sqrt{n}]\} \\
 E &= E_{clique} \cup E_V \cup E_A \cup E_B \\
 E_{clique} &= \{(u, v) \mid u, v \in U, u \neq v\} \\
 E_V &= \bigcup_{i \in [\sqrt{n}]} \{(u, v) \mid u, v \in V_i, u \neq v\} \\
 E_A &= \bigcup_{i \in [\sqrt{n}]} \left(\bigcup_{j \in [\sqrt{n}]} \{(v_j^{(i)}, u_y^{(x)}) \mid x \in [\sqrt{n}], x \neq j, y \in A_x^{(i)}\} \right) \\
 E_B &= \bigcup_{i \in [\sqrt{n}]} \left(\bigcup_{j \in [\sqrt{n}]} \{(v_j^{(i)}, u_y^{(j)}) \mid y \in B_j^{(i)}\} \right)
 \end{aligned}$$

We first prove that if there exists $i \in [\sqrt{n}]$ such that for any $j \in [\sqrt{n}]$, $B_j^{(i)} \subseteq A_j^{(i)}$, then there exists a vertex cut of size $n/2$. We claim that $N(V_i)$ is the vertex cut that we want. Firstly, notice that $V_j \cap N(V_i) = \emptyset$ for any $j \neq i$, so $N(V_i)$ is a valid vertex cut. Further, according to E_A, E_B and the fact that $B_j^{(i)} \subseteq A_j^{(i)}$ for any i , we know $N(V_i) = \{u_y^{(x)} \mid x \in [\sqrt{n}], y \in A_x^{(i)}\}$. Thus, $|N(V_i)| = \sum_{x \in [\sqrt{n}]} |A_x^{(i)}| = n/2$ (remember that $|A_x^{(i)}| = \sqrt{n}/2$).

Now we prove that if for any $i \in [\sqrt{n}]$, there exists $j \in [\sqrt{n}]$, $B_j^{(i)} \not\subseteq A_j^{(i)}$, then any vertex cut has size at least $n/2 + 1$. Taking any two vertices $u, v \in V$, we will prove that there are at least $n/2 + 1$ internal vertex disjoint paths connecting u and v . If $u, v \in V_i$ for some i or $u, v \in U$, then there is an edge between u, v . Thus, we only need to consider the following two cases.

Case 1. If $u \in V_i$ for some i and $v \in U$. We will first construct $n/2 + 1$ paths $p_1^u, \dots, p_{n/2+1}^u$ such that (i) they start with vertex u , (ii) they are vertex disjoint except the starting vertex u . Suppose $j \in [\sqrt{n}]$ satisfies $B_j^{(i)} \not\subseteq A_j^{(i)}$. Suppose $u = v_{j'}^{(i)}$ for some j' . The paths contain three types

1. Type 1: length 1 paths from u to every vertex in $\{u_y^{(x)} \mid x \in [\sqrt{n}], x \neq j', y \in A_x^{(i)}\}$. There are $(\sqrt{n}/2) \cdot (\sqrt{n} - 1)$ many type 1 paths.
2. Type 2: pick arbitrary $\sqrt{n}/2$ vertices in V_i not identical to v_j and $v_{j'}$ (remember that $\sqrt{n} > 10$ and $|V_i| = \sqrt{n}$), denoted as P . Notice that every vertex in P has edges to any node in $\{u_y^{(j')} \mid y \in A_{j'}^{(i)}\}$ according to the definition of E_A . We one-to-one match these nodes to $\{u_y^{(j')} \mid y \in A_{j'}^{(i)}\}$. Each type two path is $(u, v, m(v))$ where $v \in P$ and $m(v)$ is the matched node. There are $\sqrt{n}/2$ type 2 paths.
3. Type 3: Let $z \in B_j^{(i)} \setminus A_j^{(i)}$, such z must exist since $B_j^{(i)} \not\subseteq A_j^{(i)}$. The type 3 path is (u, v_j, u_z^j) .

There are in total $n/2 + 1$ paths, and one can verify that they are vertex disjoint except u . Now since U is a clique, we can find $n/2 + 1$ internal vertex disjoint path from u to v by connecting the $n/2 + 1$ end points of the $n/2 + 1$ paths to v .

Case 2. If $u \in V_i$ and $v \in V_j$ for some $i \neq j$. According to case 1, both u and v has paths $p_1^u, \dots, p_{n/2+1}^u$ with end point set $P_1 \subseteq U$ and $p_1^v, \dots, p_{n/2+1}^v$ with end point set $P_2 \subseteq U$. For each $w \in P_1 \cap P_2$, we connect the two paths p_i^u, p_j^v that both take w as end point, which leads to a path connecting u and v . For the rest path, we match them one-by-one arbitrarily and connect p_i^u, p_j^v by connecting their end point by an edge (remember that U is a clique). Thus, we get $n/2 + 1$ internal vertex disjoint path from u to v .

Conclusion. The graph G can be constructed with $O(\log n)$ communication: $E_{\text{clique}}, E_V, E_A$ is known by Alice and E_B is known by Bob. The value answer to VC solves the $\sqrt{n}$ -OR $\sqrt{n}$ -AND subset problem. According to [Lemma D.85](#), VC has lower bound $\Omega(n^{1.5})$. □

D.10 Reductions Between Problems and Proofs of Main Theorems

For convenience, all the functions appearing in this section (for example, $W(m, n)$ in [Lemma D.88](#)), we assume they are *smooth*, in the sense that $W(Km, Kn) = \text{poly}(K) \cdot W(m, n)$.

From (S, T) -vertex connectivity to BMC. We describe the reduction from the (S, T) vertex connectivity problem to BMC (bipartite minimum vertex cover) using the construction from [\[Sch03\]](#)(Section 16.7c), see also [\[Bli+22; Bra+20a\]](#).

Definition D.86 (Construction). Let $G = (V, E)$ be a graph with two vertex sets $A, B \subseteq V$. We construct the bipartite matching instance as follows. We may assume that $A \cap B = \emptyset$. We define the bipartite graph $H = (V', V'', E')$ as follows. For each $v \in V - A$, we add a new vertex v' to V' , and for each $v \in V - B$ we add a new vertex v'' to V'' . We define E' as the set of pairs (u', v'') with $u \in V - A, v \in V - B$ with the property that $(u, v) \in E$ or $u = v$. We define the capacity of each $v' \in V'$ to be ∞ if $v \in B$, and 1 otherwise. Similarly, we define the capacity of each $v'' \in V''$ to be ∞ if $v \in A$ and 1 otherwise.

We can extract a min (A, B) -separator in G from the minimum vertex cover of H as follows. For any $U \subseteq V$, we denote $U' = \{v' \mid v \in U\}$, and $U'' = \{v'' \mid v \in U\}$.

Lemma D.87 ([\[Sch03\]](#)(Section 16.7c)). *Let $U \subseteq V - B, W \subseteq V - A$ be such that $D = U \cup W$ is a minimum vertex cover in H . Then, the set $C = (U \cap A) \cup (U \cap W) \cup (W \cap B)$ is a min (A, B) -separator in G .*

Observe that the bipartite graph H in [Definition D.86](#) has $2n - |A| - |B| = O(n)$ vertices. Also, the construction above contains some nodes with ∞ capacity. We can reduce to unit capacity WLOG as follows. First, we replace ∞ with n . For each node with capacity n , we simulate it to behave as if it is a duplicate of n unit-capacity nodes. With appropriate implementation, we apply vertex reduction algorithm in Appendix B of [\[Ass+22\]](#) to reduce to $O(n)$ nodes while preserving the maximum matching. Roughly speaking, the vertex reduction algorithm greedily finds $O(\log n)$ maximal matchings and preserves only the nodes in all these maximal matchings. Finding a maximal matching in a node-capacity graph can be done efficiently.

In conclusion, we have the following.

Lemma D.88. *If the BMC problem with n vertices, m vertices and diameter D can be solved in*

- (PRAM) $W(m, n)$ work and $D(m, n)$ depth,
- (CONGEST) $T(m, n, D)$ rounds,
- (Communication) $f(n)$ bits of communication,
- (Streaming) $s(n)$ space, $p(n)$ passes,

then (S, T) -vertex connectivity problem can be solved in

- (PRAM) $O(W(m, n))$ work and $O(D(m, n))$ depth,
- (CONGEST) $O(T(m, n, D))$ rounds,
- (Communication) $O(f(n))$ bits of communication,
- (Streaming) $O(s(n))$ space, $O(p(n))$ passes,

From (global) vertex connectivity to single sink vertex connectivity.

Lemma D.89 (Global to Single Source). *Suppose there exist algorithms solving SSSVC with the following complexities in different models.*

1. $T(m, n)$ work and $D(m, n)$ depth in the PRAM model.
2. $R(m, n, D)$ rounds in CONGEST model.
3. $C(n)$ communication bits in the two-party communication model,
4. $S(n)$ space and $P(n)$ passes in the streaming model,

Then there exist algorithms solving VC in

1. $\tilde{O}(T(m, n))$ work and $\tilde{O}(D(m, n))$ depth in the PRAM model.
2. $\tilde{O}(R(m, n, D))$ rounds in CONGEST model.
3. $\tilde{O}(C(n))$ communication bits in the two-party communication model,
4. $\tilde{O}(S(n))$ space and $\tilde{O}(P(n))$ passes in the streaming model,

Proof. The general idea is to first compute the minimum degree δ , then sample $z = 10n \log n / (n - \delta)$ nodes $s_1, \dots, s_z$, and denote $B_i = V - N_G[s_i]$. Now construct a graph $G' = (V', E')$ where

$$V' = \{s\} \cup V' \cup \left(\bigcup_{i \in [z]} B_i \right)$$

here $V' = \{v' \mid v \in V\}$ is a copy of V , and s is a new node.

$$E' = \{(s, v') \mid v' \in V'\} \cup \left(\bigcup_{i \in [z]} \{(v, u) \mid u, v \in B_i, (u, v) \in E\} \cup \{(u', v) \mid u' \in V', v \in B_i, (v, u) \in E\} \right)$$

Now we query the SSSVC on G' and source s .

To prove the correctness, suppose (L, S, R) is a minimum vertex cut of G . We first prove G' has a cut separating s to another node with size $|S|$. We have $\delta \geq |S|$, otherwise, the neighborhoods of the minimum degree node should be a smaller cut. Thus, $|L| + |R| = n - |S| \geq n - \delta$. By sampling $z = 10n \log n / (n - \delta)$ nodes, one of them (suppose it is s_i) will be in R w.h.p. since $|R| \geq |L|$. Thus, $(S \cap B_i) \cup \{u' \mid u \in S \cap N(S)\}$ is a cut in G' which has size $|S|$. Now we prove that $\kappa_{G'}(s, t) \geq |S|$ for any $t \in V'$. If $t \in N[s]$, then $\kappa_{G'}(s, t) = |V'| - 1$. Otherwise, $t \in B_i$ for some i . If there exists $|S|$ internal vertex disjoint path in G from t to s , then the $|S|$ paths corresponds to $|S|$ internal vertex disjoint path in G' from s to t .

To recover the cut for G from the cut for G' , notice that the cut (L, S, R) in G' with $s \in R$ has the property that $L \subseteq B_i$ if L is connected. Thus, S corresponds to a cut in G .

The size of V' is at most $(n + 1) + (n - \delta) \cdot 10n \log n / (n - \delta) \leq n'$.

Now we prove the complexities in different models by showing how to implement the aforementioned algorithm.

PRAM. If $|S| < 0.1n$, the reduction is trivial since $O(\log n)$ sampled nodes suffice to hit a node in R . Thus, we assume $|S| \geq 0.1n$, in which case we have $m = \Omega(n^2)$. The minimum degree can be computed in $O(m)$ time. Sampling can be done in $O(n)$ time. Construct the graph G' cost $\tilde{O}(n^2)$ time. The total work is $\tilde{O}(T(m, n))$. The depth is constant plus the oracle depth.

CONGEST. The implementation for CONGEST is a bit different since we cannot explicitly construct G' . Instead, after sample z sinks $s_1, \dots, s_z$, we run SSSVC on the subgraphs $H_1, \dots, H_z$ defined as H_i containing all edges adjacent to $V - N_G[s_i]$ plus edges from $N_G(s_i)$ to s_i , with the sinks $s_1, \dots, s_z$. Notice that H_i

has the same diameter as the original graph since we are only ignoring edges between $N(s_i)$, which has mutual distances at most 2. We run all the algorithm on H_i simultaneously, which causes dilation at most $R(n, D)$. Now we bound the congestion, i.e., given an arbitrary edge (u, v) , we will bound the number of H_i that includes (u, v) . Specifically, we are going to bound the number of H_i including (u, v) through (i) (u, v) is an edge adjacent to s_i , this can happen at most $O(\log n)$ times w.h.p., (ii) (u, v) is an edge adjacent to $V - N_G[s_i]$, in other words s_i is not a neighborhood of u or is not a neighborhood of v , according to Chernoff bound, both event can happen for at most $O(\log n)$ times since u has degree at least δ and we sample s_i uniformly at random where each node get sampled with probability $10 \log n / (n - \delta)$. Thus, the total congestion is $\tilde{O}(R(m, n, D))$.

Communication. The minimum degree can be computed in $\tilde{O}(n)$ communication by computing $|N_G(v)|$ in $\tilde{O}(1)$ communication. Sampling can be done by Alice and then the results are sent to Bob. The vertex set B_i is known to both Alice and Bob in $\tilde{O}(n - \delta)$ communication by using [Theorem D.28](#) on V and $N_G[s]$, which in total cost $\tilde{O}(n)$ communication. Now edges in G' are split into two parts for Alice and Bob, and they can simulate the SSVC algorithm on G' in $C(n')$ communication.

Streaming. The minimum degree can be computed in one pass and $\tilde{O}(n)$ space. Sampling can be done without accessing the graph. The vertex sets B_i can be computed in one pass and $\tilde{O}(n)$ space (for all i) by [Theorem D.28](#). Then we can simulate the stream for G' by using the stream for G , in $S(n')$ space and $P(n')$ passes. □

From s-t vertex connectivity to (global) vertex connectivity.

Lemma D.90 (s-t vertex connectivity to (global) vertex connectivity). *Suppose there exist algorithms solving VC with the following complexities in different models.*

1. $T(n)$ work and $D(n)$ depth in PRAM model.
2. $C(n)$ communication bits in the two-party communication model,
3. $S(n)$ space and $P(n)$ passes in the streaming model,

Then there exist algorithms solving s-t VC in

1. $O(C(n))$ communication bits in the two-party communication model,
2. $O(S(n))$ space and $O(P(n))$ passes in the streaming model,
3. $O(T(n))$ work and $O(D(n))$ depth in PRAM model.

Proof. By [Lemma D.88](#), we only need to reduce BMC (bipartite minimum vertex cover) to VC.

Given a bipartite graph $G = (A, B, E)$ where $E \subseteq A \times B$, we construct a graph $G' = (A \cup B, E \cup E_A \cup E_B)$ where $E_A = \{(u, v) \mid u, v \in A\}$ and $E_B = \{(u, v) \mid u, v \in B\}$, i.e., we build a clique on both A and B . We run VC on G' , which gives us a minimum vertex cut S . We return (i) S in the case that $|S| < \min(|A|, |B|)$, (ii) the smaller one among A, B in the case that $|S| \geq \min(|A|, |B|)$.

We first show that every vertex cut in G' must be a vertex cover on G . Suppose (L, S, R) is a vertex cut in G' , since there are no edges between L and R , it cannot happen that both $L \cap A$ and $R \cap A$ are non-empty. W.o.l.g., assume $L \cap A$ is non-empty. If $L \cap B$ is non-empty as well, then R cannot contain any node in A or B , which is a contradiction to the fact that R cannot be empty. Thus, $R \cap A = \emptyset$, so we have $L \subseteq A$. Similarly, we have $R \subseteq B$. For every edge in G , it is either adjacent to S , or both of its endpoints are inside A (or inside B). But edges in G' that are totally inside A or B are not edges in G , thus, S is a vertex cover for G .

Then we show that if there is a vertex cover C in G which satisfies that $|C| < \min(|A|, |B|)$, then C must be a vertex cut for G' . W.o.l.g., assume $|A| \leq |B|$. We have that $|C| < |A|$. Let $L = A - C$ and $R = B - C$. We claim that $(L, V - C, R)$ is a vertex cut. It is clear that $L, V - C, R$ form a partition of $A \cup B$. Moreover, if there exists an edge between L and R , then this edge is not covered by C in G , which is a contradiction. Thus, there are no edges between L and R .

Given the last two paragraphs, suppose S^* is the returned set by our reduction algorithm, then either (i) the minimum vertex cover of G is less than $\min(|A|, |B|)$, in which case there exists a vertex cut in G' with the same size of the minimum vertex cover, so S^* cannot be larger, and cannot be smaller since it is a vertex cover itself, or (ii) the minimum vertex cover of G is $\min(|A|, |B|)$, then we returned the correct thing. $\square$

Now we are ready to prove the main theorems.

Proof of Theorem D.1. Given an s-t vertex connectivity algorithm, we can solve SSVC according to Lemma D.45, then we can solve VC using Lemma D.89. $\square$

Proof of Theorem D.2. Given an S-T vertex connectivity algorithm, we first convert it into a subgraph S-T vertex connectivity algorithm by the same reduction described in Section D.11. Then we can solve SSVC according to Lemma D.46, then we can solve VC using Lemma D.89. $\square$

Proof of Theorem D.4. We plug in Theorem D.1 with the work and depth for s-t vertex connectivity. The latter one can be derived from the maximum bipartite matching algorithm of Theorem D.92. To convert maximum bipartite matching algorithm to s-t vertex connectivity algorithm, we first convert it into bipartite minimum vertex cover algorithm by running a reachability on the residual graph to find the minimum vertex cover, then use Lemma D.88. $\square$

Proof of Theorem D.5. We plug in Theorem D.2 with the round complexity for S-T vertex connectivity. The round complexity of S-T vertex connectivity can be derived from Theorem D.93 combined with Lemma D.88. $\square$

Proof of Theorem D.6. We plug in Lemma D.89 with Lemma D.67. $\square$

Proof of Theorem D.3. We combine Lemma D.78 with Lemma D.89, and Lemma D.83. $\square$

Proof of Theorem D.9. We combine Lemma D.79 with Lemma D.89. $\square$

Acknowledgements

The project is partially supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 759557-ALGOCom), and the ERC Starting Grant (CODY 101039914), Dr. Max Rössler, the Walter Haefner Foundation, and the ETH Zürich Foundation. We thank Danupon Nanongkai for the helpful discussions and the anonymous reviewers for their helpful suggestions on the writing.

D.11 Bipartite Matching in CONGEST and PRAM

D.11.1 Parallel BMM

The following theorem is the state-of-the-art for parallel bipartite maximum matching.

Theorem D.91 ([Bra+25]). *In PRAM model, bipartite maximum matching can be solved in $\tilde{O}(m + n^{1.5})$ work and $\tilde{O}(n^{1/2})$ depth, with high probability.*

However, we cannot plug Theorem D.91 to Theorem D.1 since the work function is not a super additive function on m . Instead, if we want the work to be super-additive in m , the following theorem from [LS14] is the state-of-the-art.

Theorem D.92 ([LS14]). *In PRAM model, bipartite maximum matching can be solved in $\tilde{O}(mn^{0.5})$ work and $\tilde{O}(n^{1/2})$ depth, with high probability.*

D.11.2 A CONGEST Algorithm

In this section, we will prove the following two theorems. Notice that [Theorem D.94](#) is subsumed by [Theorem D.92](#), but we put it here as an alternative algorithm.

Theorem D.93. *In CONGEST model, maximum bipartite matching can be solved in $\tilde{O}(n^{3/4}D^{1/2} + n^{1/2}D + n^{7/10+o(1)}D^{7/10})$ rounds, with high probability. In particular, when $D = O(n^{3/7-\epsilon})$ for some constant $\epsilon > 0$, this is sublinear in n .*

Theorem D.94. *In PRAM model, bipartite maximum matching can be solved in $\tilde{O}(mn^{0.5+\epsilon})$ work and $n^{1+o(1)-\epsilon}$ depth for every $\epsilon \in [0, 0.25]$, with high probability.*

The main observation is that a semi-streaming $(1 - \epsilon)$ -approximation algorithm of [\[Ass+22\]](#) almost straightforwardly can be implemented in the CONGEST and PRAM models in roughly $\frac{1}{\epsilon}$ rounds/depth. Then, after computing a $(1 - \epsilon)$ maximum bipartite matching, we combine with known reachability algorithms [\[CF23; JLS19\]](#) in CONGEST and PRAM to find the remaining ϵn many augmenting paths one at a time. Hence, this section does not contain much technical novelty, but rather show how known algorithms can almost straightforwardly be generalized to other models and combined in order to show not yet document results for CONGEST ([Theorem D.93](#)) and PRAM ([Theorem D.92](#)).

Related Work: Bipartite Matching in CONGEST and PRAM. While bipartite matching has been extensively studied in many models, the exact version seems to resist sublinear (i.e. sublinear when $D = n^{1-\delta}$ for constant $\delta > 0$) algorithms in the CONGEST model. The work of [\[AKO18\]](#) observed that a simple extension of the Hopcroft-Karp algorithm can solve bipartite matching in $O(M^* \log M^*)$ round in CONGEST, where M^* is the size of the maximum matching (i.e., in general their algorithm runs in $O(n \log n)$ rounds). The work of [\[For+21\]](#) takes a different approach based on Laplacian solves, and show how unit-capacitated maximum flow (and hence the special case of bipartite matching) can be solved in $O(m^{3/7+o(1)}(n^{1/2}D^{1/4} + D))$ rounds in CONGEST. This, to our knowledge, was the only known sublinear algorithm in CONGEST, although only in sparse graphs $m = \tilde{O}(n)$ and when $D = n^{2/7-\delta}$ for some constant $\delta > 0$. We note that the running time of our [Theorem D.93](#) is an improvement for any regime of m and D . We also note that [\[Vos23\]](#) generalized the min-cost max-flow approach of [\[LS14\]](#) in [Theorem D.92](#) to also work in the CONGEST model, handling each of the $n^{1/2}$ rounds of solving a laplacian system in $O(\sqrt{n} + D)$ rounds in CONGEST (thus their total number of rounds is not sublinear in n).

There has also been a lot of work on bipartite maximum matching algorithms in the PRAM model, or the more general min-cost max-flow problems (see, e.g., [\[LS14; Bra+25\]](#)). It is known that the problem is in randomized NC (i.e., polylog depth and polynomial work) due to a reduction to computing matrix inverse [\[Lov79; KUW86; MVV87; GP88\]](#). Another sequence of work focuses on sublinear deterministic algorithms [\[Gol+92; GPV93\]](#).

There are also many PRAM, CONGEST, and streaming algorithms for $(1 - \epsilon)$ -approximating maximum bipartite matching, see e.g., [\[Coh95; ALT21; Ass+22; Egg+12; AKO18; LPP15\]](#), however most of these have depth/round/pass complexity at least $\frac{1}{\epsilon^2}$. A crucial property of [\[Ass+22\]](#) is that its running time only depends on a single factor of $\frac{1}{\epsilon}$, which is crucial to allow the exact algorithm to have sublinear depth/round/pass when combined with fast reachability algorithms in the different models.

Semi-Streaming to Other Models. The semi-streaming algorithm of [\[Ass+22\]](#) will, when implemented in CONGEST and PRAM, imply the below lemmas for approximating bipartite matching. Since, to our knowledge, this connection has not been observed before, we sketch the proof in [Section D.11.3](#).

Lemma D.95. *In CONGEST model, there is an algorithm that computes a $(1 - \epsilon)$ -approximate fractional matching to the maximum bipartite matching problem in $\tilde{O}(\frac{D}{\epsilon})$ round.*

Lemma D.96. *In PRAM model, there is an algorithm that computes a $(1 - \epsilon)$ -approximate fractional matching for the maximum bipartite matching problem in $\tilde{O}(\frac{1}{\epsilon})$ depth and $\tilde{O}(\frac{m}{\epsilon})$ work.*

Remark D.97. For the CONGEST algorithm, the $\frac{1}{\epsilon}$ -dependency in [Lemma D.95](#) is tight for low-diameter graphs where $D = O(\log(n))$, as shown by a corresponding lower-bound in [\[AKO18\]](#). Since there are several algorithms with $\tilde{O}(\frac{1}{\epsilon^2})$ complexity without dependency on D , e.g., [\[AKO18; ALT21; LPP15\]](#), a natural question that remains open is if there is a $\tilde{O}(\frac{1}{\epsilon})$ -round CONGEST algorithm for $(1 - \epsilon)$ -approximate bipartite matching.

Before proving these lemmas, we show how they imply [Theorem D.93](#) and [Theorem D.92](#), by combining with known techniques. Our plan is to first round the $(1 - \epsilon)$ -approximate fractional matching to integral, and then find the remaining augmenting paths one-by-one using state of the art reachability algorithms in the two models.

The CONGEST Model. In the CONGEST model, [\[AKO18\]](#) showed how to almost losslessly round a fractional matching to an integral one. In particular, they showed the following theorem.

Theorem D.98 ([\[AKO18\]](#)). *Let $G = (V, E)$ be a bipartite graph with maximum degree Δ , $y \in \mathbb{R}^E$ be a fractional matching of G , and $\epsilon > 0$ be a parameter. There is a deterministic $O\left(\frac{\log^2(\Delta/\epsilon) + \log^* n}{\epsilon}\right)$ -time CONGEST algorithm that computes a matching M of G such that $|M| \geq (1 - \epsilon)||y||_1$.*

[Theorem D.98](#) together with [Lemma D.95](#) implies that we can find, in $\tilde{O}(\frac{D}{\epsilon})$ rounds, an integral matching M that is at most ϵn far away from maximum. We will find the remaining $\leq \epsilon n$ augmenting paths one by one, by using the current fastest directed reachability algorithm in CONGEST. Indeed, note that given an integral matching M on a bipartite graph $G = (L \cup R, E)$, one can put directions on the edges as follows: $e \in E$ is oriented from R to L if $e \in M$, and otherwise oriented from L to R . Now, increasing the size of the matching M by one is equivalent to finding any directed path from an unmatched vertex in L to an unmatched vertex in R —such a path is called an augmenting path.

We use the single source shortest path algorithm of [\[CF23\]](#), setting all edge-weights to 1.

Theorem D.99 (Distributed Reachability [\[CF23\]](#)). *There is a distributed algorithm that, given an n -node directed graph with positive integer edge weights of undirected hop-diameter D , solves the single-source-shortest-paths problem with $\tilde{O}(n^{1/2} + D + n^{2/5+o(1)}D^{2/5})$ rounds of communication in the CONGEST model, with high probability.*

We need to run the reachability algorithm at most ϵn times. Hence, for solving exact bipartite matching, the total round complexity becomes:

$$\tilde{O}\left(\frac{D}{\epsilon} + \epsilon n \cdot (n^{1/2} + D + n^{2/5+o(1)}D^{2/5})\right)$$

By setting $\epsilon := \frac{D^{1/2}}{n^{3/4} + n^{1/2}D^{1/2} + n^{7/10}D^{1/5}}$ to balance out the terms, the final time complexity is

$$\tilde{O}\left(n^{3/4}D^{1/2} + n^{1/2}D + n^{7/10+o(1)}D^{7/10}\right).$$

This proves [Theorem D.93](#).

The PRAM Model. Our strategy in the PRAM model is similar. First we round the fractional matching to an integral one using the following PRAM rounding lemma.

Theorem D.100 ([\[Gol+92\]](#)). *Let $G = (V, E)$ be a bipartite graph, $y \in \mathbb{R}^E$ be a fractional matching of G . There is a deterministic $O(\log^2(n))$ -depth, $O(m \log^2 n)$ -work PRAM algorithm that computes a matching M of G such that $|M| \geq \lceil ||y||_1 \rceil$.*

[Lemma D.96](#) and [Theorem D.100](#) together thus implies a $\tilde{O}(\frac{1}{\epsilon})$ depth and $\tilde{O}(\frac{m}{\epsilon})$ work find an integral matching M that is at most ϵn far away from maximum. Again, we will find the remaining $\leq \epsilon n$ augmenting paths one by one, by using a known reachability algorithm.

Theorem D.101 (Parallel Reachability [JLS19]). *There is a parallel algorithm that, given an n -node, m -edge directed graph, solves the single-source reachability problem with work $\tilde{O}(m)$ and depth $n^{1/2+o(1)}$ with high probability in n .*

Although in [JLS19] they only solve the decision version and not how to recover the path. However this has been observed (see comment below [Ass+22, Proposition 4]) that the algorithm can actually recover the path too.

We need to run the reachability algorithm at most ϵn times. The total complexity becomes:

$$\tilde{O}\left(\frac{1}{\epsilon} + \epsilon n \cdot n^{1/2+o(1)}\right) \text{ depth,} \quad \text{and} \quad \tilde{O}\left(\frac{m}{\epsilon} + \epsilon n \cdot m\right) \text{ work.}$$

By setting $\epsilon := n^{-1/2-\delta}$ for any $\delta \in [0, 0.25]$, we get:

$$\tilde{O}\left(n^{1+o(1)-\delta}\right) \text{ depth,} \quad \text{and} \quad \tilde{O}\left(m \cdot n^{1/2+\delta}\right) \text{ work.}$$

This proves [Theorem D.93](#), since $n \leq 2m$ after removing possibly isolated vertices.

D.11.3 Generalizing a Semi-Streaming Algorithm

We now show how to generalize the semi-streaming algorithm of [Ass+22] to work in the CONGEST and PRAM settings in order to prove [Lemmas D.95](#) and [D.96](#). We note that the algorithm of [Ass+22] is based on [She17], but where [Ass+22] shows that this algorithm can be implemented in low space. We note that for our purposes in CONGEST and PRAM, the algorithm of [She17] would directly work as these models do not care about the space complexity. However, we choose to work with [Ass+22] as it is more straightforward to see how it can be implemented in our models. This section will skip over most of the proofs and intuition, as they can be found in [Ass+22], and instead focus on simply arguing why the exact algorithm used by [Ass+22] also works in CONGEST and PRAM.

The algorithm of [Ass+22] works by formulating the maximum bipartite matching problem as a ℓ_1 -regression problem and then solving it using a $\frac{1}{\epsilon}$ -round optimization method for box-simplex games.

Let $\Delta^m := \{x \in \mathbb{R}^m : \|x\|_1 = 1\}$ be the m -dimensional simplex. In particular, consider the following box-simplex optimization problem, for some $n \times m$ matrix $\mathbf{A}$, and vectors $c \in \mathbb{R}^m$ and $b \in \mathbb{R}^n$.

$$\min_{x \in \Delta^m} \max_{y \in [-1, 1]^n} y^\top \mathbf{A}^\top x + c^\top x - b^\top y, \quad (\text{D.9})$$

which is equivalent to the following ℓ_1 -regression problem:

$$\min_{x \in \Delta^m} c^\top x + \|\mathbf{A}^\top x - b\|_1. \quad (\text{D.10})$$

We restate [Ass+22, Algorithm 3] here as [Algorithm 4](#), which, according to [Ass+22, Proposition 1] outputs $\bar{x}$ that is a ϵ -approximate minimizer to the box-simplex game in (D.9).

The plan is as follows:

- Show how to set up bipartite matching problem as a box-simplex game in (D.9), by choosing appropriate $\mathbf{A}, b, c$ as in [Ass+22, Section 4].
- Show how [Algorithm 4](#) (and a simple post-processing step [Algorithm 5](#)) can be implemented in both CONGEST and PRAM efficiently, for the parameters given by the bipartite matching problem.

Setting up Bipartite Matching as a Box-Simplex Game. Let $G = (V, E)$ be a bipartite graph we wish to solve bipartite matching in. Let M^* be the size of the maximum matching. We follow [Ass+22, Section 4] to set it up as a box-simplex game, see their article for more details and discussion.

Let $M \in \mathbb{Z}$ be a 2-approximation of the size of the maximum matching, i.e., it satisfies $M \leq M^* \leq 2M$. There are multiple ways to obtain this in our models: for example we can guess $M = 2^i$ for every $i = 0 \dots \lceil \log n \rceil$, and run the algorithm for all guesses and output the best achieved matching. Alternatively, we can find a

Algorithm 4: LowSpaceACMirrorProx($\mathbf{A}, b, c, \varepsilon$) [Ass+22]

- 1 **Input:** $\mathbf{A} \in \mathbb{R}_{\geq 0}^{m \times n}$, $c \in \mathbb{R}^m$, $b \in \mathbb{R}^n$, $0 \leq \varepsilon \leq \|\mathbf{A}\|_\infty$
 2 **Output:** $\{v'_t\}_{0 \leq t < T} \subset \mathbb{R}^n$, $\{u'_t\}_{0 \leq t < T} \subset \mathbb{R}^n$, $\{\lambda'_t\}_{0 \leq t < T} \subset \mathbb{R}$, $\{y'_t\}_{0 \leq t < T} \subset \mathbb{R}^n$ so that for

$$\bar{y} := \frac{1}{T} \sum_{0 \leq t < T} y'_t, \quad \bar{x} := \frac{1}{T} \sum_{0 \leq t < T} \frac{\exp(\mathbf{A}v'_t + |\mathbf{A}|u'_t + \lambda'_t c)}{\|\exp(\mathbf{A}v'_t + |\mathbf{A}|u'_t + \lambda'_t c)\|_1},$$

the pair $(\bar{x}, \bar{y})$ is an ε -approximate saddle point to (D.9)

- 3 $T \leftarrow O(\frac{\|\mathbf{A}\|_\infty \log m}{\varepsilon})$, $K \leftarrow O(\log \frac{n\|\mathbf{A}\|_\infty}{\varepsilon})$
 4 $t \leftarrow 0$, $\lambda_0 \leftarrow 0$, $v_0 \leftarrow \mathbf{0}_n$, $u_0 \leftarrow \mathbf{0}_n$, $y_0 \leftarrow \mathbf{0}_n$
 5 **while** $t < T$ **do**
 6 $(v^{(0)}, u^{(0)}, \lambda^{(0)}, y^{(0)}) \leftarrow (v_t, u_t, \lambda_t, y_t)$
 7 $\gamma^y \leftarrow \frac{1}{3}(b - \mathbf{A}^\top \frac{\exp(\mathbf{A}v_t + |\mathbf{A}|u_t + \lambda_t c)}{\|\exp(\mathbf{A}v_t + |\mathbf{A}|u_t + \lambda_t c)\|_1}) - 2\text{diag}(y_t) |\mathbf{A}|^\top \frac{\exp(\mathbf{A}v_t + |\mathbf{A}|u_t + \lambda_t c)}{\|\exp(\mathbf{A}v_t + |\mathbf{A}|u_t + \lambda_t c)\|_1}$
 8 **for** $0 \leq k < K$ **do**
 9 $v^{(k+1)} \leftarrow -\frac{1}{10\|\mathbf{A}\|_\infty}(\frac{1}{3}y_t - 10\|\mathbf{A}\|_\infty v^{(0)})$
 10 $u^{(k+1)} \leftarrow -\frac{1}{10\|\mathbf{A}\|_\infty}(-(y^{(0)})^2 + (y^{(k)})^2 - 10\|\mathbf{A}\|_\infty u^{(0)})$
 11 $\lambda^{(k+1)} \leftarrow -\frac{1}{10\|\mathbf{A}\|_\infty}(\frac{1}{3} - 10\|\mathbf{A}\|_\infty \lambda^{(0)})$
 12 $d^{(k+1)} \leftarrow 2|\mathbf{A}|^\top \frac{\exp(\mathbf{A}v^{(k+1)} + |\mathbf{A}|u^{(k+1)} + \lambda^{(k+1)}c)}{\|\exp(\mathbf{A}v^{(k+1)} + |\mathbf{A}|u^{(k+1)} + \lambda^{(k+1)}c)\|_1}$
 13 $y^{(k+1)} \leftarrow \text{med}(-1, 1, -\frac{\gamma^y}{d^{(k+1)}})$ entrywise
 14 $(v'_t, u'_t, \lambda'_t, y'_t) \leftarrow (v^{(K)}, u^{(K)}, \lambda^{(K)}, y^{(K)})$
 15 $(v^{(0)}, u^{(0)}, \lambda^{(0)}, y^{(0)}) \leftarrow (v_t, u_t, \lambda_t, y_t)$
 16 $\gamma^y \leftarrow \frac{1}{3}(b - \mathbf{A}^\top \frac{\exp(\mathbf{A}v'_t + |\mathbf{A}|u'_t + \lambda'_t c)}{\|\exp(\mathbf{A}v'_t + |\mathbf{A}|u'_t + \lambda'_t c)\|_1}) - 2\text{diag}(y_t) |\mathbf{A}|^\top \frac{\exp(\mathbf{A}v_t + |\mathbf{A}|u_t + \lambda_t c)}{\|\exp(\mathbf{A}v_t + |\mathbf{A}|u_t + \lambda_t c)\|_1}$
 17 **for** $0 \leq k < K$ **do**
 18 $v^{(k+1)} \leftarrow -\frac{1}{10\|\mathbf{A}\|_\infty}(\frac{1}{3}y'_t - 10\|\mathbf{A}\|_\infty v^{(0)})$
 19 $u^{(k+1)} \leftarrow -\frac{1}{10\|\mathbf{A}\|_\infty}(-(y^{(0)})^2 + (y^{(k)})^2 - 10\|\mathbf{A}\|_\infty u^{(0)})$
 20 $\lambda^{(k+1)} \leftarrow -\frac{1}{10\|\mathbf{A}\|_\infty}(\frac{1}{3} - 10\|\mathbf{A}\|_\infty \lambda^{(0)})$
 21 $d^{(k+1)} \leftarrow 2|\mathbf{A}|^\top \frac{\exp(\mathbf{A}v^{(k+1)} + |\mathbf{A}|u^{(k+1)} + \lambda^{(k+1)}c)}{\|\exp(\mathbf{A}v^{(k+1)} + |\mathbf{A}|u^{(k+1)} + \lambda^{(k+1)}c)\|_1}$
 22 $y^{(k+1)} \leftarrow \text{med}(-1, 1, -\frac{\gamma^y}{d^{(k+1)}})$ entrywise
 23 $(v_{t+1}, u_{t+1}, \lambda_{t+1}, y_{t+1}) \leftarrow (v^{(K)}, u^{(K)}, \lambda^{(K)}, y^{(K)})$
 24 $t \leftarrow t + 1$

maximal matching (hence a 2-approximation) in $\tilde{O}(1)$ rounds in CONGEST,¹² or in $\tilde{O}(1)$ depth and $O(m)$ work in PRAM, by using Luby's folklore (randomized) maximal independent set algorithm (applied to the line graph of G).

Let $\mathbf{B}$ be the $m \times n$ adjacency matrix of G . Let $\tilde{\mathbf{B}}$ be the $(m+1) \times (n+2)$ matrix obtained by adding with one extra row (a fake edge), and two extra columns (two fake vertices), all filled with zeroes, to $\mathbf{B}$. Let $d_{\text{MCM}} \in \{0, 1\}^{n+2}$ be 1 everywhere except 0 in the indices corresponding to the two fake vertices.

Now consider the following ℓ_1 -regression problem:

$$\min_{x \in \Delta^{m+1}} \|\mathbf{A}^\top x - b\|_1, \quad \text{where } \mathbf{A} := M\tilde{\mathbf{B}}, \quad b := \frac{1}{2}d_{\text{MCM}}. \quad (\text{D.11})$$

Use Algorithm 4 to solve it up to $\Theta(\varepsilon M)$ additive accuracy to obtain a solution x . Set $\tilde{x} := 2Mx_E$, where x_E is x but we dropped the coordinate of the fake edge. Now $\tilde{x}$ will almost be an approximate maximum fractional matching, however it is not necessarily feasible. See [Ass+22, Section 4.1] for an argument why $\|\tilde{x}\|_1 - \text{Overflow}(\tilde{x}) \geq (1 - \varepsilon)M^*$, where $\text{Overflow}(x) := \sum_{v \in V} \max\left(0, \left(\sum_{u: (u,v) \in E} x_{uv}\right) - 1\right) =$

¹²In the CONGEST model, we would need an additional $O(D)$ rounds to calculate the size of this maximal matching and broadcast it to every vertex, however this is dominated by the running time $\tilde{O}(\frac{D}{\varepsilon})$ we are aiming to prove.

$\left\| (B^\top x - d)_+ \right\|_1$ is the sum of overflow at each vertex.

So, like in [Ass+22], we need a simple post-processing step where we remove the overflow of $\tilde{x}$ to make it into a feasible matching without removing too much. We restate a simplified version of [Ass+22, Algorithm 4] below as **Algorithm 5**.

Algorithm 5: RemoveOverflow(x, G) [Ass+22]

- 1 **Input:** Bipartite graph $G = (V, E)$ with incidence matrix B , $x \in \mathbb{R}_{\geq 0}^E$
 - 2 **Output:** $\tilde{x} \in \mathbb{R}_{\geq 0}^E$ with $\tilde{x} \leq x$ entrywise, $B^\top \tilde{x} \leq \mathbf{1}$, and $\|\tilde{x}\|_1 \geq \|x\|_1 - \left\| (B^\top x - d)_+ \right\|_1$ (see [Ass+22, Lemma 5])
 - 3 $d^x \leftarrow B^\top x$
 - 4 $f \leftarrow (d^x - \mathbf{1})_+$
 - 5 $\tilde{x}_e \leftarrow x_e \left(1 - \max \left(\frac{f_a}{d_a^x}, \frac{f_b}{d_b^x} \right) \right)$ for all $e = (a, b) \in E$ with $x_e > 0$
 - 6 **return** $\tilde{x}$
-

In particular, we get a fractional matching $x^* := \text{RemoveOverflow}(\tilde{x}, G)$ of size $\|x^*\| > (1 - \epsilon)M$.

Implementations in CONGEST and PRAM. What remains is to argue that **Algorithms 4** and **5** can be implemented efficiently in CONGEST and PRAM.

Algorithm 5 can easily be implemented in $O(1)$ rounds in CONGEST, as all it does is compute the overflow at each vertex, and then decrease the value of the fractional matching on an edge (u, v) based on the overflow at its endpoints. Similarly, it can be computed in $\tilde{O}(1)$ depth and $O(m)$ work in PRAM.

Algorithm 4 is a bit more complicated. It runs in $T = O(\frac{\|A\|_\infty \log m}{\epsilon})$ phases, which is $\tilde{O}(\frac{1}{\epsilon})$ as $A := \tilde{B}$ so $\|\tilde{B}\|_\infty = O(M)$ by construction, and we solve it up to error $\epsilon := \epsilon M$ (note the different epsilons). Similarly, note that $K = O(\log \frac{n\|A\|_\infty}{\epsilon}) = O(\log n)$. Hence, each line in the algorithm is run at most $\tilde{O}(\frac{1}{\epsilon})$ times.

We will show that each statement inside the loops in **Algorithm 4** can be implemented in $\tilde{O}(D)$ rounds in CONGEST, or $\tilde{O}(1)$ depth and $\tilde{O}(m)$ work in PRAM.

The algorithm keeps tracks of many variables on supported on vertices. It also computes intermediary values on the edges as expressions like:

$$\frac{\exp(Av'_t + |A|u'_t + \lambda'_t c)}{\|\exp(Av'_t + |A|u'_t + \lambda'_t c)\|_1}$$

Computing matrix products Av or $A^\top v$ for some vector v can be done efficiently¹³ in parallel and locally, as A is essentially the incidence matrix (the additional fake vertices and edges have corresponding zero-rows/columns, and hence can also be handled very efficiently). Similarly computing and multiplying by a diagonal matrix $\text{diag}(\cdot) y_t$ is efficient to do locally and in parallel. All the entrywise computations (like $\exp(v)$, or computing the entrywise median) is also efficiently locally/parallel computable.

The normalization, i.e., computing $\|v\|_1$ (as in $\|\exp(Av'_t + |A|u'_t + \lambda'_t c)\|_1$) is a bit more tricky for the CONGEST algorithm. It essentially boils down to computing the sum of values stored at each vertex, and requires $\tilde{O}(D)$ rounds to aggregate and sum up the value, and then broadcast it to all vertices. In PRAM, this normalization step is still easy to do in $\tilde{O}(m)$ work and $\tilde{O}(1)$ depth.

Remark D.102. We note that this is the main bottleneck in the implementation of the CONGEST algorithm, and why it depends on D . An interesting open question is if there is an alternative $(1 - \epsilon)$ -approximation algorithm in CONGEST that only needs $\tilde{O}(\frac{1}{\epsilon})$ rounds, i.e., without dependency on D .

We conclude that, even though **Algorithm 4** might look initially intimidating, and it's analysis is non-trivial (see [Ass+22]), it is straightforward to see how it can be implemented efficiently in CONGEST and PRAM models when A is (almost) the incidence matrix of a graph. In particular, it can be implemented in $\tilde{O}(D/\epsilon)$

¹³One potential worry is numerical stability, as it might require many bits to represent the numbers. However, the algorithm is numerically stable, so values much smaller than the largest values can safely be rounded; see the comment in [Ass+22] about numerical stability.

rounds in CONGEST, or $\tilde{O}(\frac{1}{\epsilon})$ depth and $\tilde{O}(\frac{m}{\epsilon})$ work in PRAM to compute (together with Algorithm 5) an $(1 - \epsilon)$ -approximate fractional matching. This proves Lemmas D.95 and D.96.

D.11.4 Solving Bipartite Matching on a Subgraph in CONGEST

In our reductions, we need to solve bipartite matching on a bipartite subgraph H of the original (not-necessarily bipartite) graph G . In most models of computation, we can just focus on solving bipartite matching on H directly, however, in the CONGEST model the diameter of H might be much larger than the diameter of G , and we are in fact allowed to use edges of $G \setminus H$ for the purpose of communication. While most algorithms can easily be adapted to solve bipartite matching on a subgraph, we can in fact show a perfect reduction irregardless of the underlying algorithm.

Lemma D.103. *Suppose there is a $R(n, m, D)$ round CONGEST algorithm for solving maximum bipartite matching on n -vertex, m -edge, D -diameter bipartite graphs G_{bip} . Then there is a $R(O(n), O(m), O(D))$ round CONGEST algorithm that on n -vertex, m -edge, D -diameter (not-necessarily bipartite) graph G , solves maximum bipartite matching on a given bipartite subgraph H (of potentially much larger diameter).*

Proof Sketch. For each vertex $v \in V$, make 5 copies of it: $v^{(H)}, v^{(a)}, v^{(a')}, v^{(b)}, v^{(b')}$. Let (A, B) be the vertex bi-partition of V induced by the bipartite subgraph H . Add the following edges:

$$\begin{aligned} & (u^{(H)}, v^{(H)}) \text{ for all } (u, v) \in E(H) \\ & (u^{(a)}, v^{(b)}) \text{ and } (u^{(b)}, v^{(a)}) \text{ for all } (u, v) \in E(G) \\ & (v^{(a)}, v^{(a')}) \text{ and } (v^{(b)}, v^{(b')}) \text{ for all } v \in V \\ & (v^{(H)}, v^{(a)}) \text{ for all } v \in A \\ & (v^{(H)}, v^{(b)}) \text{ for all } v \in B \end{aligned}$$

Let this graph be called G' . That is $G'[V^{(H)}]$ is H , and $G'[V^{(a)} \cup V^{(b)}]$ is the binary lift of G . It is easy to verify that the graph is bipartite, by construction. Moreover, the diameter of any connected component¹⁴ of G' is at most $O(D)$. This is since, if G is not bipartite, let g be its shortest odd-cycle and note that $|g| \leq 2D + 1$, which means that any vertex v on this cycle will have $v^{(a)}$ and $v^{(b)}$ at most distance $2D + 1$ from each other in G' .

Any maximum matching M_H of H can be extended to a matching $M_{G'}$ of G' , of size $|M_{G'}| = |M_H| + 2n$, by adding all the $(v^{(a)}, v^{(a')})$ and $(v^{(b)}, v^{(b')})$ edges. For the other direction, suppose that $M_{G'}$ is a maximum matching of G' . We know that all vertices $v^{(a)}$ and $v^{(b)}$ must be matched (else an edge $(v^{(a)}, v^{(a')})$ can be added to the matching). We can without loss of generality change the matching edge of $v^{(a)}$ to be matched to $v^{(a')}$ (and similar for $(v^{(b)}, v^{(b')})$). Then let $M_H = M_{G'} \setminus (\cup_{v \in V} \{(v^{(a)}, v^{(a')}), (v^{(b)}, v^{(b')})\})$, and note that M_H is a matching on H of size $|M_{G'}| - 2n$.

Thus, solving the matching problem on G' and on H is essentially equivalent. The CONGEST algorithm running on G will simulate the bipartite matching CONGEST algorithm running on each connected component of G' , where each node v simulates the work of all its copies. $\square$

D.12 Graph Sparsification

We fix source $x \in C$ throughout this section.

For all node $u \in C$, we denote $\mu_C(u) := \min_{L' \subseteq C: u \in L'} |N_G(L')|$.

Definition D.104 (G_x). Given a source $x \in C$ where $C \subseteq V$ in a graph $G = (V, E)$, we define the *local graph* G_x as the induced graph $G[N_G[C]]$ after adding a sink t and the set of edges (v, t) for all $v \in N_G(C)$.

¹⁴If G is connected, G' will have two connected components if and only if G is bipartite, and else it will have one connected component.

Lemma D.105. *If (L, S, R) is a minimum (x, t) -separator in G_x where $x \in L, t \in R$, then L is a minimizer of $\mu_C(x) = \min_{L' \subseteq C': x \in L'} |N_G(L')|$ and vice versa. In particular, $\kappa_{G_x}(x, t) = \mu_C(x)$.*

Proof. First, we prove that $\kappa_{G_x}(x, t) \geq \mu_C(x)$. Let (L, S, R) be an (x, t) -min-separator in G_x where $x \in L, t \in R$. We claim that $L \subseteq C$. Suppose L contains a vertex $v \in N(C)$. This means $N_{G_x}(L)$ contains t , and thus $t \notin R$, a contradiction. Observe that $N(L) = S$ because since (L, S, R) is a minimum (x, t) -separator. Since $x \in L \subseteq C$ and $N(L) = S$, we have

$$\kappa_{G_x}(x, t) = |S| = |N_{G_x}(L)| = |N_G(L)| \geq \min_{L' \subseteq C': x \in L'} |N_G(L')| = \mu_C(x).$$

Next, we prove $\mu_C(x) \geq \kappa_{G_x}(x, t)$. Let L^* be a minimizer of $\min_{L' \subseteq C': x \in L'} |N_G(L')|$. Observe that $N_G(L^*) \subseteq N_G[C]$. We now prove that $N_G(L^*)$ is an (x, t) -separator in G_x . Since $N_G(L^*)$ is a separator in G and $N_G[L^*] \subseteq N_G[C]$, it is enough to prove that the set of edges from t to every vertex in $N_G(C)$ will not destroy this cut. Indeed, since $L^* \subseteq C$, the new edges will not incident to L^* . Therefore, $N_G(L^*) = N_{G_x}(L^*)$ is an (x, t) -separator in G_x . So,

$$\mu_C(x) = |N_G(L^*)| = |N_{G_x}(L^*)| \geq \kappa_{G_x}(x, t).$$

This concludes the proof. $\square$

We cannot yet implement G_x in a distributed network because connecting s to every node in $N_G(C)$ is a global operation. Instead, we simulate s by adding a perfect matching from each node in $N_G(C)$ to a new node (so we can remove s).

Definition D.106 (G'_x). Given G_x , define G'_x as follows. First, remove t from G'_x . Then, for each vertex $u \in N_G(C)$, add a new vertex y along with an edge (u, y) . Denote Y as the set of new vertices.

Lemma D.107. *Every (s, t) -cut in G_x is an (s, Y) -cut in G'_x and vice versa. In particular, $\kappa_{G_x}(s, t) = \kappa_{G'_x}(s, Y)$.*

Proof. If S is an (s, t) -separator in G'_x , then denote a vertex cut (L, S, R) in G'_x where $s \in L, t \in R$. Since $N_{G'_x}(t) \subseteq S \cup R$, every vertex in Y in G'_x does not have an edge to L . Therefore, $(L, S, (R \cup Y) - \{t\})$ is a vertex cut in G'_x . If S is an (s, Y) -separator in G'_x , then the same separator must be an (s, t) -separator in G'_x by contracting all Y into t . $\square$

Corollary D.108. *If (L, S, R) is a minimum (x, Y) -separator in G'_x where $x \in L, Y \subseteq R$, then L is a minimizer of $\min_{L' \subseteq C': x \in L'} |N_G(L')|$. In particular, $\kappa_{G'_x}(x, Y) = \min_{L' \subseteq C': x \in L'} |N_G(L')|$.*

Definition D.109 (Sparsification). Give G_x , define $\tilde{H}_x$ after the following reductions.

1. Remove $Z := N_G(x) \cap N_G(t)$ from G_x .
2. Remove every edge $(u, v) \in E_G(N_G(x), N_G(x)) \cup E_G(N_G(t), N_G(t))$.

Lemma D.110. $\kappa_{G_x}(x, t) = \kappa_{\tilde{H}_x}(x, t) + |Z|$. Furthermore, If S is an (s, t) -min-separator in H , then $Z \cup S$ is an (s, t) -min-separator in G .

Proof. The first step follows because every vertex $v \in N_G(x) \cap N_G(t)$ belongs to every (x, t) -separator in G . The second step also follows from the same reduction rules in [Li+21]. $\square$

Finally, we define the graph H_x as follows.

Definition D.111 (H_x). Give $\tilde{H}_x$, define H_x as follows. First, remove t from $\tilde{H}_x$. Then, for each vertex $u \in N_G(C)$, add a new vertex y along with an edge (u, y) . Denote Y as the set of new vertices.

Lemma D.112. *Every (s, t) -cut in G_x is an (s, Y) -cut in G'_x and vice versa. In particular, $\kappa_{G_x}(s, t) = \kappa_{G'_x}(s, Y)$.*

Proof. The proof is similar to Lemma D.107. $\square$

We are ready to prove [Lemma D.58](#).

Proof of [Lemma D.58](#). Given $x \in C$, we construct H_x as defined in [Definition D.111](#). We can verify that this construction in [Definition D.109](#) can be done in $O(1)$ rounds. We can compute Z in two rounds. First, s and t broadcast its id along its neighbors, and in the second round, if a node receives both s and t then it belongs to Z . Every node $v \in N_G(C) - Z$ can simulate another node in Y in the matching by itself. Removing every edge $(u, v) \in E_G(N_G(x), N_G(x)) \cup E_G(N_G(C), N_G(C))$ can be done in $O(1)$ rounds.

We verify each of the following properties. First we show that for every vertex $v \in C$, $\deg_{H_x}(v, V_x - (N_G(x) \cap C)) \leq O(2^{\log^{0.7} n} \cdot \ell)$. This follows from [Equation \(D.2\)](#) and the fact that the common neighbors Z were removed. Next, $\kappa_{H_x}(x, Y_x) + |Z_x| \geq \min_{L' \subseteq C: L' \neq \emptyset} |N_G(L')|$. the fact that if $x \in L$, then $\kappa_{H_x}(x, Y_x) + |Z_x| = \min_{L' \subseteq C: L' \neq \emptyset} |N_G(L')|$, can be derived from [Lemmas D.47](#) and [D.112](#) and [Corollary D.108](#). $\square$

Paper E

Deterministic Vertex Connectivity via Common-Neighborhood Clustering and Pseudorandomness

Yonggang Jiang, Chaitanya Nalam, Thatchaphol Saranurak,
Sorrachai Yingchareonthawornchai

Published at ACM Symposium on Theory of Computing (STOC) 2025.

Full version available at <https://arxiv.org/abs/2503.20985>.

Abstract

We give a deterministic algorithm for computing a global minimum vertex cut in a *vertex-weighted* graph n vertices and m edges in $\hat{O}(mn)$ time.¹ This breaks the long-standing $\hat{\Omega}(n^4)$ -time barrier in dense graphs, achievable by trivially computing all-pairs maximum flows. Up to subpolynomial factors, we match the fastest randomized $\tilde{O}(mn)$ -time algorithm by [HRG00], and affirmatively answer the question by [Gab06] whether deterministic $O(mn)$ -time algorithms exist even for unweighted graphs. Our algorithm works in directed graphs, too.

In unweighted undirected graphs, we present a faster deterministic $\hat{O}(m\kappa)$ -time algorithm where $\kappa \leq n$ is the size of the global minimum vertex cut. For a moderate value of κ , this strictly improves upon all previous deterministic algorithms in unweighted graphs with running time $\hat{O}(m(n + \kappa^2))$ [Eve75], $\hat{O}(m(n + \kappa\sqrt{n}))$ [Gab06], and $\hat{O}(m2^{O(\kappa^2)})$ [SY22]. Recently, a linear-time algorithm has been shown by [Kor24] for small κ .

Our approach applies the *common-neighborhood clustering*, recently introduced by [Bli+25], in novel ways, e.g., on top of weighted graphs and on top of vertex-expander decomposition. We also exploit pseudorandom objects often used in computational complexity communities, including *crossing families* based on dispersers from [WZ99; TUZ01] and *selectors* based on linear lossless condensers [GUV09; Che11]. To our knowledge, this is the first application of selectors in graph algorithms.

¹We use $\tilde{O}(\cdot)$ and $\hat{O}(\cdot)$ to hide $\text{poly} \log(n)$ and $n^{o(1)}$ factors, respectively.

E.1 Introduction

In the (*global*) *minimum vertex cut* problem, given an n -vertex, m -edge graph $G = (V, E)$ with vertex weights $w : V \rightarrow \{1, \dots, W\}$, we must find a vertex set $S \subseteq V$ with minimum total weight $w(S) := \sum_{u \in S} w(u)$ whose removal disconnects G . The weight $w(S)$ is called the *weighted vertex connectivity* of G . In the unweighted case, we must find S with minimum size $|S|$ whose removal disconnects G , and the size $|S|$ is called the *unweighted vertex connectivity* of G .

In this paper, we study fast algorithms for computing minimum vertex cuts, a topic that algorithm designers have extensively studied for over half a century [Kle69]. Before we survey its development below, observe that we can straightforwardly find a minimum vertex cut by solving (s, t) -maximum flow between all pairs of vertices. This algorithm requires $\binom{n}{2}$ maxflow calls, which takes $\widehat{O}(n^4)$ time by [Bra+23]. We will refer to this time bound as a *baseline*.²

Early Development Within the Baseline. Since the 60s, there has been an active line of work on fast unweighted vertex connectivity [Kle69; Pod73; ET75; Eve75; Gal80; EH84; Mat87]. All of these algorithms are deterministic. The fastest among them [Eve75; Gal80] make $O(n + \kappa^2)$ maxflow calls, where κ denotes the vertex connectivity. However, when $\kappa = \Omega(n)$, they still require $\Omega(n^2)$ maxflow calls, which is as slow as the straightforward algorithm.

Beating the Baseline via Randomization. In the 80s, Becker et al. [Bec+82] were the first to surpass the baseline for unweighted vertex connectivity. Their randomized algorithm makes $\tilde{O}(n)$ maxflow calls, which takes $\tilde{O}(mn^{1.5}) = \tilde{O}(n^{3.5})$ time using the available flow algorithm. Then, Linial, Lovász, and Wigderson [LLW88] discovered a connection between vertex connectivity and rubber band networks, leading to a randomized algorithm with $\tilde{O}(n^\omega + n\kappa^\omega) = O(n^{3.373})$ time.³

Later in the 90s, Henzinger, Rao, and Gabow [HRG00] surpassed the baseline in the *weighted* case. Their randomized algorithm in weighted graphs takes $\tilde{O}(mn) = \tilde{O}(n^3)$ time, improving upon [LLW88]. Also, they showed a *deterministic* algorithm in weighted graphs with $O(m^2)$ time. This bound, however, still does not beat the baseline on dense graphs.

Beating the Baseline Deterministically: Unweighted Case. The first and only deterministic algorithm that surpassed the baseline was discovered by Gabow [Gab06]. He introduced a novel use of Ramanujan expanders and developed an algorithm for unweighted graphs that makes $O(n + \kappa\sqrt{n})$ maxflow calls, which takes $\widehat{O}(m(n + \kappa\sqrt{n})) = \widehat{O}(mn^{1.5})$ time [Bra+23].⁴

Gabow also posed an open problem: whether there exists a $O(mn)$ -time *deterministic* algorithm for unweighted graphs. There has been no progress toward answering this question until now.

Near-Optimal Time via Randomization. Around 2020, there was again significant progress in randomized algorithms. For unweighted graphs, a $\tilde{O}(m\kappa^2)$ -time algorithm was developed [NSY19a; For+20] using novel local cut algorithms. Later, through a new kernelization technique, Li et al. [Li+21] showed how to compute unweighted vertex connectivity using polylogarithmic maxflow calls, leading to an almost-optimal $\widehat{O}(m)$ time. Additionally, [CQ21] presented a randomized algorithm for $(1 + \epsilon)$ -approximate weighted vertex connectivity using polylogarithmic maxflow calls via the minimum isolating cut technique [LP20; AKT21].

The Current State. So far, progress on deterministic algorithms has been much slower, and to advance any further, we need to answer one of the long-standing open problems listed below:

1. Can we beat Gabow's $\widehat{O}(mn^{1.5})$ -time algorithm for unweighted vertex connectivity?
2. Can we even beat the $O(n^4)$ -time baseline for weighted vertex connectivity?

²We use this bound solely for comparison purposes and do not apply it unfairly. Specifically, we do not claim an algorithm is slower than the baseline, while it can actually surpass the baseline using faster maximum flow algorithms.

³ ω is the matrix multiplication exponent.

⁴The time bound stated in [Gab06] was $O(\min\{n^{0.75}, \kappa^{1.5}\}\kappa m + mn) = O(n^{3.75})$ because he utilized the flow algorithms available at that time. In Section E.6, we give a new analysis of his running time in terms of maxflow calls.

In contrast, for randomized algorithms, there is a $\tilde{O}(m)$ -time unweighted algorithm [Li+21] and a $\tilde{O}(mn)$ -time weighted algorithm [HRG00]. Can we narrow this randomized-deterministic gap?

Randomized vs. Deterministic Gaps. In the area of graph algorithms, closing the gaps between deterministic and randomized algorithms has been a fruitful research program. In the closely related problem of minimum *edge* cut, significant efforts [KT15; HRW17; Sar21; LP20; Li21; Hen+24] to match Karger’s $\tilde{O}(m)$ -time algorithm [Kar00] with deterministic algorithms have generated several influential techniques, including the minimum isolating cut [LP20; AKT21; Abb+22], new applications of local flow techniques [KT15; HRW17; SW19; For+20; Cha+21], strong decomposition [KT15; Hen+24], and expander decomposition [Sar21; Li21]. Other related examples include deterministic algorithms for expander decomposition and their numerous applications [Chu+20; LS21; Gor+21] and deterministic min-cost flow algorithms [Bra+23; Che+24; Van+24].

Directed Case. The minimum vertex cut problem can be generalized to *directed* graphs in both weighted and unweighted cases. The goal here is to find the smallest vertex set S such that $G \setminus S$ is not *strongly* connected. Since Gabow’s algorithm and the straightforward algorithm work on directed graphs with the same guarantee, the two questions above also capture the state of the art for deterministic algorithms in directed graphs.⁵

E.1.1 Our Results

We give a strong affirmative answer to both questions simultaneously, even for directed graphs.

Theorem E.1. *There is a deterministic algorithm that, given a directed graph with n vertices, m edges, and vertex weights in $\{1, \dots, W\}$, outputs a minimum vertex cut in $\widehat{O}(mn \log^4 W)$ time.*

Thus, up to subpolynomial factors, Theorem E.1 resolved the open problem posed by Gabow whether a $O(mn)$ -time deterministic algorithm exists for even unweighted graphs. Compared with the state of the art in weighted graphs, it matches the fastest randomized $\tilde{O}(mn)$ -time algorithm [HRG00] and improves upon the deterministic $O(m^2)$ -time algorithm [HRG00].

Furthermore, we present an even faster algorithm for undirected unweighted graphs.

Theorem E.2. *There is a deterministic algorithm that, given an undirected unweighted graph with n vertices, m edges, and vertex connectivity $\kappa \leq n$, outputs a minimum vertex cut in $\widehat{O}(m\kappa)$ time.*

This algorithm is strictly faster than *all* known deterministic algorithms for unweighted vertex connectivity for moderate κ , including the $\widehat{O}(m2^{O(\kappa^2)})$ -time algorithm by Saranurak and Yingchareonthawornchai [SY22], Even’s algorithm [Eve75], which requires $O(n + \kappa^2)$ max-flow calls, and Gabow’s algorithm [Gab06], which requires $O(n + \kappa\sqrt{n})$ max-flow calls. Recently, Korhonen [Kor24] has shown an $O(m \cdot \kappa^{O(\kappa^2)})$ -time deterministic algorithm, which is linear when κ is small.

Deterministic Barrier to $o(mn)$ Time. When $\kappa = \Omega(n)$, Theorems E.1 and E.2 take $\Omega(mn)$ time. This suggests a natural question: Can we strictly beat the $O(mn)$ bound deterministically? Below, we highlight that the bottleneck is essentially captured by the hardness of this question.⁶

Let $G = (V, E)$ be an undirected unweighted graph. We say that (L, S, R) is a vertex cut if there is no edge between L and R . Our question is how to solve the problem even if all minimum cuts are “perfectly balanced.”

Question E.3. *Suppose that all minimum vertex cuts (L, S, R) of a graph G are such that $|L|, |S|, |R| = \Omega(n)$. Compute the vertex connectivity of G in $o(mn)$ time.*

⁵The state of the art for randomized algorithms on directed graphs is slightly different. The almost-linear time unweighted algorithm by [Li+21] also works in directed graphs but has a slower running time of $\widehat{O}(n^2)$ instead of $\widehat{O}(m)$. The $(1 + \epsilon)$ -approximation weighted $\widehat{O}(m)$ -time algorithm by [CQ21] does not work in directed graphs; however, [Cen+22] works in $\widehat{O}(n^2)$ time.

⁶The only other bottleneck is a technicality related to deterministic sparse recovery, which seems solvable.

With randomization, this question is easily solvable in $\widehat{O}(m)$ time by random sampling.⁷ Indeed, the $\widehat{O}(m)$ -time algorithm [Li+21] highly exploits this. It is striking that none of the deterministic techniques can yield a $o(mn)$ -time algorithm; we leave this as a challenging open problem.

Historical Note. The preliminary version of this paper [NSY23] only solves undirected unweighted deterministic vertex connectivity in $\widehat{O}(m\kappa^2)$ time.

E.1.2 Our Tools

Next, we highlight the four technical tools that we developed and may be useful for future applications. The first two are employed for both [Theorems E.1](#) and [E.2](#), while the last two are used only in [Theorem E.2](#). To introduce the techniques in an intuitive way, we focus on unweighted undirected graphs in this section.

Crossing Family.

At a high level, a *crossing family* $\mathcal{P}$ is a family of pairs of elements such that, for every two sufficiently large sets L and R , there must exist a “crossing” pair $(x, y) \in \mathcal{P}$ where $x \in L$ and $y \in R$.

Below, we provide a formal definition of our crossing family, where L and R can belong to two different universes A and B , respectively.

Definition E.4 (Asymmetric crossing family). Given two sets A, B and two integers ℓ, r where $\ell \leq |A|$, $r \leq |B|$, and $\ell \leq r$, an (A, B, ℓ, r) -crossing family $\mathcal{P}$ is a subset of $A \times B$ such that, for any $L \subseteq A$ and $R \subseteq B$ with sizes at least ℓ and r , respectively, we have $\mathcal{P} \cap (L \times R) \neq \emptyset$. The degree of an element $x \in A$ in $\mathcal{P}$ is $|\{y \in B \mid (x, y) \in \mathcal{P}\}|$.

Our algorithm in weighted graphs relies crucially on the asymmetric crossing family. We show a construction with the following guarantee.

Theorem E.5. *There exists a deterministic algorithm that, given two sets A, B and two integers ℓ, r with $\ell \leq |A|$, $r \leq |B|$, and $\ell \leq r$, outputs an (A, B, ℓ, r) -crossing family in $\tilde{O}(|A| \cdot \max\{1, \frac{|B|-r}{\ell}\})$ time such that the degree of every element in A is at most $\tilde{O}(\max\{1, \frac{|B|-r}{\ell}\})$.*

This family immediately implies the symmetric version of a crossing family that appears more applicable for finding a minimum vertex cut.

Corollary E.6 (Symmetric crossing family). *There exists a deterministic algorithm that, given an integer n and a parameter $\alpha \geq 1$, outputs $\mathcal{P}$ of size $\tilde{O}(\alpha n)$, a subset of $[n] \times [n]$ in time $\tilde{O}(\alpha n)$, such that for any partition (L, S, R) of $[n]$ that satisfies $|R| \geq |L| \geq |S|/\alpha$, we have $\mathcal{P} \cap (L \times R) \neq \emptyset$. Moreover, for each $x \in [n]$, the out-degree of x in $\mathcal{P}$, i.e., $|\{y \mid (x, y) \in \mathcal{P}\}|$, is at most $\tilde{O}(\alpha)$.*

We will also refer to the above $\mathcal{P}$ as a (n, α) -crossing family.

Use Case. To see why [Corollary E.6](#) is useful, consider the graph from [Question E.3](#). There must be a pair from the $(n = |V|, \alpha = \Omega(1))$ -crossing family $\mathcal{P}$ from [Corollary E.6](#) that crosses a minimum vertex cut (L, S, R) . Therefore, if we solve the (x, y) -maximum flow for each $(x, y) \in \mathcal{P}$, we will find the minimum vertex cut using $\tilde{O}(n)$ max flow calls instead of trivially using $\binom{n}{2}$ max flow calls.

Our Construction: Dispersers. Both [Theorem E.5](#) and [Corollary E.6](#) are proved in [Section E.5.1](#) using *dispersers* [WZ99; TUZ01], which are well-known objects for constructing randomness extractors. To our knowledge, this is the first direct application of dispersers in fast graph algorithms.

⁷Sample $O(\log n)$ pairs of vertices. With high probability, one of the pairs (a, b) is such that $a \in L$ and $b \in R$ for some minimum vertex cut (L, S, R) . Assuming this, the (a, b) -maximum flow must reveal a minimum vertex cut.

Previous Construction: Ramanujan Expanders. Gabow [Gab06] implicitly showed that the edges of Ramanujan expanders can be used to construct a crossing family in the setting of [Corollary E.6](#) that consists of $\tilde{O}(n^3/(|L||R|))$ pairs (see [Lemma E.75](#)).

Our crossing family from [Corollary E.6](#) subsumes this bound and strictly improves upon it, particularly when $|L| \approx |S| \ll n$. In this situation, the family size from [Corollary E.6](#) would be $\tilde{O}(n)$, while Gabow's family size would be $\tilde{O}(n^2/|L|)$. This improvement is crucial for our purposes.

Common Neighborhood Clustering

Crossing families are useful when there exists a minimum vertex cut (L, S, R) where $|L| = \Omega(|S|)$. To address the opposite case where $|L| = o(|S|)$, we exploit the concept of *common neighborhood clustering*, introduced recently in [Bli+25]. Below, $N_G(u)$ denotes the neighbors of u in G .

Lemma E.7 (Common-Neighborhood Clustering). *For any graph $G = (V, E)$ and parameter ℓ , there exists a collection of clusters $V_1, \dots, V_z \subseteq V$ with the following properties. For any minimum vertex cut (L, S, R) where $|L| \leq \ell$,*

1. $L \subseteq V_i$ for some cluster V_i ,
2. for every cluster V_i and for every $u, v \in V_i$, the symmetric difference

$$|N_G(u) \triangle N_G(v)| = O(\ell \log n), \text{ and}$$

3. every vertex belongs to at most $O(\log n)$ clusters.

Use Case. [Lemma E.7](#) is surprisingly useful in many ways. The intuition is as follows. First, we know that the minimum vertex cut (L, S, R) is “captured” by some cluster V_i because, by [Item 1](#), both L and S are contained in $N_G[V_i] := N_G(V_i) \cup V_i$ for some V_i . Second, the structure of inside each cluster V_i can greatly speed up the flow computation. Roughly speaking, suppose we want to compute a (s, t) -minimum vertex cut where $s, t \in V_i$, by using the existing idea from [Li+21; Bli+25], we are able to delete all edges⁸ from a node $u \in V_i$ to a node in $N(s)$, without changing the (s, t) -minimum vertex cut. But after deletion, by [Item 2](#), there are only $O(\ell \log n)$ edges left adjacent to every vertex except s, t . Thus, this clustering is very effective in the case when ℓ is small, e.g., when $|L| \ll |S|$.

Lastly, even if we try to work on each cluster separately, the total work is small because the total number of edges of the graph induced by $N_G[V_i]$ overall clusters V_i is at most $\tilde{O}(m)$ by [Item 3](#).

Previous Usage: Randomized, Unweighted, Undirected. Common-neighborhood clustering in [Bli+25] was used in randomized reductions from global to (s, t) -minimum vertex cuts in parallel and distributed models. However, their algorithms are randomized and work only for unweighted undirected graphs.

Our Usage: Deterministic, Weighted, Directed. To prove [Theorem E.1](#), we extend the reach of this technique much further into the deterministic setting, and also to directed and weighted graphs. See [Section E.3](#) for details on how we combine both crossing families and common-neighborhood clustering to work in weighted directed graphs.

Terminal Reduction

The next tool is specific to unweighted undirected graphs. It handles the case when $|L| = \Omega(|S|)$, where (L, S, R) is a minimum vertex cut. Below, we say that a terminal set $T \subseteq V$ is α -balanced if there is a minimum vertex cut (L, S, R) such that $|T \cap L| \geq \alpha|S|$, and $|T \cap R| \geq \alpha|S|$.

⁸We are omitting details here. The correct operation is to first make each undirected as two bi-direction directed edges, then delete every directed edge from u to $N(s)$, then make every remaining directed edge as undirected.

Lemma E.8 (Terminal Reduction (Simplified version of Lemma E.25)). *There is a deterministic algorithm that, given an n -vertex m -edge undirected graph $G = (V, E)$ along with a $2^{\sqrt{\log n}}$ -balanced terminal set $T \subseteq V$, returns a new terminal set $T' \subseteq V$ of size at most $0.9|T|$ and a separator S^* such that either*

- T' is 0.1-balanced, or
- $|S^*| = \kappa_G$.

The algorithm runs in $\widehat{O}(m\kappa)$ time.

Use Case. Let us see why Lemma E.8 is useful. Initially, set $T \leftarrow V$. If T is not 0.1-balanced, then $|L| \leq 0.1|S|$, which corresponds to the “unbalanced” case that can be handled using common neighborhood clustering. Next, as long as T is $2^{\sqrt{\log n}}$ -balanced, a call to Lemma E.8 will either reduce the size of T or yield a minimum vertex cut S^* . After at most $O(\log n)$ rounds, we will either obtain a minimum vertex cut or a terminal set T that is not $2^{\sqrt{\log n}}$ -balanced but is 0.1-balanced. This is precisely when our last tool, called a *selector*, becomes useful, as we will explain last.

Our Proof: New Use of Common-Neighborhood Clustering. The proof of Lemma E.8 turns out to be the most technical part of the paper. Surprisingly, the key ingredient is again common-neighborhood clustering, but we apply it *on top of vertex-expander decompositions*. This gives a completely novel use of the technique. See Section E.4.1 for details.

Previous Terminal Reductions. Theorem 3.3 in [SY22] gives a $\widehat{O}(m2^{O(\kappa^2)})$ -time algorithm for performing *vertex reduction*. It serves the same purpose as Lemma E.8, but instead reduces the total graph size rather than the number of terminal vertices. However, the $2^{O(\kappa^2)}$ -factor seems very challenging to improve to $\text{poly}(\kappa)$, as doing so would imply a breakthrough for vertex sparsifiers.

The similar terminal reduction for minimum *edge* cuts [LP20] (Section 4.2) is much simpler to obtain. Ultimately, the underlying reason is that vertex mincuts can separate a graph into many connected components, while edge mincuts always separate a graph into two components.

Selectors

The final tool for our $\widehat{O}(m\kappa)$ -time algorithm for unweighted undirected graphs is a pseudorandom object called *selectors*. Intuitively, a selector is a family $\mathcal{S}$ of subsets that, given any two disjoint sets L and S where $|L| \geq \varepsilon|S|$, there must exist a set $U \in \mathcal{S}$ that “hits L once” and “misses S .”

Definition E.9 ((n, k, ε) -selector). For every positive integers n, k with $k < n$ and parameters $\varepsilon \in [0, 1]$, an (n, k, ε) -selector $\mathcal{S}$ is a family of subsets of $[n]$ such that, for every two disjoint subsets L and S of $[n]$ with sizes satisfying $\varepsilon k < |L| \leq k$ and $|S| \leq k$, there exists a set $U \in \mathcal{S}$ such that $|U \cap L| = 1$ and $|U \cap S| = 0$.

We prove the following theorem about selectors in Section E.5.2.

Theorem E.10. *For any integer n and parameter $\varepsilon \geq 1/n^{o(1)}$, there exists $\tilde{k} \geq n^{1-o(1)}$ depending on ε such that, given any $k \leq \tilde{k}$, we can compute a (n, k, ε) -selector $\mathcal{S}$ of size at most $|\mathcal{S}| = \widehat{O}(k)$ in $\widehat{O}(n)$ time, and every set $U \in \mathcal{S}$ has size at least 2.*

Use Case. Following the discussion below Lemma E.8, the remaining case to handle is when the terminal set T_0 is not $2^{\sqrt{\log n}}$ -balanced but is 0.1-balanced. That is, there is a minimum vertex cut (L, S, R) where $0.1|S| \leq |T_0 \cap L| \leq 2^{\sqrt{\log n}}|S| = \widehat{O}(\kappa)$.

Construct a $(n = |T_0|, k = |T_0 \cap L|, \varepsilon = 0.1/2^{\sqrt{\log n}})$ -selector $\mathcal{F}$ on T_0 . By Theorem E.10, $|\mathcal{F}| \leq \widehat{O}(|T_0 \cap L|)$, and for some $U \in \mathcal{F}$, a minimum vertex cut (L, S, R) must *isolate* U , meaning that $|L \cap U| = 1$, $|S \cap U| = 0$, and $|R \cap U| > 0$ because $|U| \geq 2$. Thus, for each $U \in \mathcal{F}$, we compute minimum isolating vertex cuts of U in $O(\log |U|)$ maxflow calls by [Li+21].⁹ This must reveal the minimum vertex cut. The total running time is $\widehat{O}(m) \cdot |\mathcal{F}| \leq \widehat{O}(m\kappa)$.

⁹The minimum isolating cuts algorithm of [Li+21] only works on undirected graphs.

Our Construction: Linear Lossless Condensers. It is crucial that our selector has size $|\mathcal{S}| = \widehat{O}(k)$ so that our final running time is $\widehat{O}(m\kappa)$. Selectors of this size exist in the literature [CGR02; Ind02; CK05], but they do not explicitly guarantee that each set $U \in \mathcal{S}$ has size $|U| \geq 2$.¹⁰

To address this issue, we construct the selectors ourselves. To ensure that $|U| \geq 2$, we use a lossless condenser from [GUV09] that is also *linear*, as constructed by [Che11]. Given the right object, the construction is straightforward.

Related Pseudorandom Objects. Hit-and-miss families from [KP21] or splitters (e.g., [LP20] and many results in the FPT literature) have been used in other graph algorithms and can be used to construct selectors.¹¹ Unfortunately, both objects are too strong and would imply a selector of size $\Omega(k^2)$ [KP21], resulting in a final running time of $\Omega(m\kappa^2)$, which is too slow.

E.2 Preliminaries

We use $\text{polylog}(f)$ to denote $\log^c(f)$ for some constant c . We use $\widetilde{O}(f)$ to hide poly-logarithmic factors and $\widehat{O}(f)$ to hide sub-polynomial factors, i.e., $\widetilde{O}(f) = f \cdot \text{polylog}(f)$ for some constant c and $\widehat{O}(f) = f \cdot f^{o(1)}$.

For a set A , we use 2^A to denote the power set of A , i.e., each element in 2^A is a subset of A . For two sets A, B , we use $A \Delta B = (A - B) \cup (B - A)$ to denote the symmetric difference between A and B .

Directed weighted graph. A directed weighted graphs is denoted by $G = (V, E, w)$ where V is the vertex set, $E \subseteq V \times V$ is the edges set and $w : V \rightarrow \mathbb{N}^+$ assigns a positive integer weight to each vertex. We always use $n = |V|, m = |E|$ to denote the number of nodes and edges. We use W to denote the maximum weighted, i.e., $W = \max_{v \in V} w(v)$. We use G^R to denote the reversed graph $(V, E^R = \{(v, u) \mid (u, v) \in E\}, w)$.

For $u \in V$, we use $N_G^{\text{out}}(u) = \{v \in V \mid (u, v) \in E\}$ to denote the out-neighborhood set of u . For simplicity, we also use $N_G(u)$ to denote $N_G^{\text{out}}(u)$. We write $N_G[u] = N_G(u) \cup \{u\}$. For a vertex set $A \subseteq V$, we use $G[A] = (A, \{(u, v) \in E \mid u, v \in A\})$ to denote the induced subgraph on A . We write $w(A) = \sum_{v \in A} w(v)$. We use $N_G(A) = (\cup_{v \in A} N_G(v)) - A$ to denote the neighborhood set of A . We write $N_G[A] = N_G(A) \cup A$. For two vertex sets $A, B \subseteq V$, we use $\overrightarrow{E}_G(A, B)$ to denote the set of edges from A to B , and $\overleftarrow{E}_G(A, B)$ to denote the edges from B to A . When the context is clear, G is omitted in the subscript.

Undirected unweighted graphs. Let $G = (V, E)$ be an undirected unweighted graph. Let $A, B \subseteq V$. We use $E(A, B)$ to denote the set of edges (u, v) where $u \in A, v \in B$. We use $N_G(u)$ to denote the neighborhood set of u in G , and $N_G[u] = N_G(u) \cup \{u\}$. We use $N_G(A) = (\cup_{u \in A} N(u)) - A$ to denote the neighborhoods of A and $N_G[A] = N_G(A) \cup A$. The minimum degree $\delta_G = \min_{u \in V} |N_G(u)|$.

Vertex cut. A *vertex cut* is a tri-partition of the graph (L, S, R) ¹² with the guarantee that there are no edges from L to R . We call S a *separator*. The weight of a vertex cut (L, S, R) is $w(S)$, and the size of the vertex cut is $|S|$. A vertex cut or separator is *minimum* if its weight is minimum (or the size is minimum if the graph is unweighted).

(s, t) -vertex cut. A vertex cut (L, S, R) is a (s, t) -vertex cut for two vertices $s, t \in V$ if $s \in L, t \in R$. We call S as a (s, t) -separator. We use $\kappa_G(s, t)$ to denote the weight of the minimum weight (s, t) -vertex cut (or the size if the graph is unweighted). When the context is clear, we omit G in the subscript. By folklore reduction, we can find a minimum (s, t) -separator by one call to max flow.

Lemma E.11 ([Bra+23]). *There is a deterministic algorithm finding a minimum (s, t) -separator in $\widehat{O}(m) \cdot \log W$ time.*

¹⁰Definition E.9 generalizes the selector defined in [CGR02; Ind02], which takes $\varepsilon = 1/2$. A different notation for the (n, k, r) -selector defined in [CK05] implies our (n, k, ε) -selector above when $r = (1 - \varepsilon)k$.

¹¹At a high level, a hit-and-miss family is the same as a selector but does not require that $|L| \geq \varepsilon|S|$. A splitter is even stronger; it further guarantees that, for every $x \in L$, there exists a set U such that $U \cap L = \{x\}$ and $U \cap S = \emptyset$.

¹²Which means $L \cup S \cup R = V$ and L, S, R are mutually disjoint.

E.3 An $\hat{O}(mn)$ -Time Algorithm for Weighted Directed Graphs

Suppose (L, S, R) is one of the minimum vertex cuts. We will assume $w(L) \leq w(R)$ by working on both G and the reversed graph G^R . Let λ be a sufficiently large constant throughout this section. In [Sections E.3.2](#) and [E.3.3](#), we will show how to handle the case when $w(R) > \lambda w(L) \log n$ and $w(R) \leq \lambda w(L) \log n$, separately.

Lemma E.12 (Lopsided Case). *There is a deterministic algorithm that given a directed weighted graph $G = (V, E, w)$, outputs a vertex cut in $\hat{O}(mn) \cdot \log^5 W$ time. Furthermore, if one of the minimum vertex cut (L, S, R) in G satisfies $w(R) > \lambda w(L) \log n$, then the output is a minimum vertex cut of G .*

Lemma E.13 (Symmetric Case). *There is a deterministic algorithm that given a directed weighted graph $G = (V, E, w)$, outputs a vertex cut (L, S, R) in $\hat{O}(mn) \cdot \log^4 W$ time. Furthermore, if one of the minimum vertex cut (L, S, R) in G satisfies $w(L) \leq w(R) \leq \lambda w(L) \log n$, then the output is a minimum vertex cut of G .*

The above two lemmas immediately imply [Theorem E.1](#). We give the overview for proving both lemmas next.

E.3.1 Overview

Fix a vertex mincut (L, S, R) in the input weighted directed graph G . We can assume we know the weights $w(L), w(S)$, and $w(R)$ up to a factor of 2 by guessing. Assume also that $w(L) \leq w(R)$. Otherwise, the argument is symmetric.

We improve upon the straightforward algorithm that computes (s, t) -maximum flow overall $\binom{n}{2}$ all pairs (s, t) using the following strategy:

1. Compute an (s, t) -maximum flow only for pairs (s, t) from the certain crossing family $\mathcal{P}$, and
2. Before computing (s, t) -maximum flow, “compress” the graph to reduce the *amortized* size of the maxflow instances. More precisely, each edge appears in at most $\tilde{O}(n)$ maxflow instances.

We emphasize that our goal is to bound the *amortized* size of maxflow instances instead of the *worst-case* size as done in prior work [\[Li+21\]](#). This is a key difference between our approach and the one of [\[Li+21\]](#).

Graph Compression. First, we show how to compress maxflow instances and obtain a $\hat{O}(mn)$ -time algorithm given the following crossing family $\mathcal{P}$ such that

1. There exists $(s, t) \in \mathcal{P}$ with $s \in L$ and $t \in R$, and
2. The *degree* of u in $\mathcal{P}$ defined as $\deg_{\mathcal{P}}(u) := |\{v \mid (u, v) \in \mathcal{P}\}|$ is at most $\tilde{O}(n \cdot w(u)/w(R))$.

For each $(s, t) \in \mathcal{P}$, we will compress the graph without changing the (s, t) -vertex mincut by deleting the edges between vertices in $N_G^{out}(s)$ as also done in [\[Li+21\]](#). This is safe because any (s, t) -vertex maxflow never need to use edges between vertices in $N_G^{out}(s)$.

Now, we argue that, for each edge (u, v) , the total number of maxflow instances that (u, v) is not deleted is $\tilde{O}(n)$. Consider the maxflow instance for $(s, t) \in \mathcal{P}$, if (u, v) is not deleted, this means that either $u \notin N_G^{out}(s)$ or $v \notin N_G^{out}(s)$. That is, either $s \notin N_G^{in}(u)$ or $s \notin N_G^{in}(v)$. It remains to show that the number of pairs $(s, t) \in \mathcal{P}$ where $s \notin N_G^{in}(u)$ or $s \notin N_G^{in}(v)$ is at most $\tilde{O}(n)$.

Indeed, according to the degree bound of $\mathcal{P}$, the total number of such s, t pair is at most

$$\sum_{a \in (V - N_G^{in}(u)) \cup (V - N_G^{in}(v))} \deg_{\mathcal{P}}(a) = \tilde{O} \left(n \cdot \frac{\sum_{a \in (V - N_G^{in}(u)) \cup (V - N_G^{in}(v))} w(a)}{w(R)} \right).$$

Note that $N_G^{in}(u), N_G^{in}(v)$ are vertex cuts and so $w(N_G^{in}(u)), w(N_G^{in}(v)) \geq w(S)$. Hence,

$$\sum_{a \in (V - N_G^{in}(u)) \cup (V - N_G^{in}(v))} w(a) \leq 2 \cdot (w(V) - w(S)) = 2 \cdot (w(L) + w(R)) \leq 4w(R).$$

where the last inequality is by the assumption $w(L) \leq w(R)$. Combining the two inequalities above, we conclude that the total number of $(s, t) \in \mathcal{P}$ where $s \notin N_G^{in}(u)$ or $s \notin N_G^{in}(v)$ is

$$\sum_{a \in (V - N_G^{in}(u)) \cup (V - N_G^{in}(v))} \deg_{\mathcal{P}}(v) = \tilde{O}\left(n \cdot \frac{4w(R)}{w(R)}\right) = \tilde{O}(n).$$

Crossing Family Construction. Now, we construct the desired crossing family $\mathcal{P}$.

Suppose for a moment that we are in the unweighted case. If we apply [Corollary E.6](#), we would get the family $\mathcal{P}$ such that there exists $(s, t) \in \mathcal{P}$ with $s \in L$, $t \in R$ and (ii) the degree of u in $\mathcal{P}$ is at most $\tilde{O}(\alpha) = \tilde{O}(|S|/|L|)$. By weight bucketing and using the fact that $|S| \leq n$, a natural generalization of [Corollary E.6](#) to the weighted case will give us the similar family $\mathcal{P}$ where the degree of u in $\mathcal{P}$ is at most $\tilde{O}(n \cdot w(u)/w(L))$.

In the symmetric setting of [Lemma E.13](#), i.e., $w(R) = \tilde{\Theta}(w(L))$, then the degree bound of u in $\mathcal{P}$ becomes $\tilde{O}(n \cdot w(u)/w(R))$, exactly the property that we want.

Thus, the hard case is in the lopsided setting of [Lemma E.12](#) where possibly $w(L) \ll w(R)$. We sketch our strategy in this case below. The key step is that, without knowing R , we can find a vertex set R' that “approximates R well” in the sense that $w(R' \triangle R) = \tilde{O}(w(L))$. This key step is explained in the next paragraph.

Given R' , we will use the asymmetric crossing family from [Theorem E.5](#) to construct $\mathcal{P}$. Suppose for a moment again that we are in the unweighted case. So $|R' \triangle R| = \tilde{O}(|L|)$. Consider an (A, B, ℓ, r) -crossing family where $A \leftarrow V$, $B \leftarrow R'$, $\ell \leftarrow |L|$, and $r \leftarrow |R \cap R'|$. Using [Theorem E.5](#), we would obtain $\mathcal{P}$ where there is $(s, t) \in \mathcal{P}$ where $s \in L$ and $t \in R \cap R'$ and, for each $u \in V$, the degree of u in $\mathcal{P}$ is at most $\tilde{O}(\frac{|B| - r}{\ell}) = \tilde{O}(\frac{|R'| - |R \cap R'|}{\ell}) = \tilde{O}(\frac{|R' - R|}{\ell}) = \tilde{O}(1)$. Note that it is valid to apply [Theorem E.5](#), which requires $\ell \leq r$, because we are in the lopsided case where R is much larger than L . Formally, the condition $|L| \leq |R \cap R'|$ is valid because R is much larger than L and $|R' \triangle R| = \tilde{O}(|L|)$, i.e., R and R' are basically the same (up to $\tilde{O}(|L|)$ elements).

By weight bucketing, a natural generalization of [Theorem E.5](#) to the weighted case (formally shown in [Lemma E.17](#)) will give a degree bound of $\tilde{O}(n \cdot w(u)/w(R))$, exactly what we want. To see this, since $R' - R$ takes a fraction of $\tilde{O}(w(L)/w(R))$ weight in R , there must exist a bucket such that $R - R'$ takes a fraction of $\tilde{O}(w(L)/w(R))$ vertices in R' . However, R' can be as large as n , so the degree bound becomes $\tilde{O}\left(\frac{n \cdot (w(L)/w(R))}{w(L)}\right) = \tilde{O}(n/w(R))$. The additional $w(u)$ factor comes from the fact that we also bucket the weights in $A \leftarrow V$, which is similar to duplicating a vertex u for $w(u)$ times.

Approximate R without Knowing It. To explain the idea of how to find R' such that $w(R \triangle R') = \tilde{O}(w(L))$ without knowing R . We will further assume here that the graph satisfies *common-neighborhood* property, i.e., for every $u, v \in V$, we assume $w(N_G^{out}(u) \triangle N_G^{out}(v)) \leq \tilde{O}(w(L))$. This assumption can be removed using the common-neighborhood clustering from [Lemma E.7](#), generalized to the weighted case in [Lemma E.14](#).

First, observe that, for every node $u \in L$, since $w(N_G^{out}(u)) \geq w(S)$ and $N_G^{out}(u) \subseteq L \cup S$, we have $w(N_G^{out}(u) \triangle S) = O(w(L))$. With the common-neighborhood property, we further deduce that, for every $v \in V$ we must have $w(N_G^{out}(v) \triangle S) = O(w(L))$. Intuitively, this means that S receives a lot of edges from every node in V , which gives us enough information for S so that we can approximately find S' with $w(S' \triangle S) = \tilde{O}(w(L))$. Simply by setting $R' = V - S'$, we can get $w(R' \triangle R) = \tilde{O}(w(L))$. This is formally explained in [Lemma E.16](#).

E.3.2 Lopsided Case: Proof of [Lemma E.12](#)

Step 1: common-neighborhood clustering. We first compute a common-neighborhood clustering described in [Lemma E.14](#) below. In [Lemma E.14](#), we should think of ℓ as an approximation of $w(L)$ where (L, S, R) is a minimum lopsided vertex cut. ℓ can be guessed by the powers of 2.

Lemma E.14 (Common-neighborhood Clustering). *There is a deterministic algorithm that given a directed weighted graph $G = (V, E, w)$ and an integer ℓ , outputs a set of clusters $\mathcal{C} \subseteq 2^V$ in $\tilde{O}(mn)$ time such that*

1. (sparse) for every $v \in V$, there are at most $O(\log n)$ sets in $\mathcal{C}$ that contains v ,
2. (common-neighborhood) for every $C \in \mathcal{C}$ and $u, v \in C$, $w(N_G^{\text{out}}(u) \Delta N_G^{\text{out}}(v)) = O(\ell \log n)$.
3. (cover) for every minimum vertex cut (L, S, R) with $w(L) \leq \ell$, there is $C \in \mathcal{C}$ where $L \subseteq C$.

Proof. Define an undirected graph $G' = (V, E')$ where $E' = \{(u, v) \mid w(N_G^{\text{out}}(u) \Delta N_G^{\text{out}}(v)) \leq 2\ell\}$. G' can be constructed in $O(mn)$ time by explicitly computing $N_G^{\text{out}}(u) \Delta N_G^{\text{out}}(v)$ for any u, v .

We run *sparse neighborhood cover* [Awe+98] on G' to get a set of clusters $\mathcal{C} \subseteq 2^V$, which has running time nearly linear on the size of G' , which is $\tilde{O}(n^2) = \tilde{O}(mn)$. According to [Awe+98], $\mathcal{C}$ has the following properties.

1. **For every $v \in V$, there are at most $O(\log n)$ sets in $\mathcal{C}$ that contains v .** This gives the (sparse) property as in Lemma E.14.
2. **The diameter of each $C \subseteq \mathcal{C}$ is at most $O(\log n)$.** This means there is a path with length $O(\log n)$ from u to v in C for any $u, v \in C \in \mathcal{C}$. According to the definition of G , if there is an edge $(a, b) \in E'$, then we can add and delete vertices with weighted at most 2ℓ from $N_G^{\text{out}}(a)$ to get $N_G^{\text{out}}(b)$. Thus, we can add and delete vertices with weight at most $2\ell \cdot O(\log n)$ from $N_G^{\text{out}}(u)$ to get $N_G^{\text{out}}(v)$ since there is a path with length $O(\log n)$ connecting them, which means $w(N_G^{\text{out}}(u) \Delta N_G^{\text{out}}(v)) = O(\ell \log n)$, giving (common-neighborhood).
3. **For every $v \in V$, there exists $C \in \mathcal{C}$ with $N_{G'}[v] \subseteq C$.** We will prove that, for every two vertices $u, v \in L$, we have $w(N_G^{\text{out}}(u) \Delta N_G^{\text{out}}(v)) \leq 2\ell$. This implies that $L \subseteq N_{G'}[u]$. Since we know that $N_{G'}[u] \subseteq C$ for some $C \in \mathcal{C}$, this implies (cover). Notice that this proof appears in many previous works [Li+21; Bli+25], and we prove it here for completeness.

To prove $w(N_G^{\text{out}}(u) \Delta N_G^{\text{out}}(v)) \leq 2\ell$, notice that $w(N_G^{\text{out}}(u) \Delta N_G^{\text{out}}(v)) = 2w(N_G^{\text{out}}(u) \cup N_G^{\text{out}}(v)) - w(N_G^{\text{out}}(u)) - w(N_G^{\text{out}}(v))$. We have $w(N_G^{\text{out}}(u) \cup N_G^{\text{out}}(v)) \leq w(S) + w(L)$ since $N_G^{\text{out}}(u), N_G^{\text{out}}(v) \subseteq L \cup S$. We also have $w(N_G^{\text{out}}(u)) \geq w(S)$ since (L, S, R) is a minimum vertex cut of G and $N_G^{\text{out}}(u)$ is a separator of G . Thus, we get $w(N_G^{\text{out}}(u) \Delta N_G^{\text{out}}(v)) \leq 2(w(L) + w(S)) - w(S) - w(S) = 2w(L) \leq 2\ell$.

The output of the sparse neighborhood cover algorithm on G' gives the common-neighborhood clustering as desired. $\square$

To better convey the idea, the rest of the algorithm will be described based on the following assumption.

Assumption E.15. *(L, S, R) is a minimum vertex cut of G which satisfies $w(R) > \lambda w(L) \log n$, and we are given a cluster $C \in \mathcal{C}$ with $L \subseteq C$ (where $\mathcal{C}$ is constructed by Lemma E.14).*

Assumption E.15 can be removed by running the rest of the algorithm for every $C \in \mathcal{C}$. Moreover, we assume that we know the approximated value of $w(L)$ and $w(R)$ denoted by ℓ and r . To remove this assumption, we can guess ℓ, r by the powers of 2.

Step 2: identifying a set largely intersecting R . In this step, we will construct a set that can be modified from R by adding or deleting vertices with total weight at most $O(w(L) \log n)$, i.e., it intersects R a lot.

Define $V_{\text{high}} = \{v \in V \mid w(N_G^{\text{in}}(v) \cap C) > 0.9w(C)\}$, i.e., V_{high} contains all the nodes which has in-neighbors almost covering C . Define $V_{\text{low}} = V - V_{\text{high}}$. We claim that V_{low} is the set that largely intersects R .

Lemma E.16. *Assume Assumption E.15, we have $w(V_{\text{low}} \Delta R) = O(w(L) \log n)$.*

Proof. We first show that it suffices to prove $w(V_{\text{high}} \Delta S) = O(w(L) \log n)$. Suppose we have $w(V_{\text{high}} \Delta S) = O(w(L) \log n)$. We show that the total weight changes due to swapping in and swapping out vertices from the set V_{low} to get the set R is $O(w(L) \log n)$. Since $V_{\text{low}} = V - V_{\text{high}}$, we can add or delete vertices with

weight at most $O(w(L) \log n)$ from V_{low} to get $V - S$. Notice that $V - S = L \cup R$, thus, by further deleting L from $V - S$ (which has weight $w(L)$), we get R . Thus, $w(V_{low} \triangle R) = O(w(L) \log n)$ as desired.

We prove that $w(S - V_{high}) = O(w(L) \log n)$. Define the potential function $Q = \sum_{(u,v) \in E(C,S)} w(u) \cdot w(v)$. One way to express Q is by $Q = \sum_{v \in C} w(v) w(N^{out}(v) \cap S)$. Let $x \in L \subseteq C$ (remember that L, S, R are defined in [Assumption E.15](#)) be an arbitrary vertex in L , we have $w(N^{out}(x) \cap S) \geq w(S) - w(L)$ since $w(N^{out}(x)) \geq w(S)$ (as S is a minimum separator) and $N^{out}(x) \subseteq L \cup S$. According to (common-neighborhood) property of [Lemma E.14](#), for any $v \in C$, we have $w(N^{out}(v) \triangle N^{out}(x)) \leq O(w(L) \log n)$, which means

$$w(N^{out}(v) \cap S) \geq w(S) - w(L) - O(w(L) \log n) \geq w(S) - O(w(L) \log n). \quad (\text{E.1})$$

Thus, we get

$$\begin{aligned} Q &= \sum_{v \in C} w(v) w(N^{out}(v) \cap S) \\ &\geq \sum_{v \in C} w(v) \cdot (w(S) - O(w(L) \log n)) \\ &= w(C) \cdot (w(S) - O(w(L) \log n)) \end{aligned}$$

On the other hand, Q can be expressed by $Q = \sum_{v \in S} w(v) \cdot w(N^{in}(v) \cap C)$, which according to the definition of V_{high} is

$$\begin{aligned} Q &= \sum_{v \in S} w(v) w(N^{in}(v) \cap C) \\ &\leq \left(\sum_{v \in S \cap V_{high}} w(v) \cdot (w(C)) \right) + \left(\sum_{v \in S \cap V_{low}} w(v) \cdot (0.9w(C)) \right) \\ &= (w(S \cap V_{high}) + 0.9w(S \cap V_{low})) \cdot w(C) \end{aligned}$$

By combining the two inequalities and canceling the $w(C)$ factor, we get

$$\begin{aligned} w(S \cap V_{high}) + 0.9w(S \cap V_{low}) &\geq w(S) - O(w(L) \log n) \\ O(w(L) \log n) &\geq 0.1w(S \cap V_{low}) \end{aligned}$$

i.e., $w(S - V_{high}) = O(w(L) \log n)$.

Next, we prove that $w(V_{high} - S) = O(w(L) \log n)$. Define $Q' = \sum_{(u,v) \in E(C, V_{high} - S)} w(u) w(v)$. One way to express Q' is $Q' = \sum_{v \in C} w(v) w(N^{out}(v) \cap (V_{high} - S))$. For all $v \in C$, we claim that

$$w(N^{out}(v) \cap (V_{high} - S)) \leq w(N^{out}(v)) - w(N^{out}(v) \cap S) \leq O(w(L) \log n).$$

The last inequality follows because $w(N^{out}(v) \cap S) \geq w(S) - O(w(L) \log n)$ by [\(E.1\)](#). Also, we have $w(N^{out}(v)) \leq w(L \cup S) + O(w(L) \log n)$. This is because for any vertex $x \in L \subseteq C$, $w(N^{out}(x)) \leq w(L \cup S)$ and $w(N^{out}(v) \triangle N^{out}(x)) \leq O(w(L) \log n)$ by the (common-neighborhood) property of [Lemma E.14](#). Thus,

$$Q' = \sum_{v \in C} w(v) \cdot w(N^{out}(v) \cap (V_{high} - S)) \leq w(C) \cdot O(w(L) \log n)$$

On the other hand, Q' can be expressed as

$$Q' = \sum_{v \in V_{high} - S} w(v) \cdot w(N^{in}(v) \cap C) \geq w(V_{high} - S) \cdot 0.9w(C)$$

The last inequality follows from the definition of V_{high} .

By combining the two inequalities, we get $w(V_{high} - S) = O(w(L) \log n)$. □

Note that V_{low} can be constructed in $O(m)$ time by iterating all edges.

Step 3: constructing a crossing family. In this step, we will construct a crossing family, which is a set of vertex pairs such that one of them will cross L, R . Formally, we will prove the following lemma.

Lemma E.17. *Given V_{low} which is constructed in step 2 satisfying Lemma E.16, there is a deterministic algorithm that constructs a family of pairs $P \subseteq C \times V_{low}$ such that*

1. *If Assumption E.15 is true, then there exists $(u, v) \in P$ such that $u \in L, v \in R$,*
2. *for every $u \in C$, define $\deg_P(u) = |\{v \in V_{low} \mid (u, v) \in P\}|$, we have $\deg_P(u) = \tilde{O}\left(1 + n \cdot \frac{w(u)}{w(R)}\right) \cdot \log^3 W$.*

The running time is $O(n|C|)$.

Proof. (Algorithm): Split the weights in V into $q = \lceil \log W \rceil$ buckets, denoted the vertices in the i -th bucket as V_i , i.e., nodes in V_i have weights between 2^{i-1} to 2^i . Define $C_i = V_i \cap C, D_i = V_{low} \cap V_i$. For every $i, j \in [q]$, apply Theorem E.5 on vertices sets C_i, D_j with $l = w(L)/(2^i \log W), r = |D_j| - nw(L) \log W \log^2 n/w(R)$. Let the returned edge set be $P_{i,j}$. Add all $(u, v) \in P_{i,j}$ to P .

(Correctness): Let $L_i = V_i \cap L$. There must exists i such that $w(L_i) \geq w(L)/\log W$. Remember that $L \subseteq C$, which means $|C_i \cap L| = |L_i| \geq w(L)/2^i \log W$. Remember that $w(V_{low} \triangle R) = O(w(L) \log n)$ by Lemma E.16 and $w(R) > \lambda w(L) \log n$ for sufficiently large constant λ by Assumption E.15. Thus, we have $w(V_{low} \cap R) \geq w(R)/2$. There must exists j such that $w(D_j \cap R) \geq w(R)/(2 \log W)$. Notice that $|D_j \cap R| \geq w(D_j \cap R)/2^j \geq w(R)/(2^{j+1} \log W)$, implying $2^j = \Omega(w(R)/|D_j \cap R| \log W) = \Omega(w(R)/n \log W)$. Thus, we have $|D_j - R| \leq w(D_j - R)/2^{j-1} = O(nw(L) \log W \log n/w(R))$. From Theorem E.5, $P_{i,j}$ must crosses L, R .

Now, we show the bound for $\deg_P(u)$. According to Theorem E.5, the left degree for the set $P_{i,j}$ is bounded by

$$\tilde{O}\left(1 + \frac{nw(L) \log W \log^2 n/w(R)}{w(L)/2^i \log W}\right) = \tilde{O}\left(1 + n \cdot \frac{2^i}{w(R)} \cdot \log^2 W\right)$$

Notice that 2^i is roughly the weight of every node in C_i . Another $\log W$ factor comes from the fact that each node is included in $P_{i,j}$ for $\log W$ times.

(Running time): According to Theorem E.5, the running time for constructing $P_{i,j}$ is bounded by at most $|C_i| \cdot |D_j|$, thus, the total running time is at most $\tilde{O}(n|C|)$. $\square$

Step 4: running max-flows on sparsified graphs. The idea is to find (s, t) -separator using Lemma E.11 for every pair $(s, t) \in P$ where P is constructed by Lemma E.17, and the minimum among them is the minimum cut we want. However, remember that in Lemma E.17, we only guarantee $\deg_P(u) = \tilde{O}(1 + n \cdot w(u)/w(R))$, which means when $w(R)$ is small, it can be as large as n . Thus, P can be as large as n^2 . Calling n^2 max flows may take $\Omega(n^2 m)$ time, which is too slow.

To handle the issue specified above, we need to sparsify each max flow instance by deleting “useless” edges (i) the edges between nodes in $N(s)$, (ii) the edges between nodes outside C . We also need to add an edge from every vertex in $N_G^{out}(C)$ to t in order to make sure the cut we found in $G_{s,t,C}$ has the left-hand side in C . Formally, we define the sparsified graph $G_{s,t,C}$ as follows:

$$E_t = \bigcup_{u \in N_G^{out}(C)} \{(u, t)\},$$

$$G_{s,t,C} = (C \cup N_G^{out}(C) \cup \{t\}, (E_G(C) - E_G(N_G^{out}(s), N_G^{out}(s)) \cup E_t)).$$

The following two lemmas show that $G_{s,t,C}$ preserves the minimum separator in G .

Lemma E.18. *Any (s, t) -separator in $G_{s,t,C}$ is a (s, t) -separator in G .*

Proof. Suppose (L', S', R') is a (s, t) -vertex cut in $G_{s,t,C}$. Since any vertex in $N_G^{out}(C)$ has an edge to t , we have $L' \subseteq C$. Now we prove that $N_G^{out}(L') \subseteq N_{G_{s,t,C}}^{out}(L')$, certifying that S' is a (s, t) -separator in G .

If $v \in N_G^{out}(L') \cap N_G^{out}(s)$, then $v \in N_{G_{s,t,C}}^{out}(L')$ since every edge from s to neighbors of s are preserved in $G_{s,t,C}$; if $v \in N_G^{out}(L') - N_G^{out}(s)$, there exists $u \in L'$ such that $(u, v) \in E$, since $v \notin N_G^{out}(s)$, (u, v) is preserved in $G_{s,t,C}$. So, $v \in N_{G_{s,t,C}}^{out}(L')$. $\square$

The above lemma implies that the minimum (s, t) -separator in $G_{s,t,C}$ has weight at least the weight of the minimum (s, t) -separator in G .

Lemma E.19. *Let (L, S, R) be the minimum vertex cut satisfying [Assumption E.15](#). If $s \in L, t \in R$, then the minimum (s, t) -separator in $G_{s,t,C}$ is a minimum (s, t) -separator in G .*

Proof. We have $N_{G_{s,t,C}}^{out}(L) \subseteq N_G^{out}(L)$ because the only edges that exist in $G_{s,t,C}$ but do not exist in G are the edges from $N_G^{out}(C)$ to t , but we know $L \subseteq C$ by [Assumption E.15](#). Since $N_G^{out}(L) \subseteq S$, we have $N_{G_{s,t,C}}^{out}(L) \subseteq S$. This implies that S is a (s, t) -separator in $G_{s,t,C}$.

Thus, the minimum (s, t) -separator in $G_{s,t,C}$ has weight at most the weight of the minimum (s, t) -separator in G . By [Lemma E.18](#), we know the two weights must be equal. $\square$

Next, we will prove that the cumulative number of edges in all max flow calls is roughly mn . To do this, instead of giving an upper bound on the number of edges in $G_{s,t,C}$ for every s, t, C (which actually can be very large) as done in [\[Li+21\]](#), we count how many times each edge is in a max flow call. This is the crucial difference between our argument and the one in [\[Li+21\]](#).

Lemma E.20. *The total number of edges in all $G_{s,t,C}$ among $(s, t) \in P, C \in \mathcal{C}$ is at most $\tilde{O}(mn) \cdot \log^2 W$, where P is constructed in [Lemma E.17](#) and $\mathcal{C}$ is constructed in [Lemma E.14](#). That is,*

$$\sum_{(s,t) \in P, C \in \mathcal{C}} |E(G_{s,t,C})| = \tilde{O}(mn) \cdot \log^3 W.$$

Moreover, with $O(mn)$ time of preprocessing, each $G_{s,t,C}$ can be constructed in time proportional to the number of edges in $G_{s,t,C}$.

Proof. Given an arbitrary edge $(u, v) \in E$, we prove that there are at most $\tilde{O}(n)$ different $G_{s,t,C}$'s that contain it. Notice that besides edges in E , there are edges in E_t that are also in $G_{s,t,C}$. However, the number of these edges is $|N_G^{out}(C)|$, which is charged to edges in E intersecting $G_{s,t,C}$. Thus, proving there are at most $\tilde{O}(n)$ different $G_{s,t,C}$ that contains an arbitrary edge $(u, v) \in E$ suffices.

For (u, v) to be included in $G_{s,t,C}$, one of u, v must be in C . Since u, v can be included in at most $O(\log n)$ different C according to [Lemma E.14](#), we just need to prove that for a fixed C with $u \in C$ (or $v \in C$), (u, v) is included in $\tilde{O}(n)$ different $G_{s,t,C}$. According to the construction of $G_{s,t,C}$, either $u \notin N_G^{out}(s)$ or $v \notin N_G^{out}(s)$. According to [Lemma E.17](#), the total number of pairs $(s, t) \in P$ satisfying $u \notin N_G^{out}(s)$ (which is equivalent to $s \notin N_G^{in}(u)$) can be calculated as

$$\begin{aligned} & \sum_{s \notin N_G^{in}(u)} \tilde{O} \left(1 + n \cdot \frac{w(s)}{w(R)} \right) \cdot \log^3 W \\ &= \tilde{O} \left(n + n \cdot \frac{\sum_{s \notin N_G^{in}(u)} w(s)}{w(R)} \right) \cdot \log^3 W \\ &= \tilde{O} \left(n + n \cdot \frac{2w(R)}{w(R)} \right) \cdot \log^3 W \\ &= \tilde{O}(n) \cdot \log^3 W \end{aligned}$$

The last equality is because $w(N_G^{in}(u)) \geq w(S)$ (because $N_G^{in}(u)$ is a vertex separator), which implies $w(V - N_G^{in}(u)) \leq w(L) + w(R) \leq 2w(R)$. The same argument goes for the case when $v \notin N_G^{out}(s)$.

Now we show fast construction of $G_{s,t,C}$. For the $O(mn)$ time pre-processing, we compute exactly $N_G^{out}(u) - N_G^{out}(v)$ and $N_G^{in}(u) - N_G^{in}(v)$ for any $u, v \in V$. Now for any s, t, C , in order to construct $G_{s,t,C}$, we need to find all edges for $G_{s,t,C}$. Notice that all edges for $G_{s,t,C}$ are either (1) edges adjacent to s , which can be constructed directly from $N_G^{out}(s)$, (2) edges adjacent to vertices in C minus the edges between

$N_G^{out}(s)$, this can be found by including all edges adjacent to vertices in $C - N_G^{out}(s)$, and then for any $u \in C \cap N_G^{out}(s)$, including all edges in $N_G^{in}(u) - N_G^{out}(s)$ and $N_G^{out}(u) - N_G^{out}(s)$, the latter is directly found from the pre-computed sets (it is too slow to compute them again as $N_G^{in}(u), N_G^{out}(u)$ could be much larger than $N_G^{in}(u) - N_G^{out}(s)$ and $N_G^{out}(u) - N_G^{out}(s)$), (3) edges adjacent to t , this is essentially determined by the neighbors of t and $N_G^{out}(C)$, where the latter is found in the previous step. $\square$

In summary, the algorithm is as follows. Firstly, guess $w(L)$ and $w(R)$ by powers of 2 (leading to $\tilde{O}(\log^2 W)$ many possibilities). For each guess, run the following algorithm to find a cut and the minimum cut among all cuts is the final output.

- (Step 1) construct common-neighborhood clustering using [Lemma E.14](#), which gives $\mathcal{C}$ in $\tilde{O}(mn)$ time,
- for every $C \in \mathcal{C}$ (notice that $|\mathcal{C}| = \tilde{O}(n)$), do the following steps to find a vertex cut, the final output will be the minimum among them,
 - (Step 2) find V_{low} with respect to C in $O(m)$ time,
 - (Step 3) construct a crossing family P in $\tilde{O}(n|C|)$ time according to [Lemma E.17](#). Note that the total time is $\sum_{C \in \mathcal{C}} \tilde{O}(n|C|) = \tilde{O}(n^2)$.
 - (Step 4) for every $(s, t) \in P$, construct $G_{s,t,C}$ and find the minimum (s, t) -separator in $G_{s,t,C}$, which must be a separator in G according to [Lemma E.18](#), and the total running time is $\hat{O}(mn) \log^3 W$ according to [Lemma E.20](#).

Now we argue the correctness. Let (L, S, R) be the minimum vertex cut in G satisfying $w(R) > \lambda w(L) \log n$. According to [Lemma E.14](#), there exists $C \in \mathcal{C}$ such that $L \subseteq C$. Thus, [Assumption E.15](#) is true for at least one $C \in \mathcal{C}$, for which [Lemma E.17](#) holds. So, there exists $(s, t) \in P$ with $s \in L, t \in R$. According to [Lemma E.19](#), it will return a valid vertex cut of G with minimum weight $w(S)$. Moreover, any cut found by the algorithm must be a valid vertex cut according to [Lemma E.18](#). Therefore, the output must be a valid vertex cut with minimum weight $w(S)$.

E.3.3 Symmetric Case: Proof of [Lemma E.13](#)

In the case when $w(L) \leq w(R) \leq \lambda w(L) \log n$, we will first use [Lemma E.21](#) to construct a family of pairs such that one of them crosses L and R . As in the previous section, we assume ℓ is an approximation of $w(L)$.

Lemma E.21. *When a minimum vertex cut (L, S, R) satisfies $w(L) \leq w(R) \leq \lambda w(L) \log n$, there is a deterministic algorithm with running time $\tilde{O}(n^2 \log^2 W)$ that constructs a family of pairs $P \subseteq V \times V$ such that*

1. *there exists $(u, v) \in P$ such that $u \in L, v \in R$ or $v \in L, u \in R$,*
2. *for every $u \in V$, define $\deg_P(u) = |\{v \in V \mid (u, v) \in P\}|$, we have $\deg_P(u) = \tilde{O}\left(1 + n \cdot \frac{w(u)}{w(R)}\right) \cdot \log^2 W$.*

Proof. (Algorithm): Split the weights into $q = \lceil \log W \rceil$ buckets, denote the vertices in the i -th bucket as V_i , i.e., nodes in V_i have weights between 2^{i-1} to 2^i . For every $i, j \in [q]$, apply [Corollary E.6](#) on the vertex set $V_i \cup V_j$ (treat $|V_i \cup V_j|$ as n in [Corollary E.6](#) and relates every integer in $[n]$ to a vertex in $V_i \cup V_j$) with $\alpha = \min(n, n \cdot (2^{\max(i,j)}/\ell) \cdot \log W \cdot \lambda^2 \log n)$ (recall that ℓ is a 2-approximation of $w(L)$), let the returned $(|V_i \cup V_j|, \alpha)$ crossing family be $P_{i,j}$. If $i \geq j$, add all $(u, v) \in P_{i,j}$ with $u \in V_i$ to P ; otherwise, add all $(u, v) \in P_{i,j}$ with $u \in V_j$ to P . The total running time is $\tilde{O}(n^2 \log^2 W)$ since constructing each crossing family cost $\tilde{O}(n^2)$ as $\alpha \leq n$.

(Correctness): Let $L_i = V_i \cap L$ and $R_i = V_i \cap R$. There must exist i such that $w(L_i) \geq w(L)/\log W$ and j such that $w(R_j) \geq w(R)/\log W$. We will prove that $P_{i,j}$ crosses L, R . Notice that $|(V_i \cup V_j) \cap L| \geq |L_i| \geq w(L_i)/2^i$, and $|(V_i \cup V_j) \cap R| \geq |R_j| \geq w(R_j)/2^j$. Thus, we have

$$|(V_i \cup V_j) \cap S|/|(V_i \cup V_j) \cap L| \leq n \cdot (2^i/w(L)) \cdot \log W \leq \alpha$$

and

$$|(V_i \cup V_j) \cap S|/|(V_i \cup V_j) \cap R| \leq n \cdot (2^j/w(R)) \cdot \log W \leq \alpha$$

(the inequality holds when $\alpha < n$, but when $\alpha \geq n$, the crossing family is complete which must cross L and R). This implies that there is $(u, v) \in P_{i,j}$ with $u \in L, v \in R$ and another $(u', v') \in P_{i,j}$ with $u' \in R, v' \in L$. In any case, the correctness for (1) is proved. Moreover, according to [Corollary E.6](#), we have the out-degree of u in $P_{i,j}$ is at most $\tilde{O}\left(1 + n \cdot \frac{2^{\max(i,j)} \log W}{\ell}\right)$. Remember that we only add $(u, v) \in P_{i,j}$ to P if $u \in V_i$ with $i \geq j$, which implies $w(u) = \Omega(2^{\max(i,j)})$. Also remember that $\ell = \tilde{\Theta}(w(R))$. Notice that each u can be included in at most $\log W$ many $P_{i,j}$, thus, we get $\deg_P(u) \leq \tilde{O}\left(1 + n \cdot \frac{w(u)}{w(R)}\right) \cdot \log^2 W$. $\square$

Similar to [Lemma E.18](#), the total number of edges in the sparsified max flow instances is $\tilde{O}(mn \log^2 W)$.

Lemma E.22. *Let P be the crossing family from [Lemma E.21](#). For $s, t \in V$, define $G_{s,t}$ as the graph after deleting edges between $N_G^{out}(s)$, and the edges between $N_G^{in}(t)$. We have*

1. *the minimum (s, t) -vertex cut does not change in $G_{s,t}$ compared to G ,*
2. *the total number of edges of $G_{s,t}$ among all $(s, t) \in P$ and $(t, s) \in P$ is $\tilde{O}(mn) \cdot \log^2 W$.*

Moreover, $G_{s,t}$ can be constructed in time proportional to the number of edges in $G_{s,t}$ with $O(mn)$ time preprocessing.

Proof. The first point is easy to see: vertex cut in G is clearly vertex cut in $G_{s,t}$, and suppose (L, S, R) is a (s, t) -vertex cut in $G_{s,t}$, we have $N_G^{out}(s) \subseteq L \cup S$ and $N_G^{in}(t) \subseteq S \cup R$, which means edges between $N_G^{out}(s)$ or between $N_G^{in}(t)$ cannot cross L, R , implying that (L, S, R) is a (s, t) -vertex cut in G .

To see the second point, we will prove that for any edge $(u, v) \in E$, there are at most $\tilde{O}(n \log^2 W)$ possible $(s, t) \in P$ or $(t, s) \in P$ such that $G_{s,t}$ includes (u, v) . There are two possibilities for (u, v) to be included in $G_{s,t}$, the first one is that (u, v) is an edge from s to $N_G^{out}(s)$ or from $N_G^{in}(t)$ to t . This case can happen at most $2n$ times, since either u is fixed to s or v is fixed to t , and P is not a multiset. The second case is that one of u, v is in $V - N_G^{out}[s]$. Let us assume $u \in V - N_G^{out}[s]$, which implies $s \in V - N_G^{in}(u)$. The total number of such s, t pair in P is at most

$$2 \sum_{v \in V - N_G^{in}(u)} \deg_P(v) = \tilde{O}\left(n + n \cdot \frac{\sum_{v \in V - N_G^{in}(u)} w(v)}{w(R)}\right) \cdot \log^2 W = \tilde{O}(n) \cdot \log^2 W$$

The last equality is because $N_G^{in}(u)$ is a vertex cut, which implies $w(N_G^{in}(u)) \geq w(S)$ and $w(V - N_G^{in}(u)) \leq w(L) + w(R) \leq 2w(R)$.

To see fast construction of $G_{s,t}$, notice that we can use $O(mn)$ time preprocessing to get the symmetric difference between neighborhood set of every two vertices, then to construct $G_{s,t}$ we simply need to find neighbors of vertices $V_{s,t}$ minus the neighbors of s, t . $\square$

In summary, the algorithm first generate $\tilde{O}(\log W)$ guesses of $w(L)$, for each of them, use [Lemma E.21](#) to get a crossing family P with a pair $(u, v) \in P$ satisfying either $u \in L, v \in R$ or $v \in L, u \in R$, so we need to construct $G_{u,v}$ for every $(u, v) \in P$ or (v, u) and find the minimum (s, t) -separator, the total running time and correctness is guaranteed by [Lemma E.22](#).

E.4 An $\widehat{O}(m\kappa)$ -Time Algorithm for Unweighted Undirected Graphs

The goal of this section is prove [Theorem E.2](#). The algorithm consists of three ingredients: (1) the unbalanced algorithm, (2) the terminal reduction algorithm, and (3) the terminal-balanced algorithm.

The guarantee of the algorithm for the unbalanced case is summarized as follows. One can view the minimum degree δ as an approximate size of a vertex mincut.

Lemma E.23. *There is a deterministic algorithm that takes inputs an n -nodes m -edges undirected graph $G = (V, E)$, and outputs a vertex cut of G . If there exists a minimum vertex cut (L, S, R) of G satisfying*

$|L| \leq \lambda \cdot \delta$ for a sufficiently large constant λ and $|L| \leq |R|$, then the output is a minimum vertex cut of G . The algorithm runs in $\widehat{O}(m\delta)$ time.

For the balanced case, we first define balanced terminal sets.

Definition E.24. For an undirected graph $G = (V, E)$, a terminal set $T \subseteq V$ is called α -strongly-balanced if for every minimum vertex cut of G denoted by S and every connected component of $G[V - S]$ denoted by L , we have $|T \cap L| \geq \alpha|S|$.

The terminal reduction algorithm is as follows.

Lemma E.25. There is a deterministic algorithm $\text{TERMINALREDUCTION}(G, T, k)$ (Algorithm 7) that takes as inputs an n -vertex m -edge undirected graph $G = (V, E)$, along with a $2^{\sqrt{\log n}/2}$ -strongly-balanced terminal set $T \subseteq V$ and a cut parameter $k \geq \kappa_G$ and $k = O(\kappa_G)$, outputs (S', T') where

- $|T'| \leq 0.9|T|$,
- either S' is a minimum vertex cut of G or $T' \subseteq V$ is a 0.01-strongly-balanced terminal set.

The algorithm runs in time $\widehat{O}(mk)$.

The final ingredients handle the case when the terminal set is balanced.

Lemma E.26. There is a deterministic algorithm $\text{BALANCEDTERMINALVC}(G, T, k)$ that takes as inputs an undirected graph $G = (V, E)$, a terminal set $T \subseteq V$ and a cut parameter k , outputs a vertex cut of G . If there exists a vertex cut (L, S, R) of G such that $|T \cap L|, |T \cap R| = \widehat{\Omega}(|T \cap S|)$ and $|T \cap L| = \widehat{O}(k)$, then the output cut is a minimum vertex cut of G . The algorithm runs in $\widehat{O}(mk)$ time.

Remark E.27. The proof is based on the isolating vertex cut lemma [Li+21], which requires the graph to be undirected.

Via the three algorithmic components from Lemmas E.23, E.25 and E.26, we are ready to prove Theorem E.2.

Proof of Theorem E.2.

Proof. By the sparsification algorithm of [NI92], we can assume the input undirected graph G has number of edges $m \leq \kappa_G n$ by $O(m)$ time preprocessing. Thus, the minimum degree $\delta \leq 2\kappa_G$. Notice that $\delta \geq \kappa_G$. Thus, by setting $k = \delta$ we have $k \geq \kappa_G$ and $k = O(\kappa_G)$.

Algorithm. The algorithm first use Lemma E.23 on G to get a vertex cut S^* . Then the algorithm set $T_0 = V$, and for $i = 0, 1, 2, \dots$ run

$$S_i \leftarrow \text{BALANCEDTERMINALVC}(G, T_i, k)$$

and

$$(S'_i, T_{i+1}) \leftarrow \text{TERMINALREDUCTION}(G, T_i, k)$$

until T_{i+1} becomes empty. The returned cut will be the minimum one among S^* and S_i, S'_i for all i .

Correctness. If there exists a minimum vertex cut S and a minimum size connected component L of $G[V - S]$ (that makes $|V - L - S| \geq |L|$) such that $|L| \leq \lambda|S| = O(\delta)$ for a large constant λ , then according to Lemma E.23, S^* is a minimum vertex cut. Otherwise, a terminal set $T_0 = V$ must be 0.01-strongly-balanced.

Let i be the minimum index such that T_i is not a $2^{\sqrt{\log n}/2}$ -strongly-balanced terminal set. Such i must exist because the size $|T_i|$ is strictly decreasing. Suppose that none of $S^*, S_0, S'_0, \dots, S_{i-1}, S'_{i-1}$ is a minimum vertex cut; otherwise, we are done. We will show that S_i must be a minimum vertex cut. If $i = 0$, $T_0 = V$ is 0.01-strongly-balanced as observed above. If $i > 0$, since T_{i-1} is a $2^{\sqrt{\log n}/2}$ -strongly-balanced terminal set, according to Lemma E.25, T_i must be a 0.01-strongly-balanced terminal set because we assume

S'_{i-1} is not a minimum vertex cut. In any case, T_i is a 0.01-strongly-balanced terminal set and not a $2\sqrt{\log n/2}$ -strongly-balanced terminal set.

By the definition, there exists a vertex cut S and a connected component of $G[V - S]$ denoted by L such that $|T_i \cap L| < 2\sqrt{\log n/2}|S|$. We also have $|T_i \cap L| > \Omega(|S|)$ and $|V - S - L| > \Omega(|S|)$. By [Lemma E.26](#), S_i is a minimum vertex cut of G .

Complexity. The sparsification by [\[NI92\]](#) takes $O(m)$ time. [Lemma E.23](#) takes $\widehat{O}(m\delta) = \widehat{O}(m\kappa_G)$ time. According to [Lemma E.25](#), $|T_{i+1}| \leq 0.9|T_i|$, so there can be at most $O(\log n)$ loop of algorithm. Each loop takes time $\widehat{O}(m\kappa_G)$ according to [Lemmas E.25](#) and [E.26](#). $\square$

E.4.1 Organization and Overview

Organization. The rest of this section is devoted for proving [Lemmas E.23](#), [E.25](#) and [E.26](#). [Lemma E.26](#) serves as subroutines for [Lemmas E.23](#) and [E.25](#) so we prove it first in [Section E.4.2](#). Another subroutine we are going to use is common-neighborhood clustering, which we define and show an algorithm in [Section E.4.3](#). After that, we prove [Lemma E.23](#) in [Sections E.4.4](#) and [E.4.5](#) and [Lemma E.25](#) in [Section E.4.6](#).

We give the technical overview of [Lemma E.23](#) and [Lemma E.25](#) in this section as the proofs are quite involved.

Overview of Unbalanced Case: [Lemma E.23](#)

Let (L, S, R) be a vertex mincut where $|L| \leq |R|$. In this section, we assume that $G[L]$ is connected, and $|L|, |S|$ are known (otherwise, we can approximate the sizes by a factor of 2 using binary search) and

$$|L| \leq O(|S|) \tag{E.2}$$

Fast Common-Neighborhood Clustering. The key technical tool is the fast common-neighborhood clustering ([Lemma E.14](#)). We compute collection of clusters $V_1, \dots, V_z$ such that

1. $L \subseteq V_i$ for some cluster V_i ,
2. for all cluster V_i , and for every pair $u, v \in V_i$, $|N(u) \triangle N(v)| \leq O(|L| \log n)$, and
3. every vertex belongs to $O(\log n)$ clusters.

The construction in [Lemma E.14](#) takes $\tilde{O}(mn)$ time because we explicitly compare the neighborhood set of every pair of vertices. To obtain $\tilde{O}(m\kappa)$ time, we define a different intermediate graph from the one in the proof of [Lemma E.14](#), open the box of sparse-neighborhood cover algorithm and apply *deterministic sparse recovery sketching* from [\[NS17\]](#). We show the fast construction in [Section E.4.3](#).

We assume we obtain a collection of clusters $V_1, \dots, V_z \subseteq V$ satisfying the three properties stated above. In addition, the clusters $V_1, \dots, V_z$ have important structural properties.

Lemma E.28. *If $L \subseteq V_i$ for some i , then either $|V_i| \leq O(|S| \log n)$ or $|S \cap V_i| \leq O(|L| \log n)$.*

Proof. We prove that if $|V_i| = \omega(|S| \log n)$, then $|S \cap V_i| = O(|L| \log n)$. Suppose to the contrary that $|S \cap V_i| = \omega(|L| \log n)$. We use double counting on the size of $E_G(V_i \cap S, V_i \cap R)$. we have

$$|E_G(V_i \cap S, V_i \cap R)| \leq |V_i \cap S| \cdot O(|L| \log n)$$

since $|N_G(u) \triangle N_G(v)| = O(|L| \log n)$ according to the second property of the clusters for any $u \in V_i \cap S$ (and notice that any node in $V_i \cap R$ is not a neighbor of an arbitrary vertex $v \in L$). On the other hand, consider a node $s \in L$, since $|N_G(s)| \geq |S|$ and $N_G(s) \subseteq L \cup S$, we have $|S \setminus N_G(s)| \leq |L|$. According to the second property of the clusters, for any $u \in R$, we have $|N_G(s) - N_G(u)| = O(|L| \log n)$. Thus, for any $u \in R$ we have $|(V_i \cap S) \cap N_G(u)| \geq |(V_i \cap S) \cap N_G(s)| - O(|L| \log n) \geq |V_i \cap S| - |L| - O(|L| \log n) \geq |V_i \cap S|/2$. The last inequality is due to $|S \cap V_i| = \omega(|L| \log n)$. Now we get

$$|E_G(V_i \cap S, V_i \cap R)| \geq |V_i \cap R| \cdot |V_i \cap S|/2$$

By combining the inequalities above, we get $|V_i \cap R| = O(|L| \log n)$. Thus, we have

$$|S| \geq |V_i \cap S| = |V_i| - |V_i \cap L| - |V_i \cap R| = \omega(|S| \log n).$$

This is a contradiction. $\square$

Based on [Lemma E.28](#) and the property of clusters, we have two cases:

Case 1: There is a cluster V_i such that $L \subseteq V_i$ and $|V_i| = O(|S| \log n)$. In this situation, we will use crossing families based on dispersers [\[TUZ01\]](#) with kernelization techniques [\[Li+21\]](#).

We construct a crossing family $\mathcal{P}$ by applying [Corollary E.6](#) using $n = |V(G)|$ and $\alpha = \Theta(|S|/|L|)$ in $\tilde{O}(n\alpha) = \tilde{O}(n \cdot \frac{\kappa}{|L|})$ time. By [Corollary E.6](#), there exists a pair $(s, t) \in \mathcal{P}$ such that $s \in L, t \in R$, and thus

$$\kappa = \min_{(s,t) \in \mathcal{P}} \kappa_G(s, t) \text{ and} \quad (\text{E.3})$$

$$|\mathcal{P}| = \tilde{O}(n\alpha) = \tilde{O}(n \cdot \frac{\kappa}{|L|}). \quad (\text{E.4})$$

By max-flow mincut theorem, it suffices to compute (s, t) -vertex-capacity max-flow for each $(s, t) \in \mathcal{P}$. Before running (s, t) -vertex-capacity max-flow, we compute a *reduced* instance (called *kernel*) of G .

Our task is to compute the *kernel* of G with the following guarantee:

Given s, t in $G = (V, E)$, return a graph H with $\tilde{O}(\kappa \cdot |L|)$ edges such that

1. if $s \in L, t \in R$, then $\kappa_G(s, t) = \kappa_H(s, t)$,
2. for any s, t , we have $\kappa_G(s, t) \leq \kappa_H(s, t)$.

Given such kernels, we can solve Case 1 by computing (s, t) -max-flows on the kernel of G for all $(s, t) \in \mathcal{P}$. The running time is

$$|\mathcal{P}| \cdot \widehat{O}(\kappa \cdot |L|) \stackrel{(\text{E.4})}{=} \tilde{O}(n \cdot \frac{\kappa}{|L|}) \cdot \widehat{O}(\kappa \cdot |L|) = \widehat{O}(n\kappa^2) = \widehat{O}(m\kappa). \quad (\text{E.5})$$

The last equality follows because the minimum degree $\delta \geq \kappa$. In this paper, we show a deterministic construction by exploiting the property of the cluster V_i . In [Section E.4.4](#), we prove the following: Given $s \in L, t \in R$, and a cluster V_i such that $L \subseteq V_i$, we can deterministically compute a kernel of size $\tilde{O}(|V_i||L|)$. By the assumption of Case 1, the kernel is of size $\tilde{O}(|V_i||L|) = \tilde{O}(|S||L| \log n) = \tilde{O}(\kappa \cdot |L|)$ as desired. We remark that the kernel of size $\tilde{O}(\kappa|L|)$ exists by the randomized construction of [\[Li+21\]](#).

Case 2: There is a cluster V_i such that $L \subseteq V_i, |S \cap V_i| \leq O(|L| \log n)$ and $|V_i| = \omega(|S| \log n)$ (which implies $|R \cap V_i| \geq |S \cap V_i|$ because of [Equation \(E.2\)](#)). This case is similar to the proof of [Lemma E.26](#), which is based on selectors (based on linear lossless condensers [\[GUV09; Che11\]](#)) and the isolating cut lemma [\[Li+21\]](#).

Let $T = V_i$ be a terminal set. Since $|L \cap T|, |R \cap T| \geq |S \cap T|$ and $|L \cap T| = |L| \stackrel{(\text{E.2})}{\leq} O(|S|)$, the vertex cut (L, S, R) is terminal-balanced. Therefore, applying [Lemma E.26](#) with $T = V_i$ on every cluster V_i would obtain a vertex mincut in one of the clusters. However, one might notice that the running time is $z \cdot \widehat{O}(m\kappa)$, which is too large for us. To address this, we will use [Lemma E.26](#) not on the whole graph, but only on the subgraph containing all edges incident to V_i , which is enough for us since $L \subseteq V_i$. The details are stated in [Lemma E.36](#).

Overview of Terminal Reduction: [Lemma E.25](#)

Fix a vertex mincut (L, S, R) in G . We say that a terminal set T is α -balanced if $|T \cap L|, |T \cap R| \geq \alpha|S|$. We use α -balanced instead of α -strong-balanced for simplicity. The same argument goes through for α -strong-balanced terminal set. In this overview, we outline an $\widehat{O}(m\kappa)$ -time algorithm for the following version.

Terminal Reduction: Given a $2^{\sqrt{\log n}}$ -balanced terminal set $T \subseteq V$, compute $T' \subseteq V$ and a separator S^* where $|T'| \leq 0.9|T|$ such that

- T' is 0.1-balanced, or
- S^* is a minimum separator in G .

We now outline the high-level ideas for the terminal reduction procedure. We define $\gamma = 2^{\sqrt{\log n}}$.

1. Given T , we compute $(X, \mathcal{U})$ a *terminal vertex expander decomposition* on graph G (see [Lemma E.51](#) for a formal statement). That is, we obtain a vertex set X and a collection of vertex disjoint sets $\mathcal{U}$ such that

- X and all vertex sets in $\mathcal{U}$ form a vertex partition. That is, they are disjoint and the union of them form the vertex set V ,
- There are no edges between two different sets $U_1, U_2 \in \mathcal{U}$,
- For each $U \in \mathcal{U}$, the induced subgraph $G[U]$ is an expander with respect to T , and
- $|X| \leq 0.1|T|$.

Simplification. In this technical overview, we assume that for each vertex set $U \in \mathcal{U}$, $U \subseteq L$, $U \subseteq S$, or $U \subseteq R$. This setting conveys most of the main ideas. This assumption is close to be true because each vertex set U is an expander with respect to T , the mincut (L, S, R) cannot cross U in a balanced way, i.e., either most terminals in U are contained in L , S , or R , which is similar to our assumption.

2. **Main goal.** We will construct three terminal sets $T_{\text{big}}, T_{\text{small}}, T_{\bar{X}}$ so that the terminal set $T' := X \cup T_{\text{big}} \cup T_{\text{small}} \cup T_{\bar{X}}$ is of size at most $0.9|T|$ and compute a vertex cut S^* of G so that either

- (i) The new terminal set T' is 0.1-balanced, i.e., it satisfies $|L \cap T'|, |R \cap T'| \geq 0.1|S|$, or
- (ii) $|S^*| = \kappa$.

For simplicity, in the following analysis, our main goal will be to show that we will only

$$|L \cap T'| \geq 0.1|S|,$$

the other side $|R \cap T'| \geq 0.1|S|$ can be achieved by the symmetric argument.

For each $U \in \mathcal{U}$, we denote $T_U := U \cap T$ as the terminal set inside U . Let $\mathcal{U}_{\text{small}} := \{U \in \mathcal{U} : |T_U| = 1\}$, and $\mathcal{U}_{\text{big}} := \{U \in \mathcal{U} : |T_U| > 1\}$. Below, we will describe how to construct T_{small} and T_{big} based on $\mathcal{U}_{\text{small}}$ and $\mathcal{U}_{\text{big}}$.

3. **We Can Focus on $\mathcal{U}_{\text{small}}$ by T_{big} .** We construct T_{big} as follows: For each $U \in \mathcal{U}_{\text{big}}$, add to T_{big} arbitrary $\lceil |T_U|/2 \rceil$ terminals from T_U . It is easy to see that $|T_{\text{big}}| \leq (2/3)|T|$. The following claim shows the reason why we define T_{big} in this way.

Claim E.29. *If most terminals in L are contained in $\mathcal{U}_{\text{big}}$, i.e., $\sum_{U \in \mathcal{U}_{\text{big}}} |T_U \cap L| \geq |T \cap L|/2$, then $|L \cap T_{\text{big}}| \geq |T \cap L|/4 \geq 0.1|S|$.*

This is because at least half of the terminals in $T_U \cap L$ are included to T_{big} according to our simplification that $T_U \subseteq L$ if $T_U \cap L \neq \emptyset$. The inequality $|T \cap L|/4 \geq 0.1|S|$ is because T is γ -balanced.

Let $L_C := \{U \in \mathcal{U}_{\text{small}} \mid U \subseteq L\}$. From now, we assume most terminals in $T \cap L$ are contained in $\mathcal{U}_{\text{small}}$, i.e., $\sum_{U \in \mathcal{U}_{\text{small}}} |T_U \cap L| \geq 1/2|T \cap L| \geq \frac{\gamma}{2} \cdot |S|$. Since $\sum_{U \in \mathcal{U}_{\text{small}}} |T_U \cap L| = |L_C|$, we have

$$|L_C| \geq \frac{\gamma}{2} \cdot |S|. \quad (\text{E.6})$$

Our next goal becomes this: pick a small number of clusters in $\mathcal{U}_{\text{small}}$ (at most $0.01|T|$) so that they hit L_C a lot (at least $|L_C|/\gamma$) and include all the terminals in these clusters into a terminal set T_{small} .

4. **An Easy Case for X .** Before we describe how to pick a small number of clusters in $\mathcal{U}_{\text{small}}$, it is instructive to first solve an easy case of X where $|X \cap S| = \tilde{O}(|X \cap L|)$. Here, we assume that $|X \cap L| \leq 0.1|S|$ (otherwise $|T' \cap L| > 0.1|S|$ and we are done).

In this case, we can almost apply [Lemma E.26](#) on X as a terminal set except that it requires $|X \cap S| \leq |X \cap R|$, but X may not intersect with R at all. To handle this situation, we define $T_{\bar{X}}$ that contains an arbitrary $0.01|T|$ nodes from $T - X$. If $|T_{\bar{X}} \cap L| > 0.1|S|$, then we are done. Assume $|T_{\bar{X}} \cap L| \leq 0.1|S|$, and thus $|T_{\bar{X}} \cap R| \geq |S|$ since $|T| \geq \gamma|S|$ and $|T \cap X| \leq 0.1|T|$. We claim that $|T_{\bar{X}} \cap S| \leq |X \cap L|$. If true, then the terminal $T'' := X \cup T_{\bar{X}}$ satisfies the conditions in [Lemma E.26](#) and we can apply [Lemma E.26](#) on T'' to get a vertex mincut S^* in $\hat{O}(m \cdot |T'' \cap L|) \leq \hat{O}(m \cdot (|X \cap L| + |T_{\bar{X}} \cap L|)) \leq \hat{O}(m\kappa)$ time.

We now prove $|T_{\bar{X}} \cap S| \leq |X \cap L|$. Fix a vertex set $U \in L_C$. By definition of the expander decomposition, $N_G(U) \subseteq X \cap (L \cup S)$. So, $|N_G(U)| \leq |X \cap S| + |X \cap L|$. On the other hand, since $N_G(U)$ is a valid vertex cut, $|N_G(U)| \geq \kappa = |S|$. So $|S| \leq |X \cap S| + |X \cap L|$, which is equivalent to $|S - X| \leq |X \cap L|$. The claim follows as $|T_{\bar{X}} \cap S| \leq |S - X|$.

5. **Pruning X or Concentrated T_{small} .** To handle the general case of X , we define $T_{\bar{X}}$ using X in the same way, and then we define $X' \subseteq X$ by removing vertices in X in such a way that we never remove any vertex in $X \cap L$, and at the end, all but $\tilde{O}(|X \cap L|)$ vertices from $X \cap S$ have been removed. The process of carving out nodes of X into X' is called *the pruning step*. To implement the pruning step, we will construct $T_{\text{small}} \subseteq T$ of size at most $0.1|T|$ and $X' \subseteq X$ such that either

$$|T_{\text{small}} \cap L| \geq 0.1|S| \text{ or } X' \cap L = X \cap L, |X' \cap S| \leq \tilde{O}(|X \cap L|). \quad (\text{E.7})$$

That is, either we find a small number of clusters in $\mathcal{U}_{\text{small}}$ so that they hit L_C a lot (and include all terminals in these clusters in T_{small}) or we manage to prune X into X' satisfying [Equation \(E.7\)](#) (we say that T_{small} is *concentrated* if $|T_{\text{small}} \cap L| \geq 0.1|S|$). Therefore, we can add T_{small} to the final terminal set and run the algorithm for the easy case on X' (using $T_{\bar{X}}$ defined by X'), and we are done.

The main challenge for the pruning step is to characterize the conditions when a vertex $v \in X$ is in $X \cap L$ and in $X \cap S$ *without the knowledge* of set L and S .

The rest of the section is devoted to the pruning step.

6. **Common-neighborhood Clustering on the Expander Decomposition.** The solution, which is the crux of our algorithm, is that we run the sparse neighborhood cover algorithm on the common neighborhood graph obtained from the bipartite graph between X and $\mathcal{U}$ obtained by contracting each component in $\mathcal{U}$. More precisely, we define a bipartite graph $H := (X \cup \mathcal{U}_{\text{small}}, E')$ where there is an edge $(x, U) \in E'$ if and only if there is an edge between $x \in X$ to $U \in \mathcal{U}_{\text{small}}$ in G . To see the intuition why it helps, we have the following claim. Let $a = |X \cap L| < 0.1|S|$ (if $|X \cap L| \geq 0.1|S|$, then we are done). We assume the algorithm knows a (we can estimate a up to a constant factor by guessing the size $O(\log n)$ times by the powers of 2).

Claim E.30. *For any $U_1, U_2 \in L_C$, we have $|N_H(U_1) \Delta N_H(U_2)| \leq 2a$.*

Proof. Notice that $N_H(U_1) \subseteq X$ is a vertex cut in the original graph G , since different clusters in $\mathcal{U}$ do not have edges connecting them. Moreover, $N_H(U_1) \subseteq (L \cup S) \cap X$. The same holds for U_2 . Thus, we have $\kappa \leq |N_H(U_1)|, |N_H(U_2)| \leq \kappa + a$ and $|N_H(U_1) \cup N_H(U_2)| \leq \kappa + a$. This gives us $|N_H(U_1) \Delta N_H(U_2)| \leq 2a$. $\square$

By running common-neighborhood clustering on $H[\mathcal{U}_{\text{small}}]$ (i.e., running sparse neighborhood cover on the common-neighborhood graph where there is an edge between U_1, U_2 if $|N_H(U_1) \Delta N_H(U_2)| \leq 2a$), we can get $V_1, \dots, V_z$ with the following properties.

Claim E.31. *We can obtain clusters $V_1, \dots, V_z \subseteq \mathcal{U}_{\text{small}}$ satisfying*

- (a) $L_C \subseteq V_i$ for some cluster V_i ,
- (b) for every cluster V_i and every two vertices $U_1, U_2 \in V_i$, $|N_H(U_1) \Delta N_H(U_2)| \leq O(a \log n)$.

Remark E.32. We need an $\hat{O}(m\kappa)$ algorithm for computing $V_1, \dots, V_z$. However, we cannot apply the same intermediate graph because $H[L_C]$ is an independent set. To fix this, the idea is to build

new virtual edges between $\mathcal{U}_{\text{small}}$ carefully so that, the total number of edges is bounded by $O(m)$, and after adding those edges, L_C is connected. We refer to [Lemma E.49](#) for details.

7. **Proof Strategy for the Pruning Step.** We exploit the structure of the common-neighborhood clustering on $H[\mathcal{U}_{\text{small}}]$ to help us identify the nodes in $X \cap S$ while avoiding the nodes in $X \cap L$ as follows. We denote V^* as a cluster that contains L_C . Note that V^* exists by [Claim E.31\(a\)](#).

At high level, we argue that all but $\tilde{O}(a)$ vertices in $X \cap S$ have at least τ neighbors in V^* in graph H (where τ is a parameter chosen to be closer to $|V^*|$) using the properties of the common-neighborhood clustering on $H[\mathcal{U}_{\text{small}}]$. At the same time, we ensure that every node in $X \cap L$ has fewer than τ neighbors in V^* via T_{small} (to be defined). Therefore, deleting all nodes in X having at least τ neighbors in V^* would complete the pruning step. Since the algorithm does not know the set V^* , we will try all clusters $V_1, \dots, V_z$ of a certain size and argue that we never delete any vertex in $X \cap L$.

We next formalize the ideas.

8. **Structural Property of V^* .** We start with a key property from the common-neighborhood clustering on $H[\mathcal{U}_{\text{small}}]$. For any vertex $x \in X$ and $U \subseteq \mathcal{U}_{\text{small}}$, we denote the degree of x in U in graph H as $\deg_H(x, U) := |N_H(x) \cap U|$.

Lemma E.33. *For all $\delta \in (0, 1)$, all but $O(a \cdot \delta^{-1} \log n)$ vertices in $X \cap S$ have at least $(1 - \delta)|V^*|$ neighbors in V^* in graph H . That is, $|Z_\delta| \leq O(a \cdot \delta^{-1} \log n)$ where $Z_\delta := \{x \in X \cap S : \deg_H(x, V^*) < (1 - \delta)|V^*|\}$.*

Proof. Fix $U \in L_C \subseteq V^*$. We have $|N_H(U) \cap (X \cap S)| \geq |S| - |X \cap L| \geq |S \cap X| - a$. By [Claim E.31\(ii\)](#), $|N_H(U) \triangle N_H(U')| \leq O(a \log n)$ for all $U' \in V^*$. Therefore, for all $U' \in V^*$, $|N_H(U') \cap (X \cap S)| \geq |S \cap X| - O(a \log n)$. In other words, for all $U \in V^*$, the number of non-edges from U to $X \cap S$ is at most $O(a \log n)$. Define $\text{NN}_H(X \cap S, V^*)$ to be the number of non-edges between $X \cap S$ and V^* in H . We have the following.

$$\delta \cdot |V^*| \cdot |Z_\delta| \leq \text{NN}_H(X \cap S, V^*) \leq |V^*| \cdot O(a \log n).$$

The lower bound follows since each vertex in Z_δ has at least $\delta \cdot |V^*|$ non-edges to V^* . Therefore, $|Z_\delta| = O(a \cdot \delta^{-1} \log n)$. $\square$

Suppose we are given the set V^* . How to design an algorithm that removes all but $\tilde{O}(a)$ vertices in $X \cap S$ and none in $X \cap L$? We almost have the pruning step except that we need the following property for some $\delta \in (0, 1)$ to be specified:

$$(1 + \frac{2}{\gamma})|L_C| < (1 - \delta)|V^*|. \quad (\text{E.8})$$

Given [Equation \(E.8\)](#), we can characterize $X \cap L$ and almost all nodes in $X \cap S$ using $\tau := (1 - \delta)|V^*|$. Indeed, for all $y \in X \cap L$ and for all but $\tilde{O}(a \cdot \delta^{-1})$ vertices $x \in X \cap S$, we have

$$\deg_H(y, V^*) \leq (1 + \frac{2}{\gamma})|L_C| \stackrel{(\text{E.8})}{<} \tau = (1 - \delta)|V^*| \leq \deg_H(x, V^*).$$

The first inequality follows since every vertex in $X \cap L$ has degree at most $|L_C| + |S| \stackrel{(\text{E.6})}{\leq} (1 + \frac{2}{\gamma})|L_C|$ in H , and thus it has at most $(1 + \frac{2}{\gamma})|L_C|$ neighbors in V^* (w.r.t. H). The last inequality follows since, by [Lemma E.33](#), for all $\delta \in (0, 1)$, all but $\tilde{O}(a \cdot \delta^{-1})$ vertices in $X \cap S$ have a lot of neighbors, i.e., at least $(1 - \delta)|V^*|$ in V^* (w.r.t. H).

Therefore, by deleting all nodes $x \in X$ such that $\deg_H(x, V^*) \geq \tau$, we delete all but $\tilde{O}(a \cdot \delta^{-1})$ vertices in $X \cap S$ while we do not delete any node in $X \cap L$.

Next, we establish [Equation \(E.8\)](#) by using T_{small} and we will set $\delta^{-1} = O(\log^2 n)$.

9. **Small L_C by T_{small} .** We construct T_{small} as follows. For each V_i , we pick arbitrary $|V_i|/\log^2 n$ elements in V_i . Recall that an element in V_i corresponds to an expander in $\mathcal{U}_{\text{small}}$. For each of these

expanders, include its unique terminal into T_{small} .

The size of T_{small} is negligible compared to the size of T because $|T_{\text{small}}| \leq O(\log n) \cdot (|T|/\log^2 n) = o(|T|)$ where the $O(\log n)$ factor is from the fact that each element is contained in at most $O(\log n)$ clusters V_i .

If $|L_C| \geq (1 - \frac{1}{2\log^2 n})|V^*|$, then $|T_{\text{small}} \cap L| \geq |L_C| \cdot \frac{1}{2\log^2 n}$. Since $|L_C| \stackrel{(E.6)}{\geq} \frac{\gamma}{2} \cdot |S|$, we have $|L \cap T'| \geq 0.1|S|$ as desired. From now on, we assume

$$|L_C| < (1 - \frac{1}{2\log^2 n})|V^*|. \quad (\text{E.9})$$

Therefore, for all $y \in X \cap L$,

$$\deg_H(y, V^*) \leq (1 + \frac{2}{\gamma})|L_C| < (1 - \frac{1}{3\log^2 n})|V^*|.$$

By setting $\delta = \frac{1}{3\log^2 n}$ in [Lemma E.33](#), we have for all $x \in (X \cap S) - Z_\delta$,

$$(1 - \frac{1}{3\log^2 n})|V^*| \leq \deg_H(x, V^*).$$

We now fix $\delta := \frac{1}{3\log^2 n}$ and let Z_δ be the set defined in [Lemma E.33](#) and define $\tau := (1 - \frac{1}{3\log^2 n})|V^*|$. We conclude with the following lemma.

Lemma E.34. *For all $y \in X \cap L$, and $x \in (X \cap S) - Z_\delta$, $\deg_H(y, V^*) < \tau \leq \deg_H(x, V^*)$.*

Therefore, deleting all nodes in X which have at least τ neighbors in V^* would complete the pruning step. The remaining piece of the puzzle is that the algorithm does not know the cluster V^* , which we describe how to deal with it next.

10. **Finalize the Pruning Step.** For simplicity, let us assume that $V_1, \dots, V_z$ is a partition of the graph.¹³ The procedure for deleting vertices in X is as follows: For every cluster V_i from [Claim E.31](#), delete all vertices $x \in X$ such that

$$\deg_H(x, V_i) \geq \max\left\{(1 - \frac{1}{3\log^2 n})|V_i|, 2|S|\right\}.$$

Let X' be the remaining vertices in X . We now prove the two properties of X' .

- (a) $|X' \cap S| \leq \tilde{O}(|X \cap L|)$. Since we try for all V_i , we show that the iteration $V_i = V^*$ will delete a lot of vertices in X . If $V_i = V^*$, then

$$\tau = (1 - \frac{1}{3\log^2 n})|V^*| \stackrel{(E.9)}{>} |L_C| \stackrel{(E.6)}{\geq} \frac{\gamma}{2}|S| \geq 2|S|.$$

Therefore, [Lemma E.34](#) and [Lemma E.33](#) imply that $|X' \cap S| \leq |Z_\delta| \leq O(a \log^3 n) = \tilde{O}(a)$.

- (b) $X' \cap L = X \cap L$. We prove that no vertices in $X \cap L$ are deleted. If $V_i = V^*$, then V^* cannot delete vertices in $X \cap L$ by [Lemma E.34](#). If $V_i \neq V^*$, then notice that $X \cap L$ has at most $|S|$ neighbors in V_i . This is because $X \cap L$ can only be adjacent to vertex sets in L_C or vertex set that intersects with S . Moreover, the former case cannot happen since $L_C \subseteq V^*$ which is disjoint from V_i . Thus, $X \cap L$ has at most $|S|$ neighbors in V_i and so V_i cannot delete vertices in $X \cap L$.

E.4.2 Terminal-Balanced Case: Proof of [Lemma E.26](#)

In this section, we show the algorithm for the case when a vertex mincut is balanced with respect to a terminal set ([Lemma E.26](#)) and its extension that will be used later ([Lemma E.36](#)). Our strategy is to employ pseudorandom objects (i.e., [Corollary E.6](#) and [Theorem E.63](#)).

¹³In fact, a sparse neighborhood cover is a collection of $O(\log n)$ partitions of the graph. To remove this assumption, it is enough to repeat the algorithm for all the $O(\log n)$ partitions.

We first need to introduce the vertex version of the isolating cuts lemma, which is from [Li+21].

Lemma E.35 (Lemma 4.2 of [Li+21]). *There exists an algorithm that takes an input graph $G = (V, E)$ and an independent set $I \subset V$ of size at least 2, and outputs for each $v \in I$, a $(v, I \setminus v)$ -min-separator C_v . The algorithm makes (s, t) max-flow calls on unit-vertex-capacity graphs with $O(m \log |I|)$ total number of vertices and edges and takes $O(m)$ additional time.*

We are now ready to prove Lemma E.26. The proof is based on an application of selector Theorem E.63 and crossing family Corollary E.6, which shows κ calls to max flow are possible when $|L|$ is roughly the same as $|S|$ and $|R|$ larger than $|S|$.

Proof of Lemma E.26. Suppose $|T \cap L| \leq k \cdot n^{g(n)}$ and $|T \cap L|, |T \cap R| \geq n^{-g(n)} \cdot |T \cap S|$ where $g(n) = o(1)$. Let $\varepsilon = n^{-2g(n)}$ and let $\mathcal{S}$ be a $(|T|, k/\varepsilon, \varepsilon)$ -selector which is a family of subsets of $[|T|]$. Here we map each number in $1, \dots, |T|$ to a vertex in T , and with a little abuse of notation, each $X \subseteq \mathcal{S}$ is a vertex subset of T . Here we assume $k/\varepsilon \leq |T|^{1-o(1)}$ for any function $o(1)$. the case when k is large will be solved in the next paragraph. According to Theorem E.10, we can construct $\mathcal{S}$ of size $|\mathcal{S}| = k \cdot n^{o(1)}$ in $\hat{O}(n)$ time, and every set $X \subseteq \mathcal{S}$ has size at least 2. Notice that $k < |T \cap L| \leq k/\varepsilon$ and $|T \cap S| \leq k/\varepsilon$. Now according to the definition of selector, there exists $X \subseteq \mathcal{S}$ such that $|X \cap L| = 1$ and $|X \cap S| = 0$ and $|X| \geq 2$. According to Lemma E.35, by setting the independent set I to be a maximal independent set of X , which must satisfy $|I \cap L| = 1, |I \cap S| = 0$ and $|I| \geq 2$, we can get a vertex cut of size at most S since S itself is an isolating cut. Notice that the running time is $\hat{O}(m|\mathcal{S}|) = \hat{O}(mk)$.

Now suppose $k/\varepsilon > |T|^{1-o(1)}$, which means $k \cdot n^{o(1)} \geq |T|$. We construct a $(|T|, 1/\varepsilon)$ -crossing family $\mathcal{P}$ according to Corollary E.6. Similarly, we abuse the notation a bit and assume each $P \subseteq \mathcal{P}$ is a subset of $T \times T$. According to the definition of crossing family, there exists $(s, t) \in P$ such that $s \in L, t \in R$. Thus, by running $s - t$ min cut over all $(s, t) \in P$ and find the minimum one, we get the minimum vertex cut of G . The algorithm runs in time $\hat{O}(m|T|)$ which is fine since $k \cdot n^{o(1)} \geq |T|$. $\square$

The following lemma is a more fine-grained version of Lemma E.26, which works on a subgraph. The proof is similar to that of Lemma E.26. This will be needed later for the unbalanced algorithm in Section E.4.5. It is incomparable to Lemma E.26 because it further requires L to be a subset of T , where Lemma E.26 does not have this requirement.

Lemma E.36. *There is a deterministic algorithm SUBGRAPHBALANCEDTERMINALVC(G, T, k) that takes as inputs an undirected graph $G = (V, E)$, a terminal set $T \subseteq V$ and a cut parameter k , outputs a vertex cut of G . If there exists a vertex cut (L, S, R) of G such that $L \subseteq T$ and $|L|, |T \cap R| = \hat{\Omega}(|T \cap S|)$ and $|L| = \hat{O}(k)$, then the output cut is a minimum vertex cut of G . The algorithm runs in $\hat{O}(k \sum_{t \in T} |N_G(t)|)$ time.*

Proof. The proof is similar to the proof of Lemma E.26. Suppose $|T \cap L| = |L| \leq k \cdot n^{g(n)}$ and $|L|, |T \cap R| \geq n^{-g(n)} \cdot |T \cap S|$ where $g(n) = o(1)$ and define ε and $\mathcal{S}$ and I in the same way as in the proof of Lemma E.26. We will explain what to do when k is large later. Notice that I has the property $|I \cap L| = 1, |I \cap S| = 0$ and $|I| \geq 2$. Now we cannot afford to run isolating cuts on the whole graph because we want the running time to be smaller. Instead, we construct a graph

$$G' = (T \cup N_G(T) \cup \{s\}, E(T, T \cup N_G(T)) \cup \{(s, u) \mid u \in N_G(T)\})$$

Define $I' = I \cup \{s\}$. Notice that $I \cup \{s\}$ is still an independent set because $I \subseteq T$. The algorithm use Lemma E.35 on I', G' . The running time is bounded by $\hat{O}(k \sum_{t \in T} |N_G(t)|)$ where $O(\sum_{t \in T} |N_G(t)|)$ is the size of G' . Now we prove that the minimum isolating cut of I' (denoted by C_v which is a $(v, I' \setminus v)$ -min-separator for some $v \in I'$) on G' is equal to $|S|$. Notice that $S = N_G(L) = N_{G'}(L)$ since $L \subseteq T$. Also, notice that $S \cap I' = \emptyset$. Thus, S' is an isolating cut of I' , which means $|S| \geq |C_v|$ for some $v \in I'$. On the other way, suppose the connected component containing v after deleting C_v is L_v . We have $L_v \subseteq T$, otherwise $s \in N(L_v)$. Thus, we have $N_G(L_v) = N_{G'}(L_v)$. Notice that $N_G(L_v) \cup L_v \neq V$ since $|I| \geq 2$, and there must exists a vertex in I other than v inside T outside $N_G(L_v) \cup L_v$. Thus, C_v is a vertex cut of G , which implies $|S| \leq |C_v|$.

Now suppose $k/\epsilon > |T|^{1-o(1)}$, which means $k \cdot n^{o(1)} \geq |T|$. Similar to the proof of [Lemma E.26](#), we construct an $(|T|, 1/\epsilon)$ -crossing family $\mathcal{P}$. There exists $(a, b) \in P$ such that $a \in L, b \in R$. Instead of running (a, b) min cut on G' over all $(a, b) \in P$, we run $(a, b \cup \{s\})$ min cut on G' . Any $(a, b \cup \{s\})$ min cut (L', S', R') satisfying $L' \subseteq T$ since s connects to all nodes in $N_G(T)$. Thus, $N_G(L') = N_{G'}(L')$ and $t \in R'$ where $t \in V$. Thus, S' is a vertex cut of G . On the other hand, the minimum vertex cut S of G is also a vertex cut of $(a, b \cup \{s\})$ for some $(a, b) \in P$ since $L \subseteq T$. The total running time is $\widehat{O}(k \sum_{t \in T} |N_G(t)|)$. $\square$

E.4.3 Faster Common-Neighborhood Clustering Algorithm

In [Section E.3.2](#) step 1, we presented a common-neighborhood clustering algorithm that runs in $O(mn)$ deterministic time. This running time is too slow as our goal for this section is $\widehat{O}(m\kappa)$. Thus, we present a faster algorithm in this section. We restate the common-neighborhood clustering lemma as in [Lemma E.14](#) that fits our purpose here.

In the following lemma, the symmetric distance oracle $\mathcal{A}_{dist} : V \times V \rightarrow \mathbb{N}$ can be viewed as an algorithm computing the value of $|N_G(u) \triangle N_G(v)|$. The (sparse) property further requires $\mathcal{C}$ to be a collection of $O(\log n)$ partitions of the graph, which is crucial for our usage. The (cover) property can cover all $L \subseteq V$ that is connected and has a low neighborhood symmetric distance, which covers the minimum cut L as we want.

Lemma E.37. *There is a deterministic algorithm $\text{CNC}(G, \mathcal{A}_{dist}, d)$ ([Algorithm 6](#)) that takes as inputs an n -vertex m -edge undirected graph $G = (V, E)$, a symmetric distance oracle $\mathcal{A}_{dist} : V \times V \rightarrow \mathbb{N}$ and $d \in \mathbb{N}^+$ which satisfies triangle inequality $\mathcal{A}_{dist}(u, v) + \mathcal{A}_{dist}(v, w) \geq \mathcal{A}_{dist}(u, w)$, outputs a set of partitions of V denoted by $\mathcal{P}$ such that*

1. (sparse) $|\mathcal{P}| = O(\log n)$, for every $P \in \mathcal{P}$, $P \subseteq 2^V$ is a partition of V , which means $\cup_{C \in P} C = V$ and $C_1 \cap C_2 = \emptyset$ for $C_1, C_2 \in P$ with $C_1 \neq C_2$, we call C a cluster,
2. (cover) for any $L \subseteq V$ where $G[L]$ is connected and $\mathcal{A}_{dist}(u, v) \leq d$ for any $u, v \in L \cap U$, there exists a cluster C such that $L \subseteq C$,
3. (common-neighborhood) for any cluster C , for any $u, v \in C$, we have $\mathcal{A}_{dist}(u, v) = O(d \log n)$.

The algorithm uses $\widetilde{O}(m)$ calls to $\mathcal{A}_{dist}$ and in addition $\widetilde{O}(n \max_{v \in V} |N_G(v)|)$ time.

Algorithm 6: $\text{CNC}(G, \mathcal{A}_{dist}, d)$

```

1 Let  $\mathcal{P} \leftarrow \emptyset, U \leftarrow V$ 
2 Let  $G' = (V, H = \{(u, v) \in E \mid \mathcal{A}_{dist}(u, v) \leq d\})$ 
3 while  $U \neq \emptyset$  do
4    $\mathcal{C} \leftarrow \emptyset$ 
5    $U_{rem} \leftarrow U, U_{next} \leftarrow \emptyset$ 
6   while  $U_{rem} \neq \emptyset$  do
7     Let  $u$  be an arbitrary node in  $U_{rem}$ 
8     Set  $T_0 = \emptyset, T_1 = \{u\}, i = 1$ 
9     while  $|N_{G'[U_{rem}]}(T_i)| \geq |T_{i-1}|$  or  $|T_i| \geq 1.1|T_{i-1}|$  do
10       $i \leftarrow i + 1$ 
11      Build BFS tree  $T_i$  on  $G'[U_{rem}]$  with root  $u$  and only include  $v$  to  $T$  if  $\mathcal{A}_{dist}(u, v) \leq 5id$ 
12      Build BFS tree  $T$  on  $G'[U_{rem}]$  with root  $u$  and only include  $v$  to  $T$  if  $\mathcal{A}_{dist}(u, v) \leq (5i - 3)d$ 
13       $\mathcal{C} \leftarrow \mathcal{C} \cup \{T\}$ 
14       $U_{rem} \leftarrow U_{rem} - T$ 
15       $U_{next} \leftarrow U_{next} \cup (T_i - T_{i-1})$ 
16    $U \leftarrow U_{next}$ 
17    $\mathcal{P} \leftarrow \mathcal{P} \cup \{\mathcal{C}\}$ 
18 return  $\mathcal{P}$ 

```

Proof of Lemma E.37. We first prove (*sparse*). Notice that in the same while loop (Algorithm 6), the tree nodes included in $\mathcal{C}$ (Algorithm 6) are disjoint since Algorithm 6 creates T on $G'[U_{rem}]$ where T is deleted from U_{rem} (Algorithm 6). They form a partition of V since the loop ends if $U_{rem} = \emptyset$. Thus, we only need to prove that the outer while loop (Algorithm 6) only contains $O(\log n)$ loops. We will prove that in each loop, $|U_{next}| \leq 0.1|U|$, which implies the $O(\log n)$ loop numbers since we set $U \leftarrow U_{next}$ at the end of each loop. Notice that U_{next} includes $T_i - T_{i-1}$ (Algorithm 6) where $T_{i-1} \subseteq T$ which is deleted from U_{rem} , and from the condition of Algorithm 6 we have $|T_i| < 1.1|T_{i-1}|$ which implies $|T_i - T_{i-1}| \leq 0.1|T|$. Thus, we can charge $|T_i - T_{i-1}|$ into $0.1|T|$. Notice that the sum of all $|T|$ is at most $|U|$ (Algorithm 6), we have $|U_{next}| \leq 0.1|U|$.

Next, we prove (*cover*). Suppose $L \subseteq V$ where $G[L]$ is connected and $\mathcal{A}_{dist}(u, v) \leq d$ for any $u, v \in L$. We will prove that there exists $C \in \mathcal{C} \in \mathcal{P}$ such that $L \subseteq C$. For this purpose, we only need to prove the following loop invariance: in the while loop Algorithm 6, if $L \subseteq U$ in the beginning, then at the end of this loop, either there exists $C \in \mathcal{C}$ such that $L \subseteq C$ or $L \subseteq U_{next}$. Since U_{next} will be U in the next loop, the loop invariance will imply (*cover*). Now suppose $L \subseteq U$. Notice that the trees T included in $\mathcal{C}$ (Algorithm 6) in this while loop form a partition of U , there must exist T such that $L \cap T \neq \emptyset$. Let T be the first one in the loop of Algorithm 6 that $L \cap T \neq \emptyset$ (which means $L \subseteq U_{rem}$ at this point). If $L \subseteq T$ then we are done since T is included in $\mathcal{C}$. Suppose $L \not\subseteq T$, we claim that $L \cap T_{i-1} = \emptyset$. Otherwise, there exists $w \in L \cap T_{i-1}$ such that $\mathcal{A}_{dist}(u, w) \leq (5i - 5)d$, which implies $\mathcal{A}_{dist}(u, v) \leq (5i - 4)d$ for any $v \in L$. Also notice that for any edge $(u, v) \in G[L]$ we have $\mathcal{A}_{dist}(u, v) \leq d$, thus, $(u, v) \in H$ and $G[L] = G'[L]$ which is connected. Thus, we get $L \subseteq T$ since T is a BFS tree that includes a node v if $v \in N_{G'[U_{rem}]}(T)$ and $\mathcal{A}_{dist}(u, v) \leq (5i - 3)d$ (Algorithm 6), a contradiction. Now we have $L \cap T_{i-1} = \emptyset$. Let $w \in L \cap T$, we have $\mathcal{A}_{dist}(u, w) \leq (5i - 3)d$ and $\mathcal{A}_{dist}(u, v) \leq d$ for any $v \in L$, which implies $\mathcal{A}_{dist}(u, v) \leq (5i - 2)d$ for any $v \in L$. Therefore, $L \subseteq T_i$ since T_i is a BFS tree with root u which includes all nodes v with $\mathcal{A}_{dist}(u, v) \leq 5id$. Therefore, $L \subseteq T_i - T_{i-1}$, which means $L \subseteq U_{next}$.

Next, we prove (*common-neighborhood*). This is because $\mathcal{C}$ only includes T where T contains all nodes v satisfying $\mathcal{A}_{dist}(u, v) \leq (5i - 3)d$ for a specific node u . If we can prove that $i = O(\log n)$, then we are done. For that purpose, we need to prove the while loop Algorithm 6 only contains $O(\log n)$ loops. There are two cases for the while loop to continue. One is $|T_i| \geq 1.1|T_{i-1}|$, which can happen at most $O(\log n)$ times since $|T_i| \leq n$ for any i . In the case that $|N_{G'[U_{rem}]}(T_i)| \geq |T_{i-1}|$, we will prove that $|T_{i+1}| \geq |T_i| + |T_{i-1}| \geq 2|T_{i-1}|$. Indeed, for every vertex $v \in N_{G'[U_{rem}]}(T_i)$, there exists $w \in T_i$ such that $(w, v) \in H$, which means $\mathcal{A}_{dist}(w, v) \leq d$, thus, $\mathcal{A}_{dist}(u, v) \leq \mathcal{A}_{dist}(u, w) + \mathcal{A}_{dist}(w, v) \leq 5(i + 1)d$ and $v \in T_{i+1}$. Thus, $|T_{i+1}| \geq |T_i| + |T_{i-1}|$ as claimed. In both cases, each loop of Algorithm 6 implies that $|T_{i+1}| \geq 1.1|T_{i-1}|$, which can happen at most $O(\log n)$ times.

Next, we prove the running time. To construct G' , we need m calls to $\mathcal{A}_{dist}$ and $O(m)$ running time. As proven above, there are at most $O(\log n)$ loops of Algorithm 6. In each loop, as proven above, there are at most $O(\log n)$ loops of Algorithm 6. Each of them builds a BFS tree with root u , while building the tree T_i will cost $|T_i \cup N_{G'[U_{rem}]}(T_i)|$ many $\mathcal{A}_{dist}$ calls (since we calculate the distance of each of them with u) and in addition the number of edges inside $T_i \cup N_{G'[U_{rem}]}(T_i)$ time, which is at most $|T_i \cup N_{G'[U_{rem}]}(T_i)| \cdot \max_{v \in V} |N_{G'}(v)|$. Notice that T_{i-1} is deleted from U_{rem} and $|T_i \cup N_{G'[U_{rem}]}(T_i)| = O(|T_{i-1}|)$ at the end of loop Algorithm 6. Thus, we can charge the oracles and running time to $\tilde{O}(|T_{i-1}|)$. Since $|U_{rem}| \leq n$, the number of oracle calls is bounded by $\tilde{O}(n)$ and the running time is bounded by $\tilde{O}(n \max_{v \in V} |N_G(v)|)$. In total, the number of oracle calls is $m + \tilde{O}(n) = \tilde{O}(m)$. $\square$

E.4.4 Oracle for the Unbalanced Case

For the unbalanced case, we wish to use the crossing family which gives roughly n/ℓ pairs of nodes where ℓ is roughly the size of L , and one of them will cross L, R . We then want to know $\kappa(s, t)$ for each (s, t) pair of them. However, the trivial way gives use a running time of $\hat{O}(m) \cdot n\kappa/\ell$, which is far from our goal of $\hat{O}(m\kappa)$. The following data structure gives us a way to answer each $\kappa(s, t)$ in $\hat{O}(\delta\ell)$ time, which is sufficient for us as $\hat{O}(\delta\ell) \cdot n\kappa/\ell = \hat{O}(m\kappa)$. We will show how to use this data structure in the next section.

Lemma E.38. *There is a deterministic algorithm that, given an undirected graph G and an integer ℓ , in $\hat{O}(m\delta)$ time returns*

1. vertex sets $V_1, \dots, V_z$ where each vertex $v \in V$ is contained in at most $O(\log n)$ many V_i for different i ,
2. a data structure $\mathcal{D}$ that, given any vertices s and t , it returns an integer $\tilde{\kappa} \geq \kappa(s, t)$ in $\tilde{O}(\delta \ell)$ time.

Suppose (L, S, R) is a minimum vertex cut of G . If $|L| \leq |R|$ and $\ell = \lceil |L|, 2|L| \rceil$, then one of the following two events happens.

1. $\exists i, L \subseteq V_i, |V_i \cap S| = \tilde{O}(|L|), |V_i \cap R| = \tilde{\Omega}(|L|)$.
2. If $\mathcal{D}$ is given $s \in L, t \in R$, it will return $\tilde{\kappa} = \kappa(G)$.

We need the following sparse recovery tool from [NS17].

Lemma E.39 (Deterministic sparse recovery, corollary of Theorem 4.14 [NS17]). *There are deterministic algorithms $\mathcal{A}_{\text{construct}}(G, x)$ and $\mathcal{A}_{\text{recover}}(\vec{v}, \vec{u})$ such that*

- $\mathcal{A}_{\text{construct}}$ takes as inputs an undirected graph $G = (V, E)$ and a positive integer x outputs a vector $\vec{v}$ with length $\hat{O}(x)$ for each $v \in V$ in $\hat{O}(m)$ time,
- $\mathcal{A}_{\text{recover}}$ takes as inputs two vectors $\vec{v}, \vec{u}$ computed by $\mathcal{A}_{\text{construct}}$, outputs $N_G(u) \triangle N_G(v)$ in $\hat{O}(x)$ time if $|N_G(u) \triangle N_G(v)| \leq x$.

Proof. In Theorem 4.14 [NS17], we set $k = x$ and $n = |V|$, we can construct $\Phi \in \{0, 1\}^{d \times n}$ ($d = \hat{O}(x)$) implicitly in $\hat{O}(n)$ time by knowing all the non-zero entities. Now we abuse the notation a bit and define $N_G(u) \in \{0, 1\}^n$ for each u to be the identity vector where the i -th entry is 1 iff the i -th node in V is in $N_G(u)$. $\mathcal{A}_{\text{construct}}$ will output $v = \Phi N_G(u)$ for any $u \in V$, which takes time $\hat{O}(m)$ in total since $\Phi N_G(u)$ takes time $\hat{O}(|N_G(u)|)$ (each column of Φ has $\hat{O}(1)$ non-zero entries).

$\mathcal{A}_{\text{recover}}$ will take $\Phi N_G(u), \Phi N_G(v) \in \{0, 1\}^d$ and compute $\Phi(N_G(u) - N_G(v))$. According to Theorem 4.14 [NS17], if $N_G(u) - N_G(v)$ contains at most x non-zero entries (i.e., $|N_G(u) \triangle N_G(v)| \leq x$), then all the non-zero entries are returned, which give us $N_G(u) \triangle N_G(v)$. $\square$

We show our algorithm for Lemma E.38 as follows, i.e., how to compute the vertex sets $V, V_1, \dots, V_z$ and the memory space of the data structure $\mathcal{D}$. We will show how $\mathcal{D}$ uses the memory space to answer a query later.

Algorithm for Lemma E.38. The algorithm takes inputs an undirected graph $G = (V, E)$ and an integer ℓ . The following procedure will find the data structure $\mathcal{D}$.

1. Compute the minimum degree of the graph, denoted by δ . Let V_{low} contain all vertices v with $|N_G(v)| = O(\delta)$.
2. Define distance $\mathcal{A}_{\text{dist}}(u, v)$ for every $u, v \in V_{\text{low}}$ as $\mathcal{A}_{\text{dist}}(u, v) = |N_G(u) \triangle N_G(v)|$. Notice that $\mathcal{A}_{\text{dist}}(u, v)$ can be computed in time $\tilde{O}(\delta)$.
3. Let $\mathcal{P} \leftarrow \text{CNC}(G[V_{\text{low}}], \mathcal{A}_{\text{dist}}, 4\ell)$. Let $V_1, \dots, V_z$ be all vertex sets in $\mathcal{C}$ for all $\mathcal{C} \in \mathcal{P}$, i.e., $z = \sum_{\mathcal{C} \in \mathcal{P}} |\mathcal{C}|$.
4. Use $\mathcal{A}_{\text{construct}}(G, \ell \log^2 n)$ in Lemma E.39 to get a vector $\mathbf{v}$ for every $v \in V$.
5. Data structure $\mathcal{D}$ memorizes (i) $V_1, \dots, V_z$ and the sizes of them, (ii) for every $u \in V$, the index set $I_u = \{i \mid u \in V_i\}$, (iii) the vectors $\mathbf{v}$ for every $v \in V$.

The algorithm returns vertex sets $V, V_1, \dots, V_z$ and the data structure $\mathcal{D}$. We will show the query algorithm for $\mathcal{D}$ later. The following claim is a crucial property.

Claim E.40. For every $u, v \in V_i$, $|N_G(u) \triangle N_G(v)| = O(\ell \log n)$.

Proof. From Lemma E.37, we have $\mathcal{A}_{\text{dist}}(u, v) = O(\ell \log n)$. According to the definition of $\mathcal{A}_{\text{dist}}$, we have $|N_G(u) \triangle N_G(v)| = O(\ell \log n)$. $\square$

The following lemma shows the running time of the algorithm.

Lemma E.41. The algorithm for constructing vertex sets and $\mathcal{D}$ described above runs in time $\tilde{O}(m\delta)$.

Proof. We analyze each step.

1. Computing the minimum degree costs $O(m)$.
2. The distance oracle uses time $\tilde{O}(\delta)$ for each query.
3. By [Lemma E.37](#), the running time is $\tilde{O}(m\delta)$.
4. By [Lemma E.39](#), the running time is $\hat{O}(m)$.

□

Intuition for Kernel Graphs. Suppose we are given $s \in L, t \in R$. Let V' be one of the clusters such that $L \subseteq V'$. Our goal is to compute a graph H such that $\kappa_G(s, t) = \kappa_H(s, t)$. At the same time, we will guarantee that $\kappa_G(s, t) \leq \kappa_H(s, t)$ always holds even if $s, t \in R$ and $L \subseteq V'$ do not hold.

WLOG, we assume that $t \notin N_G[V']$. If $t \in N_G[V']$, we delete all neighbors of t and t itself from V' , as they cannot be in L , after which we still have $L \subseteq V'$ and the common neighborhood guarantee of V' . We define the graph H in the following way. Define $E_G(A, B) = \{(u, v) \in E \mid u \in A, v \in B\}$ for two vertex sets $A, B \subseteq V$.

Let H' be the graph with vertex set $N_G[V'] \cup \{t\}$ and edge set $E_G(V', N_G[V']) \cup \{(v, t) \mid v \in N_G(V')\}$, i.e., all edges in G incident to vertices in V' plus new edges from every node in $N_G(V')$ to t .

Let H be obtained from H' as follows. Vertex s and t mark all of their incident edges. Other vertices in $V' \setminus \{s\}$ mark edges not incident to $N_G(s)$. Then, we delete all unmarked edges from H' .

The following claim is crucial to the correctness and shows the reason for constructing H in the above manner.

Claim E.42. *For any vertex set $A \subseteq V'$ with $s \in A$, we have $N_H(A) = N_G(A)$.*

Proof. We first show that $N_H(A) = N_{H'}(A)$. It is easy to see that $N_H(A) \subseteq N_{H'}(A)$. Now, suppose for contradiction that there exists $v \in N_{H'}(A) \setminus N_H(A)$. Thus, there is $u \in A \subseteq V'$ where $u \neq s$ and (u, v) is unmarked. But this happens only when $v \in N_G(s)$. Since $s \in A$, we have $v \in N_H(A)$, a contradiction.

Now we show $N_{H'}(A) = N_G(A)$. This follows from the fact that all edges incident to $A \subseteq V'$ are preserved in H' compared to G . □

Correctness. We need to prove that the $s - t$ minimum vertex cut, denoted by (L', S', R') where $s \in L', t \in R'$ in H has the same size as (L, S, R) , which is the minimum vertex cut of G with $s \in L, t \in R$. We need to prove two directions.

1. ($|S| \geq |S'|$) for that we need to prove S is also a $s - t$ vertex cut in H . Remember that $L \subseteq V'$ and $N_G(L) = S$. Thus, $N_G(L) \subseteq N_H[V']$, which does not contain t , is an $s - t$ vertex cut.
2. ($|S| \leq |S'|$) for that we need to prove S' is also a $s - t$ vertex cut in G . Since t has an edge to every vertex in $N_G(V')$, we have $L' \subseteq V'$, otherwise $t \in N_H(L')$. Thus, $S' = N_H(L') = N_G(L')$ is a $s - t$ vertex cut in G .

Moreover, the second argument holds even if L is not a subset of V' or $s \in L, t \in R$ does not hold. Thus, we always have $\kappa_G(s, t) \leq \kappa_H(s, t)$.

Query algorithm for the data structure $\mathcal{D}$. Before stating our algorithm, we first define our *kernel graph* with respect to some V_i and $s \in V_i, t \in V$ as follows.

Definition E.43. For $i \in [z], s \in V_i, t \in V$, let $V'_i = V_i \setminus N_G[t]$, define $G_{s,t}^i = (V_{s,t}^i, E_{s,t}^i)$ where

$$\begin{aligned} V_{s,t}^i &= N_G[V'_i] \cup \{t\} \\ E_{s,t}^i &= \{(s, v) \mid v \in N_G(s)\} \\ &\quad \cup \{(u, v) \mid u \in V'_i, v \in N_G(u) \setminus N_G(s)\} \\ &\quad \cup \{(u, t) \mid u \in N_G(V'_i), u \neq t\} \end{aligned}$$

Note that the construction of $G_{s,t}^i$ is the same as the one described in the overview. We state the following key properties of $G_{s,t}^i$ whose proofs are essentially the same as shown above.

Lemma E.44. $\kappa_{G_{s,t}^i}(s, t) \geq \kappa_G(s, t)$.

Lemma E.45. Let (L, S, R) be a minimum vertex cut of G . If $s \in L, t \in R, L \subseteq V_i$, then $\kappa_{G_{s,t}^i}(s, t) = \kappa(G)$.

Now we are ready to state our query algorithm for $\mathcal{D}$.

Query Algorithm for $\mathcal{D}$

1. Let I_s be all indices i with $s \in V_i$. If $I_s = \emptyset$, **return** n .
2. $I'_s = \{i \in I_s : |V_i| \leq O(\delta \log n)\}$. If $I'_s = \emptyset$, **return** n .
3. For every $i \in I'_s$, construct a graph $G_{s,t}^i = (V_{s,t}^i, E_{s,t}^i)$ defined in [Definition E.43](#).
4. Let S' be the minimum (s, t) -separator in $G_{s,t}^i$ (which can be done by solving a vertex-capacity max flow on $G_{s,t}^i$).
5. **return** $|S'|$.

Next, we prove that $G_{s,t}^i$ can be constructed efficiently.

Lemma E.46. For every $i \in I'_s$, $G_{s,t}^i$ can be constructed in $\widehat{O}(|V_i|\ell) = \widehat{O}(\delta\ell)$ time.

Proof. We can find $V'_i = V_i \setminus N_G[t]$ in time $\tilde{O}(|V_i|)$ by checking whether each node in V_i is in $N_G[t]$ or not. To find $N_G[V'_i]$, for each $u \in V'_i$, list $N_G(u) \setminus N_G(s)$ and include them in $V_{s,t}^i$ and also add (u, v) for all $v \in N_G(u) \setminus N_G(s)$ to the set of edges. To list $N_G(u) \setminus N_G(s)$ efficiently, notice that according to [Claim E.40](#), $N_G(u) \setminus N_G(s)$ is guaranteed to have size $O(\ell \log n)$, and according to [Lemma E.39](#), we can use $\mathcal{A}_{\text{recover}}$ to find $N_G(u) \setminus N_G(s)$ in $\widehat{O}(\ell)$ time. In total, we spent time $\widehat{O}(\ell|V_i|)$ on $\mathcal{A}_{\text{recover}}$. Finally, add all the edges from t to $N_G(V_{s,t}^i)$. Each step takes $\widehat{O}(|V_i|\ell)$ time, so the lemma is proved. $\square$

We are ready to prove [Lemma E.38](#).

Proof of Lemma E.38. According to [Lemma E.37](#), each $v \in V$ is contained in at most $O(\log n)$ many V_i for different i . The algorithm also returns a data structure $\mathcal{D}$, according to [Lemma E.41](#), it costs $\widehat{O}(m\delta)$ time to construct. Given (s, t) , according to [Lemma E.46](#), $\mathcal{D}$ takes $\widehat{O}(\delta\ell)$ time to construct the kernel $G_{s,t}^i$ for any $i \in I'_s$ (recall that $|I'_s| = O(\log n)$ according to [Lemma E.37](#)) and we run vertex-capacity max flow calls on $G_{s,t}^i$. According to [Lemma E.44](#), the returned value of $\mathcal{D}$ on each query s, t is at least $\kappa(G)$.

Now suppose (L, S, R) is a minimum vertex cut of G where $|L| \leq |R|$ and $\ell \in [|L|, 2|L|]$. Notice that if $|L| = \Omega(|S|)$, then the first event happens by taking $V_i = V$. Thus, in what follows we assume $|L| = o(|S|)$. We need the following claim.

Claim E.47. For any $u, v \in L$, $|N_G(u) \triangle N_G(v)| \leq 2|L|$.

Proof. We have $|N(u) \triangle N(v)| = |N(u) \cup N(v)| - |N(u) \cap N(v)|$, where $|N(u) \cup N(v)| \leq |L| + |S|$, and $|N(u) \cap N(v)| = |N(u)| + |N(v)| - |N(u) \cup N(v)|$. Here $|N(u)| \geq |S|$ since $|S|$ is a minimum separator, same for $|N(v)|$. Thus, $|N(u) \cap N(v)| \geq 2|S| - (|L| + |S|)$. We have $|N(u) \triangle N(v)| \leq (|L| + |S|) - (2|S| - (|L| + |S|)) = 2|L|$. $\square$

We can assume WLOG that $G[L]$ is connected; otherwise, by taking a connected component of $G[L]$ we get another minimum vertex cut such that $G[L]$ is connected. According to [Lemma E.37](#) property (cover), $L \subseteq V_i$ for some $i \in [z]$.

We claim that if $|V_i| = O(\delta \log n)$, then the second event happens. Suppose $\mathcal{D}$ is given $s \in L, t \in R$, we have $|N_G(s)| = O(|L| + |S|) = O(\delta)$ since $|L| = o(|S|)$ and $|S| \leq \delta$. Besides, we have $i \in I_s$ and $|V_i| \leq O(\delta \log n)$. Thus, according to the query algorithm for $\mathcal{D}$, $G_{s,t}^i$ is constructed. According to [Lemma E.45](#), $\mathcal{D}$ will return exactly $|S| = \kappa(G)$.

Now assume $|V_i| = \omega(\delta \log n)$, we will prove that the first event happens, i.e., $|L| = \tilde{\Omega}(|V_i \cap S|)$, $|V_i \cap R| = \tilde{\Omega}(|V_i \cap S|)$ (remember that $L \subseteq V_i$ is already proved). If $|V_i \cap S| = O(\ell \log n) = O(|L| \log n)$, then we are done since $|V_i \cap R| = |V_i| - |L| - |V_i \cap S| = \Omega(\delta \log n)$ (remember that $\ell = \Theta(|L|) = o(|S|) = o(\delta)$).

Now we will prove that $|V_i \cap S| = \omega(|L| \log n)$ cannot happen. We use double counting on the size of $E_G(V_i \cap S, V_i \cap R)$. We have

$$|E_G(V_i \cap S, V_i \cap R)| \leq |V_i \cap S| \cdot O(\ell \log n).$$

In fact, every $u \in V_i \cap S$ has at most $O(\ell \log n)$ neighbors in $V_i \cap R$. This is because u is a neighbor $v \in L$, which has no neighbor in $V_i \cap R$, but the neighborhood of u and v are similar; more precisely, $|N_G(u) \Delta N_G(v)| = O(\ell \log n)$ according to [Claim E.40](#).

On the other hand, consider a node $v \in L$, since $|N_G(v)| \geq |S|$ and $N_G(v) \subseteq L \cup S$, we have $|S \setminus N_G(v)| \leq |L|$. Thus, we have $|V_i \cap N_G(v)| = \Theta(|V_i \cap S|)$. Now we get

$$|E_G(V_i \cap S, V_i \cap R)| \geq \Omega(|V_i \cap R| \cdot |V_i \cap S|)$$

since $|N_G(u) \Delta N_G(v)| = O(\ell \log n)$ for any $u \in V_i \cap R$ and $|V_i \cap N_G(v)| = \Theta(|V_i \cap S|) = \omega(\ell \log n)$. By combining the inequalities above, we get $|V_i \cap R| = O(\ell \log n)$. Thus, we have $|V_i \cap S| = |V_i| - |V_i \cap L| - |V_i \cap R| = \omega(\delta \log n)$. This is a contradiction since $|S| \leq \delta$. $\square$

E.4.5 Unbalanced Case: Proof of [Lemma E.23](#)

We are now ready to prove [Lemma E.23](#). We describe an algorithm and analysis below.

Algorithm. For every $\ell = 2^i$ from $i = 1$ to $i = \lceil \log(\delta \log n) \rceil$ (which serves as an estimate of $|L|$), the algorithm uses the following procedure (called the i -th iteration) to find the separator $S^{(i)}$ as follows.

The algorithm firstly uses [Corollary E.6](#) with parameters $\alpha = \frac{2\delta}{\ell}$ to compute an (n, α) -crossing family P . Then, it uses [Lemma E.38](#) on G, ℓ to get (i) $V_1, V_2, \dots, V_z$, (ii) the data structure $\mathcal{D}$. Then the algorithm uses [Lemma E.36](#) on $G, V_i, \delta \log n$ for every $i \in [z]$ to get separators $S_1, \dots, S_z$, and uses the data structure $\mathcal{D}$ with queries $(s, t) \in P$ to get $\tilde{\kappa}_{s,t} \geq \kappa(s, t)$ for every $(s, t) \in P$. If among $|S_1|, \dots, |S_z|$ and $\tilde{\kappa}_{s,t}$ for any $s, t \in P$, the minimum value is $|S_{j_i}|$, then set $S^{(i)} = S_{j_i}$; otherwise, suppose the minimum value is $\tilde{\kappa}_{s,t}$ for some $(s, t) \in P$, we set $S^{(i)}$ to be the minimum separator between s, t , denoted by $S_{s,t}$, which can be found by one call to unit-vertex-capacity max flow on the whole graph.

Finally, return the minimum $S^{(i)}$ among all i as the output.

Correctness. According to [Lemma E.36](#), the returned cuts $S_1, \dots, S_z$ are always a valid separator. Besides, $S_{s,t}$ must be a valid separator since it is deduced from a vertex-capacity max flow from s to t . Thus, we just need to prove that one of $S^{(i)}$ is a minimum separator. Suppose (L, S, R) is one of the minimum separators of G with $|L| \leq |R|$ and $|L|, |S| = O(\delta)$. There must exist i such that $\ell = 2^i \in [|L|, 2|L|]$. We will prove that $S^{(i)}$ is a minimum cut. In order to prove this, we just need to show that in the i -th iteration, either one of $S_1, \dots, S_z$ is a minimum cut, or one of $\tilde{\kappa}_{s,t}$ for $(s, t) \in P$ satisfies $\tilde{\kappa}_{s,t} = \kappa(G)$. If this is true, then $S^{(i)}$ is either $|S_j|$ for some j such that $|S_j| = \kappa(G)$, or some s, t minimum separator for some $(s, t) \in P$ with $\tilde{\kappa}_{s,t} = \kappa(G) \geq \kappa(s, t)$ (in which case $\kappa(s, t) = \kappa(G)$).

According to [Lemma E.38](#), there are two events to consider.

Case 1: $\exists i, L \subseteq V_i, |V_i \cap S| = \tilde{O}(|L|), |V_i \cap R| = \tilde{\Omega}(|L|)$. In this case, since $|V_i \cap L| \leq |L| \leq \delta \log n$, by using [Lemma E.36](#) on $G, V_i, \delta \log n$, we can get S_i which is a minimum separator of G .

Case 2: $\mathcal{D}$ is given $s \in L, t \in R$. We prove that it will return $\tilde{\kappa} = \kappa(G)$. Indeed,

Lemma E.48. *there exists $(s, t) \in P$ such that $s \in L, t \in R$.*

Proof. Let (L, S, R) be a vertex mincut in G where $|R| \geq |L|$. Since S is a minimum separator, $|S| \leq \delta$ (otherwise the neighborhoods of the vertex with a minimum degree should be the minimum separator). We

have $|S|/|L| \leq (\delta)/(\ell/2) = \alpha$. According to [Corollary E.6](#), since we have $|R| \geq |L| \geq |S|/\alpha$, there exists $(s, t) \in P$ such that $s \in L, t \in R$. $\square$

By [Lemma E.38](#), $\mathcal{D}$ will return $\tilde{\kappa}_{s,t} = \kappa(G)$ on such $s \in L, t \in R$.

Time complexity. For each iteration, [Corollary E.6](#) and [Lemma E.38](#) use $\tilde{O}(m\delta)$ time. Then [Lemma E.36](#) on each V_i takes time $\hat{O}\left(\sum_{v \in V_i} |N_G(v)|\right)$, which in total takes time $\hat{O}(m\delta)$ since each vertex in V is included in at most $O(\log n)$ V_i 's. One call to s, t vertex-capacity max flow takes time $\hat{O}(m)$. It remains to bound the running time of queries to $\mathcal{D}$, which is

$$|P| \cdot \hat{O}(\delta\ell) = O(n\alpha) \cdot \hat{O}(\delta\ell) = \hat{O}(n\delta^2) = \hat{O}(m\delta).$$

The first equality follows from [Corollary E.6](#) and [Lemma E.38](#).

E.4.6 Terminal Reduction: Proof of [Lemma E.25](#)

This section is devoted to proving [Lemma E.25](#). We first restate the omitted bottlenecks in the overview and explain how we solve it.

In the overview, we mentioned that common-neighborhood clustering is not efficient as trivially it needs $\tilde{O}(mn)$ time. The algorithm presented in [Section E.4.3](#) is efficient enough as $\log n$ as the input graph has degree κ which we can guarantee because all nodes with degrees much larger than κ must not be in L and can be deleted. However, according to [Lemma E.37\(2\)](#), it requires L to be connected in G . It is not guaranteed that L_U is a connected induced subgraph. Remember the definition of L_U : all clusters that are completely inside L . $G[L]$ is connected indeed, but $G[L]$ contains three parts (i) L_U , (2) $X \cap L$, (3) nodes in other clusters crossing S but not in L_U . Deleting the second and third parts could cause L_U to be disconnected.

Our solution. We try to add (not too many) extra edges to make L_U connected. This is reflected from [Algorithm 7](#) to [Algorithm 7](#) in the algorithm. The idea is: for every $x \in X \cap L$, we want to make sure that all neighbors of x inside L_U are connected by extra edges. A trivial way to do this is by adding a spanning subgraph on neighbors of x . However, this simple strategy does not solve the problem: assume neighbors of x are $v_1, \dots, v_k$, after adding a spanning subgraph connecting them (for example, a star rooted at v_1), if v_1 is not in L_U , then the star helps nothing for connecting L_U : there are no edges added to the induced subgraph on L_U ! We solve this issue by the shaving lemma [Lemma E.49](#). At a high level, it helps to shave nodes out of the neighbors of x , so that the remaining nodes are "roughly in L_U " in the sense that they have a small neighborhood symmetric difference just as the nodes in L_U , which is enough for us. For this shaving lemma, think of A as the neighbors of x , and A' as the neighbors of x inside L_U , we want to find R covering A' such that R itself has a small neighborhood symmetric difference.

Lemma E.49. *There is a deterministic algorithm $\text{SHAVING}(G, A, a)$ that takes as inputs an undirected graph $G = (V, E)$, a vertex set A and a integer a , return $R \subseteq A$. Moreover, if there exists $B \subseteq V, A' \subseteq A$ satisfying the following conditions*

- $|A'| \geq 0.99|A|$,
- for any $u \in A'$ we have $N_G(u) \subseteq B$ and $|B - N_G(u)| \leq a$,

then the returned set R further satisfies

- $A' \subseteq R$,
- $|N_G(u) \Delta N_G(v)| \leq 5a$ for any $u, v \in R$.

The algorithm runs in time $O(\sum_{u \in A} |N_G(u)|)$.

A careful reader might notice that even if we connect all neighbors of x inside L_U for every $x \in X \cap L$, L_U still could be disconnected: remember L has three parts, and the third part (nodes in clusters crossing S but not in L_U) is also removed from L , which could disconnect L_U . The solution is to give a better analysis: if after removing the third part L is disconnected into many components, instead of trying to cover the whole

L_U using common-neighborhood clustering, we can try to cover each of the components, which are indeed connected. This does not change the algorithm but puts more challenges on the analysis.

Now we start describing the algorithm. We first introduce the notion of vertex expanders.

Definition E.50. We define the *terminal vertex expansion* h_T of a vertex cut (L, S, R) in graph G with respect to the terminal set T as follows:

$$h_T(L, S, R) = \frac{|S|}{\min\{|T \cap (L \cup S)|, |T \cap (R \cup S)|\}}.$$

We say the cut (L, S, R) is (T, ϕ) -vertex expanding in G if $h_T(L, S, R) \geq \phi$ and (T, ϕ) -vertex sparse if $h_T(L, S, R) < \phi$. We say that a graph G is a (T, ϕ) -vertex expander if every vertex cut (L, S, R) such that $\min(|T \cap (L \cup S)|, |T \cap (R \cup S)|) > 0$ in the graph G is (T, ϕ) -vertex expanding.

We need the following lemma as a subroutine of our algorithm [Algorithm 7](#).

Lemma E.51 (Vertex expander decomposition, [\[LS22\]](#)). *There is a deterministic algorithm EXPANDERDECOMPOSITION(G, T) that takes as inputs an n -vertex m -edge undirected graph $G = (V, E)$ and a terminal set $T \subseteq V$, return $(X, \mathcal{U})$ where $X \subseteq V$ and $\mathcal{U}$ is a set of subsets of V , such that*

1. X along with all sets in $\mathcal{U}$ form a partition of V , i.e., they are disjoint and the union of them is V ,
2. there are no edges between different sets in $\mathcal{U}$, i.e., $E(U_1, U_2) = \emptyset$ for any $U_1, U_2 \in \mathcal{U}$ with $U_1 \neq U_2$,
3. $G[U]$ is a $(T, 2^{-\log^{0.1} n})$ -vertex expander for any $U \in \mathcal{U}$,
4. $|X| < 0.01|T|$.

The algorithm runs in time $\hat{O}(m)$.

Proof. We first calculate $N_G(u) \cap A$ for any $u \in V$. This can be done by the following way in time $O(\sum_{u \in A} |N_G(u)|)$: for every $x \in A$, we increase the count for each $v \in N_G(x)$ by 1. Now let V' contain all v with $|N_G(v) \cap A| \geq 0.1|A|$. We claim that $V' \subseteq B$ and $|B - V'| \leq 1.2a$. To see $V' \subseteq V$, notice that at most $0.01|A|$ nodes in A have edges to $V - B$. To see $|B - V'| \leq a$, notice that for a vertex in B to be not included in V' , it needs to have more than $0.9|A|$ non-edges to A , which includes at least 0.89 non-edges A' . However, the total number of non-edges from A' to B' is at most $a \cdot |A|$. Thus, there are at most $a \cdot |A| / 0.9|A| \leq 1.2a$ nodes in B that is not in V' . Now we set R to be all nodes $u \in A$ with $|N_G(u) \triangle V'| \leq 2.2a$. For every node $u \in A'$, we have $|N_G(u) \triangle V'| \leq a + 1.2a \leq 2.2a$, which means $A' \subseteq R$. For any $u, v \in R$, we have $|N_G(u) \triangle N_G(v)| \leq 4.4a \leq 5a$. $\square$

To prove [Lemma E.25](#), we describe the algorithm, shown in [Algorithm 7](#), and analysis.

Algorithm 7: TERMINALREDUCTION(G, T, k)

```

1 Let  $(X, \mathcal{U}) \leftarrow \text{EXPANDERDECOMPOSITION}(G, T)$  (see Lemma E.51)
2 Let  $S^* = V, T_{\text{big}} = \emptyset, T_{\text{small}} = \emptyset$  and  $T_{\bar{X}}$  be arbitrary  $\lfloor |T|/100 \rfloor$  nodes in  $T - X$ 
3 Let  $\mathcal{U}_{\text{small}} = \{U \in \mathcal{U} \mid 0 < |T \cap U| < 5\}, \mathcal{U}_{\text{big}} = \{U \in \mathcal{U} \mid |T \cap U| \geq 5\}$ 
4 foreach  $U \in \mathcal{U}_{\text{big}}$  do
5   Include arbitrary  $\lceil \frac{2}{3}|T \cap U| \rceil$  vertices in  $T \cap U$  into  $T_{\text{big}}$ 
6 Construct  $G' = (X \cup \mathcal{U}_{\text{small}}, E')$  where  $E' = \{(x, U) \in X \times \mathcal{U}_{\text{small}} \mid \exists u \in U, (x, u) \in E\}$ 
7 Let  $\mathcal{U}_{\text{low}}^{\text{small}} \leftarrow \{U \in \mathcal{U}_{\text{small}} \mid |N_{G'}(U)| \leq 2k\}$  and  $H_{\text{low}} = G'[X \cup \mathcal{U}_{\text{low}}^{\text{small}}]$ 
8 foreach integer  $1 \leq i \leq \lceil \log k \rceil$  do
9   Let  $a \leftarrow 2^i$ 
10  Let  $X_{\text{low}} \leftarrow \{x \in X \mid |N_{H_{\text{low}}}(x)| < 1000a\}$ 
11  foreach  $x \in X$  do
12    Let  $R_x \leftarrow \text{SHAVING}(H_{\text{low}}, N_{H_{\text{low}}}(x), a)$  (see Lemma E.49)
13    Let  $H_x$  be arbitrary  $|R_x| - 1$  edges (not in the original graph) that span  $R_x$ 
14  Build a graph  $H_{\text{connected}}$  with vertex set  $\mathcal{U}_{\text{low}}^{\text{small}}$ , and edge set be the union of  $H_x$  for all  $x \in X$ 
15  Define distance function  $\mathcal{A}_{\text{dist}}(u, v) = |N_{G'}(u) \Delta N_{G'}(v)|$ 
16  Let  $\mathcal{P} \leftarrow \text{CNC}(H_{\text{connected}}, \mathcal{A}_{\text{dist}}(u, v), 15a)$  (see Lemma E.37)
17  foreach  $P \in \mathcal{P}$  do
18    Let  $X' \leftarrow X$ 
19    foreach  $C \in P$  do
20      For arbitrary  $\lfloor \frac{|C|}{\log^2 n} \rfloor$  nodes  $U \in C \subseteq \mathcal{U}_{\text{low}}^{\text{small}}$ , include  $|T \cap U|$  into  $T_{\text{small}}$ 
21      if  $|C| > 10a \log^2 n$  then
22        foreach  $x \in X' \cap N_{G'}(C)$  do
23          Delete  $x$  from  $X'$  if  $|N_{G'}(x) \cap C| > (1 - \frac{1}{\log^3 n}) |C|$ 
24    Let  $S_1 \leftarrow \text{BALANCEDTERMINALVC}(G, X' \cup T_{\bar{X}}, k)$  (see Lemma E.26)
25    Let  $S_2 \leftarrow \text{BALANCEDTERMINALVC}(G, X_{\text{low}} \cup T_{\bar{X}}, k)$ 
26    If  $|S_1| < |S^*|$ , let  $S^* \leftarrow S_1$ . If  $|S_2| < |S^*|$ , let  $S^* \leftarrow S_2$ 
27 Let  $T' \leftarrow X \cup T_{\text{big}} \cup T_{\text{small}} \cup T_{\bar{X}}$ 
28 return  $(S^*, T')$ 

```

We now analyze [Algorithm 7](#).

Correctness. We first show that $|T'| \leq 0.9|T|$. Remember that $T' = X \cup T_{\text{big}} \cup T_{\text{small}} \cup T_{\bar{X}}$. We bound each term as follows.

1. According to [Lemma E.51](#), $|X| < 0.01|T|$.
2. According to [Algorithm 7](#), T_{big} includes $\lceil \frac{2}{3}|T \cap U| \rceil$ vertices for every U with $|T \cap U| \geq 5$, which is at most $4|T \cap U|/5$ nodes. Remember that U are disjoint for different $U \in \mathcal{U}$, therefore, $|T_{\text{big}}| \leq 4|T|/5$.
3. According to [Algorithm 7](#), we include $T \cap U$ into T_{small} for at most $|C|/\log^2 n$ number of U for every $C \in \mathcal{C}$. Since $|T \cap U| < 5$, we includes at most $4|C|/\log^2 n$ nodes into T_{small} for each C . According to [Lemma E.37](#), $\sum_{C \in \mathcal{C}} |C| = O(|\mathcal{U}_{\text{low}}^{\text{small}}| \log n)$. Since every $U \in \mathcal{U}_{\text{small}}$ contains at least one terminal in T , we have $|T_{\text{small}}| = O(|T|/\log n)$, which is at most $0.01|T|$ for sufficiently large n .
4. According to [Algorithm 7](#), we have $|T_{\bar{X}}| \leq 0.01|T|$.

In total, $|T'| \leq (0.01 + 0.8 + 0.01 + 0.01)|T| \leq 0.9|T|$.

It remains to prove either S^* is a minimum vertex cut of G or $T' \subseteq V$ is a 0.01-strongly-balanced terminal set. Let S be an arbitrary vertex cut of G and L be a connected component after deleting S from G . Since T is $2^{\sqrt{\log n}/2}$ -strongly-balanced and $k \in [\kappa_G, C\kappa_G]$ for sufficient large constant C , according to [Definition E.24](#), we have $|T \cap L| \geq 2^{\sqrt{\log n}/2} k$. Our goal is to prove that if $|T' \cap L| < 0.01k$, then S^* (the output of [Algorithm 7](#)) must be a minimum vertex cut of G . Since $T' = X \cup T_{\text{big}} \cup T_{\text{small}} \cup T_{\bar{X}}$, we can assume $|X \cap L| < 0.01k$,

$|T_{\text{big}} \cap L| < 0.01k$ and $|T_{\text{small}} \cap L| < 0.01k$. Write $T_L = T \cap L, T_S = T \cap S, T_R = T \cap (V - L - S)$. In the following lemmas, we will specify what happens if $|T_{\text{big}} \cap L| < 0.01k$ and $|T_{\text{small}} \cap L| < 0.01k$, separately.

Lemma E.52. *If $|X \cap L| < k/100$ and $|T_{\text{big}} \cap L| < k/100$, then $|\cup_{U \in \mathcal{U}_{\text{small}}} (T_L \cap U)| > 2\sqrt{\log n/2}k/4$.*

Proof. Suppose to the contrary, $|\cup_{U \in \mathcal{U}_{\text{small}}} (T_L \cap U)| \leq 2\sqrt{\log n/2}k/4$. Then we have $|\cup_{U \in \mathcal{U}_{\text{big}}} (T_L \cap U)| \geq 2\sqrt{\log n/2}k/5$ since $|T_L| \geq 2\sqrt{\log n/2}k$ and nodes in T can be either $U \in \mathcal{U}_{\text{small}} \cup \mathcal{U}_{\text{big}}$ or X where $|X| < 0.01|T|$.

Let $\tilde{\mathcal{U}}$ contain all $U \in \mathcal{U}_{\text{big}}$ with (i) $|U \cap T_L| \geq |U \cap T_R|$, (ii) $|U \cap T_L| \geq 2|U \cap S|$. We first prove that $|\cup_{U \in \tilde{\mathcal{U}}} (T_L \cap U)| \geq 2\sqrt{\log n/2}k/10$. For that purpose, we will bound the size of $|\cup_{U \in \mathcal{U}_{\text{big}} - \tilde{\mathcal{U}}} (T_L \cap U)|$. If (i) is violated, since $G[U]$ is a $(T, 2^{-\log^{0.1} n})$ -vertex expander according to [Lemma E.51](#) and according to the definition of expander [Definition E.50](#), we have $\frac{|U \cap S|}{|U \cap (T_L \cup T_S)|} \geq 2^{-\log^{0.1} n}$ since $U \cap S$ is a vertex cut of $G[U]$. This gives us $|U \cap T_L| \leq 2^{\log^{0.1} n} |U \cap S|$. Thus, the sum of $|U \cap T_L|$ for all U violating (i) is at most $2^{\log^{0.1} n} k$. If (ii) is violated, then $|U \cap T_L| < 2|U \cap S|$. Thus, the sum of $|U \cap T_L|$ for all U violating (ii) is at most $2k$. We get $|\cup_{U \in \tilde{\mathcal{U}}} (T_L \cap U)| \geq 2\sqrt{\log n/2}k/5 - 2^{\log^{0.1} n} k - 2k \geq 2\sqrt{\log n/2}k/10$.

Notice that $U \cap T_L$ contains at least $0.4|U \cap T|$ nodes for $U \in \tilde{\mathcal{U}}$ according to the above properties (i),(ii). According to [Algorithm 7](#), we will include at least $2/3|T \cap U|$ many terminals in $T \cap U$ into T_{big} for every $U \in \tilde{\mathcal{U}}$, which contain at least $0.05|T \cap U|$ nodes from $T_L \cap U$. Since $|\cup_{U \in \tilde{\mathcal{U}}} (T_L \cap U)| \geq 2\sqrt{\log n/2}k/10$, we have $|T_{\text{big}} \cap L| \geq 0.05 \cdot 2\sqrt{\log n/2}k/10 > k/100$, a contradiction. $\square$

Lemma E.53. *If $|X \cap L| < k/100$ and $|T_{\text{big}} \cap L|, |T_{\text{small}} \cap L| < k/100$, then one of the following events happens.*

1. *There exists a loop [Algorithm 7](#) such that X_{low} in [Algorithm 7](#) satisfies $|X_{\text{low}} \cap S| = \tilde{O}(|X_{\text{low}} \cap L|)$ and $|T_{\tilde{X}} \cap S| = \tilde{O}(|X_{\text{low}} \cap L|)$.*
2. *There exists a loop [Algorithm 7](#) such that X' in [Algorithm 7](#) satisfies $|X' \cap S| = \tilde{O}(|X' \cap L|)$ and $|T_{\tilde{X}} \cap S| = \tilde{O}(|X' \cap L|)$.*

Proof. Let $\mathcal{U}_{\text{inside}} \subseteq \mathcal{U}_{\text{low}}^{\text{small}}$ contain all U such that $U \subseteq L$. Let $\mathcal{U}_{\text{cross}} \subseteq \mathcal{U}_{\text{low}}^{\text{small}}$ contain all U such that $U \cap (L \cup S) \neq \emptyset$ and $U \not\subseteq L$. Notice that for any $x \in X \cap L$, we have $N_{H_{\text{low}}}(x) \subseteq \mathcal{U}_{\text{inside}} \cup \mathcal{U}_{\text{cross}}$.

Claim E.54. $|\mathcal{U}_{\text{inside}}| > k \log^9 n$.

Proof. For any $U \in \mathcal{U}_{\text{small}}$, if $U \subseteq L$, then $|N_{G'}(U)| \leq |X \cap L| + |S| \leq 1.01k$ (that is because S is a vertex cut), which implies $U \in \mathcal{U}_{\text{low}}$. Also notice that for any $U \in \mathcal{U}_{\text{small}}$ with $T_L \cap U \neq \emptyset$, either $U \in \mathcal{U}_{\text{inside}}$ or $U \cap S \neq \emptyset$ where the latter case can only happen for at most k different U as they are disjoint and $|S| \leq k$. Each U contain at most 4 terminals in T , and due to [Lemma E.52](#), we have $|\cup_{U \in \mathcal{U}_{\text{small}}} (T_L \cap U)| > 2\sqrt{\log n/2}k/4$, which means terminals inside $\mathcal{U}_{\text{inside}}$ is at least $2\sqrt{\log n/2}k/4 - 4k$, and $|\mathcal{U}_{\text{inside}}| > 2\sqrt{\log n/2}k/16 - k > k \log^9 n$. $\square$

We create $O(\log n)$ buckets for elements $U \in \mathcal{U}_{\text{inside}}$ according to $|N_{G'}(U) \cap (X \cap L)|$. Let $\mathcal{U}_{\text{inside}}^j$ contain all $U \in \mathcal{U}_{\text{inside}}$ with $|N_{G'}(U) \cap (X \cap L)| \in [2^j, 2^{j+1}]$. There exists j such that $|\mathcal{U}_{\text{inside}}^j| > k \log^8 n$. Let j be such an index and write $\mathcal{U}^* = \mathcal{U}_{\text{inside}}^j$. Consider the loop in [Algorithm 7](#) when $2^i \in [2^{j-2}, 2^{j-1}]$. We write $a^* = 2^i$.

Claim E.55. *In the loop i described above, $|X_{\text{low}} \cap S| \leq 16a^*$.*

Proof. For every $U \in \mathcal{U}^*$, we have $|N_{G'}(U)| \geq |S|$ and $|N_{G'}(U) \cap (X \cap L)| \leq 8a^*$ and $N_{G'}(U) \subseteq (X \cap L) \cup (X \cap S)$. Thus, $|N_{G'}(U) \cap S| \geq |S| - 8a^*$, in other words, U has at most $8a^*$ non-edges to S . If $x \in X_{\text{low}} \cap S$, then it has at most $1000a^*$ edges to $\mathcal{U}^*$, which is at most $1000k$. The number of non-edges from x to $\mathcal{U}^*$ is at least $|\mathcal{U}^*| - 1000k \geq |\mathcal{U}^*|/2$. Thus, $|X_{\text{low}} \cap S| \leq (8a^*|\mathcal{U}^*|)/(|\mathcal{U}^*|/2) = 16a^*$. $\square$

We also need the following useful claim.

Claim E.56. $|U_{\text{cross}}| \leq 8a^*$ and $|T_{\tilde{X}}| \leq 8a^*$.

Proof. Take an arbitrary $U \in \mathcal{U}^*$, we have $|N_{G'}(U)| \geq |S|$ and $|N_{G'}(U) \cap L| \leq 8a^*$. Notice that $N_{G'}(U) \subseteq (L \cup S) \cap X$. Thus, $|N_{G'}(U) \cap S| \geq |S| - 8a^*$, which implies $S - X \leq 8a^*$. Each $U \in U_{\text{cross}}$ must contain at

least one element in S , and they are disjoint. Remember that $T_{\bar{X}} \subseteq T - X$. Therefore, $|U_{\text{cross}}| \leq 8a^*$ and $|T_{\bar{X}}| \leq 8a^*$. $\square$

Now if $|X_{\text{low}} \cap L| > 0.01a^*$, according to [Claim E.55](#) and [Claim E.56](#), we have $|X_{\text{low}} \cap S| = \tilde{O}(|X_{\text{low}} \cap L|)$ and $|T_{\bar{X}} \cap S| = \tilde{O}(|X_{\text{low}} \cap L|)$, which completes the proof. In what follows we assume $|X_{\text{low}} \cap L| \leq 0.01a^*$. Write $X_{\text{high}L} = (X - X_{\text{low}}) \cap L$ and $X_{\text{low}L} = X_{\text{low}} \cap L$.

We first define a graph on vertex set $X_{\text{high}L}$ where $x_1, x_2 \in X_{\text{high}L}$ have an edge connecting them if $N_{G'}(x_1) \cap N_{G'}(x_2) \cap \mathcal{U}_{\text{inside}} \neq \emptyset$. Suppose this graph has ℓ connected components denoted by $X_1, \dots, X_\ell$. We write $\mathcal{U}_q = N_{H_{\text{low}}}(X_q) \subseteq \mathcal{U}_{\text{inside}} \cup \mathcal{U}_{\text{cross}}$ and $\mathcal{U}_q^{\text{inside}} = \mathcal{U}_q \cap \mathcal{U}_{\text{inside}}$.

Claim E.57. *For any $q \in [\ell]$ with $|X_q| \geq a^*$, one of the loop of [Algorithm 7](#) when 2^i (which we denote by a) satisfies $2|X_q| \leq a < 4|X_q|$ (which must happen for exactly one i), there exists a partition $P \subseteq \mathcal{P}$ (returned in [Algorithm 7](#)) and a cluster $C \in P$ such that $\mathcal{U}_q^{\text{inside}} \subseteq C$. Denote such partition as P_q and such cluster as C_q .*

Proof. Remember R_x is defined in [Algorithm 7](#) for every $x \in X$. We first prove that for any $x \in X_q$, the condition described in [Lemma E.49](#) is satisfied if we let $B = X_{\text{low}L} \cup X_q \cup (X \cap S)$ and $A' = N_{H_{\text{low}}}(x) \cap \mathcal{U}_{\text{inside}}$. Notice that $A = N_{H_{\text{low}}}(x)$ and $A - A' \subseteq \mathcal{U}_{\text{cross}}$. According to [Claim E.56](#), we have $|A| - |A'| \leq 8a^*$. Remember that $x \in X_q \subseteq X_{\text{high}L}$. According to the definition of $X_{\text{high}L}$, we have $N_{H_{\text{low}}}(x) \geq 1000a^*$ (recall that the *low*, *high* is defined with respect to a^*), thus, $|A'| \geq 0.99|A|$. For any $U \in A' = N_{H_{\text{low}}}(x) \cap \mathcal{U}_{\text{inside}}$, we have $N_{G'}(U) \cap L \subseteq X_{\text{low}L} \cup X_q$ since X_q is a connected component where all neighbors of $N_{G'}(U) \cap X_{\text{high}L}$ should be in the same connected component X_q . Thus, $N_{G'}(U) \subseteq X_{\text{low}L} \cup X_q \cup (X \cap S) = B$. Besides, we have $|N_{G'}(U)| \geq |S|$, which means $|B - N_{G'}(U)| \leq |X_{\text{low}L}| + |X_q| + |S| - |S| \leq 0.01a^* + 0.5a \leq 0.51a$. Thus, all conditions are satisfied [Algorithm 7](#). The conclusion we get is: R_x satisfies $N_{H_{\text{low}}}(x) \cap \mathcal{U}_{\text{inside}} \subseteq R_x$ and $|N_{G'}(u) \triangle N_{G'}(v)| \leq 5a$ for any $u, v \in R_x$.

Write $R_q = \cup_{x \in X_q} R_x$. We will prove that (i) $\mathcal{U}_q^{\text{inside}} \subseteq R_q$, (ii) $H_{\text{connected}}[R_q]$ is connected, (iii) $|N_{G'}(u) \triangle N_{G'}(v)| \leq 15a$ for any $u, v \in R_q$. If all of them are correct, then according to [Lemma E.37](#) property (cover), there exists $C \in P \subseteq \mathcal{P}$ such that $\mathcal{U}_q^{\text{inside}} \subseteq R_q \subseteq C$ and the proof is finished.

(i) $\mathcal{U}_q^{\text{inside}} \subseteq R_q$. This follows from $N_{H_{\text{low}}}(x) \cap \mathcal{U}_{\text{inside}} \subseteq R_x$ for any $x \in X_q$ and $\mathcal{U}_q^{\text{inside}} = \mathcal{U}_q \cap \mathcal{U}_{\text{inside}}$.

(ii) $H_{\text{connected}}[R_q]$ is connected. Remember that in $H_{\text{connected}}$, there is a path on R_x for every $x \in X_q$. Recall that X_q is connected in the sense that $x, y \in X_q$ have an edge if they share a neighbor in $\mathcal{U}_{\text{inside}}$, in which case the path R_x, R_y are connected together since $N_{H_{\text{low}}}(x) \cap \mathcal{U}_{\text{inside}} \subseteq R_x$ for any $x \in X_q$. Therefore, $H_{\text{connected}}[R_q]$ (which is the union of paths R_x for $x \in X_q$) is connected.

(iii) $|N_{G'}(u) \triangle N_{G'}(v)| \leq 15a$ for any $u, v \in R_q$. For any $u \in \mathcal{U}_q^{\text{inside}}$, we have $N_{G'}(u) \subseteq X_{\text{low}L} \cup X_q \cup S$ and $|N_{G'}(u)| \geq |S|$, which implies $|N_{G'}(u) \triangle N_{G'}(v)| \leq 2|X_{\text{low}L} \cup X_q \cup S| - |S| - |S| \leq 1.02a$. For any $w \in R_q - \mathcal{U}_q^{\text{inside}}$, there exists $u \in \mathcal{U}_q^{\text{inside}}$ such that they are in the same path R_x for some $x \in X_q$ and $|N_{G'}(u) \triangle N_{G'}(w)| \leq 5a$. Thus, according to the triangle inequality for symmetric difference, we have $|N_{G'}(u) \triangle N_{G'}(v)| \leq 15a$ for any $u, v \in R_q$. $\square$

Now we have P_q and C_q for every $q \in [\ell]$ with $|X_q| \geq a^*$ and $\mathcal{U}_q^{\text{inside}} \subseteq C_q$, we have the following claim.

Claim E.58. *There exists $q \in [\ell]$ with $|X_q| \geq a^*$ and $|C_q| \geq (\log^4 n)|X_q|$ where $|\mathcal{U}_q^{\text{inside}}| < (1 - 1/2 \log^2 n)|C_q|$.*

Proof. If to the contrary, $|\mathcal{U}_q^{\text{inside}}| \geq (1 - 1/2 \log^2 n)|C_q|$ holds for any of them, then according to [Algorithm 7](#), at least $|\mathcal{U}_q^{\text{inside}}|/2 \log^2 n$ nodes in $T \cap L$ is included in T_{small} . If we can prove $\sum_{|X_q| \geq a^*} |\mathcal{U}_q^{\text{inside}}| \geq k \log^5 n$, then $\sum_{|X_q| \geq a^*, |C_q| \geq (\log^4 n)|X_q|} |\mathcal{U}_q^{\text{inside}}| \geq k \log^5 n - \sum_q (\log^4 n)|X_q| \geq k \log^5 n - 0.01k \log^4 n$, violating the fact that $|T_{\text{small}}| < 0.01k$. Remember that $|\mathcal{U}^*| \geq k \log^8 n$ where $\mathcal{U}^*$ only contain $U \in \mathcal{U}_{\text{inside}}$ with $|N_{G'}(U) \cap (X \cap L)| \geq 2a^*$. Any $U \in \mathcal{U}^*$ satisfies $N_{G'}(U) \cap (X \cap L) \subseteq X_{\text{low}L} \cup X_{\text{high}L}$ where the part in $X_{\text{low}L}$ has size at most $0.01a^*$, and the part in $X_{\text{high}L}$ must be in the same X_q for some q , implying $|X_q| \geq 1.99a^*$. Thus, $\sum_{|X_q| \geq a^*} |\mathcal{U}_q^{\text{inside}}| \geq k \log^5 n$. $\square$

Now we have $q \in [\ell]$ with $|X_q| \geq a^*$ and $|C_q| \geq (\log^4 n)|X_q|$ where $|\mathcal{U}_q^{inside}| < (1 - 1/2 \log^2 n)|C_q|$. We will prove that X' (in [Algorithm 7](#)) satisfies $|X' \cap S| = \tilde{O}(|X' \cap L|)$ in this specific loop when $P = P_q$, which finishes the proof. We first show that $X_q \subseteq X'$, i.e., [Algorithm 7](#) does not delete any vertex in X_q . This is because for any $x \in X_q$, $N_{G'}(x) \subseteq \mathcal{U}_q^{inside} \cup \mathcal{U}_{cross}$ where $|\mathcal{U}_q^{inside}| < (1 - 1/2 \log^2 n)|C_q|$ and $|\mathcal{U}_{cross}| \leq 8a^* \leq 8|X_q| \leq |C_q|/\log^4 n$. It remains to prove $|X' \cap S| = \tilde{O}(|X_q|) = O(|X' \cap L|)$. Let $u \in \mathcal{U}_q^{inside}$ be an arbitrary element. We have $|N_{G'}(u) \cap S| \geq |S| - |X_{lowL}| - |X_q| \geq |S| - 0.1a$ (remember that $|X_q| \geq a/4$ and $a \geq a^*$). Notice that for any $v \in C_q$, we have $|N_{G'}(u) \triangle N_{G'}(v)| = O(a \log n)$ according to [Lemma E.37](#) property (common-neighborhood). Thus, we have $|N_{G'}(v) \cap S| \geq |S| - O(a \log n)$ for any $v \in C_q$. For a vertex $s \in S$ to not be deleted, it needs to have at least $|C_q|/\log^3 n$ many non-edges to C_q . However, there are at most $O(a \log n)|C_q|$ many non-edges from C_q to S . Thus, the number of nodes in S that is not deleted is at most $O(a \log n)|C_q|/(|C_q|/\log^3 n) = \tilde{O}(a) = \tilde{O}(|X_q|) = O(|X' \cap L|)$. Besides, according to [Claim E.56](#), we have $|T_{\bar{X}} \cap S| = O(a^*) = O(|X' \cap L|)$, which finishes the proof. $\square$

With [Lemma E.53](#), further assume $|T_{\bar{X}} \cap L| < k/100$ (otherwise we are done), the S_1 in [Algorithm 7](#) or S_2 in [Algorithm 7](#) will be a minimum vertex cut of G : $|(X' \cup T_{\bar{X}}) \cap S| = \tilde{O}(|X' \cap L|)$ holds due to [Lemma E.53](#), $|(X' \cup T_{\bar{X}}) \cap L| \leq 0.02k$ due to $|X \cap L| < 0.01k$ and $|T_{\bar{X}} \cap L| < 0.01k$, $|T_{\bar{X}} \cap (V - L - S)| > k$ due to $|T - X| > (\log n)k$ and $|T_{\bar{X}}| \geq |T|/101$, similar for X_{low} .

Complexity. One can verify that most lines of the algorithm can be run in time $\tilde{O}(mk)$. We will explain some non-trivial parts as follows.

1. **Construction of auxiliary graphs.** G' is constructed by first deleting all vertex set in $\mathcal{U}_{big}$ from G , and then contract every vertex set in $\mathcal{U}_{small}$. H_{low} is an induced subgraph of G' .
2. **Call of Shaving.** In [Algorithm 7](#), the total running time according to [Lemma E.49](#) is $\sum_{x \in X} \sum_{U \in N_{H_{low}}(x)} |N_{G'}(U)|$. Here $|N_{G'}(U)| \leq 2k$ since $U \in \mathcal{U}_{low}^{small}$. Thus, it is at most $k \sum_{x \in X} |N_{H_{low}}(x)| = O(km)$ since the number of edges in H_{low} is at most m .
3. **Size of $H_{connected}$ and call of CNC.** $H_{connected}$ is a union of path H_x , where H_x is a path with length at most $|N_{H_{low}}(x)|$. Thus, the number of edges in $H_{connected}$ is at most m . Moreover, the maximum degree is at most $4k$ since for every $u \in \mathcal{U}_{low}^{small}$, we have $|N_{G'}(u)| \leq 2k$, where each $x \in N_{G'}(u)$ will add at most 2 edges to u in $H_{connected}$. The distance oracle $\mathcal{A}_{dist}(u, v)$ for any $u, v \in \mathcal{U}_{low}^{small}$ (which is the vertex set of $H_{connected}$) can be computed in time $O(k)$ since $|N_{G'}(u)|, |N_{G'}(v)| \leq 2k$ according to the definition of $\mathcal{U}_{low}^{small}$. Therefore, according to [Lemma E.37](#), the running time for the CNC call is at most $\tilde{O}(mk)$.
4. **Delete vertex from X' .** The size $\mathcal{P}$ is at most $O(\log n)$ according to [Lemma E.37](#), where each $P \in \mathcal{P}$ is a partition. Now we explain the total running time of [Algorithm 7](#). To implement it for a specific $P \in \mathcal{P}$, we first iterate over all $x \in X$ and $u \in N_{H_{low}}(x)$, add a count 1 to the $C \in P$ for $u \in C$, which cost $O(m)$ time. After that, we get $|N_{H_{low}}(x) \cap C|$ for every $x \in X$ and $C \in P$. For each C , we can find $N_{G'}(C)$ in time the sum of the degree of every node in C , and decide whether to delete each node or not in $O(1)$ time. Since P is a partition, it costs time $O(m)$.

E.5 Construction of Pseudorandom Objects

In this section, we discuss the construction of the crossing family ([Theorem E.5](#) and [Corollary E.6](#)) and the selectors ([Theorem E.10](#)).

E.5.1 Crossing Family

The goal of this section is to prove [Theorem E.5](#) below. Recall the definition of the crossing family from [Definition E.4](#).

Theorem E.5. *There exists a deterministic algorithm that, given two sets A, B and two integers ℓ, r with $\ell \leq |A|$, $r \leq |B|$, and $\ell \leq r$, outputs an (A, B, ℓ, r) -crossing family in $\tilde{O}(|A| \cdot \max\{1, \frac{|B|-r}{\ell}\})$ time such that the degree of every element in A is at most $\tilde{O}(\max\{1, \frac{|B|-r}{\ell}\})$.*

The degree of an element $x \in A$ in a crossing family, $\mathcal{P}$ is $|\{y \in B \mid (x, y) \in \mathcal{P}\}|$.

The Crossing Family uses an explicit construction of a pseudorandom object named disperser defined below, and the theorem regarding the explicit construction of disperser follows.

Definition E.59 ((k, ε) -Disperser [TUZ01]). A bipartite graph $G = (V, W, E)$ is a (k, ε) -disperser if every subset $A \subseteq V$ of size at least k has at least $(1 - \varepsilon)|W|$ distinct neighbors in W . A disperser is explicit if, for every vertex $v \in V$ and i , the i^{th} neighbor of v can be computed in $\tilde{O}(1)$ time.

Theorem E.60 (Paraphrase of Theorem 1.4 from [TUZ01]). *For every integer n and $k < n$ that are powers of 2, and a constant $\varepsilon \in (0, 1)$, there is an explicit (k, ε) -disperser $G = (V = [n], W, E)$ with left-degree $d = (\log n)^{c_{E.60}}$ and we have $|W| \in [c_{\text{low}}kd / \log^3 n, c_{\text{high}}kd]$ for some universal constants $c_{E.60}, c_{\text{low}}, c_{\text{high}}$.*

We extend the construction of disperser in Theorem E.60 to Lemma E.61 so that, the inputs n, k can be arbitrary integers and not just powers of 2 and the left-degree of the disperser can be amplified arbitrarily to any given $d > (\log n)^{c_{E.60}}$.

Lemma E.61. *Given three positive integers n, k, d with $k < n$, $0 < d < n$ and a constant $\varepsilon \in (0, 1)$, we can construct an explicit (k, ε) -disperser $D = (V, W, E)$ with $|V| = n$ and $c_{\text{low}} \frac{k \cdot d'}{\log^3 n} \leq |W| \leq c_{\text{high}} k \cdot d'$ where $d' = \Theta(\max(d, (\log n)^{c_{E.60}}))$ is the left-degree of the graph G .*

The proof of Lemma E.61 is mostly by adjusting parameters appropriately and is deferred to the Section E.7. Given Lemma E.61, we first prove a relaxed version of Theorem E.5 below.

Lemma E.62. *There exists a deterministic algorithm that, given two finite sets A, B and two integers l, r with $l \leq |A|$, $r \leq |B|$ and $l \leq r$ outputs an (A, B, l, r) -crossing family in $\tilde{O}(|A| \cdot |B|/l)$ time. The degree of every element in A is at most $\tilde{O}(|B|/l)$.*

Proof. (Construction) We first construct two dispersers. Let $D_1 = (V_1 = B, W_1, E_1)$ be the disperser constructed using Lemma E.61 with $n = |B|, k = r, d = (\log n)^{c_{E.60}}, \varepsilon = 1/8$ as inputs and $D_2 = (V_2 = A, W_2, E_2)$ be the disperser constructed using Lemma E.61 with $n = |A|, k = l, d = (r/l) \cdot (\log n)^{c_{E.60}}, \varepsilon = 1/8$ as inputs.

By Lemma E.61 we have the guarantee that $|W_1| \in [r \cdot (\log n)^{c_{E.60}-3}, r \cdot (\log n)^{c_{E.60}}]$ since $d' = (\log n)^{c_{E.60}}$. Similarly, we have $d' = (r/l) \cdot (\log n)^{c_{E.60}}$ for D_2 as $r \geq l$ thus $|W_2| \in [r \cdot (\log n)^{c_{E.60}-3}, r \cdot (\log n)^{c_{E.60}}]$ as $kd' = l \cdot (r/l) \cdot (\log n)^{c_{E.60}} = r \cdot (\log n)^{c_{E.60}}$.

Note that although the sizes of W_1, W_2 are in the same range they need not be equal. Assume $|W_1| \geq |W_2|$ (other case is similar) and $\beta = \lfloor |W_1|/|W_2| \rfloor \in [1, \log^3 n]$. Each vertex in W_2 is duplicated β times and then choose $|W_1| - \beta|W_2|$ arbitrary vertices in W_2 and duplicate one more time to reach the size $|W_1|$ exactly. Hence every vertex is duplicated at least β times and at most $\beta + 1$ times. Since the set with duplicates of W_2 has size exactly $|W_1|$, we can arbitrarily map both sets and without loss of generality assume they are same say W . We define $D'_2 = (V_2 = A, W, E'_2)$ where E'_2 is obtained by duplicating incident edges when duplicating vertices in W_2 .

Any subset of size at least l in D_2 has at least $(1 - \varepsilon)|W_2|$ neighbors in W_2 , so the number of non-neighbors are at most $\varepsilon|W_2|$. So the number of non-neighbors of the same set of size at least l in D'_2 are at most $\varepsilon(\beta + 1)|W_2|$ and the fraction of non-neighbors in the W is $\varepsilon(\beta + 1)|W_2|/|W| \leq \varepsilon(1 + 1/\beta) \leq 2\varepsilon$. The first inequality follows as $|W| \geq \beta|W_2|$, second inequality follows due to $\beta \geq 1$. Thus D'_2 is $(l, 2\varepsilon)$ -disperser.

We now bound the right-degree of D_1 . The total number of edges in D_1 is $|B| \cdot (\log n)^{c_{E.60}}$. The average degree of the right vertices is $|B| \cdot (\log n)^{c_{E.60}}/|W|$. Hence, by an averaging argument there cannot be more than $|W|/4$ many vertices of degree greater $4|B| \cdot (\log n)^{c_{E.60}}/|W|$. Therefore, at least $3|W|/4$ many vertices in W have right-degree at most $4|B| \cdot (\log n)^{c_{E.60}}/|W| \leq \tilde{O}(|B|/r)$ we call this set W_{small} .

Let $N_D(v)$ be the set of neighbors of vertex v in graph D . We extend this definition to sets where $N_D(S)$

is the union of set of neighbors of vertices present in S . Recall $D_1 = (B, W, E_1)$ and $D'_2 = (A, W, E'_2)$. We define the (A, B, l, r) -crossing family $\mathcal{P}$ as

$$\mathcal{P} = \{(u, v) \mid u \in A, v \in N_{D_1}(N_{D'_2}(u) \cap W_{\text{small}})\}$$

(Running time and degree bound) For each vertex $u \in A$ we have $|N_{D'_2}(u)| \leq r(\beta+1)(\log n)^{(c_{E.60})}/l$. Since we are only considering neighbors of u that are also in W_{small} , the size of neighborhood of $N_{D'_2}(u) \cap W_{\text{small}}$ in D_1 is $|N_{D_1}(N_{D'_2}(u) \cap W_{\text{small}})| \leq \frac{r(\beta+1)(\log n)^{(c_{E.60})}}{l} \cdot \tilde{O}(|B|/r) = \tilde{O}(|B|/l)$ which is the out-degree of $u \in A$ in $\mathcal{P}$. Since each edge takes $\tilde{O}(1)$ time to compute, it takes $\tilde{O}(|A| \cdot |B|/l)$ time to compute $\mathcal{P}$.

(Correctness) For all $L \subset A$, $R \subset B$ which are of size at least l, r respectively, we need to show that $\mathcal{P} \cap (L \times R) \neq \emptyset$. Since D'_2 is a $(l, 2\varepsilon)$ -disperser we have $|N_{D'_2}(L)| \geq (1 - 2\varepsilon)|W| = 3|W|/4$ as $\varepsilon = 1/8$. Similarly, $|N_{D_1}(R)| \geq (1 - \varepsilon)|W| = 7|W|/8$ as D_1 is a (r, ε) -disperser. Since $|W_{\text{small}}| \geq 3|W|/4$, we have $N_{D'_2}(L) \cap W_{\text{small}} \cap N_{D_1}(R) \neq \emptyset$ and let w be any vertex in this intersection. We conclude $\mathcal{P} \cap (L \times R) \neq \emptyset$ because $w \in N_{D'_2}(u) \cap W_{\text{small}}$ for some $u \in L$ which also has an edge to some vertex $v \in R$ as $w \in N_{D_1}(R)$ which concludes $(u, v) \in \mathcal{P}$. $\square$

We can extend the lemma above to get a better bound when $r \geq |B|/2$ using a simple trick and prove [Theorem E.5](#).

Proof of Theorem E.5. If $r \leq |B|/2$, we know that $|B| - r \geq |B|/2$ so we apply [Lemma E.62](#) with A, B, l, r as inputs to get crossing family $\mathcal{P}$ of degree at most $\tilde{O}(|B|/l) \leq \tilde{O}((|B| - r)/l)$.

If $r > |B|/2$ let $B' \subset B$ be an arbitrary subset of size exactly $2(|B| - r)$. Let $r' = |B| - r$ and $l' = \min(|B| - r, l)$, we have $r' \geq l'$, thus we can apply the [Lemma E.62](#) with A, B', l', r' as inputs to get crossing family $\mathcal{P}$ of degree at most $\tilde{O}((|B| - r)/l') = \max(1, (|B| - r)/l)$ as required.

We will now prove that $\mathcal{P}$ is also an A, B, l, r crossing family. For any subset $L \subset A, R \subset B$ of size at least l, r respectively, let $R' = R \cap B'$. As $R \geq r$ we have the guarantee that $|R'| = |R \cap B'| \geq |B| - r = r'$. We also have $|L| \geq l \geq l'$. Thus from [Lemma E.62](#) we have $\mathcal{P} \cap (L \times R') \neq \emptyset$ which implies $\mathcal{P} \cap (L \times R) \neq \emptyset$ as $R' \subset R$. $\square$

Given [Theorem E.5](#), [Corollary E.6](#) follow immediately.

Corollary E.6 (Symmetric crossing family). *There exists a deterministic algorithm that, given an integer n and a parameter $\alpha \geq 1$, outputs $\mathcal{P}$ of size $\tilde{O}(\alpha n)$, a subset of $[n] \times [n]$ in time $\tilde{O}(\alpha n)$, such that for any partition (L, S, R) of $[n]$ that satisfies $|R| \geq |L| \geq |S|/\alpha$, we have $\mathcal{P} \cap (L \times R) \neq \emptyset$. Moreover, for each $x \in [n]$, the out-degree of x in $\mathcal{P}$, i.e., $|\{y \mid (x, y) \in \mathcal{P}\}|$, is at most $\tilde{O}(\alpha)$.*

Proof. Algorithm: Guess the size of $|L|, |R|$ by powers of 2, denoted by l, r . For each guess l, r satisfying (1) $l \leq r$, (2) $l + r < n$, and (3) $(n - r)/l \leq \alpha + 1$, we run [Theorem E.5](#) with parameters $A = V, B = V$ and the corresponding l, r and take the union of all of those crossing families as $\mathcal{P}$.

Running time and degree bound: There can be at most $\log^2 n$ such crossing families, each taking at most $\tilde{O}(|A| \cdot (|B| - r)/l) = \tilde{O}(n(n - r)/l) = \tilde{O}(n\alpha)$ time to construct. The total running time is still $\tilde{O}(n\alpha)$. The out-degree of every vertex in each crossing family is bounded by $\tilde{O}((n - r)/l) = \tilde{O}(\alpha)$ based on condition (3). Hence the out-degree of every element in A is at most $\log^2 n \cdot \tilde{O}(\alpha) = \tilde{O}(\alpha)$.

Correctness: For every partition (L, S, R) of $[n]$ that satisfies $|R| \geq |L| \geq |S|/\alpha$ let l, r be the highest powers of 2 that are at most $|L|, |R|$. It is easy to verify that all the above three conditions are satisfied. Thus the crossing family constructed with this l, r as inputs has non-empty intersection with $L \times R$, so does the union $\mathcal{P}$. $\square$

E.5.2 Selectors

We mainly prove [Theorem E.10](#) in this sub-section, which follows from [Theorem E.63](#) stated below. Recall the definition of the selector.

Definition E.9 ((n, k, ϵ) -selector). For every positive integers n, k with $k < n$ and parameters $\epsilon \in [0, 1]$, an (n, k, ϵ) -selector $\mathcal{S}$ is a family of subsets of $[n]$ such that, for every two disjoint subsets L and S of $[n]$ with sizes satisfying $\epsilon k < |L| \leq k$ and $|S| \leq k$, there exists a set $U \in \mathcal{S}$ such that $|U \cap L| = 1$ and $|U \cap S| = 0$.

Let $T \in \mathcal{S}$ be the set such that $T \cap L = \{x\}$ and $T \cap S = \emptyset$ then we say that T selects x from L avoiding S .

Theorem E.63. *There exists an algorithm that, given integers n, k and parameters $\epsilon \in [0, 1], \alpha \in (1/n, 1)$ where $k \leq (\frac{n}{2})^{\frac{1}{1+\alpha}} \cdot (\frac{\epsilon}{\log^2 n})^{\frac{2}{\alpha}}$, return a (n, k, ϵ) -selector $\mathcal{S}$ of size at most $|\mathcal{S}| \leq k^{1+\alpha} \cdot (\frac{\log n \log k}{\epsilon})^{3+3/\alpha}$ in $\tilde{O}(n) \cdot (\frac{\log n \log k}{\epsilon})^{1+1/\alpha}$ running time and every set $T \in \mathcal{S}$ has size at least $n/(k^{1+\alpha} \cdot (\frac{\log n \log k}{\epsilon})^{2+2/\alpha})$.*

We give the proof of [Theorem E.10](#) (recalled below) assuming [Theorem E.63](#).

Theorem E.10. *For any integer n and parameter $\epsilon \geq 1/n^{o(1)}$, there exists $\tilde{k} \geq n^{1-o(1)}$ depending on ϵ such that, given any $k \leq \tilde{k}$, we can compute a (n, k, ϵ) -selector $\mathcal{S}$ of size at most $|\mathcal{S}| = \hat{O}(k)$ in $\hat{O}(n)$ time, and every set $U \in \mathcal{S}$ has size at least 2.*

Proof. Let $\psi(n) = \log(1/\epsilon)/\log n = o(1)$ as $\epsilon \geq 1/n^{o(1)}$ and $\alpha = \max(\sqrt{\psi(n)}, 1/\sqrt{\log n})$. Let $\tilde{k} = n^{1-f(n)} \geq n^{1-o(1)}$ with $f(n) = O(\alpha + \frac{\psi(n)}{\alpha} + \frac{\log \log n}{\alpha \log n}) = o(1)$ based on our choice of parameters. Our choice of $f(n)$ satisfies the premise of [Theorem E.63](#) that is $\tilde{k} = n^{1-f(n)} \leq (\frac{n}{2})^{\frac{1}{1+\alpha}} \cdot (\frac{\epsilon}{\log^2 n})^{\frac{2}{\alpha}}$ as $k \leq \tilde{k}$. Apply [Theorem E.63](#) with n, k, ϵ, α as input and return the output (n, k, ϵ) -selector $\mathcal{S}$.

Based on our choice of parameters we have

$$\begin{aligned} \left(\frac{\log n \log k}{\epsilon}\right)^{1+1/\alpha} &\leq \left(\frac{\log^2 n}{\epsilon}\right)^{2/\alpha} \\ &\leq n^{(\psi(n) + \frac{2 \log \log n}{\log n})(2/\alpha)} \\ &\leq n^{o(1)} \end{aligned} \quad (\alpha = \max(\sqrt{\psi(n)}, 1/\sqrt{\log n})) \quad (1)$$

From [Theorem E.63](#) we have

$$|\mathcal{S}| \leq k^{1+\alpha} \cdot \left(\frac{\log n \log k}{\epsilon}\right)^{3+3/\alpha} \leq \frac{n}{2} \cdot \left(\frac{\epsilon}{\log^2 n}\right)^{1+1/\alpha} \cdot \left(\frac{\log n \log k}{\epsilon}\right)^{3+3/\alpha} \leq k \cdot n^{f(n)} \cdot n^{o(1)} \leq k \cdot n^{o(1)},$$

the second inequality follows from premise, the third inequality follows as $k = n^{1-f(n)}$ and [Equation \(1\)](#). The time taken is $\tilde{O}(n) \cdot (\frac{\log n \log k}{\epsilon})^{1+1/\alpha} \leq n^{1+o(1)}$. The size of every set $T \in \mathcal{S}$ is at least $n/m \geq n/(k^{1+\alpha} \cdot (\frac{\log n \log k}{\epsilon})^{2+2/\alpha}) \geq 2$ as $k \leq \tilde{k} \leq (\frac{n}{2})^{\frac{1}{1+\alpha}} \cdot (\frac{\epsilon}{\log^2 n})^{\frac{2}{\alpha}}$. $\square$

We construct the selector family from a lossless expander, which is defined as follows:

Definition E.64 ((k, α) -lossless expander). A d -left regular bipartite graph $G = (V_L, V_R, \Gamma)$ with left vertex set V_L , right vertex set V_R and the neighborhood function $\Gamma : V_L \times [d] \rightarrow V_R$ where $\Gamma_i(v) := \Gamma(v, i)$ for $i \in [d], v \in V_L$ denotes the i^{th} neighbor of vertex v ; is a (k, α) -lossless expander for some $k < |V_L|$ and $\alpha \in [0, 1]$, if every subset S of V_L of size at most k has the size of its neighborhood $|N_G(S)| \geq \alpha |S|$.

$\Gamma_i^{-1}(r)$ denotes the set of all left vertices whose i^{th} neighbor is r for some $i \in [d], r \in [m]$. A lossless expander is said to be *explicit* if $\Gamma_i(v)$ can be computed in $\text{polylog}(|V_L| + |V_R|)$ time for any i, v .

[\[Che11\]](#) shows how to construct an explicit lossless expander stated below.

Lemma E.65 (Corollary 2.23 from [\[Che11\]¹⁴](#)). *Let $\alpha > 1/\text{poly}(n)$ be an arbitrary number¹⁵. Then, for parameters $N \in \mathbb{N}$, $K \leq N$, and $\epsilon > 0$, there is an explicit 2^D -left regular $(2^K, (1 - \epsilon))$ -lossless expander $G = (\{0, 1\}^N, \{0, 1\}^{D+M}, \Gamma)$ with $D \leq (1 + 1/\alpha)(\log(NK/\epsilon) + O(1))$ and $M \leq 2D + (1 + \alpha)K$ ¹⁶. Moreover, Γ_i is a linear function (over $\mathbb{F}_2$) for every fixed choice of $i \in \{0, 1\}^D$.*

¹⁴The original lemma in [\[Che11\]](#) constructs a lossless condenser, however, according to Lemma 2.2.1 of [\[TUZ01\]](#) lossless condenser is equivalent to a lossless expander. The original version also states the lemma over any field of fixed prime power p , however we set $p = 2$ as it is enough for our purpose.

¹⁵Although the original version of the lemma states α to be constant, lemma still holds for any $\alpha > 1/\text{poly}(n)$

¹⁶One additive factor of D comes from the condenser construction of [\[Che11\]](#), the other D comes from the equivalence of condenser and expander in [\[TUZ01\]](#)

The neighborhood function of lossless expander Γ in [Lemma E.65](#) is constructed based on evaluations of polynomials in $\mathbb{F}_p$ for prime p . We state the following property of linear functions, which is required to prove a lower bound on the size of the sets in the selector family.

Proposition E.66. *For some prime number p , let $f : \mathbb{F}_p^a \rightarrow \mathbb{F}_p^b$ be a linear function for some integers a, b with $a \geq b$ in the field $\mathbb{F}_p$. For $r \in \mathbb{F}_p^b$, let $f^{-1}(r)$ be the set of all vectors in $\mathbb{F}_p^a$ that map to r , then every nonempty set $f^{-1}(r)$, has size at least p^{a-b} .*

This proposition follows from the property that $f(x+y) = f(x) + f(y)$ for any $x, y \in \mathbb{F}_p^a$ where operations are performed in the field $\mathbb{F}_p$.

We conclude the following theorem using the [Lemma E.65](#) and [Proposition E.66](#).

Lemma E.67. *There exists an algorithm given $n, k, \varepsilon, \alpha$ as inputs with $k \leq O(\frac{n\varepsilon}{\log^2 n})^{1/(1+\alpha)}$ satisfying $n = 2^N$ for some $N \in \mathbb{N}$, $\varepsilon \in (0, 1)$ and $\alpha \in (1/n, 1)$, constructs a d -left regular bi-partite graph $G = ([n], [m], E)$ is a $(k, 1 - \varepsilon)$ -lossless expander in $\tilde{O}(nd)$ time. We have $d \leq O(\frac{\log n \cdot \log k}{\varepsilon})^{1+\frac{1}{\alpha}}$ and $m \leq d^2 \cdot k^{1+\alpha}$. Let Γ be the neighborhood function of G , we have the guarantee that every non-empty set $|\Gamma_i^{-1}(r)|$ has size at least n/m for $i \in [d], r \in [m]$.*

Proof. Let's provide $N, K(= \log k), \varepsilon, \alpha$ as inputs to [Lemma E.65](#), the output G is a $(k, (1 - \varepsilon))$ -lossless expander.

From [Lemma E.65](#) we have the following bound on the size of the right vertices $m = p^M \leq 2^{2D} \cdot 2^{K(1+\alpha)} = d^2 \cdot k^{1+\alpha}$ and the left degree $d = 2^D \leq O(\frac{\log n \cdot \log k}{\varepsilon})^{1+\frac{1}{\alpha}}$. As mentioned in [Lemma E.65](#), it takes $\tilde{O}(1)$ time to compute each edge. Since there are at most nd edges, it takes $\tilde{O}(nd)$ time.

For any fixed $i \in [d]$, from the construction in [Lemma E.65](#) we have Γ_i is a linear function that maps $\mathbb{F}_2^N$ to $\mathbb{F}_2^M$. Thus, for every right vertex $r \in [m], i \in [d]$ such that $\Gamma_i^{-1}(r) \neq \emptyset$ has size at least $2^{N-M} = n/m$ according to [Proposition E.66](#). $\square$

We define another type of expander called unique neighbor expander with guarantees that follow from the lossless expander and are enough for proving [Theorem E.63](#).

Definition E.68 ((k, α) -unique neighbor expander). A d -left regular bipartite graph $G = (V_L, V_R, E)$ is a (k, α) -unique neighbor expander for some $k < |V_L|$ and $\alpha \in [0, 1]$, if every subset S of V_L of size at most k has at least $\alpha d|S|$ many vertices in its neighborhood $N_G(S)$ which have unique neighbor in S .

Proposition E.69. *A bipartite graph $G = (V_L \cup V_R, E)$ that is a $(k, (1 - \varepsilon))$ -lossless expander is also a $(k, (1 - 2\varepsilon))$ -unique neighbor expander.*

This proposition follows from the definitions. We are now ready to prove [Theorem E.63](#).

Proof of Theorem E.63. Given integers n, k as inputs let $n' \in [n, 2n]$ be a power of 2. We duplicate arbitrary $n' - n$ many elements from $[n]$ and add them to $[n]$ so the size is exactly n' . Let $\Gamma : [n'] \times [d] \rightarrow [m]$ be the neighborhood function of lossless expander $G = (V_L \cup V_R, E)$ obtained by giving $n', k' = 4k, \varepsilon' = \varepsilon/8, \alpha$ as inputs to [Lemma E.67](#). We have $|V_L| = n'$ and let $|V_R| = m$ and d is the left-degree of G . For each $i \in [d], x \in \mathbb{F}_2^{\log n'}$, let $\Gamma_i(x) := \Gamma(x, i)$. We construct the selector family as follows

$$\mathcal{S} := \{\Gamma_i^{-1}(j) : i \in [d], j \in [m]\}.$$

From [Lemma E.67](#) we know that $d \leq O(\frac{\log n' \cdot \log k'}{\varepsilon'})^{1+\frac{1}{\alpha}} \leq O(\frac{\log n \cdot \log k}{\varepsilon})^{1+\frac{1}{\alpha}}$ and $m \leq d^2(k')^{1+\alpha} \leq O(d^2 k^{1+\alpha})$. So the size of the family is at most $dm \leq O(k^{1+\alpha} d^3) \leq O(k^{1+\alpha} \cdot (\frac{\log n \cdot \log k}{\varepsilon})^{3+3/\alpha})$ and it takes $\tilde{O}(nd) \leq \tilde{O}(n) \cdot (\frac{\log n \cdot \log k}{\varepsilon})^{1+1/\alpha}$. We also have that each non-empty set $\Gamma_i^{-1}(j)$ has size at least $n/m \geq O(n/(k^{1+\alpha} \cdot (\frac{\log n \cdot \log k}{\varepsilon})^{2+2/\alpha}))$ based on the bound of m .

From [Proposition E.69](#) we have the guarantee that G is also a $(k', (1 - 2\varepsilon'))$ -unique neighbor expander. For any set $A \subset V_L$ of size at most k' , we have the guarantee that there are at least $(1 - 2\varepsilon')d|A|$ vertices in V_R that have unique neighbors in A . There are at least $(1 - 2\varepsilon')$ fraction of vertices in A that are unique neighbors to some vertex in V_R . Otherwise, the number of vertices in the neighborhood of A that have

unique neighbors is strictly less than $(1 - 2\varepsilon')d|A|$, as each left vertex can have at most d neighbors, leading to contradiction.

Let $v \in A$ be the vertex that is the unique neighbor of some vertex r in V_R . There exists an $i \in [d]$ such that $\Gamma(v, i) = r$. Then we have the guarantee that $A \cap \Gamma_i^{-1}(r) = \{v\}$. Hence, at least $(1 - 2\varepsilon')$ fraction of vertices in A can be *selected* by some set in $\mathcal{S}$.

We now prove that the $\mathcal{S}$ is a (n, k, ε) -selector. Let $L, S \subset [n]$ be two disjoint sets of sizes $\varepsilon k < |L| \leq k$ and $S \leq k$. Let $A \subset V_L$ be the set of vertices corresponding to L, S along with their duplicates if they exist. We consider that an element is selected even if its duplicate is selected. We have $|A| \leq 2|L \cup S| \leq 4k = k'$. Since less than $2\varepsilon'|A| \leq \varepsilon k$ cannot be selected by $\mathcal{S}$ and the size of L is greater than εk , there exists at least one vertex in L that can be selected by $\mathcal{S}$, avoiding all other vertices in $A = L \cup S$. Hence, $\mathcal{S}$ is (n, k, ε) -selector family. $\square$

E.6 Exposition of Gabow's Algorithm

Through this section, we aim to provide an exposition of Gabow's algorithm for finding the minimum vertex cut in light of recent developments in maxflow literature. We prove the following theorem.

Theorem E.70. *There exists an algorithm that takes an undirected graph $G = (V, E)$ and a cut parameter k as inputs and outputs either a vertex cut of size less than k or guarantees that the graph is k -vertex connected in $\tilde{O}(n + k\sqrt{n})$ maxflows on graphs of size at most m where $|V| = n, |E| = m$.*

The main idea of Gabow's algorithm is to find a pair of vertices (s, t) such that $s \in L, t \in R$ so that applying a unit-vertex-capacity maxflow across the pair would return the separator S of size less than k . Gabow uses the Ramanujan expander to find such a pair.

E.6.1 Ramanujan Expander

We start with defining the Ramanujan expander. Let A denote the adjacency matrix of the graph G , and D represent the diagonal matrix with the i -th diagonal entry as $\deg(v_i)$, where v_i is the i -th vertex of the graph G . The eigenvalues of the matrix AD^{-1} are denoted by $\lambda_1(G), \lambda_2(G), \dots, \lambda_n(G)$, sorted from large to small. Notably, AD^{-1} is often called the column-wise normalized adjacency matrix or the random walk matrix. Alon and Boppana famously showed that $\lambda_2(G) \geq 2\sqrt{\frac{d-1}{d}} - o_n(1)$ for any d -regular graph G of n vertices. The graphs that (almost) achieve this lower bound are called the Ramanujan expanders.

Definition E.71 (Ramanujan Expander). A d -regular graph $X = (V_X, E_X)$ is a Ramanujan expander if $\lambda_2(X) \leq 2\sqrt{d-1}/d$.

We use the following explicit construction of the Ramanujan graphs from [LPS86; Mar88].

Lemma E.72 (Explicit construction of Ramanujan Expander). *Given integers n and $d \in \mathbb{N}$, there exists a d' -regular Ramanujan expander $X = (V_X, E_X)$ with $|V_X| \leq c_{E.72}n$ for some universal constant $c_{E.72}$, where $d \leq d' \leq 4d$. This expander can be constructed in $O(nd)$ time.*

The critical property of the Ramanujan graph that is exploited in Gabow's algorithm is that any two large disjoint subsets of the graph are connected by an edge across them. This is implied by the *expander mixing lemma* (see, e.g., [Vad12]) stated below.

Lemma E.73 (Expander mixing lemma). *Let $G = (V, E)$ be any d -regular graph. For all $S, T \subseteq V$ we have,*

$$\left| |E(S, T)| - \frac{|S| \cdot |T| \cdot d}{|V|} \right| \leq \lambda_2(G) \cdot d \cdot \sqrt{|S| \cdot |T|}$$

Intuitively, the expander mixing lemma states that for any d -regular graph with small λ_2 value behaves close to a random graph, i.e., the number of edges between any two subsets (S, T) is close to the "expected"

number of edges between them in a random graph with $p = d/n$. The expander mixing lemma implies the following corollary for Ramanujan expanders whose proof is left as an exercise to the readers.

Corollary E.74. *Let $X = (V_X, E_X)$ be a Ramanujan expander. For any $L, R \subseteq V_X$, if $|L| \cdot |R| \cdot d' \geq 4|V_X|^2$, then $E_X(L, R) \neq \emptyset$.*

The technical issue of [Lemma E.72](#) is that, given a number n , it does not give a graph with precisely n vertices. Below, we show that this problem can be easily fixed by taking an induced subgraph of the Ramanujan graph of the required size n . Since any two large sets in the sub-graph also belong to the original Ramanujan graph and the edge between them is retained as we take the induced sub-graph.

Lemma E.75 (Two large sets are connected). *For an integer n and a degree parameter d , we can construct a graph $H = (V_H, E_H)$ with n vertices and a maximum degree of $4d$ in $O(nd)$ time such that, for any $A, B \subseteq V_H$, if $|A| \cdot |B| \cdot d \geq 4c_{E.72}^2 n^2$, then $E_H(A, B) \neq \emptyset$ for some universal constant $c_{E.72}$.*

Proof of Lemma E.75. Given integers n, d apply [Lemma E.72](#) and construct d' -regular Ramanujan expander $X = (V_X, E_X)$ in time $O(nd)$. We construct $H = (V_H, E_H)$ from X as follows. Let V_H be any arbitrary subset of V_X of size n and $E_H = E_X(V_H, V_H)$ which is the subset of edges E_X whose both endpoints are in V_H . Hence, it takes at most $O(nd)$ total time to construct the graph H . Maximum degree of H is at most $d' \leq 4d$ from [Lemma E.72](#).

For any $L, R \subseteq V_H \subseteq V_X$, if we have

$$|L| \cdot |R| \cdot d' \geq |L| \cdot |R| \cdot d \geq 4c_{E.72}^2 n^2 \geq 4|V_X|^2$$

as $d' \geq d$, $|L||R|d \geq 4c_{E.72}^2 n^2$ and $|V_X| \leq c_{E.72} n$ respectively. From [Corollary E.74](#) we have guarantee that $E_X(L, R) \neq \emptyset$, hence $E_H(L, R) \neq \emptyset$. $\square$

E.6.2 Overview of Gabow's algorithm

For any mincut (L, S, R) the key observation is that both L and R contain at least $\delta_G - \kappa_G$ many vertices, where δ_G is the minimum degree of a vertex in the graph G , and κ_G is the vertex connectivity of the graph G . We define the difference between $\delta_G - \kappa_G$ as the gap γ_G of the graph G . We omit the subscript G when it is clear from the given context.

In the following lemma, we demonstrate how to leverage the gap and find a cut smaller than k for any given parameter k .

Lemma E.76. *Given an undirected graph $G = (V, E)$ with a gap γ and a cut parameter k , there exists an algorithm that either outputs a cut of size less than k or declares that the graph is k -connected in $O(n + \frac{\delta^2}{\gamma})$ maxflows on graphs of size at most $|E|$.*

From the above lemma, it is clear that the graph should have a large gap γ for a smaller runtime. We can modify the graph G to construct a new graph G' with a large gap. We then use [Lemma E.76](#) on G' to find a cut of size less than k' , say S' , and then construct the separator S in G from the separator S' . Since any general graph need not have a large gap, we create a new one by modifying the original one, as proven in [Lemma E.77](#).

Lemma E.77. *Given an undirected graph $G = (V, E)$, there exists an algorithm that constructs a graph $H = (V_H, E_H)$ of size at most $|E|$ with a gap γ , a vertex cut (L, S, R) of the graph G in $O(\gamma n)$ maxflows. If $\kappa_G < k$, we also have the guarantee that $|S| < k$, or $V \setminus V_H$ is contained in every mincut of G .*

Given these two lemmas, let us first prove [Theorem E.70](#).

Proof of Theorem E.70. Using [\[NI92\]](#), we can reduce the number of edges in the graph to $O(nk)$ without affecting the vertex mincut if a mincut of size less than k exists. Thus, $\delta = O(k)$.

If $k < \sqrt{n}$, then $\delta = O(\sqrt{n})$, and we can directly apply [Lemma E.76](#). The number of maxflows is $O(n + \delta^2/\gamma) = O(n)$ and is dominated by $O(n + k\sqrt{n})$, so we are done. Hence, $k \geq \sqrt{n}$.

We can assume $\delta \geq \sqrt{n}$ as otherwise $\delta < \sqrt{n} \leq k$, and we return the minimum degree cut. Given a graph G , we use [Lemma E.77](#) to increase the gap to $\delta/\sqrt{n}$, which outputs H , a cut (L, S, R) and takes $O(\delta\sqrt{n})$ maxflows. If $|S| < k$, then we are done.

Otherwise, we apply [Lemma E.76](#) with H and $k' = k - |V \setminus V_H|$ as inputs to either find a cut (L', S', R') of H of size less than k' or declare that $\kappa_H \geq k'$. If $|S'| < k'$, then $(L', S' \cup (V \setminus V_H), R')$ is a valid vertex cut of G of size less than k . If $\kappa_H \geq k'$ implies $\kappa_G \geq k$, because if $\kappa_G < k$, from [Lemma E.77](#), we have the guarantee that $|V \setminus V_H|$ is contained in every mincut of G , which implies $\kappa_H < k'$, and we arrive at a contradiction.

This takes $O(n + \delta^2/\gamma) \leq O(n + \delta\sqrt{n})$ maxflows as $\gamma \geq \delta/\sqrt{n}$. Note that the number of edges in H is at most that of G . Hence, the runtime is at most $O(n + \delta\sqrt{n})$ maxflows on graphs of size at most G . Since $\delta = O(k)$, the runtime is at most $O(n + k\sqrt{n})$ maxflows. $\square$

We now prove [Lemmas E.76](#) and [E.77](#), starting with [Lemma E.76](#).

E.6.3 Exploiting a Large Gap

The critical idea in proving [Lemma E.76](#) is to use Ramanujan expanders to find a pair of vertices (s, t) that lies across a vertex mincut (L, S, R) , i.e., $s \in L, t \in R$. Below, we recall the lemma regarding Ramanujan expanders in [Section E.5.1](#).

Let H be the Ramanujan expander constructed with $|V_H|, d$ as inputs to the above lemma for any subset V_H of V . From the above lemma, if L, R of a mincut (L, S, R) of G have a large intersection with V_H , i.e., $A = L \cap V_H, B = R \cap V_H$, then the degree d needed to satisfy the condition in [Lemma E.75](#) is small, thus optimizing the size of the Ramanujan graph. Thus, we need to find subsets V_H with large $|A|, |B|$, which brings us to the following definition.

Definition E.78 (ρ -rich). A set $T \subset V$ is said to be ρ -rich in a graph G for some integer parameter ρ if there exists a mincut (L, S, R) of G such that $|T \cap L| \geq \rho$ and $|T \cap R| \geq \rho$.

Using [Lemma E.75](#), we conclude the following lemma from [\[Gab06\]](#), rewritten here for completeness.

Lemma E.79 (Lemma 2.4 from [\[Gab06\]](#)). Take any $k \geq \kappa_G$ and let T be a ρ -rich set of size t . Let $X_{t,d} = (V_X, E_X)$ be the Ramanujan expander constructed with $t, d = 1 + \frac{4(c_{E.72}t)^2}{\rho\rho'}$ as inputs to [Lemma E.75](#), where $\rho' = \max(\rho, \frac{t-k}{2})$. Then

$$\kappa_G = \min\{\kappa_G(x, y) : (x, y) \in E_X\}$$

Proof. Since T is ρ -rich, let (L, S, R) be a vertex mincut such that $\alpha = |L \cap T| \geq \rho$ and $\beta = |R \cap T| \geq \rho$. Since (L, S, R) is a partition, we have $\alpha + \beta + |S| = t$. Without loss of generality, let $\alpha \geq \beta$, which implies α has a size of at least $\frac{t-|S|}{2} \geq \frac{t-k}{2}$. Thus, $\alpha\beta d \geq \rho'\rho d \geq 4(c_{E.72}t)^2$, which implies there exists an edge across $E_X(L \cap T, R \cap T) \neq \emptyset$ according to [Lemma E.75](#). Hence, the result follows. $\square$

We now see how to find a very rich set (high ρ) but small in size. Below, we show how to construct such a set. We have the following lemma for any vertex $a \in V$.

Lemma E.80. For any vertex $a \in V$, if $\kappa(a) \geq k$, then for any vertex mincut (L, S, R) , $|N_G(a) \cap L|, |N_G(a) \cap R| \geq \min(\delta, \kappa(a)) - \kappa + 1$.

Proof. Let (A, C, B) be the vertex cut corresponding to $\kappa(a)$. Assume, for contradiction, $|N_G(a) \cap L| < \min(\kappa(a), \delta) - \kappa + 1$. Let $l \in L$ be any vertex; we have $|L| + |S| \geq \delta + 1$, implying $|L| \geq \delta - \kappa + 1$. Hence, $L \setminus N_G(a) \neq \emptyset$. Thus, $(L \setminus N_G(a), (N_G(a) \cap L) \cup S \setminus \{a\}, \{a\} \cup R)$ separates a from the rest of the graph with a separator of size at most $|N_G(a) \cap L| + |S| - 1 < \min(\kappa(a), \delta)$, which is a contradiction. $\square$

According to [Lemma E.80](#), if the graph has a large gap, i.e., $\delta - \kappa \geq \tau$, and $\kappa(a) \geq k + \tau \geq \kappa + \tau$, then the neighborhood set of a is τ -rich.

We define $\kappa_W(T)$ for a set $T \subset V$ as the minimum number of vertices we need to remove, including the vertices in T , to disconnect the set T from the rest of the graph. It can be computed by computing the

rooted vertex connectivity of a new vertex t in a graph G_T constructed by joining the vertices of T to t with an edge.

Let (A, C, B) be the vertex cut corresponding to $\kappa(a)$. Since we have (A, C, B) as a vertex cut, the idea is to either use them to construct a set whose weak separation gives us the mincut or return a "small" set that is rich. We first note the following property of a mincut and how it intersects with a rooted mincut for some vertex a .

Lemma E.81. *A mincut (L, S, R) exists such that one of the following is true.*

- A contains L or R
- (or) C contains L or R
- (or) A or B is contained in S , and S is $\min(|A|, |B|)$ -rich.

Proof. Assume A and C do not contain L or R . Hence, L, R span across A, B, C . $L \cap C, R \cap C \neq \emptyset$; otherwise, $L \cap A, L \cap B$ does not have an edge across them, and $(L \cap A, S, R \cup (L \cap B))$ is a vertex mincut that satisfies the first condition.

Since L spans across A, B, C and $L \cap C \neq \emptyset$, we have either $L \cap A \neq \emptyset$ or $L \cap B \neq \emptyset$. Similarly, with R , we have $R \cap A \neq \emptyset$ or $R \cap B \neq \emptyset$.

If $L \cap A \neq \emptyset$ and $R \cap B \neq \emptyset$, then we have

$$|S \cap B| + |S \cap C| + |R \cap C| = |N_G(R \cap B)| \geq \kappa(a) = |C|$$

$$|S \cap B| \geq |C \cap L|$$

We now consider $N_G(L \cap A)$,

$$|N_G(L \cap A)| = |S \cap A| + |S \cap C| + |L \cap C| \leq |S| = \kappa$$

Thus, $(L \cap A, S, V \setminus ((L \cap A) \cup S))$ is a mincut that satisfies the first condition. Hence, at least one of $L \cap A, R \cap B$ is empty. Similarly, at least one of $L \cap B, R \cap A = \emptyset$.

Thus, if we have $L \cap A = \emptyset$, then $R \cap B \neq \emptyset$. Since $L \cap A = \emptyset$, we have $L \cap B \neq \emptyset$ as otherwise C contains L . Since $L \cap B \neq \emptyset$, we have $R \cap A = \emptyset$, thus $A \subset S$.

Any subset of L, R has its neighborhood set of size at least κ as it cannot be smaller. We have $|N_G(L \cap B)| = |L \cap C| + |S \cap C| + |S \cap B| \geq |S|$ which implies $|L \cap C| \geq |S \cap A| = |A|$. Similarly, $|R \cap C| \geq |A|$.

If we start with assuming $L \cap A \neq \emptyset$, then we will have $|L \cap C|, |R \cap C| \geq |B|$. Thus, C is $\min(|A|, |B|)$ -rich. $\square$

Given the above lemma, we are ready to define the algorithm. The algorithm takes a graph with a large gap, and either returns a τ -rich set or a cut of size less than k . The critical steps of the algorithm are as follows:

Algorithm 8: $\text{KCUTORRICHSET}(G, k)$ **Input:** A graph $G = (V, E)$ and connectivity parameter k .**Output:** Either a vertex cut of size less than k or a τ -rich set T .

```

1 if  $\delta > n/2$  then
2   return  $V$ 
3 Let  $a$  be a vertex of degree  $\delta$ .
4 Let  $(A, C, B)$  be the minimum rooted vertex separator for  $a$ .
5 if  $\kappa(a) < k$  then
6   return  $(A, C, B)$ 
7 else if  $\kappa(a) \geq k + \tau$  then
8   return  $N_G(a)$ 
9 else
10  Let  $T \subset A \cup B$  and  $|T| = k$ 
11  if  $\kappa_W(C) < k$  or  $\kappa_W(T) < k$  then
12    return Corresponding vertex cut
13  else
14    return  $C$ 

```

Lemma E.82. *There exists an algorithm that takes a graph G and a connectivity parameter k as inputs and finds either a separator of size less than k or a τ -rich set of size at most 2δ for $\tau = \frac{\delta-k}{2}$ in at most $O(n)$ maxflows.*

Proof. We know from the previous discussion that the vertex set V is $(\delta - \kappa)$ -rich, which implies it is also $(\delta - k)/2$ -rich. If the minimum degree is very high, i.e., $\delta > n/2$, we obtain a τ -rich set of size $n \leq 2\delta$.

As shown in [Algorithm 8](#), we compute the rooted vertex connectivity of a vertex a with degree δ , which gives the vertex cut (A, C, B) where C is a separator of size $\kappa(a)$. Note that any subset $\hat{B} \subset B$ has $|N_G(\hat{B})| \geq \kappa(a)$.

If $\kappa(a) < k$, we return (A, C, B) and we are done. Otherwise, if $\kappa(a) \geq k$, and it is significantly larger than k (i.e., $\kappa(a) \geq k + \tau$), then, according to [Lemma E.80](#), $N_G(a)$ is τ -rich, which is of size δ .

If $\kappa(a) < k + \tau$ and we have $\delta = k + 2\tau$, then there exist at least k vertices in $A \cup B$, and we construct a set T of size $k < 2\delta$. Note that we can increase the size of a ρ -rich set to 2δ by adding arbitrary vertices while still keeping it τ -rich.

Let (L, S, R) be a mincut of size less than k , and from [Lemma E.81](#), we have the following cases:

- If A contains L or R , in which case C spans across only S, R or L, S , the weak separator of C gives the mincut, i.e., $\kappa = \kappa_W(C)$.
- If C contains L or R , in which case $A \cup B$ spans across only S, R or L, S , the weak separator of $A \cup B$ gives the mincut. Since we are interested only in a cut smaller than k , we take k vertices from $A \cup B$ as the set T and find its weak separator $\kappa_W(T)$.
- Otherwise, C is $\min(|A|, |B|)$ -rich. Since $\delta = k + 2\tau$ and $\kappa(a) < k + \tau$, we have $\min(|A|, |B|) \geq \tau$. Hence, C is a τ -rich set, and $|C| < k + \tau = \frac{\delta+k}{2} < 2\delta$.

Thus, [Algorithm 8](#) either returns a cut of size less than k or a τ -rich set. We first compute the rooted vertex connectivity for a vertex, which takes n maxflows. Then, based on the $\kappa(a)$ value, we run at most two more weak separator calls, each taking $O(n)$ maxflows. $\square$

Proof of Lemma E.76. Given a graph G with a gap γ , we apply [Algorithm 8](#) to either find a cut of size less than k or a τ -rich set T of size $t \leq 2\delta$ in at most $O(n)$ maxflows. We then apply [Lemma E.79](#) with inputs T and k . The algorithm guarantees that applying (s, t) -maxflows on edges of $X_{t,d}$ returns mincuts, which are

at most

$$\begin{aligned}
|E_X| &\leq t \cdot 4d && \text{From Lemma E.75} \\
&\leq 2\delta \cdot \frac{O(\delta^2)}{\tau \max(\tau, \frac{t-k}{2})} && \text{From Lemma E.79} \\
&\leq O\left(\frac{\delta^2}{\gamma}\right) && \tau = \frac{\gamma}{2}, \max(\tau, \frac{t-k}{2}) = \frac{t-k}{2} \geq \frac{\delta}{2}
\end{aligned}$$

Hence, the total number of maxflows is at most $O(n + \frac{\delta^2}{\gamma})$ on graphs of size at most $|E|$. If the mincut is less than k , according to Lemma E.81, we either get a vertex cut of size less than k or a τ -rich set. According to Lemma E.79, we find the mincut if a τ -rich set is returned. Thus, we find a cut of size less than k ; otherwise, we conclude that the graph is k -connected. $\square$

E.6.4 Gap Enlargement

We now see how to increase the gap and construct a graph with a large gap. We first do the following preliminary observations.

Lemma E.83 (Lemma 3.1 from [Gab06]). *Let $G = (V, E)$ be a graph.*

1. *For any vertex $x \in V$ with $\kappa(x) > \kappa_G$, x belongs to the vertex mincut of G .*
2. *For any pair $(x, y) \in V \times V$ with $\kappa(x, y) > \kappa_G$, $E \cup \{(x, y)\}$ does not destroy any vertex mincut in G .*

Proof. • If $\kappa(x) = \kappa_G$, then there exists a mincut that separates x from the rest of the graph G . Since $\kappa(x) > \kappa_G$, every mincut of G contains x .

- For any pair (x, y) , if $\kappa(x, y) > \kappa_G$, then there exists no mincut such that x and y lie on both sides of the mincut. Thus, adding an edge does not destroy any mincut of the graph G .

$\square$

These two important observations play a key role in constructing the graph H from G that has a large gap. Based on Item 1, we have the guarantee that the vertex x belongs to the mincut of the graph G when we assume the mincut is of size less than k . Let (L, S, R) be any mincut of the graph G . We can remove x from the graph $G \setminus \{x\}$, and the set $S \setminus \{x\}$ remains the mincut of the residual graph “ $G - x$ ”. This modification reduces δ_G by 1. However, removing x from G might also reduce δ_G by 1.

Let F be the set of vertices whose degree is $\delta_G - 1$ in the residual graph. For any $u \in F$, we can increase the degree back to δ_G by adding an edge with one of its non-neighbors, say v , after confirming that they always lie on the same side of every mincut by computing $\kappa(u, v)$. If $\kappa(u, v) \geq k$ and we have $\kappa_G < k$, then we know according to Item 2 that adding edge (u, v) does not destroy any mincut, thereby increasing the degree of u to δ_G . Thus, it takes $|F|$ maxflows to increase all the degrees in F .

We formally define the algorithm below that increases the gap by 1 and prove its correctness and runtime.

Algorithm 9: *IncreaseGap*

Input: A graph $H = (V_H, E_H)$ with gap $\gamma(H)$, a cut (L, S, R)
Output: A graph $H' = (V_{H'}, E_{H'})$ with gap $\gamma(H) + 1$, an updated cut (L, S, R)

- 1 Choose any $x \in V_H$. Compute $\kappa_H(x)$ and the corresponding cut (X, Z, Y)
- 2 **if** $|Z \cup (V \setminus V_H)| < |S|$ **then**
- 3 $(L, S, R) \leftarrow (X, Z \cup (V \setminus V_H), Y)$
- 4 **end**
- 5 Let $H' = H \setminus x$ removing x from the graph H
- 6 Let $F = \{y \in N_H(x) \mid \deg_{H'}(y) = \delta(H) - 1\}$
- 7 **for** $u \in F$ **do**
- 8 Iterate over $V_{H'}$ to find w such that $(u, w) \notin E_{H'}$
- 9 Compute $\kappa_{H'}(u, w)$ and the corresponding cut (U, Q, W)
- 10 **if** $|Q \cup (V \setminus V_{H'})| < |S|$ **then**
- 11 $(L, S, R) \leftarrow (U, Q \cup (V \setminus V_{H'}), W)$
- 12 **end**
- 13 $H' = H' \cup (u, w)$ adding the edge (u, w) to H'
- 14 **end**
- 15 **return** $H', (L, S, R)$

Claim E.84. The *Algorithm 9* takes a graph H with gap $\gamma(H)$ as input and outputs a graph H' with gap $\gamma(H) + 1$, both of size at most H , in $O(|V_H|)$ maxflows on graphs of size at most m , where m is the size of the original graph. It also maintains the smallest vertex cut in the original graph G .

Proof. We maintain the invariant that (L, S, R) is a valid vertex cut in the original graph G and update it whenever we find a smaller one. Assuming $\kappa_G < k$ and conditioned on not finding a cut of size less than k so far, we have the guarantee that $V \setminus V_H$ is contained in every mincut.

Hence, $(X, Z \cup (V \setminus V_H), Y)$ is a valid cut in G . Therefore, we update (L, S, R) with the newly found cut. Conditioned on not finding the mincut, we have the guarantee that x belongs to every mincut in G according to [Item 1](#). Thus, we can safely remove it from the graph H , constructing a new graph H' . We have $\kappa(H') = \kappa(H) - 1$.

However, $\delta(H')$ can also decrease by 1, keeping the gap the same. Now, we increase the minimum degree of H' by adding edges to the vertices with degree $\delta(H) - 1$, checking if (s, t) -vertex mincut across the two vertices in H' occurs after extending the cut to the graph G by taking the union with $V \setminus V_{H'}$. Thus, we increase the minimum degree by 1, which increases $\gamma(H')$ by one. The number of edges in H' is, at most, the number of edges in H , as we only add edges to a vertex if an edge is deleted from it due to the removal of x .

Computing $\kappa(x)$ takes n maxflows. It takes a maxflow call to increase the degree of a vertex in F . Hence, the total number of maxflow calls is at most $n + |F| \leq 2n = O(n)$. $\square$

We are now ready to prove [Lemma E.77](#) by using [Claim E.84](#).

Proof of Lemma E.77. We initialize H with $G = (V, E)$ and (L, S, R) with the minimum degree cut in G . Applying the INCREASEGAP subroutine γ times constructs a graph $H' = (V_{H'}, E_{H'})$ with gap γ of size at most $|E|$, and (L, S, R) is a valid vertex cut in G .

If $|S| < k$, we are done; otherwise, $|S| \geq k$. If $\kappa_G < k$, then from [Claim E.84](#), we have the guarantee that the smallest cut found so far is not the mincut; hence, the vertices removed from G , i.e., $T = V \setminus V_{H'}$, belong to every mincut of G . $\square$

E.7 Omitted Proofs from Section E.5.1

E.7.1 Proof of Lemma E.61

Before we give the proof we state and prove two simple claims that help in proving Lemma E.61. The following claim generalizes the Theorem E.60 to work for any given n, k and not just the powers of 2.

Claim E.85. *Given two positive integers n, k with $k < n$ and a constant $\varepsilon \in (0, 1)$, we can construct an explicit (k, ε) -disperser $D = (V, W, E)$ with $|V| = n$, left-degree $d = \Theta((\log n)^{c_{E.60}})$ and $|W| \in [c_{low} \frac{kd}{\log^3 n}, c_{high} kd]$ where c_{low}, c_{high} are some universal constants.*

Proof. Let n', k' be the powers of 2 such that $n \leq n' < 2n$ and $k/2 < k' \leq k$. It follows that $k' < n'$. Let $D' = (V', W', E')$ be the explicit (k', ε) -disperser obtained with n', k', ε as inputs to Theorem E.60 with left-degree $d = (\log n')^{c_{E.60}}$ and $|W| \in [c'_{low} \frac{k'd}{\log^3 n}, c'_{high} k'd]$.

Consider any arbitrary subset V of V' of size exactly n and consider the induced subgraph of D' restricted to V, W' and let E be the edge set of the induced graph of D' on $V \cup W'$. Any subset S of size at least k of V (of V' as well), is also a subset of V' of size at least k' as $k' \leq k$, so the neighborhood of S in D' is at least of size $(1 - \varepsilon)|W'|$. Since we take the induced subgraph of $V \cup W'$, the neighborhood of S in D is same, thus $D = (V, W = W', E)$ is a (k, ε) -disperser with number of left vertices exactly n . Since $\log n \leq \log n' \leq 1 + \log n$ the left-degree $d = (\log n')^{c_{E.60}} \leq c \cdot (\log n)^{c_{E.60}}$ for some constant c , $d \geq (\log n)^{c_{E.60}}$ and W satisfies the size requirements for some universal constants given by Theorem E.60. $\square$

The left-degree d of the disperser is $\Theta((\log n)^{c_{E.60}})$ as given in Theorem E.60. Lemma E.61 (recalled below) generalizes Theorem E.60 by making the degree arbitrarily large as needed. Note that the degree cannot be made smaller than $c(\log n)^{c_{E.60}}$ where c is some universal constant.

Lemma E.61. *Given three positive integers n, k, d with $k < n$, $0 < d < n$ and a constant $\varepsilon \in (0, 1)$, we can construct an explicit (k, ε) -disperser $D = (V, W, E)$ with $|V| = n$ and $c_{low} \frac{k \cdot d'}{\log^3 n} \leq |W| \leq c_{high} k \cdot d'$ where $d' = \Theta(\max(d, (\log n)^{c_{E.60}}))$ is the left-degree of the graph G .*

Proof. If $d \leq (\log n)^{c_{E.60}}$ then let $D = (V, W, E)$ be the explicit disperser obtained by giving n, k, ε as inputs to Claim E.85 and just output D . From the guarantees of Claim E.85 we have D as an explicit (k, ε) -disperser with $|V| = n$, the left-degree $d = \Theta((\log n)^{c_{E.60}}) = d'$ as $d \leq (\log n)^{c_{E.60}}$ and the bounds of W are satisfied.

If $d > (\log n)^{c_{E.60}}$, let $\gamma = \lceil d/(\log n)^{c_{E.60}} \rceil$. Construct a disperser $D = (V, W, E)$ with $\gamma n, \gamma k, \varepsilon$ as inputs to Claim E.85. Let V' be the set of super vertices obtained by contracting arbitrary partition $\mathcal{F}$ of V where each set contained in the partition is of size exactly γ , we have $|V'| = n$ as $|V| = \gamma n$. For any $u \in V'$, let $S_u \subset V$ be the set in $\mathcal{F}$ that contains u .

$$E' = \{(u, v) \mid u \in V', v \in W \text{ and there exists an edge from some vertex in } S_u \text{ to } W \text{ in } E\}$$

For every $u \in V'$ the set of edges in E' are precisely those obtained by taking the union of edges in E that are incident to every vertex $s \in S_u$. We output $D' = (V', W, E')$. The disperser might have multi edges and need not be simple.

We now prove the required properties of D' . For any $u \in V'$ the left-degree in D' is $|S_u| \cdot d''$ where d'' is left-degree of D . From the guarantee of Claim E.85 we have $d'' = \Theta((\log(\gamma n))^{c_{E.60}}) = \Theta((\log n)^{c_{E.60}})$ as $\log(\gamma n) = \Theta(\log n)$ and $|S_u| = \gamma$. Thus the left-degree of D' is $\gamma \cdot \Theta((\log n)^{c_{E.60}}) = \Theta(\max(d, (\log n)^{c_{E.60}}))$ as required.

It is an explicit disperser i.e., each edge in E' can be computed in $\tilde{O}(1)$. Let there be some ordering among the vertices in V , the same ordering applies to vertices in S_u for any $u \in V'$. We can compute i^{th} edge of vertex u for $i \in [\gamma \cdot d'']$ by computing $(i \bmod d'')^{th}$ edge of $(i/d'')^{th}$ vertex in S_u of D in $\tilde{O}(1)$ time. We have $|V'| = |V|/\gamma = n$ as $|V| = \gamma n$ is guaranteed from Claim E.85.

We have $|W| \in [c_{low} \frac{(\gamma k) d''}{\log^3(\gamma n)}, c_{high}(\gamma k) d''] = [c'_{low} \frac{k d'}{\log^3 n}, c'_{high} k d']$ as $\gamma d'' = \Theta(d)$ and $d' = d = \max(d, (\log n)^{c_{E.60}})$ where c'_{low}, c'_{high} are some universal constants.

It is left to prove that D' is a (k, ε) -disperser. Any subset T of V' of size at least k corresponds to a subset of V of size at least γk . From the guarantees of [Claim E.85](#) and based on the construction of E' the size of the neighborhood of T is at least $(1 - \varepsilon)|W|$ as required. $\square$

Bibliography

- [AR20a] S. Aaronson and P. Rall. „Quantum Approximate Counting, Simplified“. In: *SOSA*. SIAM, 2020, pp. 24–32.
- [AB24] A. Abboud and G. Bodwin. „Reachability preservers: New extremal bounds and approximation algorithms“. In: *SIAM J. Comput.* 53.2 (2024), pp. 221–246.
- [ABP18] A. Abboud, G. Bodwin, and S. Pettie. „A hierarchy of lower bounds for sublinear additive spanners“. In: *SIAM J. Comput.* 47.6 (2018), pp. 2203–2236.
- [Abb+18] A. Abboud, L. Georgiadis, D. Graf, G. F. Italiano, R. Krauthgamer, N. Parotsidis, O. Trabelsi, and P. Uznanski. „Faster Algorithms for All-Pairs Bounded Min-Cuts“. In: *CoRR* abs/1807.05803 (2018).
- [Abb+22] A. Abboud, R. Krauthgamer, J. Li, D. Panigrahi, T. Saranurak, and O. Trabelsi. „Breaking the Cubic Barrier for All-Pairs Max-Flow: Gomory-Hu Tree in Nearly Quadratic Time“. In: *FOCS*. IEEE, 2022, pp. 884–895.
- [AKT21] A. Abboud, R. Krauthgamer, and O. Trabelsi. „Subcubic algorithms for Gomory-Hu tree in unweighted graphs“. In: *STOC*. ACM, 2021, pp. 1725–1737.
- [Abb+23] A. Abboud, J. Li, D. Panigrahi, and T. Saranurak. „All-Pairs Max-Flow is no Harder than Single-Pair Max-Flow: Gomory-Hu Trees in Almost-Linear Time“. In: *FOCS*. IEEE, 2023, pp. 2204–2212.
- [AWW16] A. Abboud, V. V. Williams, and J. R. Wang. „Approximation and Fixed Parameter Subquadratic Algorithms for Radius and Diameter in Sparse Graphs“. In: *SODA*. SIAM, 2016, pp. 377–391.
- [ABN08] I. Abraham, Y. Bartal, and O. Neiman. „Nearly Tight Low Stretch Spanning Trees“. In: *FOCS*. IEEE Computer Society, 2008, pp. 781–790.
- [Aga+24] A. Agarwal, S. Khanna, H. Li, P. Patil, C. Wang, N. White, and P. Zhong. „Parallel Approximate Maximum Flows in Near-Linear Work and Polylogarithmic Depth“. In: *SODA*. SIAM, 2024, pp. 3997–4061.
- [AKO18] M. Ahmadi, F. Kuhn, and R. Oshman. „Distributed Approximate Maximum Matching in the CONGEST Model“. In: *DISC*. Vol. 121. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 6:1–6:17.
- [AG11] K. J. Ahn and S. Guha. „Linear Programming in the Semi-streaming Model with Application to the Maximum Matching Problem“. In: *ICALP (2)*. Vol. 6756. Lecture Notes in Computer Science. Springer, 2011, pp. 526–538.

- [AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
- [AK99] A. Ahuja and S. Kapoor. „A quantum algorithm for finding the maximum“. In: *arXiv preprint quant-ph/9911082* (1999).
- [Ahu+92] R. K. Ahuja, A. V. Goldberg, J. B. Orlin, and R. E. Tarjan. „Finding minimum-cost flows by double scaling“. In: *Mathematical programming* 53.1 (1992), pp. 243–266.
- [Alo+06] N. Alon, B. Awerbuch, Y. Azar, N. Buchbinder, and J. Naor. „A general approach to online network optimization problems“. In: *ACM Transactions on Algorithms (TALG)* 2.4 (2006), pp. 640–660.
- [AMS99] N. Alon, Y. Matias, and M. Szegedy. „The Space Complexity of Approximating the Frequency Moments“. In: *J. Comput. Syst. Sci.* 58.1 (1999), pp. 137–147.
- [Alt+93] I. Althöfer, G. Das, D. P. Dobkin, D. Joseph, and J. Soares. „On Sparse Spanners of Weighted Graphs“. In: *Discret. Comput. Geom.* 9 (1993), pp. 81–100.
- [AL02] A. Amit and N. Linial. „Random Graph Coverings I: General Theory and Graph Connectivity“. In: *Combinatorica* 22.1 (2002), pp. 1–18.
- [ALM02] A. Amit, N. Linial, and J. Matoušek. „Random Lifts of Graphs: Independence and Chromatic Number“. In: *Random Structures & Algorithms* 20.1 (2002), pp. 1–22.
- [ACL06] R. Andersen, F. R. K. Chung, and K. J. Lang. „Local Graph Partitioning using PageRank Vectors“. In: *FOCS*. IEEE Computer Society, 2006, pp. 475–486.
- [ACL08] R. Andersen, F. R. K. Chung, and K. J. Lang. „Local Partitioning for Directed Graphs Using PageRank“. In: *Internet Mathematics* 5.1 (2008), pp. 3–22.
- [AL08] R. Andersen and K. J. Lang. „An algorithm for improving graph partitions“. In: *SODA*. SIAM, 2008, pp. 651–660.
- [AP09] R. Andersen and Y. Peres. „Finding sparse cuts locally using evolving sets“. In: *STOC*. ACM, 2009, pp. 235–244.
- [AB23] D. Anderson and G. E. Blelloch. „Parallel Minimum Cuts in $O(m \log^2 n)$ Work and Low Depth“. In: *ACM Trans. Parallel Comput.* 10.4 (2023), 18:1–18:28.
- [ASZ20] A. Andoni, C. Stein, and P. Zhong. „Parallel approximate undirected shortest paths via low hop emulators“. In: *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*. 2020, pp. 322–335.
- [Ape+22] S. Apers, Y. Efron, P. Gawrychowski, T. Lee, S. Mukhopadhyay, and D. Nanongkai. „Cut Query Algorithms with Star Contraction“. In: *FOCS*. IEEE, 2022, pp. 507–518.
- [AL21] S. Apers and T. Lee. „Quantum Complexity of Minimum Cut“. In: *CCC*. Vol. 200. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 28:1–28:33.
- [AGG88] P. Arbenz, W. Gander, and G. H. Golub. „Restricted rank modification of the symmetric eigenvalue problem: Theoretical considerations“. In: *Linear Algebra and its Applications* 104 (1988), pp. 75–95.
- [Ary+95] S. Arya, G. Das, D. M. Mount, J. S. Salowe, and M. H. M. Smid. „Euclidean spanners: short, thin, and lanky“. In: *STOC*. ACM, 1995, pp. 489–498.
- [Ash+24a] V. Ashvinkumar, A. Bernstein, N. Cao, C. Grunau, B. Haeupler, Y. Jiang, D. Nanongkai, and H. Su. „Parallel, Distributed, and Quantum Exact Single-Source Shortest Paths with Negative Edge Weights“. In: *ESA*. Vol. 308. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, 13:1–13:15.

- [Ash+24b] V. Ashvinkumar, A. Bernstein, C. Deng, J. Gao, and N. Wein. „Low Sensitivity Hopsets“. In: *arXiv preprint arXiv:2407.10249* (2024).
- [ABK25] V. Ashvinkumar, A. Bernstein, and A. Karczmarz. „Faster approximation algorithms for restricted shortest paths in directed graphs“. In: *Proceedings of the 2025 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM. 2025, pp. 5263–5277.
- [AHW25] S. Assadi, G. Hoppenworth, and N. Wein. „Covering Approximate Shortest Paths with DAGs“. In: *STOC*. ACM, 2025, pp. 2269–2280.
- [Ass+22] S. Assadi, A. Jambulapati, Y. Jin, A. Sidford, and K. Tian. „Semi-Streaming Bipartite Matching in Fewer Passes and Optimal Space“. In: *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM. 2022, pp. 627–669.
- [ALT21] S. Assadi, S. C. Liu, and R. E. Tarjan. „An Auction Algorithm for Bipartite Matching in Streaming and Massively Parallel Computation Models“. In: *SOSA*. SIAM, 2021, pp. 165–171.
- [AR20b] S. Assadi and R. Raz. „Near-Quadratic Lower Bounds for Two-Pass Graph Streaming Algorithms“. In: *FOCS*. IEEE, 2020, pp. 342–353.
- [AS23] S. Assadi and V. Shah. „Tight Bounds for Vertex Connectivity in Dynamic Streams“. In: *SOSA*. SIAM, 2023, pp. 213–227.
- [aut25] A. authors. *Deterministic Negative-Weight Single-Source Shortest Path in Nearly Linear Time*. 2025.
- [Awe85] B. Awerbuch. „Complexity of Network Synchronization“. In: *J. ACM* 32.4 (1985). Announced at STOC’84, pp. 804–823.
- [Awe+92] B. Awerbuch, B. Berger, L. Cowen, and D. Peleg. „Fast Network Decomposition (Extended Abstract)“. In: *PODC*. ACM, 1992, pp. 169–177.
- [Awe+98] B. Awerbuch, B. Berger, L. Cowen, and D. Peleg. „Near-Linear Time Construction of Sparse Neighborhood Covers“. In: *SIAM J. Comput.* 28.1 (1998), pp. 263–277.
- [Awe+89] B. Awerbuch, A. V. Goldberg, M. Luby, and S. A. Plotkin. „Network Decomposition and Locality in Distributed Computation“. In: *FOCS*. IEEE Computer Society, 1989, pp. 364–369.
- [AP92] B. Awerbuch and D. Peleg. „Routing with Polynomial Communication-Space Trade-Off“. In: *SIAM J. Discret. Math.* 5.2 (1992), pp. 151–162.
- [AMV20] K. Axiotis, A. Madry, and A. Vladu. „Circulation control for faster minimum cost flow in unit-capacity graphs“. In: *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE. 2020, pp. 93–104.
- [AMV22] K. Axiotis, A. Mađry, and A. Vladu. „Faster sparse minimum cost flow by electrical flow localization“. In: *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE. 2022, pp. 528–539.
- [BFS86] L. Babai, P. Frankl, and J. Simon. „Complexity classes in communication complexity theory (preliminary version)“. In: *FOCS*. IEEE Computer Society, 1986, pp. 337–347.
- [BL98] I. Bárány and D. G. Larman. „The convex hull of the integer points in a large ball“. In: *Mathematische Annalen* 312 (1998), pp. 167–181.
- [Bar96a] Y. Bartal. „Probabilistic approximation of metric spaces and its algorithmic applications“. In: *Proceedings of 37th Conference on Foundations of Computer Science*. IEEE. 1996, pp. 184–193.

- [Bar96b] Y. Bartal. „Probabilistic Approximations of Metric Spaces and Its Algorithmic Applications“. In: *37th Annual Symposium on Foundations of Computer Science, FOCS '96, Burlington, Vermont, USA, 14-16 October, 1996*. IEEE Computer Society, 1996, pp. 184–193.
- [BFN19] Y. Bartal, N. Fandina, and O. Neiman. „Covering Metric Spaces by Few Trees“. In: *46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. 2019, pp. 20–1.
- [BM03] Y. Bartal and M. Mendel. „Multi-embedding and path approximation of metric spaces.“ In: *SODA*. Vol. 3. 2003, pp. 424–433.
- [Bec+82] M. Becker, W. Degenhardt, J. Doenhardt, S. Hertel, G. Kaninke, W. Kerber, K. Mehlhorn, S. Näher, H. Rohnert, and T. Winter. „A Probabilistic Algorithm for Vertex Connectivity of Graphs“. In: *Inf. Process. Lett.* 15.3 (1982), pp. 135–136.
- [Bel58] R. Bellman. „On a Routing Problem“. In: *Quarterly of Applied Mathematics* 16.1 (1958), pp. 87–90.
- [BP20] U. Ben-Levy and M. Parter. „New (α, β) spanners and hopsets“. In: *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. 2020, pp. 1695–1714.
- [Ber+14] P. Berman, A. Bhattacharyya, E. Grigorescu, S. Raskhodnikova, D. P. Woodruff, and G. Yaroslavtsev. „Steiner transitive-closure spanners of low-dimensional posets“. In: *Comb.* 34.3 (2014), pp. 255–277.
- [BRR10] P. Berman, S. Raskhodnikova, and G. Ruan. „Finding sparser directed spanners“. In: *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2010)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik. 2010.
- [Ber+25a] A. Bernstein, J. Blikstad, J. Li, T. Saranurak, and T. Tu. „Combinatorial Maximum Flow via Weighted Push-Relabel on Shortcut Graphs“. In: *FOCS*. IEEE, 2025, pp. 464–485.
- [Ber+24] A. Bernstein, J. Blikstad, T. Saranurak, and T. Tu. „Maximum Flow by Augmenting Paths in $n^{2+o(1)}$ Time“. In: *FOCS*. IEEE, 2024, pp. 2056–2077.
- [Ber+20] A. Bernstein, J. v. d. Brand, M. Probst Gutenberg, D. Nanongkai, T. Saranurak, A. Sidford, and H. Sun. „Fully-Dynamic Graph Sparsifiers Against an Adaptive Adversary“. In: *CoRR* abs/2004.08432 (2020).
- [Ber+25b] A. Bernstein, H. L. Fleischmann, M. P. Gutenberg, B. Haeupler, G. Hoppenworth, Y. Jiang, G. Z. Li, S. Pettie, T. Saranurak, and L. Schiller. „Reviving Thorup’s Shortcut Conjecture“. In: *CoRR* abs/2510.24954 (2025).
- [BGS20] A. Bernstein, M. P. Gutenberg, and T. Saranurak. „Deterministic Decremental Reachability, SCC, and Shortest Paths via Directed Expanders and Congestion Balancing“. In: *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*. Ed. by S. Irani. IEEE, 2020, pp. 1123–1134.
- [BGS21] A. Bernstein, M. P. Gutenberg, and T. Saranurak. „Deterministic Decremental SSSP and Approximate Min-Cost Flow in Almost-Linear Time“. In: *FOCS*. IEEE, 2021, pp. 1000–1008.
- [BNW22] A. Bernstein, D. Nanongkai, and C. Wulff-Nilsen. „Negative-Weight Single-Source Shortest Paths in Near-linear Time“. In: *FOCS*. IEEE, 2022, pp. 600–611.

- [BPW20] A. Bernstein, M. Probst Gutenberg, and C. Wulff-Nilsen. „Near-Optimal Incremental SSSP in Dense Weighted Digraphs“. In: *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*. Ed. by S. Irani. IEEE, 2020, pp. 1112–1122.
- [BW23] A. Bernstein and N. Wein. „Closing the gap between directed hopsets and shortcut sets“. In: *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 2023, pp. 163–182.
- [Bha+12] A. Bhattacharyya, E. Grigorescu, K. Jung, S. Raskhodnikova, and D. P. Woodruff. „Transitive-closure spanners“. In: *SIAM J. Comput.* 41.6 (2012), pp. 1380–1425.
- [Bho+26] S. Bhore, H. Chang, J. Conroy, A. Filtser, E. Oh, N. Wein, and D. W. Zheng. „DAG Covers: The Steiner Point Effect“. In: *CoRR* abs/2604.04186 (2026).
- [BKR03] M. Bienkowski, M. Korzeniowski, and H. Räcke. „A practical algorithm for constructing oblivious routing schemes“. In: *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*. 2003, pp. 24–33.
- [Ble+14] G. E. Blelloch, A. Gupta, I. Koutis, G. L. Miller, R. Peng, and K. Tangwongsan. „Nearly-Linear Work Parallel SDD Solvers, Low-Diameter Decomposition, and Low-Stretch Subgraphs“. In: *Theory Comput. Syst.* 55.3 (2014), pp. 521–554.
- [Bli+22] J. Blikstad, J. van den Brand, Y. Efron, S. Mukhopadhyay, and D. Nanongkai. „Nearly Optimal Communication and Query Complexity of Bipartite Matching“. In: *FOCS*. IEEE, 2022, pp. 1174–1185.
- [Bli+25] J. Blikstad, Y. Jiang, S. Mukhopadhyay, and S. Yingchareonthawornchai. „Global vs. s-t Vertex Connectivity Beyond Sequential: Almost-Perfect Reductions & Near-Optimal Separations“. In: *To appear in STOC* (2025).
- [Bod17] G. Bodwin. „Linear size distance preservers“. In: *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2017, pp. 600–615.
- [BH23] G. Bodwin and G. Hoppenworth. „Folklore Sampling is Optimal for Exact Hopsets: Confirming the $\sqrt{n}$ Barrier“. In: *Proceedings of the 64th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. 2023, pp. 701–720.
- [Bra+22] J. van den Brand, Y. Gao, A. Jambulapati, Y. T. Lee, Y. P. Liu, R. Peng, and A. Sidford. „Faster maxflow via improved dynamic spectral vertex sparsifiers“. In: *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*. 2022, pp. 543–556.
- [Bra+25] J. van den Brand, H. Gholizadeh, Y. Jiang, and T. de Vos. „Parallel Minimum Cost Flow in Near-Linear Work and Square Root Depth for Dense Instances“. In: *SPAA*. ACM, 2025, pp. 1–16.
- [BLS23] J. van den Brand, Y. P. Liu, and A. Sidford. „Dynamic Maxflow via Dynamic Interior Point Methods“. In: *STOC*. ACM, 2023, pp. 1215–1228.
- [BNS19] J. van den Brand, D. Nanongkai, and T. Saranurak. „Dynamic Matrix Inverse: Improved Algorithms and Matching Conditional Lower Bounds“. In: *FOCS*. IEEE Computer Society, 2019, pp. 456–480.
- [Bra20] J. v. d. Brand. „A Deterministic Linear Program Solver in Current Matrix Multiplication Time“. In: *SODA*. SIAM, 2020, pp. 259–278.
- [Bra+23] J. Brand, L. Chen, M. P. Gutenberg, R. Kyng, Y. P. Liu, R. Peng, S. Sachdeva, and A. Sidford. „A Deterministic Almost-Linear Time Algorithm for Minimum-Cost Flow“. In: *FOCS*. 2023.

- [Bra+21] J. v. d. Brand, Y. T. Lee, Y. P. Liu, T. Saranurak, A. Sidford, Z. Song, and D. Wang. „Minimum cost flows, MDPs, and ℓ_1 -regression in nearly linear time for dense instances“. In: *STOC*. ACM, 2021, pp. 859–869.
- [Bra+20a] J. v. d. Brand, Y. T. Lee, D. Nanongkai, R. Peng, T. Saranurak, A. Sidford, Z. Song, and D. Wang. „Bipartite Matching in Nearly-linear Time on Moderately Dense Graphs“. In: *FOCS*. IEEE, 2020, pp. 919–930.
- [Bra+20b] J. v. d. Brand, Y. T. Lee, A. Sidford, and Z. Song. „Solving Tall Dense Linear Programs in Nearly Linear Time“. In: *STOC*. <https://arxiv.org/pdf/2002.02304.pdf>, 2020.
- [Bra+02] G. Brassard, P. Hoyer, M. Mosca, and A. Tapp. „Quantum amplitude amplification and estimation“. In: *Contemporary Mathematics* 305 (2002), pp. 53–74.
- [BCF23] K. Bringmann, A. Cassis, and N. Fischer. „Negative-Weight Single-Source Shortest Paths in Near-Linear Time: Now Faster!“ In: *FOCS*. IEEE, 2023, pp. 515–538.
- [BF13] K. Bringmann and T. Friedrich. „Exact and Efficient Generation of Geometric Random Variates and Random Graphs“. In: *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part I*. Ed. by F. V. Fomin, R. Freivalds, M. Z. Kwiatkowska, and D. Peleg. Vol. 7965. Lecture Notes in Computer Science. Springer, 2013, pp. 267–278.
- [Buh+99] H. Buhrman, R. Cleve, R. de Wolf, and C. Zalka. „Bounds for Small-Error and Zero-Error Quantum Algorithms“. In: *FOCS*. IEEE Computer Society, 1999, pp. 358–368.
- [CF23] N. Cao and J. T. Fineman. „Parallel Exact Shortest Paths in Almost Linear Work and Square Root Depth“. In: *SODA*. SIAM, 2023, pp. 4354–4372.
- [CFR20] N. Cao, J. T. Fineman, and K. Russell. „Efficient construction of directed hopsets and parallel approximate shortest paths“. In: *STOC*. ACM, 2020, pp. 336–349.
- [Cen+22] R. Cen, J. Li, D. Nanongkai, D. Panigrahi, T. Saranurak, and K. Quanrud. „Minimum cuts in directed graphs via partial sparsification“. In: *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE. 2022, pp. 1147–1158.
- [Cen+17] K. Censor-Hillel, M. Ghaffari, G. Giakkoupis, B. Haeupler, and F. Kuhn. „Tight Bounds on Vertex Connectivity Under Sampling“. In: *ACM Trans. Algorithms* 13.2 (2017), 19:1–19:26.
- [CGK14a] K. Censor-Hillel, M. Ghaffari, and F. Kuhn. „A New Perspective on Vertex Connectivity“. In: *SODA*. SIAM, 2014, pp. 546–561.
- [CGK14b] K. Censor-Hillel, M. Ghaffari, and F. Kuhn. „Distributed connectivity decomposition“. In: *PODC*. ACM, 2014, pp. 156–165.
- [Cha+23a] D. Chakrabarty, A. Graur, H. Jiang, and A. Sidford. „Parallel Submodular Function Minimization“. In: *FOCS*. IEEE, 2023.
- [Cha+19] D. Chakrabarty, Y. T. Lee, A. Sidford, S. Singla, and S. C. Wong. „Faster Matroid Intersection“. In: *FOCS*. IEEE Computer Society, 2019, pp. 1146–1168.
- [Cha+21] P. Chalermsook, S. Das, Y. Kook, B. Laekhanukit, Y. P. Liu, R. Peng, M. Sellke, and D. Vaz. „Vertex sparsification for edge connectivity“. In: *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM. 2021, pp. 1206–1225.
- [Cha+26a] P. Chalermsook, Y. Jiang, S. Mukhopadhyay, and D. Nanongkai. „Shortcuts and Transitive-Closure Spanners Approximation“. In: *SODA*. SIAM, 2026, pp. 3536–3561.

- [Cha+26b] Y.-J. Chang, Y. Chen, D. Dey, Y. Jiang, G. Mishra, H. T. Nguyen, and M. Yang. *The Complexity of Distributed Minimum Weight Cycle Approximation*. 2026.
- [CS19] Y. Chang and T. Saranurak. „Improved Distributed Expander Decomposition and Nearly Optimal Triangle Enumeration“. In: *PODC*. ACM, 2019, pp. 66–73.
- [CS20] Y. Chang and T. Saranurak. „Deterministic Distributed Expander Decomposition and Routing with Applications in Distributed Derandomization“. In: *FOCS*. IEEE, 2020, pp. 377–388.
- [Cha+23b] A. Chattopadhyay, Y. Dahiya, N. S. Mande, J. Radhakrishnan, and S. Sanyal. „Randomized versus Deterministic Decision Tree Size“. In: *STOC*. ACM, 2023, pp. 867–880.
- [CM15] A. Chattopadhyay and S. Mukhopadhyay. „Tribes Is Hard in the Message Passing Model“. In: *STACS*. Vol. 30. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015, pp. 224–237.
- [Cha+09] K. Chaudhuri, S. Rao, S. Riesenfeld, and K. Talwar. „A push–relabel approximation algorithm for approximating the minimum-degree MST problem and its generalization to matroids“. In: *Theoretical Computer Science* 410.44 (2009), pp. 4489–4503.
- [Cha87] B. Chazelle. „Computing on a Free Tree via Complexity-Preserving Mappings“. In: *Algorithmica* 2 (1987), pp. 337–361.
- [Cha97] B. Chazelle. „A Faster Deterministic Algorithm for Minimum Spanning Trees“. In: *FOCS*. IEEE Computer Society, 1997, pp. 22–31.
- [Che+17] S. Chechik, T. D. Hansen, G. F. Italiano, V. Loitzenbauer, and N. Parotsidis. „Faster algorithms for computing maximal 2-connected subgraphs in sparse directed graphs“. In: *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM. 2017, pp. 1900–1918.
- [CZ20] S. Chechik and T. Zhang. „Dynamic Low-Stretch Spanning Trees in Subpolynomial Time“. In: *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*. Ed. by S. Chawla. SIAM, 2020, pp. 463–475.
- [CZ22] S. Chechik and T. Zhang. „Constant approximation of min-distances in near-linear time“. In: *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE. 2022, pp. 896–906.
- [CQ21] C. Chekuri and K. Quanrud. „Isolating Cuts, (Bi-)Submodularity, and Faster Algorithms for Connectivity“. In: *ICALP*. Vol. 198. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 50:1–50:20.
- [Che+24] L. Chen, R. Kyng, Y. P. Liu, S. Meierhans, and M. Probst Gutenberg. „Almost-linear time algorithms for incremental graphs: Cycle detection, sccs, st shortest path, and minimum-cost flow“. In: *Proceedings of the 56th Annual ACM Symposium on Theory of Computing*. 2024, pp. 1165–1173.
- [Che+25] L. Chen, R. Kyng, Y. P. Liu, R. Peng, M. P. Gutenberg, and S. Sachdeva. „Maximum Flow and Minimum-Cost Flow in Almost-Linear Time“. In: *J. ACM* 72.3 (2025), 19:1–19:103.
- [Che+21] L. Chen, G. Kol, D. Paramonov, R. R. Saxena, Z. Song, and H. Yu. „Almost optimal super-constant-pass streaming lower bounds for reachability“. In: *STOC*. ACM, 2021, pp. 570–583.

- [Che11] M. Cheraghchi. „Applications of Derandomization Theory in Coding“. In: *Ph. D. Thesis* (2011).
- [CKT93] J. Cheriyan, M. Kao, and R. Thurimella. „Scan-First Search and Sparse Certificates: An Improved Parallel Algorithms for k -Vertex Connectivity“. In: *SIAM J. Comput.* 22.1 (1993), pp. 157–174.
- [CR94] J. Cheriyan and J. H. Reif. „Directed s - t Numberings, Rubber Bands, and Testing Digraph k -Vertex Connectivity“. In: *Combinatorica* 14.4 (1994). Announced at SODA’92, pp. 435–451.
- [CT91] J. Cheriyan and R. Thurimella. „Algorithms for Parallel k -Vertex Connectivity and Sparse Certificates (Extended Abstract)“. In: *STOC*. ACM, 1991, pp. 391–401.
- [CLL13] H. Y. Cheung, L. C. Lau, and K. M. Leung. „Graph connectivities, network coding, and expander graphs“. In: *SIAM Journal on Computing* 42.3 (2013), pp. 733–751.
- [CK05] B. S. Chlebus and D. R. Kowalski. „Almost Optimal Explicit Selectors“. In: *Fundamentals of Computation Theory*. Ed. by M. Liśkiewicz and R. Reischuk. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 270–280.
- [Chr+20] M. Christandl, F. L. Gall, V. Lysikov, and J. Zuydam. „Barriers for rectangular matrix multiplication“. In: *CoRR* abs/2003.03019 (2020).
- [CGR02] M. Chrobak, L. Gasieniec, and W. Rytter. „Fast broadcasting and gossiping in radio networks“. In: *Journal of Algorithms* 43.2 (2002), pp. 177–189.
- [Chu21] J. Chuzhoy. „Decremental all-pairs shortest paths in deterministic near-linear time“. In: *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*. 2021, pp. 626–639.
- [Chu+20] J. Chuzhoy, Y. Gao, J. Li, D. Nanongkai, R. Peng, and T. Saranurak. „A Deterministic Algorithm for Balanced Cut with Applications to Dynamic Connectivity, Flows, and Beyond“. In: *FOCS*. IEEE, 2020, pp. 1158–1167.
- [CK19] J. Chuzhoy and S. Khanna. „A New Algorithm for Decremental Single-Source Shortest Paths with Applications to Vertex-Capacitated Flow and Cut Problems“. To appear at STOC’19. 2019.
- [CK24a] J. Chuzhoy and S. Khanna. „A faster combinatorial algorithm for maximum bipartite matching“. In: *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM. 2024, pp. 2185–2235.
- [CK24b] J. Chuzhoy and S. Khanna. „Maximum Bipartite Matching in $n^{2+o(1)}$ Time via a Combinatorial Algorithm“. In: *Proceedings of the 56th Annual ACM Symposium on Theory of Computing*. 2024, pp. 83–94.
- [CP25] J. Chuzhoy and M. Parter. „Fully dynamic algorithms for graph spanners via low-diameter router decomposition“. In: *Proceedings of the 2025 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM. 2025, pp. 785–823.
- [CT25] J. Chuzhoy and O. Trabelsi. „Breaking the $O(mn)$ -Time Barrier for Vertex-Weighted Global Minimum Cut“. In: *CoRR* abs/2506.11926 (2025).
- [CZ23] J. Chuzhoy and R. Zhang. „A new deterministic algorithm for fully dynamic all-pairs shortest paths“. In: *Proceedings of the 55th Annual ACM Symposium on Theory of Computing*. 2023, pp. 1159–1172.

- [Coh95] E. Cohen. „Approximate Max-Flow on Small Depth Networks“. In: *SIAM J. Comput.* 24.3 (1995), pp. 579–597.
- [Coh97] E. Cohen. „Size-estimation framework with applications to transitive closure and reachability“. In: *Journal of Computer and System Sciences* 55.3 (1997), pp. 441–453.
- [Coh98] E. Cohen. „Fast algorithms for constructing t-spanners and paths with stretch t“. In: *SIAM J. Comput.* 28.1 (1998), pp. 210–236.
- [Coh00] E. Cohen. „Polylog-time and near-linear work approximation scheme for undirected shortest paths“. In: *J. ACM* 47.1 (2000), pp. 132–166.
- [CLS19] M. B. Cohen, Y. T. Lee, and Z. Song. „Solving Linear Programs in the Current Matrix Multiplication Time“. In: *STOC*. <https://arxiv.org/pdf/1810.07896>. 2019.
- [Coh+17] M. B. Cohen, A. Madry, P. Sankowski, and A. Vladu. „Negative-Weight Shortest Paths and Unit Capacity Minimum Cost Flow in $O(m^{10/7} \log W)$ Time“. In: *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM. 2017, pp. 752–771.
- [Coh+14] M. B. Cohen, R. Kyng, G. L. Miller, J. W. Pachocki, R. Peng, A. B. Rao, and S. C. Xu. „Solving SDD linear systems in nearly $m \log^{1/2} n$ time“. In: *STOC*. ACM, 2014, pp. 343–352.
- [CE06] D. Coppersmith and M. Elkin. „Sparse sourcewise and pairwise distance preservers“. In: *SIAM Journal on Discrete Mathematics* 20.2 (2006), pp. 463–501.
- [Cor+09] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.
- [CF14] G. Cormode and D. Firmani. „A unifying framework for ℓ_0 -sampling algorithms“. In: *Distributed Parallel Databases* 32.3 (2014), pp. 315–335.
- [DS08] S. I. Daitch and D. A. Spielman. „Faster approximate lossy generalized flow via interior point algorithms“. In: *STOC*. ACM, 2008, pp. 451–460.
- [DK21] M. Dalirrooyfard and J. Kaufmann. „Approximation Algorithms for Min-Distance Problems in DAGs“. In: *48th International Colloquium on Automata, Languages, and Programming (ICALP 2021)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. 2021, pp. 60–1.
- [Dan51] G. B. Dantzig. „Application of the simplex method to a transportation problem“. In: *Activity analysis and production and allocation* (1951).
- [DNV16] S. F. De Rezende, J. Nordström, and M. Vinyals. „How limited interaction hinders real communication (and what it means for proof and circuit complexity)“. In: *Proceedings of the 57th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. 2016, pp. 295–304.
- [Don+22] S. Dong, Y. Gao, G. Goranci, Y. T. Lee, R. Peng, S. Sachdeva, and G. Ye. „Nested Dissection Meets IPMs: Planar Min-Cost Flow in Nearly-Linear Time“. In: *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 2022, pp. 124–153.
- [DB13] F. Dörfler and F. Bullo. „Kron Reduction of Graphs with Applications to Electrical Networks“. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 60.1 (2013), pp. 150–163.

- [Dor+21] M. Dory, Y. Efron, S. Mukhopadhyay, and D. Nanongkai. „Distributed weighted min-cut in nearly-optimal time“. In: *STOC*. Ed. by S. Khuller and V. V. Williams. ACM, 2021, pp. 1144–1153.
- [DH96] C. Durr and P. Hoyer. „A quantum algorithm for finding the minimum“. In: *arXiv preprint quant-ph/9607014* (1996).
- [Dür+06] C. Dürr, M. Heiligman, P. Høyer, and M. Mhalla. „Quantum Query Complexity of Some Graph Problems“. In: *SIAM J. Comput.* 35.6 (2006), pp. 1310–1328.
- [EK72] J. Edmonds and R. M. Karp. „Theoretical improvements in algorithmic efficiency for network flow problems“. In: *Journal of the ACM (JACM)* 19.2 (1972), pp. 248–264.
- [Egg+12] S. Eggert, L. Kliemann, P. Munstermann, and A. Srivastav. „Bipartite matching in the semi-streaming model“. In: *Algorithmica* 63.1 (2012), pp. 490–508.
- [EN19] M. Elkin and O. Neiman. „Linear-size hopsets with small hopbound, and constant-hopbound hopsets in RNC“. In: *Proceedings of the 31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2019, pp. 333–341.
- [Epp+97] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. „Sparsification - a technique for speeding up dynamic graph algorithms“. In: *J. ACM* 44.5 (1997), pp. 669–696.
- [EH84] A. Esfahanian and S. L. Hakimi. „On computing the connectivities of graphs and digraphs“. In: *Networks* 14.2 (1984), pp. 355–366.
- [Eve75] S. Even. „An Algorithm for Determining Whether the Connectivity of a Graph is at Least k “. In: *SIAM J. Comput.* 4.3 (1975), pp. 393–396.
- [EIR98] S. Even, G. Itkis, and S. Rajsbaum. „On Mixed Connectivity Certificates“. In: *Theor. Comput. Sci.* 203.2 (1998), pp. 253–269.
- [ET75] S. Even and R. E. Tarjan. „Network Flow and Testing Graph Connectivity“. In: *SIAM J. Comput.* 4.4 (1975), pp. 507–518.
- [FR06] J. Fakcharoenphol and S. Rao. „Planar graphs, negative weight edges, shortest paths, and near linear time“. In: *J. Comput. Syst. Sci.* 72.5 (2006). Announced at FOCS’01, pp. 868–889.
- [FRT04] J. Fakcharoenphol, S. Rao, and K. Talwar. „A tight bound on approximating arbitrary metrics by tree metrics“. In: *J. Comput. Syst. Sci.* 69.3 (2004), pp. 485–497.
- [Fil26] A. Filtser. „Stochastic Embedding of Digraphs into DAGs“. In: *SODA*. SIAM, 2026, pp. 105–119.
- [FL21] A. Filtser and H. Le. „Clan embeddings into trees, and low treewidth graphs“. In: *STOC*. ACM, 2021, pp. 342–355.
- [Fin20] J. T. Fineman. „Nearly Work-Efficient Parallel Algorithm for Digraph Reachability“. In: *SIAM J. Comput.* 49.5 (2020).
- [Fis+25] N. Fischer, B. Haeupler, R. Latypov, A. Royskoe, and A. L. Sulser. „A Simple Parallel Algorithm with Near-Linear Work for Negative-Weight Single-Source Shortest Path“. In: *SOSA*. SIAM, 2025, pp. 216–225.
- [Fis+26] O. Fischer, Y. Jiang, S. Mukhopadhyay, and S. Yingchareonthawornchai. „Directed and Undirected Vertex Connectivity Problems are Equivalent for Dense Graphs“. In: *SOSA*. SIAM, 2026, pp. 413–420.

- [FI03] L. Fleischer and S. Iwata. „A push-relabel framework for submodular function minimization and applications to parametric optimization“. In: *Discrete Applied Mathematics* 131.2 (2003), pp. 311–322.
- [FLL25] H. L. Fleischmann, G. Z. Li, and J. Li. „Improved Directed Expander Decompositions“. In: *CoRR* abs/2507.09729 (2025).
- [FF56] L. R. Ford and D. R. Fulkerson. „Maximal flow through a network“. In: *Canadian journal of Mathematics* 8.3 (1956), pp. 399–404.
- [For56] R. Ford. *Paper P-923*. Santa Monica, California: The RAND Corporation, 1956.
- [FG19] S. Forster and G. Goranci. „Dynamic low-stretch trees via dynamic low-diameter decompositions“. In: *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*. Ed. by M. Charikar and E. Cohen. ACM, 2019, pp. 377–388.
- [For+21] S. Forster, G. Goranci, Y. P. Liu, R. Peng, X. Sun, and M. Ye. „Minor Sparsifiers and the Distributed Laplacian Paradigm“. In: *FOCS*. IEEE, 2021, pp. 989–999.
- [FGV21] S. Forster, M. Grösbacher, and T. de Vos. „An Improved Random Shift Algorithm for Spanners and Low Diameter Decompositions“. In: *OPODIS*. Vol. 217. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 16:1–16:17.
- [For+20] S. Forster, D. Nanongkai, L. Yang, T. Saranurak, and S. Yingchareonthawornchai. „Computing and Testing Small Connectivity in Near-Linear Time and Queries via Fast Local Cut Algorithms“. In: *SODA*. SIAM, 2020, pp. 2046–2065.
- [FGM17] K. Fountoulakis, D. F. Gleich, and M. W. Mahoney. „An Optimization Approach to Locally-Biased Graph Algorithms“. In: *Proceedings of the IEEE* 105.2 (2017), pp. 256–272.
- [Fou+16] K. Fountoulakis, F. Roosta-Khorasani, J. Shun, X. Cheng, and M. W. Mahoney. „Variational perspective on local graph clustering“. In: *Mathematical Programming* (2016), pp. 1–21.
- [FM12] A. Frank and Z. Miklós. „Simple push-relabel algorithms for matroids and submodular flows“. In: *Japan Journal of Industrial and Applied Mathematics* 29.3 (2012), pp. 419–439.
- [FJY26] X. Fu, Y. Jiang, and Y. Yin. „Perfect Simulation of Las Vegas Algorithms via Local Computation“. In: *ITCS*. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2026, 63:1–63:22.
- [Ful61] D. R. Fulkerson. „An out-of-kilter method for minimal-cost flow problems“. In: *Journal of the Society for Industrial and Applied Mathematics* 9.1 (1961), pp. 18–27.
- [Fül05] O. Fülöp. „Sparse graph certificates for mixed connectivity“. In: *Discret. Math.* 294.3 (2005), pp. 285–290.
- [Gab85] H. N. Gabow. „Scaling Algorithms for Network Problems“. In: *J. Comput. Syst. Sci.* 31.2 (1985). announced at FOCS’83, pp. 148–168.
- [Gab06] H. N. Gabow. „Using expander graphs to find vertex connectivity“. In: *J. ACM* 53.5 (2006). Announced at FOCS’00, pp. 800–844.
- [GT89] H. N. Gabow and R. E. Tarjan. „Faster Scaling Algorithms for Network Problems“. In: *SIAM J. Comput.* 18.5 (1989), pp. 1013–1036.

- [Gal80] Z. Galil. „Finding the Vertex Connectivity of Graphs“. In: *SIAM J. Comput.* 9.1 (1980), pp. 197–199.
- [GP88] Z. Galil and V. Pan. „Improved processor bounds for combinatorial problems in RNC“. In: *Combinatorica* 8.2 (1988), pp. 189–200.
- [Gal14] F. L. Gall. „Powers of tensors and fast matrix multiplication“. In: *ISSAC*. ACM, 2014, pp. 296–303.
- [GGT89] G. Gallo, M. D. Grigoriadis, and R. E. Tarjan. „A fast parametric maximum flow algorithm and applications“. In: *SIAM Journal on Computing* 18.1 (1989), pp. 30–55.
- [Geo10] L. Georgiadis. „Testing 2-Vertex Connectivity and Computing Pairs of Vertex-Disjoint s - t Paths in Digraphs“. In: *ICALP (1)*. Vol. 6198. Lecture Notes in Computer Science. Springer, 2010, pp. 738–749.
- [Geo+15] L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis. „2-Vertex Connectivity in Directed Graphs“. In: *ICALP (1)*. Vol. 9134. Lecture Notes in Computer Science. Springer, 2015, pp. 605–616.
- [GNT20] M. Ghaffari, K. Nowicki, and M. Thorup. „Faster Algorithms for Edge Connectivity via Random 2-Out Contractions“. In: *SODA*. SIAM, 2020, pp. 1260–1279.
- [GP19] M. Ghaffari and J. Portmann. „Improved Network Decompositions Using Small Messages with Applications on MIS, Neighborhood Covers, and Beyond“. In: *33rd International Symposium on Distributed Computing (DISC 2019)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2019, pp. 18–1.
- [GT24] M. Ghaffari and A. Trygub. „A near-optimal low-energy deterministic distributed SSSP with ramifications on congestion and APSP“. In: *Proceedings of the 43rd ACM Symposium on Principles of Distributed Computing*. 2024, pp. 401–411.
- [GZ22] M. Ghaffari and G. Zuzic. „Universally-Optimal Distributed Exact Min-Cut“. In: *PODC*. ACM, 2022, pp. 281–291.
- [GT12] S. O. Gharan and L. Trevisan. „Approximating the Expansion Profile and Almost Optimal Local Graph Clustering“. In: *FOCS*. IEEE Computer Society, 2012, pp. 187–196.
- [GJ25] H. Gholizadeh and Y. Jiang. „A Subquadratic Two-Party Communication Protocol for Minimum Cost Flow“. In: *CoRR* abs/2510.03427 (2025).
- [GH20] L. Gianinazzi and T. Hoeffler. „Parallel Planar Subgraph Isomorphism and Vertex Connectivity“. In: *SPAA*. ACM, 2020, pp. 269–280.
- [Gol84] A. V. Goldberg. *Finding a Maximum Density Subgraph*. Tech. rep. Berkeley, CA, USA: University of California at Berkeley, 1984.
- [GT87] A. Goldberg and R. Tarjan. „Solving minimum-cost flow problems by successive approximation“. In: *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. 1987, pp. 7–18.
- [GT90] A. V. Goldberg and R. E. Tarjan. „Finding minimum-cost circulations by successive approximation“. In: *Mathematics of Operations Research* 15.3 (1990), pp. 430–466.
- [Gol95] A. V. Goldberg. „Scaling Algorithms for the Shortest Paths Problem“. In: *SIAM J. Comput.* 24.3 (1995). Announced at SODA’93, pp. 494–504.

- [Gol+92] A. V. Goldberg, S. A. Plotkin, D. B. Shmoys, and É. Tardos. „Using Interior-Point Methods for Fast Parallel Algorithms for Bipartite Matching and Related Problems“. In: *SIAM J. Comput.* 21.1 (1992), pp. 140–150.
- [GPV93] A. V. Goldberg, S. A. Plotkin, and P. M. Vaidya. „Sublinear-Time Parallel Algorithms for Matching and Related Problems“. In: *J. Algorithms* 14.2 (1993), pp. 180–213.
- [GR98] A. V. Goldberg and S. Rao. „Beyond the Flow Decomposition Barrier“. In: *J. ACM* 45.5 (1998), pp. 783–797.
- [GT88] A. V. Goldberg and R. E. Tarjan. „A new approach to the maximum-flow problem“. In: *J. ACM* 35.4 (1988), pp. 921–940.
- [GKW21] A. Golovnev, A. S. Kulikov, and R. R. Williams. „Circuit Depth Reductions“. In: *Proceedings of the 12th Innovations in Theoretical Computer Science Conference (ITCS)*. Vol. 185. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 24:1–24:20.
- [Gor+21] G. Goranci, H. Räcke, T. Saranurak, and Z. Tan. „The expander hierarchy and its applications to dynamic graph algorithms“. In: *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 2021, pp. 2212–2228.
- [Gra+12] F. Granot, S. T. McCormick, M. Queyranne, and F. Tardella. „Structural and algorithmic properties for parametric minimum cuts“. In: *Mathematical programming* 135.1 (2012), pp. 337–367.
- [GMT15] S. Guha, A. McGregor, and D. Tench. „Vertex and Hyperedge Connectivity in Dynamic Graph Streams“. In: *PODS*. ACM, 2015, pp. 241–247.
- [GUV09] V. Guruswami, C. Umans, and S. Vadhan. „Unbalanced expanders and randomness extractors from Parvaresh–Vardy codes“. In: *Journal of the ACM (JACM)* 56.4 (2009), pp. 1–34.
- [HHZ22] B. Haeupler, D. E. Hershkowitz, and G. Zuzic. „Adaptive-Adversary-Robust Algorithms via Small Copy Tree Embeddings“. In: *30th Annual European Symposium on Algorithms (ESA 2022)*. Vol. 244. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022, p. 63.
- [Hae+24] B. Haeupler, D. E. Hershkowitz, J. Li, A. Roeykoe, and T. Saranurak. „Low-Step Multi-commodity Flow Emulators“. In: *STOC*. ACM, 2024, pp. 71–82.
- [HHS23] B. Haeupler, D. E. Hershkowitz, and T. Saranurak. „Maximum Length-Constrained Flows and Disjoint Paths: Distributed, Deterministic, and Fast“. In: *STOC*. ACM, 2023, pp. 1371–1383.
- [HHT24] B. Haeupler, D. E. Hershkowitz, and Z. Tan. „New Structures and Algorithms for Length-Constrained Expander Decompositions“. In: *FOCS*. IEEE, 2024, pp. 1634–1645.
- [Hae+25a] B. Haeupler, Y. Jiang, Y. Long, T. Saranurak, and S. Wang. „Parallel $(1+\epsilon)$ -Approximate Multi-Commodity Min-Cost Flow in Almost Optimal Depth and Work“. In: *FOCS*. IEEE, 2025, pp. 1883–1893.
- [HJS25] B. Haeupler, Y. Jiang, and T. Saranurak. „Deterministic Negative-Weight Shortest Paths in Nearly Linear Time via Path Covers“. In: *CoRR* abs/2511.08551 (2025).
- [HJS26a] B. Haeupler, Y. Jiang, and T. Saranurak. „DAG Projections: Reducing Distance and Flow Problems to DAGs“. In: *Proceedings of the Annual ACM Symposium on Theory of Computing (STOC 2026)*. To appear, 2026.
- [HJS26b] B. Haeupler, Y. Jiang, and T. Saranurak. „Reducing Shortcut and Hopset Constructions to Shallow Graphs“. In: *SOSA*. SIAM, 2026, pp. 385–393.

- [HLS24] B. Haeupler, Y. Long, and T. Saranurak. „Dynamic Deterministic Constant-Approximate Distance Oracles with n^ε Worst-Case Update Time“. In: *2024 IEEE 65th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 2024, pp. 2033–2044.
- [Hae+25b] B. Haeupler, Y. Long, T. Saranurak, and S. Wang. „Length-Constrained Directed Expander Decomposition and Length-Constrained Vertex-Capacitated Flow Shortcuts“. In: *ESA*. Vol. 351. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025, 107:1–107:17.
- [HRG22] B. Haeupler, H. Räcke, and M. Ghaffari. „Hop-constrained expander decompositions, oblivious routing, and distributed universal optimality“. In: *STOC*. ACM, 2022, pp. 1325–1338.
- [HWZ20] B. Haeupler, D. Wajc, and G. Zuzic. „Network Coding Gaps for Completion Times of Multiple Unicasts“. In: *FOCS*. IEEE, 2020, pp. 494–505.
- [HMT88] A. Hajnal, W. Maass, and G. Turán. „On the Communication Complexity of Graph Properties“. In: *STOC*. ACM, 1988, pp. 186–191.
- [HO94] J. Hao and J. B. Orlin. „A faster algorithm for finding the minimum cut in a directed graph“. In: *Journal of Algorithms* 17.3 (1994), pp. 424–446.
- [HHR03] C. Harrelson, K. Hildrum, and S. Rao. „A polynomial-time tree decomposition to minimize congestion“. In: *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*. 2003, pp. 34–43.
- [HJ13] P. Harsha and R. Jain. „A Strong Direct Product Theorem for the Tribes Function via the Smooth-Rectangle Bound“. In: *FSTTCS*. Vol. 24. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013, pp. 141–152.
- [Har08] N. J. A. Harvey. „Matroid intersection, pointer chasing, and Young’s seminormal representation of S_n “. In: *SODA*. SIAM, 2008, pp. 542–549.
- [Hås86] J. Håstad. „Almost Optimal Lower Bounds for Small Depth Circuits“. In: *Proceedings of the 18th Annual ACM Symposium on Theory of Computing (STOC)*. 1986, pp. 6–20.
- [HLW21] Z. He, J. Li, and M. Wahlström. „Near-Linear-Time, Optimal Vertex Cut Sparsifiers in Directed Acyclic Graphs“. In: *29th Annual European Symposium on Algorithms (ESA 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2021, pp. 52–1.
- [HHS24] Z. He, S. Huang, and T. Saranurak. „Cactus Representation of Minimum Cuts: Derandomize and Speed up“. In: *SODA*. SIAM, 2024, pp. 1503–1541.
- [HKN14] M. Henzinger, S. Krinninger, and D. Nanongkai. „A Subquadratic-Time Algorithm for Decremental Single-Source Shortest Paths“. In: *SODA*. SIAM, 2014, pp. 1053–1072.
- [Hen+24] M. Henzinger, J. Li, S. Rao, and D. Wang. „Deterministic near-linear time minimum cut in weighted graphs“. In: *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 2024, pp. 3089–3139.
- [HRW17] M. Henzinger, S. Rao, and D. Wang. „Local Flow Partitioning for Faster Edge Connectivity“. In: *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*. 2017, pp. 1919–1938.
- [Hen97] M. R. Henzinger. „A Static 2-Approximation Algorithm for Vertex Connectivity and Incremental Approximation Algorithms for Edge and Vertex Connectivity“. In: *J. Algorithms* 24.1 (1997), pp. 194–220.

- [Hen+97] M. R. Henzinger, P. N. Klein, S. Rao, and S. Subramanian. „Faster Shortest-Path Algorithms for Planar Graphs“. In: *J. Comput. Syst. Sci.* 55.1 (1997). Announced at STOC'94, pp. 3–23.
- [HRG00] M. R. Henzinger, S. Rao, and H. N. Gabow. „Computing Vertex Connectivity: New Bounds from Old Techniques“. In: *J. Algorithms* 34.2 (2000). Announced at FOCS'96, pp. 222–250.
- [Hes03] W. Hesse. „Directed graphs requiring large numbers of shortcuts“. In: *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2003, pp. 665–669.
- [Hir64a] H. Hironaka. „Resolution of Singularities of an Algebraic Variety over a Field of Characteristic Zero: I“. In: *Annals of Mathematics* 79.1 (1964), pp. 109–203.
- [Hir64b] H. Hironaka. „Resolution of Singularities of an Algebraic Variety over a Field of Characteristic Zero: II“. In: *Annals of Mathematics* 79.2 (1964), pp. 205–326.
- [HT73] J. E. Hopcroft and R. E. Tarjan. „Dividing a Graph into Triconnected Components“. In: *SIAM J. Comput.* 2.3 (1973), pp. 135–158.
- [HSW25] G. Hoppenworth, T. Saranurak, and B. Wang. „Near-Optimal Fault-Tolerant Strong Connectivity Preservers“. In: *2025 IEEE 66th Annual Symposium on Foundations of Computer Science (FOCS)*. 2025, pp. 1610–1638.
- [HXX25] G. Hoppenworth, Y. Xu, and Z. Xu. „New Separations and Reductions for Directed Hopsets and Preservers“. In: *SODA*. SIAM, 2025, pp. 4405–4443.
- [HP19] S.-E. Huang and S. Pettie. „Thorup-Zwick emulators are universally optimal hopsets“. In: *Information Processing Letters* 142 (2019), pp. 9–13.
- [HP21] S. Huang and S. Pettie. „Lower Bounds on Sparse Spanners, Emulators, and Diameter-Reducing Shortcuts“. In: *SIAM J. Discret. Math.* 35.3 (2021), pp. 2129–2144.
- [Hua+23] Z. Huang, Y. Long, T. Saranurak, and B. Wang. „Tight Conditional Lower Bounds for Vertex Connectivity Problems“. In: *STOC*. ACM, 2023, pp. 1384–1395.
- [Ind02] P. Indyk. „Explicit constructions of selectors and related combinatorial structures, with applications“. In: *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '02. San Francisco, California: Society for Industrial and Applied Mathematics, 2002, pp. 697–704.
- [Ind06] P. Indyk. „Stable distributions, pseudorandom generators, embeddings, and data stream computation“. In: *J. ACM* 53.3 (2006), pp. 307–323.
- [ILS10] G. F. Italiano, L. Laura, and F. Santaroni. „Finding Strong Bridges and Strong Articulation Points in Linear Time“. In: *COCOA (1)*. Vol. 6508. Lecture Notes in Computer Science. Springer, 2010, pp. 157–169.
- [Iva+12] G. Ivanyos, H. Klauck, T. Lee, M. Santha, and R. de Wolf. „New bounds on the classical and quantum communication complexity of some graph properties“. In: *FSTTCS*. Vol. 18. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012, pp. 148–159.
- [Iwa03] S. Iwata. „A faster scaling algorithm for minimizing submodular functions“. In: *SIAM Journal on Computing* 32.4 (2003), pp. 833–840.
- [JLS19] A. Jambulapati, Y. P. Liu, and A. Sidford. „Parallel Reachability in Almost Linear Work and Square Root Depth“. In: *Proceedings of the 60th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. 2019, pp. 1664–1686.

- [JKS03] T. S. Jayram, R. Kumar, and D. Sivakumar. „Two applications of information complexity“. In: *STOC*. ACM, 2003, pp. 673–682.
- [Jia21] H. Jiang. „Minimizing Convex Functions with Integral Minimizers“. In: *SODA*. SIAM, 2021, pp. 976–985.
- [Jia+25a] Y. Jiang, Y. Long, T. Saranurak, and B. Wang. „Approximating Directed Minimum Cut and Arborescence Packing via Directed Expander Hierarchies“. In: *arXiv preprint arXiv:2512.05300* (2025).
- [JM23] Y. Jiang and S. Mukhopadhyay. „Finding a Small Vertex Cut on Distributed Networks“. In: *STOC*. ACM, 2023, pp. 1791–1801.
- [Jia+25b] Y. Jiang, C. Nalam, T. Saranurak, and S. Yingchareonthawornchai. „Deterministic Vertex Connectivity via Common-Neighborhood Clustering and Pseudorandomness“. In: *STOC*. ACM, 2025, pp. 2293–2304.
- [JNS26] Y. Jiang, D. Nanongkai, and P. Sawettamalya. „Minimum s t Cuts with Fewer Cut Queries“. In: *SODA*. SIAM, 2026, pp. 258–296.
- [JPP26] Y. Jiang, M. Parter, and A. Petruschka. „New Oracles and Labeling Schemes for Vertex Cut Queries“. In: *SODA*. SIAM, 2026, pp. 2760–2791.
- [JY25] Y. Jiang and C. Yun. „Parallel Small Vertex Connectivity in Near-Linear Work and Polylogarithmic Depth“. In: *CoRR* abs/2504.06033 (2025).
- [Joh77] D. B. Johnson. „Efficient Algorithms for Shortest Paths in Sparse Networks“. In: *J. ACM* 24.1 (1977), pp. 1–13.
- [KR91] A. Kanevsky and V. Ramachandran. „Improved Algorithms for Graph Four-Connectivity“. In: *J. Comput. Syst. Sci.* 42.3 (1991). announced at FOCS’87, pp. 288–306.
- [KVV04] R. Kannan, S. Vempala, and A. Vetta. „On clusterings: Good, bad and spectral“. In: *J. ACM* 51.3 (2004), pp. 497–515.
- [KK93] M. Kao and P. N. Klein. „Towards Overcoming the Transitive-Closure Bottleneck: Efficient Parallel Algorithms for Planar Digraphs“. In: *J. Comput. Syst. Sci.* 47.3 (1993), pp. 459–500.
- [KÖ22] G. Kara and C. C. Özturan. „Parallel network simplex algorithm for the minimum cost flow problem“. In: *Concurr. Comput. Pract. Exp.* 34.4 (2022).
- [KS21] A. Karczmarz and P. Sankowski. „A deterministic parallel APSP algorithm and its applications“. In: *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2021, pp. 255–272.
- [Kar99] D. R. Karger. „Random Sampling in Cut, Flow, and Network Design Problems“. In: *Math. Oper. Res.* 24.2 (1999), pp. 383–413.
- [Kar00] D. R. Karger. „Minimum cuts in near-linear time“. In: *J. ACM* 47.1 (2000). announced at STOC’96, pp. 46–76.
- [KUW86] R. M. Karp, E. Upfal, and A. Wigderson. „Constructing a perfect matching is in random NC“. In: *Comb.* 6.1 (1986), pp. 35–48.
- [KP21] C. Karthik and M. Parter. „Deterministic replacement path covering“. In: *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM. 2021, pp. 704–723.
- [KT15] K. Kawarabayashi and M. Thorup. „Deterministic Global Minimum Cut of a Simple Graph in Near-Linear Time“. In: *STOC*. ACM, 2015, pp. 665–674.

- [KRV09] R. Khandekar, S. Rao, and U. Vazirani. „Graph partitioning using single commodity flows“. In: *Journal of the ACM (JACM)* 56.4 (2009), pp. 1–15.
- [KK22] S. Khanna and C. Konrad. „Optimal Bounds for Dominating Set in Graph Streams“. In: *ITCS*. Vol. 215. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 93:1–93:23.
- [Kle67] M. Klein. „A primal method for minimal cost flows with applications to the assignment and transportation problems“. In: *Management Science* 14.3 (1967), pp. 205–220.
- [KS93] P. N. Klein and S. Subramanian. *A linear-processor polylog-time algorithm for shortest paths in planar graphs*. IEEE, 1993.
- [Kle93] P. N. Klein. „Parallelism, Preprocessing, and Reachability: A Hybrid Algorithm for Directed Graphs“. In: *J. Algorithms* 14.3 (1993), pp. 331–343.
- [KMW10] P. N. Klein, S. Mozes, and O. Weimann. „Shortest paths in directed planar graphs with negative lengths: A linear-space $O(n \log^2 n)$ -time algorithm“. In: *ACM Trans. Algorithms* 6.2 (2010). Announced at SODA’09, 30:1–30:18.
- [KS97] P. N. Klein and S. Subramanian. „A Randomized Parallel Algorithm for Single-Source Shortest Paths“. In: *J. Algorithms* 25.2 (1997), pp. 205–220.
- [Kle69] D. Kleitman. „Methods for investigating connectivity of large graphs“. In: *IEEE Transactions on Circuit Theory* 16.2 (1969), pp. 232–233.
- [KP22a] S. Kogan and M. Parter. „Beating Matrix Multiplication for $n^{1/3}$ -Directed Shortcuts“. In: *Proceedings of the 49th International Colloquium on Automata, Languages, and Programming (ICALP)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik. 2022.
- [KP22b] S. Kogan and M. Parter. „New Diameter-Reducing Shortcuts and Directed Hopsets: Breaking the Barrier“. In: *Proceedings of the 33rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2022, pp. 1326–1341.
- [KP23] S. Kogan and M. Parter. „Faster and unified algorithms for diameter reducing shortcuts and minimum chain covers“. In: *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2023, pp. 212–239.
- [Kor24] T. Korhonen. „Linear-Time Algorithms for k-Edge-Connected Components, k-Lean Tree Decompositions, and More“. In: *CoRR* abs/2411.02658 (2024).
- [KN18] S. Kratsch and F. Nelles. „Efficient and Adaptive Parameterized Algorithms on Modular Decompositions“. In: *ESA*. Vol. 112. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018, 55:1–55:15.
- [KW12] S. Kratsch and M. Wahlström. „Representative sets and irrelevant vertices: New tools for kernelization“. In: *2012 IEEE 53rd Annual Symposium on Foundations of Computer Science*. IEEE. 2012, pp. 450–459.
- [KT17] R. Krauthgamer and O. Trabelsi. „Conditional Lower Bounds for All-Pairs Max-Flow“. In: *ICALP*. Vol. 80. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, 20:1–20:13.
- [KP98] S. Kutten and D. Peleg. „Fast Distributed Construction of Small k -Dominating Sets and Applications“. In: *J. Algorithms* 28.1 (1998), pp. 40–66.
- [KSZ20] M. Kwan, L. Sauermann, and Y. Zhao. „Extension complexity of low-dimensional polytopes“. In: *CoRR* abs/2006.08836 (2020).

- [LR04] K. J. Lang and S. Rao. „A Flow-Based Method for Improving the Expansion or Conductance of Graph Cuts“. In: *IPCO*. Vol. 3064. Lecture Notes in Computer Science. Springer, 2004, pp. 325–337.
- [Lee+11] T. Lee, R. Mittal, B. W. Reichardt, R. Spalek, and M. Szegedy. „Quantum Query Complexity of State Conversion“. In: *FOCS*. IEEE Computer Society, 2011, pp. 344–353.
- [LS14] Y. T. Lee and A. Sidford. „Path Finding Methods for Linear Programming: Solving Linear Programs in $\tilde{O}(\sqrt{\text{rank}})$ Iterations and Faster Algorithms for Maximum Flow“. In: *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18–21, 2014*. 2014, pp. 424–433.
- [LS19] Y. T. Lee and A. Sidford. „Solving linear programs with $\sqrt{\text{rank}}$ linear system solves“. In: *arXiv preprint arXiv:1910.08033* (2019).
- [LP02] H. W. Lenstra and C. Pomerance. „Primality Testing with Gaussian Periods“. In: *FST TCS 2002: Foundations of Software Technology and Theoretical Computer Science*. Ed. by M. Agrawal and A. Seth. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 1–1.
- [LS10] N. Leonardos and M. E. Saks. „Lower Bounds on the Randomized Communication Complexity of Read-Once Functions“. In: *Comput. Complex.* 19.2 (2010), pp. 153–181.
- [Li20] J. Li. „Faster parallel algorithm for approximate shortest path“. In: *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*. 2020, pp. 308–321.
- [Li21] J. Li. „Deterministic mincut in almost-linear time“. In: *STOC*. ACM, 2021, pp. 384–395.
- [Li25] J. Li. *Deterministic Padded Decompositions and Negative-Weight Shortest Paths*. 2025.
- [LM24] J. Li and C. Mowry. „A Bottom-Up Algorithm for Negative-Weight SSSP with Integrated Negative Cycle Finding“. In: *arXiv preprint arXiv:2411.19449* (2024).
- [Li+21] J. Li, D. Nanongkai, D. Panigrahi, T. Saranurak, and S. Yingchareonthawornchai. „Vertex connectivity in poly-logarithmic max-flows“. In: *STOC*. ACM, 2021, pp. 317–329.
- [LP20] J. Li and D. Panigrahi. „Deterministic Min-cut in Poly-logarithmic Max-flows“. In: *FOCS*. IEEE, 2020, pp. 85–92.
- [LP21] J. Li and D. Panigrahi. „Approximate Gomory-Hu tree is faster than $n - 1$ max-flows“. In: *STOC*. ACM, 2021, pp. 1738–1748.
- [LRW24] J. Li, S. Rao, and D. Wang. „Congestion-Approximators from the Bottom Up“. In: *arXiv preprint arXiv:2407.04976* (2024).
- [LS21] J. Li and T. Saranurak. „Deterministic Weighted Expander Decomposition in Almost-linear Time“. In: *CoRR* abs/2106.01567 (2021).
- [Lin92] N. Linial. „Locality in Distributed Graph Algorithms“. In: *SIAM J. Comput.* 21.1 (1992), pp. 193–201.
- [LLW88] N. Linial, L. Lovász, and A. Wigderson. „Rubber bands, convex embeddings and graph connectivity“. In: *Combinatorica* 8.1 (1988). Announced at FOCS’86, pp. 91–102.
- [LS93] N. Linial and M. E. Saks. „Low diameter graph decompositions“. In: *Comb.* 13.4 (1993), pp. 441–454.
- [LRT79] R. J. Lipton, D. J. Rose, and R. E. Tarjan. „Generalized nested dissection“. In: *SIAM journal on numerical analysis* 16.2 (1979), pp. 346–358.

- [Liu+23] S. C. Liu, Z. Song, H. Zhang, L. Zhang, and T. Zhou. „Space-Efficient Interior Point Method, with Applications to Linear Programming and Maximum Weight Bipartite Matching“. In: *ICALP*. Vol. 261. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 88:1–88:14.
- [LS20] Y. P. Liu and A. Sidford. „Faster Energy Maximization for Faster Maximum Flow“. In: *STOC*. <https://arxiv.org/pdf/1910.14276.pdf>, 2020.
- [LM19] B. Loff and S. Mukhopadhyay. „Lifting Theorems for Equality“. In: *STACS*. Vol. 126. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 50:1–50:19.
- [LS22] Y. Long and T. Saranurak. „Near-Optimal Deterministic Vertex-Failure Connectivity Oracles“. In: *FOCS*. IEEE, 2022, pp. 1002–1010.
- [LMN21] A. López-Martínez, S. Mukhopadhyay, and D. Nanongkai. „Work-Optimal Parallel Minimum Cuts for Non-Sparse Graphs“. In: *SPAA*. ACM, 2021, pp. 351–361.
- [LPP15] Z. Lotker, B. Patt-Shamir, and S. Pettie. „Improved distributed approximate matching“. In: *Journal of the ACM (JACM)* 62.5 (2015), pp. 1–17.
- [Lov79] L. Lovász. „On determinants, matchings, and random algorithms“. In: *FCT*. Akademie-Verlag, Berlin, 1979, pp. 565–574.
- [Lu+22] K. Lu, V. Vassilevska Williams, N. Wein, and Z. Xu. „Better Lower Bounds for Shortcut Sets and Additive Spanners via an Improved Alternation Product“. In: *Proceedings of the 2022 ACM-SIAM Symposium on Discrete Algorithms, (SODA)*. 2022, pp. 3311–3331.
- [LPS86] A. Lubotzky, R. Phillips, and P. Sarnak. „Explicit Expanders and the Ramanujan Conjectures“. In: *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*. STOC ’86. 1986, pp. 240–246.
- [Mad11] A. Madry. „From graphs to matrices, and back: new techniques for graph algorithms“. PhD thesis. Massachusetts Institute of Technology, 2011.
- [Mad13] A. Madry. „Navigating Central Path with Electrical Flows: From Flows to Matchings, and Back“. In: *FOCS*. IEEE Computer Society, 2013, pp. 253–262.
- [Mad16] A. Madry. „Computing maximum flow with augmenting electrical flows“. In: *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE. 2016, pp. 593–602.
- [Mar88] G. A. Margulis. „Explicit group-theoretic constructions of combinatorial schemes and their applications in the construction of expanders and concentrators“. In: *Problemy Peredachi Informatsii* 24 (1988).
- [Mat87] D. W. Matula. „Determining Edge Connectivity in $O(nm)$ “. In: *FOCS*. IEEE Computer Society, 1987, pp. 249–251.
- [MN07] M. Mendel and A. Naor. „Ramsey partitions and proximity data structures“. In: *Journal of the European Mathematical Society* 9.2 (2007), pp. 253–275.
- [Men27] K. Menger. „Zur allgemeinen Kurventheorie“. In: *Fund. Math.* 10:1 (1927), pp. 96–115.
- [MPX13] G. L. Miller, R. Peng, and S. C. Xu. „Parallel graph decompositions using random shifts“. In: *SPAA*. ACM, 2013, pp. 196–203.
- [MR87] G. L. Miller and V. Ramachandran. „A New Graph Triconnectivity Algorithm and Its Parallelization“. In: *STOC*. ACM, 1987, pp. 335–344.
- [Moo59] E. F. Moore. „The Shortest Path Through a Maze“. In: *Proceedings of the International Symposium on the Theory of Switching*. 1959, pp. 285–292.

- [MW10] S. Mozes and C. Wulff-Nilsen. „Shortest Paths in Planar Graphs with Real Lengths in $O(n \log^2 n / \log \log n)$ Time“. In: *ESA (2)*. Vol. 6347. Lecture Notes in Computer Science. Springer, 2010, pp. 206–217.
- [MN20] S. Mukhopadhyay and D. Nanongkai. „Weighted min-cut: sequential, cut-query, and streaming algorithms“. In: *STOC*. ACM, 2020, pp. 496–509.
- [MN21] S. Mukhopadhyay and D. Nanongkai. „A Note on Isolating Cut Lemma for Submodular Function Minimization“. In: *CoRR* abs/2103.15724 (2021).
- [MVV87] K. Mulmuley, U. V. Vazirani, and V. V. Vazirani. „Matching is as easy as matrix inversion“. In: *Comb.* 7.1 (1987), pp. 105–113.
- [NI92] H. Nagamochi and T. Ibaraki. „A Linear-Time Algorithm for Finding a Sparse k -Connected Spanning Subgraph of a k -Connected Graph“. In: *Algorithmica* 7.5&6 (1992), pp. 583–596.
- [NSY23] C. Nalam, T. Saranurak, and S. Yingchareonthawornchai. „Deterministic k -Vertex Connectivity in k^2 Max-flows“. In: *CoRR* abs/2308.04695 (2023).
- [Nan24] D. Nanongkai. „Cross-Paradigm Graph Algorithms (Invited Talk)“. In: *ICALP*. Vol. 297. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, 3:1–3:1.
- [NS17] D. Nanongkai and T. Saranurak. „Dynamic spanning forest with worst-case update time: adaptive, Las Vegas, and $O(n^{1/2} - \epsilon)$ -time“. In: *STOC*. ACM, 2017, pp. 1122–1129.
- [NSW17] D. Nanongkai, T. Saranurak, and C. Wulff-Nilsen. „Dynamic Minimum Spanning Forest with Subpolynomial Worst-Case Update Time“. In: *FOCS*. IEEE Computer Society, 2017, pp. 950–961.
- [NSY19a] D. Nanongkai, T. Saranurak, and S. Yingchareonthawornchai. „Breaking quadratic time for small vertex connectivity and an approximation scheme“. In: *STOC*. ACM, 2019, pp. 241–252.
- [NSY19b] D. Nanongkai, T. Saranurak, and S. Yingchareonthawornchai. „Computing and Testing Small Vertex Connectivity in Near-Linear Time and Queries“. In: *CoRR* abs/1905.05329 (2019).
- [New91] I. Newman. „Private vs. Common Random Bits in Communication Complexity“. In: *Inf. Process. Lett.* 39.2 (1991), pp. 67–71.
- [OSV12] L. Orecchia, S. Sachdeva, and N. K. Vishnoi. „Approximating the exponential, the lanczos method and an $\tilde{O}(m)$ -time spectral algorithm for balanced separator“. In: *STOC*. ACM, 2012, pp. 1141–1160.
- [OV11] L. Orecchia and N. K. Vishnoi. „Towards an SDP-based Approach to Spectral Methods: A Nearly-Linear-Time Algorithm for Graph Partitioning and Decomposition“. In: *SODA*. SIAM, 2011, pp. 532–545.
- [OZ14] L. Orecchia and Z. A. Zhu. „Flow-Based Algorithms for Local Graph Clustering“. In: *SODA*. SIAM, 2014, pp. 1267–1286.
- [Orl88] J. Orlin. „A faster strongly polynomial minimum cost flow algorithm“. In: *Proceedings of the Twentieth annual ACM symposium on Theory of Computing*. 1988, pp. 377–387.

- [Pac+18] J. Pachocki, L. Roditty, A. Sidford, R. Tov, and V. V. Williams. „Approximating Cycles in Directed Graphs: Fast Algorithms for Girth and Roundtrip Spanners“. In: *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*. Ed. by A. Czumaj. SIAM, 2018, pp. 1374–1392.
- [Par19] M. Parter. „Small Cuts and Connectivity Certificates: A Fault Tolerant Approach“. In: *DISC*. Vol. 146. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 30:1–30:16.
- [Par20] M. Parter. „Distributed Planar Reachability in Nearly Optimal Time“. In: *34th International Symposium on Distributed Computing (DISC 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2020.
- [PP22] M. Parter and A. Petruschka. „Near-Optimal Distributed Computation of Small Vertex Cuts“. In: *36th International Symposium on Distributed Computing (DISC 2022)*. 2022.
- [Pen16] R. Peng. „Approximate undirected maximum flows in $O(m \text{polylog}(n))$ time“. In: *Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2016, pp. 1862–1867.
- [PR02] S. Pettie and V. Ramachandran. „An optimal minimum spanning tree algorithm“. In: *J. ACM* 49.1 (2002), pp. 16–34.
- [Pod73] V. Podderiyugin. „An algorithm for finding the edge connectivity of graphs“. In: *Vopr. Kibern* 2 (1973), p. 136.
- [PL07] E. Porat and O. Lipsky. „Improved Sketching of Hamming Distance with Error Correcting“. In: *Combinatorial Pattern Matching*. Ed. by B. Ma and K. Zhang. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 173–182.
- [PT11] D. Pritchard and R. Thurimella. „Fast computation of small cuts via cycle space sampling“. In: *ACM Trans. Algorithms* 7.4 (2011), 46:1–46:30.
- [Qua24] K. Quanrud. „Faster exact and approximation algorithms for packing and covering matroids via push-relabel“. In: *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM. 2024, pp. 2305–2336.
- [Rac02] H. Racke. „Minimizing congestion in general networks“. In: *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings*. IEEE. 2002, pp. 43–52.
- [Räc08] H. Räcke. „Optimal hierarchical decompositions for congestion minimization in networks“. In: *Proceedings of the fortieth annual ACM symposium on Theory of computing*. 2008, pp. 255–264.
- [RS14] H. Räcke and C. Shah. „Improved guarantees for tree cut sparsifiers“. In: *European Symposium on Algorithms*. Springer. 2014, pp. 774–785.
- [RST14] H. Räcke, C. Shah, and H. Täubig. „Computing Cut-Based Hierarchical Decompositions in Almost Linear Time“. In: *SODA*. SIAM, 2014, pp. 227–238.
- [Ram87] V. Ramachandran. „The complexity of minimum cut and maximum flow problems in an acyclic network“. In: *Networks* 17.4 (1987), pp. 387–392.
- [Ras10] S. Raskhodnikova. „Transitive-Closure Spanners: A Survey“. In: *Property Testing — Current Research and Surveys*. Ed. by O. Goldreich. Vol. 6390. Lecture Notes in Computer Science. Springer, 2010, pp. 167–196.

- [RM97] R. Raz and P. McKenzie. „Separation of the monotone NC hierarchy“. In: *Proceedings 38th Annual Symposium on Foundations of Computer Science (FOCS)*. 1997, pp. 234–243.
- [Rei11] B. Reichardt. „Reflections for quantum query algorithms“. In: *SODA*. SIAM, 2011, pp. 560–569.
- [RW12] L. Roditty and V. V. Williams. „Subquadratic time approximation algorithms for the girth“. In: *SODA*. SIAM, 2012, pp. 833–845.
- [Roz+22a] V. Rozhon, M. Elkin, C. Grunau, and B. Haeupler. „Deterministic Low-Diameter Decompositions for Weighted Graphs and Distributed and Parallel Applications“. In: *FOCS*. IEEE, 2022, pp. 1114–1121.
- [Roz+22b] V. Rozhon, C. Grunau, B. Haeupler, G. Zuzic, and J. Li. „Undirected $(1+\epsilon)$ -shortest paths via minor-aggregates: near-optimal deterministic parallel and distributed algorithms“. In: *STOC*. ACM, 2022, pp. 478–487.
- [Roz+23] V. Rozhon, B. Haeupler, A. Martinsson, C. Grunau, and G. Zuzic. „Parallel Breadth-First Search and Exact Shortest Paths and Stronger Notions for Approximate Distances“. In: *STOC*. ACM, 2023, pp. 321–334.
- [RSW18] A. Rubinstein, T. Schramm, and S. M. Weinberg. „Computing Exact Minimum Cuts Without Knowing the Graph“. In: *ITCS*. Vol. 94. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 39:1–39:16.
- [San05] P. Sankowski. *Algorithms – ESA 2005: 13th Annual European Symposium, Palma de Mallorca, Spain, October 3-6, 2005. Proceedings*. pages 770–778. Springer Berlin Heidelberg, Berlin, Heidelberg: chapter Shortest Paths in Matrix Multiplication Time, 2005.
- [San07] P. Sankowski. „Faster dynamic matchings and vertex connectivity“. In: *SODA*. SIAM, 2007, pp. 118–126.
- [Sar21] T. Saranurak. „A Simple Deterministic Algorithm for Edge Connectivity“. In: *SOSA*. SIAM, 2021, pp. 80–85.
- [SW19] T. Saranurak and D. Wang. „Expander Decomposition and Pruning: Faster, Stronger, and Simpler“. In: *SODA*. SIAM, 2019, pp. 2616–2635.
- [SY22] T. Saranurak and S. Yingchareonthawornchai. „Deterministic Small Vertex Connectivity in Almost Linear Time“. In: *FOCS*. IEEE, 2022, pp. 789–800.
- [Sch82] G. Schnitger. „A Family of Graphs with Expensive Depth Reduction“. In: *Theor. Comput. Sci.* 18 (1982), pp. 89–93.
- [Sch83] G. Schnitger. „On Depth-Reduction and Grates“. In: *Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. 1983, pp. 323–328.
- [Sch03] A. Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*. Algorithms and Combinatorics v. 1. Springer, 2003.
- [SN21] I. Shabat and O. Neiman. „A unified framework for hopsets and spanners“. In: *arXiv preprint arXiv:2108.09673* (2021).
- [She13] J. Sherman. „Nearly Maximum Flows in Nearly Linear Time“. In: *FOCS*. IEEE Computer Society, 2013, pp. 263–269.

- [She17] J. Sherman. „Area-convexity, l_1 regularization, and undirected multicommodity flow“. In: *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*. 2017, pp. 452–460.
- [SS99] H. Shi and T. H. Spencer. „Time–work tradeoffs of the single-source shortest paths problem“. In: *Journal of algorithms* 30.1 (1999), pp. 19–32.
- [Shi55] A. Shimbel. „Structure in Communication Nets“. In: *In Proceedings of the Symposium on Information Networks*. Brooklyn: Polytechnic Press of the Polytechnic Institute of Brooklyn, 1955, pp. 199–203.
- [Sho88] V. Shoup. „New algorithms for finding irreducible polynomials over finite fields“. In: *[Proceedings 1988] 29th Annual Symposium on Foundations of Computer Science*. 1988, pp. 283–290.
- [ST83] D. D. Sleator and R. E. Tarjan. „A Data Structure for Dynamic Trees“. In: *J. Comput. Syst. Sci.* 26.3 (1983), pp. 362–391.
- [Spe97] T. H. Spencer. „Time-work tradeoffs for parallel algorithms“. In: *J. ACM* 44.5 (1997), pp. 742–778.
- [ST04] D. A. Spielman and S. Teng. „Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems“. In: *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*. 2004, pp. 81–90.
- [ST11] D. A. Spielman and S. Teng. „Spectral Sparsification of Graphs“. In: *SIAM J. Comput.* 40.4 (2011), pp. 981–1025.
- [ST13] D. A. Spielman and S. Teng. „A Local Clustering Algorithm for Massive Graphs and Its Application to Nearly Linear Time Graph Partitioning“. In: *SIAM J. Comput.* 42.1 (2013), pp. 1–26.
- [SW15] X. Sun and D. P. Woodruff. „Tight Bounds for Graph Problems in Insertion Streams“. In: *APPROX-RANDOM*. Vol. 40. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015, pp. 435–448.
- [Tan+21] T. Tanaka, Y. Suzuki, S. Uno, R. Raymond, T. Onodera, and N. Yamamoto. „Amplitude estimation via maximum likelihood on noisy quantum computer“. In: *Quantum Inf. Process.* 20.9 (2021), p. 293.
- [Tar85] É. Tardos. „A strongly polynomial minimum cost circulation algorithm“. In: *Combinatorica* 5.3 (1985), pp. 247–255.
- [Tar72] R. E. Tarjan. „Depth-First Search and Linear Graph Algorithms“. In: *SIAM J. Comput.* 1.2 (1972). Announced at FOCS’71, pp. 146–160.
- [TUZ01] A. TaShma, C. Umans, and D. Zuckerman. „Loss-less condensers, unbalanced expanders, and extractors“. In: *Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing*. STOC ’01. Hersonissos, Greece: Association for Computing Machinery, 2001, pp. 143–152.
- [Tho92] M. Thorup. „On Shortcutting Digraphs“. In: *Proceedings of the 18th International Workshop on Graph-Theoretic Concepts in Computer Science (WG)*. Vol. 657. Lecture Notes in Computer Science. Springer, 1992, pp. 205–211.
- [Tho95] M. Thorup. „Shortcutting planar digraphs“. In: *Combinatorics, Probability and Computing* 4.3 (1995), pp. 287–315.

- [Tho99] M. Thorup. „Undirected Single-Source Shortest Paths with Positive Integer Weights in Linear Time“. In: *J. ACM* 46.3 (1999). Announced at FOCS’97, pp. 362–394.
- [Tho04] M. Thorup. „Compact oracles for reachability and approximate distances in planar digraphs“. In: *J. ACM* 51.6 (2004), pp. 993–1024.
- [TZ01] M. Thorup and U. Zwick. „Compact routing schemes“. In: *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*. 2001, pp. 1–10.
- [TZ05] M. Thorup and U. Zwick. „Approximate distance oracles“. In: *J. ACM* 52.1 (2005), pp. 1–24.
- [Thu97] R. Thurimella. „Sub-Linear Distributed Algorithms for Sparse Certificates and Biconnected Components“. In: *J. Algorithms* 23.1 (1997). Announced at PODC’95, pp. 160–179.
- [UY91] J. D. Ullman and M. Yannakakis. „High-Probability Parallel Transitive-Closure Algorithms“. In: *SIAM J. Comput.* 20.1 (1991), pp. 100–125.
- [Vad12] S. P. Vadhan. *Pseudorandomness*. Hanover, MA, USA: Now Publishers Inc., 2012.
- [Van+24] J. Van Den Brand, L. Chen, R. Kyng, Y. P. Liu, S. Meierhans, M. P. Gutenberg, and S. Sachdeva. „Almost-linear time algorithms for decremental graphs: Min-cost flow and more via duality“. In: *2024 IEEE 65th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE. 2024, pp. 2010–2032.
- [VXX24] V. Vassilevska Williams, Y. Xu, and Z. Xu. „Simpler and Higher Lower Bounds for Shortcut Sets“. In: *Proceedings of the 2024 ACM-SIAM Symposium on Discrete Algorithms, (SODA)*. 2024, pp. 2643–2656.
- [VGM16] N. Veldt, D. F. Gleich, and M. W. Mahoney. „A Simple and Strongly-Local Flow-Based Method for Cut Improvement“. In: *Proceedings of the 33rd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*. 2016, pp. 1938–1947.
- [Vos23] T. de Vos. „Brief Announcement: Minimum Cost Maximum Flow in the CONGEST Model“. In: *PODC*. ACM, 2023, pp. 71–74.
- [Wan+17] D. Wang, K. Fountoulakis, M. Henzinger, M. W. Mahoney, and S. Rao. „Capacity Releasing Diffusion for Speed and Locality“. In: *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*. 2017, pp. 3598–3607.
- [Wie19] C. Wie. „Simpler quantum counting“. In: *Quantum Inf. Comput.* 19.11&12 (2019), pp. 967–983.
- [WZ99] A. Wigderson and D. Zuckerman. „Expanders That Beat the Eigenvalue Bound: Explicit Construction and Applications“. In: *Combinatorica* 19.1 (1999), pp. 125–138.
- [Wil19] D. Williamson. *Network Flow Algorithms*. Cambridge University Press, 2019.
- [Woo49] M. A. Woodbury. „The stability of out-input matrices“. In: *Chicago, IL*. 1949.
- [Woo50] M. A. Woodbury. „Inverting modified matrices“. In: *Statistical Research Group, Memo. Rep. no. 42*. 1950.
- [Wul11] C. Wulff-Nilsen. „Separator Theorems for Minor-Free and Shallow Minor-Free Graphs with Applications“. In: *FOCS*. IEEE Computer Society, 2011, pp. 37–46.
- [Wul17] C. Wulff-Nilsen. „Fully-dynamic minimum spanning forest with improved worst-case update time“. In: *STOC*. ACM, 2017, pp. 1130–1143.

- [Yan91] M. Yannakakis. „Expressing Combinatorial Optimization Problems by Linear Programs“. In: *Journal of Computer and System Sciences* 43.3 (1991), pp. 441–466.
- [Yao79] A. C. Yao. „Some Complexity Questions Related to Distributive Computing (Preliminary Report)“. In: *STOC*. ACM, 1979, pp. 209–213.
- [YZ05] R. Yuster and U. Zwick. „Answering distance queries in directed graphs using fast matrix multiplication“. In: *FOCS*. IEEE Computer Society, 2005, pp. 389–396.