

Fast Dynamic Cuts, Distances and Effective Resistances via Vertex Sparsifiers

Li Chen^{*} Gramoz Goranci[†] Monika Henzinger[‡] Richard Peng[§]
 Thatchaphol Saranurak[¶]

May 6, 2020

Abstract

We present a general framework of designing efficient dynamic approximate algorithms for optimization on undirected graphs. In particular, we develop a technique that, given any problem that admits a certain notion of vertex sparsifiers, gives data structures that maintain approximate solutions in sub-linear update and query time. We illustrate the applicability of our paradigm to the following problems.

(1) A fully-dynamic algorithm that approximates all-pair maximum-flows/minimum-cuts up to a nearly logarithmic factor in $\tilde{O}(n^{2/3})$ amortized time against an oblivious adversary, and $\tilde{O}(m^{3/4})$ time against an adaptive adversary.

(2) An incremental data structure that maintains $O(1)$ -approximate shortest path in $n^{o(1)}$ time per operation, as well as fully dynamic approximate all-pair shortest path and transshipment in $\tilde{O}(n^{2/3+o(1)})$ amortized time per operation.

(3) A fully-dynamic algorithm that approximates all-pair effective resistance up to an $(1 + \epsilon)$ factor in $\tilde{O}(n^{2/3+o(1)}\epsilon^{-O(1)})$ amortized update time per operation.

The key tool behind result (1) is the dynamic maintenance of an algorithmic construction due to Madry [FOCS' 10], which partitions a graph into a collection of simpler graph structures (known as j-trees) and approximately captures the cut-flow and metric structure of the graph. The $O(1)$ -approximation guarantee of (2) is by adapting the distance oracles by [Thorup-Zwick JACM '05]. Result (3) is obtained by invoking the random-walk based spectral vertex sparsifier by [Durfee et al. STOC '19] in a hierarchical manner, while carefully keeping track of the recourse among levels in the hierarchy.

^{*}Georgia Institute of Technology, USA

[†]University of Toronto, Canada

[‡]University of Vienna, Austria

[§]Georgia Institute of Technology, USA

[¶]Toyota Technological Institute at Chicago, USA

1 Introduction

In the study of graph algorithms, there are long-standing gaps in the performances of static and dynamic algorithms. A dynamic graph algorithm is a data structure that maintains a property of a graph that undergoes edge insertions and deletions, with the goal of minimizing the time per update and query operation. Due to the prevalence of large evolving graph data in practice, dynamic graph algorithms have natural connections with network science [BKW06, PBL17], and databases [AG08, RWE13]. However, compared to the wealth of tools available for static graphs, it has proven to be much more difficult to develop algorithms for dynamic graphs, especially fully dynamic ones undergoing both edge insertions and deletions. Even maintaining connectivity undirected graphs has witnessed 35 years of continuous progress [NS17, Wul17, NSW17]. The directed version, fully dynamic transitive closure, has seen even less progress [San04, RZ08, vdBNS19], and is one of the best reflections of the difficulties of designing dynamic graph algorithms, especially in practice [HHS20].

Over the past decade, dynamic graph algorithms and their lower bounds have been studied extensively. These results led to significantly improved understandings of maintaining many basic graph properties such as connectivity, maximal matching, shortest paths, and transitive closure. However, for many of these results there are linear or polynomial conditional lower bounds for maintaining them exactly [AW14, HKNS15, AD16, Dah16]. This shifted the focus to maintaining *approximate* solutions to these problems, and/or restricting the update operations to only insertions (known as the *incremental* setting) or only deletions (known as the *decremental* setting).

While this approach has led to much recent progress on shortest path algorithms [San05, Tho05, Ber09, RZ12, ACT14, Ber16, ACK17, Che18], there has been comparatively little development in the maintenance of flows. Flows and their associated dual labels, cuts, are widely used in network analysis due to their ability to track multiple paths and more global information [HPK11, BVZ01, Zhu05]. For example, the st -maximum flow problem asks for the maximum number of edge disjoint paths between a pair of vertices [GT14], while electrical flow minimizes a congestion measure related to the sums of squares of the flow values along edges [DS84]. This need to track multiple paths has motivated the development of new dynamic tools that eschew the tree-like structures typically associated with problems such as connectivity and single-source shortest paths [Gor19]. Such tools were recently used to give the first sublinear time data structures for maintaining $(1 + \epsilon)$ -approximate electrical flows / effective resistances, which raised the optimistic possibility that all flow related problems can be maintained with $(1 + \epsilon)$ -approximation factors in subpolynomial time [DGGP19].

Motivated by interest in better understanding these problems, in this paper we present a general framework for designing efficient dynamic approximate algorithms for graph-based optimization problems in undirected graphs. In particular, we develop a technique that reduces these problems to finding a data-structure notion of vertex sparsifiers. We then utilize this framework to study dynamic graph algorithms for flows, with focus on obtaining the best approximation ratios possible, but with sub-linear time per update/query. We achieve the following results:

1. Fully dynamic all pair max-flow/min-cut, shortest path, and transshipment: $O(\log n \log \log n)$ -approx. with $\tilde{O}(n^{2/3+o(1)})^1$ amortized edge update time and query time (Theorems 6.1 and 7.1) against an oblivious adversary. Our dynamic max-flow algorithm can be extended to work against an adaptive adversary while increasing the update and query time to $\tilde{O}(m^{3/4})$ (Theorem 6.27).
2. Incremental all pair shortest path: $(2r - 1)^t$ -approximation with $\tilde{O}(m^{1/(t+1)}n^{t/r})$ worst-case update and query time, where $t, r \geq 1$ (Theorem 3.1).

¹The $\tilde{O}(\cdot)$ notation is used in this paper to hide poly-logarithmic factors.

3. Fully dynamic all pair effective resistance for general weighted graphs: $(1 + \epsilon)$ -approximation with $\tilde{O}(n^{2/3+o(1)}\epsilon^{-O(1)})$ amortized edge update time and query time, improving upon the previous running time of $\tilde{O}(n^{5/6}\epsilon^{-6})$ (Theorem 8.1).

In each of these three cases, our approximation ratios obtained match up to constants the current best known approximation ratios of oracles versions of these problems on static graphs, namely oblivious routings [R08], distance oracles [TZ05], and static computations of effective resistances [SS11]. This focus on approximation ratio is by choice: we believe just as with static approximation and optimization algorithms, approximation ratios should be prioritized over running times. However, because all current efficient construction of edge sparsifiers that preserve flows/cuts and resistances with constant or better $(1 + \epsilon)$ approximation are randomized, all above algorithms except (2) are randomized, and their guarantees are only provable against oblivious adversary (who determines the hidden sequence of updates/queries beforehand). We believe the design of more robust variants of our results hinge upon the development of more robust edge sparsification tools, which are interesting questions on their own.

Our techniques also extend to the offline dynamic setting, where the whole sequence of updates (edge insertions and deletions) and queries is given an advance. In other words, the algorithm needs to output information about the graphs at various points in this given update sequence. Specifically, we show that for graph properties that admit efficient constructions of *static* vertex sparsifiers, there are offline fully dynamic approximation algorithms with sub-linear average update and query time. We achieve the following results:

1. Offline fully dynamic all pair max-flow/min-cut: $O(\log^{4t} n)$ -approximation with $\tilde{O}(m^{1/t+1})$ average update and query time, where $t \geq 1$ (Theorem 4.11).
2. Offline fully dynamic all shortest-path: $(2r - 1)^t$ -approximation with $\tilde{O}(m^{1/t+1}n^{2/r})$ average update and query time, where $t, r \geq 1$ (Theorem 4.8).

Although the offline setting is a weaker than the standard dynamic setting, it is interesting for two reasons. First, offline algorithms are used to obtain fast static algorithms (e.g. [BKN19, LPYZ18a]). Second, many conditional lower bounds (e.g. [AW14, AD16, Dah16]) for the standard dynamic setting also hold for the offline dynamic setting. Thus, giving an efficient algorithm for the offline dynamic setting shows that no such conditional lower bound is possible. Moreover, for certain applications (e.g. computing “sensitivity information” for certain graph properties) the sequence of updates is also known beforehand.

1.1 Related work

Previous results on dynamic flow/cuts Despite the fact that all pair max-flow/min-cut is one of the cornerstone problems combinatorial optimization and has been extensively studied in the static setting, there are essentially no fast algorithms in the dynamic setting. Using previous techniques, it is possible to get dynamic algorithms with $\tilde{O}(1)$ worst-case update time and $\tilde{O}(n)$ query time under the assumption that the adversary is oblivious.² To the best of our knowledge, there is no previous algorithm with both $o(n)$ update and query time, even when we are content with only amortized guarantees.

Perhaps the closest work to this paper is the dynamic algorithm due to [CKL13] for explicitly maintaining the values of all-pairs min-cuts in $\tilde{O}(m^2)$ update time. For s - t max flow where s and t are fixed, there is an incremental algorithm with $O(n)$ amortized update time [GK18]. If we restrict to bipartite graphs

²We maintain a dynamic cut-sparsifier (against oblivious adversary) of size $\tilde{O}(n)$ due to [ADK⁺16] with $\tilde{O}(1)$ update time, and when given a query, we execute the fastest static approximation algorithms on the sparsifier in $\tilde{O}(n)$ time (using, for example, [Pen16] for $(1 + \epsilon)$ -approximate max flow, [She17] for $(1 + \epsilon)$ -approximate multi-commodity concurrent flow, and [She09] for $O(\sqrt{\log n})$ -approximate sparsest cuts).

with a certain specific structure, there is a $(1 + \epsilon)$ -approximation fully dynamic algorithm [ADK⁺16] with polylogarithmic worst-case update time. From the lower bound perspective, Dahlgaard [Dah16] showed a conditional lower bound of $\Omega(n^{1-o(1)})$ on the amortized update time for maintaining exact incremental s - t max flow in *weighted undirected* graphs. This shows that approximation is necessary to achieve sublinear running times.

The *global* minimum cut problem has been much better understood from the perspective of dynamic graphs. This is closely related to a similar phenomenon in the static setting, where in contrast to the $s - t$ min-cut problem, its global counterpart admits arguably simpler and easier algorithms. The best-known fully-dynamic algorithm is due to Thorup [Tho07], who maintains a $(1 + o(1))$ -approximation to the value of global minimum cut using $\tilde{O}(\sqrt{n})$ update and query time. When the graph undergoing updates remains planar, Lacki and Sankowski [LS11] showed an exact fully-dynamic algorithm with $\tilde{O}(n^{5/6})$ update and query time. Recently, Goranci, Henzinger and Thorup [GHT18] designed an exact incremental algorithm with $O(\log^3 n \log \log^2 n)$ update time and $O(1)$ query time.

Previous graph sparsification in the dynamic setting. Many previous works in dynamic graph algorithms are based on *edge sparsification*. This usually allows algorithms to assume that an underlying dynamic graphs is always sparse and hence speed up the running time. To the best of our knowledge, the first paper that applies edge sparsification in the dynamic setting is by Eppstein et al. [EGIN97]. This work has proven useful for several fundamental problems including dynamic minimum spanning forest and different variants of edge/vertex connectivity (e.g. [Tho07, NS17, Wul17, NSW17]). Edge sparsification has been also a key technique in dynamic shortest paths problems (e.g. [BR11] maintains distances on top of spanners, [BC16, BC17] replace “dense parts” of graphs with sparser graphs). Recently, there are works that study edge sparsification for matching problems [BS15, BS16, Sol18]. In fact, the core component of several dynamic matching algorithms is only to maintain such sparsifiers [BS15, BS16].

There are also previous developments in dynamic graph algorithms based on *vertex sparsification* which allow algorithms to work on graphs with smaller number of vertices. This usually offers a more significant speed up than edge sparsification. Earlier works [EGIS96, EGIS98, FR01] that utilize vertex sparsification in the dynamic setting are restricted to planar graphs and exploit the fact that this class of graphs admit small separators. Similar techniques are used and generalized in [GHP17, GHP18] but none of these works extend to general graphs. Several previous *offline* dynamic algorithms exploit vertex sparsification for maintaining minimum spanning forests [Epp91], small edge/vertex connectivity [PSS19], and effective resistance [LPYZ18b].

Recent Results on Dynamic Vertex Sparsification Very recently, Goranci et al. [GRST20] give a fully dynamic algorithm for maintaining a tree flow sparsifier based on a new notion of expander decomposition. One of their applications is a fully dynamic algorithm for s - t maximum flow and minimum cuts. Their algorithm is deterministic, has $n^{o(1)}$ worst-case update time and $O(\log^{1/6} n)$ query time, but their approximation ratio is $2^{O(\log^{5/6} n)} = n^{o(1)}$. Our algorithms from Theorems 6.1 and 6.27 guarantees a much better approximation ratio of $O(\log n (\log \log n)^{O(1)})$. However, our update and query times are slower, and are randomized.

Concurrent to our result there have also been several recent developments on utilizing vertex sparsifiers to maintain c -edge connectivity for small values of c [PSS19, CDLV19, LPS19, JS20].

1.2 Technical Overview

We start by discussing an incremental version of our meta theorem, which is key to our incremental all pair shortest path algorithm. The main algorithmic tool behind our construction is a data-structure version of the well-studied notion of *vertex sparsifier* [Moi09, LM10, CLLM10, Chu12, MM10, EGK⁺14], which we refer

to as *incremental vertex sparsifier*. To better convey our intuition, we start with a slightly weaker definition of such a sparsifier, which already leads to non-trivial guarantees. We then discuss the generalization and its implications.

Let $G = (V, E)$ be an n -vertex graph and, for each $u, v \in V$, let $\mathcal{P}(u, v, G)$ denote a *property* between u and v in G ³. For example, $\mathcal{P}(u, v, G)$ could be the distance between u and v in G . Let $T \subseteq V$ be a set of nodes called *terminals*. Given a parameter α , an α -*vertex sparsifier* of G w.r.t. T is a graph $H = (V', E')$ such that 1) $V' \supseteq T$, $|V'| \approx |T|$ and 2) $\mathcal{P}(u, v, H) \approx_\alpha \mathcal{P}(u, v, G)$ for all $u, v \in T$. That is, H has size close to T but still approximately preserves the property \mathcal{P} between all terminal nodes up to a factor of α .

Given a graph $G = (V, E)$ and terminals $T \subseteq V(G)$, an α -*incremental vertex sparsifier* (IVS) of G is a data structure that maintains an α -vertex sparsifier H_T and supports the following operations:

- $\text{PREPROCESS}(G, \alpha)$: preprocess the graph G ,
- $\text{ADDTERMINAL}(u)$: let $T' \leftarrow T \cup \{u\}$ and update $H_{T'}$ to an α -vertex sparsifier of G w.r.t. T' .

An *efficient* $(\alpha, f(n), g(n))$ -IVS of G is an α -IVS of G that supports the preprocessing and terminal addition operations in $O(|E|f(n))$ and $O(g(n))$ time, respectively.

The advantage of such an efficient sparsifier is that it immediately leads to a simple two-level incremental algorithm. Concretely, given an initial graph $G = (V, E)$ and an approximation parameter $\alpha \geq 1$, assume we want to design an incremental algorithm that maintains some property $\mathcal{P}(s, t, G)$ that can be computed in time $O(|E|h(n))$ on a static graph $G = (V, E)$. To achieve this, our data-structure maintains (1) an efficient $(\alpha, f(n), g(n))$ -IVS of G and (2) a set of terminals T , which is initially set to empty. We initialize our data-structure using the $\text{PREPROCESS}(G, \alpha)$ operation of the efficient IVS and rebuild from scratch every β operations, for some parameter $\beta \geq 0$. Note that after a rebuild, H_T is empty. We next describe the implementation of insertions and queries. Upon insertion of a new edge $e = (u, v)$ in G , we invoke $\text{ADDTERMINAL}(u)$ and $\text{ADDTERMINAL}(v)$, and add e to H_T . For answering (s, t) queries, we invoke $\text{ADDTERMINAL}(s)$ and $\text{ADDTERMINAL}(t)$, and run a static algorithm on H_T that computes property $\mathcal{P}(s, t, H_T)$ and return the result as an answer.

As H_T is an α -vertex sparsifier of the current graph G and $T \supseteq \{s, t\}$ by construction, we have that $\mathcal{P}(s, t, H_T)$ approximates property $\mathcal{P}(s, t, G)$ up to an α factor. The update time consists of (1) the cost for rebuilding every β operations, which is $O(mf(n)/\beta)$ and (2) the cost for adding endpoints of β edges as terminals, which is $O(\beta g(n))$. By construction, $|T| = O(\beta)$ at any time, resulting in a size of $O(\beta g(n))$ for H_T (since we start with an empty H_T after a rebuild). As the static algorithm on H_T takes time $O(|H_T|h(n))$, the query time is bounded by $O(\beta g(n)h(n))$.

Combining the above bounds on the update and query time, we obtain the following expression

$$O\left(\left(\frac{m}{\beta}\right) f(n) + \beta g(n)h(n)\right)$$

which bounds the amortized update time and worst-case query time.

A challenge we face to design a multi-level *incremental* algorithm using the above approach is the large flexibility allowed in the ADDTERMINAL operation. More concretely, given a sparsifier H_T w.r.t. T , whenever we add a new terminal to T and update H_T to $H_{T'}$, the implementation of the operation could potentially both delete and insert vertices/edges to H_T . Ideally we would like that the $H_{T'}$ is constructed by *only* adding new vertices/edges to H_T , which in turn would allow us to keep the incremental nature of the problem. To address this, we modify the operation of adding terminals in the definition of α -IVS as follows:

- $\text{ADDTERMINAL}(u)$: let T' be $T \cup \{u\}$ and update H_T to $H_{T'}$ such that

³Our approach also works for graph properties with a number of parameters that is different from 2.

- $H_{T'}$ is an α -vertex sparsifier of G w.r.t. T' .
- $H_T \subseteq H_{T'}$.

An important measure related to this new definition is the notion of *recourse*, which is the number of changes performed to the old sparsifier H_T , i.e., $|H_{T'} \setminus H_T|$. While it is straightforward to bound the recourse by the time needed to support the addition of terminals, there are scenarios where recourse can be much smaller. Equipped with the new definition of incremental vertex sparsifier and the notion of recourse, we immediately get a multi-level hierarchy for designing incremental algorithms, which is formally stated in Theorem 2.4 of Section 2.

We demonstrate the applicability of our meta theorem to the incremental all-pair shortest path problem by showing that an efficient IVS can be constructed using a deterministic variant of the distance oracle due to Thorup and Zwick [TZ05]. At a high level, the oracle preprocessing works as follows: (1) it constructs a hierarchy of centers, (2) for each vertex it finds the closest center at every level of the hierarchy and (3) for each vertex $u \in V$, it defines the notion of *bunch* $B(u)$, which is the union of all centers of u that we found in (2). Our key observation is that this construction leads to an efficient IVS with bounded recourse: (1) we preprocess the graph using the oracle preprocessing, and (2) implement the terminal addition of a vertex u by simply adding its bunch $B(u)$ to the current vertex sparsifier maintained by the data structure. This construction leads to an efficient $(\tilde{O}(n^{1/r}), \tilde{O}(n^{1/r}), 2r - 1)$ -IVS for G . The correctness of our data-structure heavily relies on the fact $\cup_{u \in T} B(u)$ is an α -vertex (distance) sparsifier of G w.r.t. T .

To extend our meta theorem to fully-dynamic graphs, we simply go back to the old implementation of the ADDTERMINAL operation (as now we can support insertions/deletions of edges) and then augment our data-structure with the operation DELETE(e), which allows deleting e from the underlying graph G and updates the maintained vertex sparsifier with respect to this deletion. Similarly to the above, bounding the recourse and performing recursive invocations of the vertex sparsifier data-structure leads to a meta theorem for fully dynamic algorithms, which is formally stated in Theorem 5.4 of Section 5.

Our fully-dynamic results on flow/cuts, distances and electrical flow/effective resistances, which are based on our generic meta-theorem, are obtained by adapting:

1. the j -tree decomposition of graphs by Madry [Mad10], which is in-turn based on the oblivious routing scheme by Racke [R08],
2. the random-walk interpretation of electrical flow preserving vertex sparsifiers (also known as Schur complements) that are also at the core of [DGGP19].

In each of these cases, the approximation ratios obtained by our data structures match the current best bounds of static variants of these problems, which are themselves well-studied. Specifically, our approximation ratios for these three problems are identical to those of sublinear time query oracles for answering (multi source/sink) min-cut queries⁴, distances, and effective resistances in static graphs. As a result, we believe our results represent natural starting points for more efficient versions of these data structures, and hope they will motivate further work on the static query versions of flows/cuts and shortest paths.

2 Incremental Algorithms via Incremental Vertex Sparsifiers

Let $G = (V, E)$ be a graph. Let \mathcal{P} be a property of graphs. Properties can be either (i) *global*, if \mathcal{P} is meant for the entire graph or (ii) *local*, if \mathcal{P} is defined with respect to a particular pair of vertices (u, v) in the graph. For example, $\mathcal{P}(u, v)$ is the distance between u and v in G or the size of the $u - v$ minimum cut.

⁴The Gomory-Hu tree on the other hand provides a much more efficient query oracle for answering $s-t$ min-cut queries, but we are not aware of generalizations of it to small sets of source/sink vertices

To simplify presentation, we assume throughout the rest of this paper that \mathcal{P} takes three parameters, two vertices, followed by the graph. Let $T \subseteq V$ be a set of nodes called *terminals*. An α -vertex sparsifier of G for \mathcal{P} w.r.t. T is a graph $H = (V', E')$ such that i) $V' \supseteq T$, $|V'| \approx |T|$, and ii) $\mathcal{P}(u, v, H) \approx_\alpha \mathcal{P}(u, v, G)$ for all $u, v \in T$. That is, H is close to T and at the same time approximately preserves the property \mathcal{P} between pairs of terminal nodes up to a factor of α .

The notion of *recourse* is used for measuring the number of changes in the target graph maintained by a data structure.

Definition 2.1 (Incremental Vertex Sparsifier). *Let $G = (V, E)$ be a graph, $T \subseteq V$ be the set of terminals, and α be an non-negative parameter. A data-structure \mathcal{D} is an α -Incremental Vertex Sparsifier (abbr. α -IVS) of G if \mathcal{D} explicitly maintains an α -vertex sparsifier H_T of G with respect to T and supports following operations:*

1. *PREPROCESS(G, T): preprocess G in time t_p*
2. *ADDTERMINAL(u): let T' be $T \cup \{u\}$ and update H_T to $H_{T'}$ in time t_u such that i) $H_{T'}$ is an α -vertex sparsifier of G with respect to T' , ii) $H_T \subseteq H_{T'}$ and iii) recourse, number of edge changes from H_T to $H_{T'}$, is at most r_u . This operation should return the set of edges $H_{T'} \setminus H_T$.*

Also, the size of the vertex sparsifier, H_T , should be $O(|T|r_u)$.

The simplest example of such *Incremental Vertex Sparsifier* can be made as follows: \mathcal{D} maintains G as the vertex sparsifier. Such \mathcal{D} is a 1-IVS of G with preprocessing time t_p update time t_u and recourse r_u being $O(|E(G)|)$.

Given a α -IVS \mathcal{D} of G , we can support edge insertion by first adding 2 endpoints to the terminal set and then add the edge directly to the vertex sparsifier H_T . The correctness comes from the decomposability. To specify the cost, the following corollary is presented to formalized the approach.

Lemma 2.2. *Let $G = (V, E)$ be a graph and \mathcal{D} be an α -IVS of G . \mathcal{D} also maintains an α -vertex sparsifier H_T of G with respect to T subject to the following operation:*

- *INSERT($e = uv$): insert edge e to G and update T and H_T . It is done in $O(t_u)$ -time with $O(r_u)$ -recourse.*

Algorithm 1: $\mathcal{D}.$ INSERT($e = uv$)

- 1 $E_u \leftarrow \mathcal{D}.$ ADDTERMINAL(u).
 - 2 $E_v \leftarrow \mathcal{D}.$ ADDTERMINAL(v).
 - 3 Add edge e to H_T .
 - 4 **return** $E_u \cup E_v \cup \{e\}$.
-

Proof. The algorithm for edge insertion is presented as Algorithm 1. The correctness comes from the decomposability of the desired graph property. The time bound is $2t_u + O(1)$ for 2 ADDTERMINAL operations and edge insertion in a graph. \square

Since our ultimate goal is to design incremental algorithms that support updates and queries in sub-linear time, we will focus on building incremental vertex sparsifiers whose update time and recourse are sub-linear in n . This requirement is made precise in the following definition.

Definition 2.3 (Efficient IVS). *Let $G = (V, E)$ be a graph, α be an non-negative parameter, and $f(n), g(n), r(n) \geq 1$ be functions. We say that \mathcal{D} is an $(\alpha, f(n), g(n), r(n))$ -efficient IVS of G if \mathcal{D} is an α -IVS of G with preprocessing time $t_p = O(m \cdot f(n))$, addTerminal time $t_u = O(g(n))$, and recourse $r_u = O(r(n))$.*

Next we show how to use efficient incremental vertex sparsifier to design online (approximate) incremental algorithms for problems with certain properties while achieving fast amortized update and query time.

Theorem 2.4. *Let $G = (V, E)$ be a graph, and for any $u, v \in V$, let $\mathcal{P}(u, v, G)$ be a solution to a minimization problem between u and v in G . Let $f(n), g(n), r(n), h(n) \geq 1$ be functions, $\alpha \geq 1$ be the approximation factor, $\ell \geq 1$ be the depth of the data structure, and let $\gamma, \mu_0, \mu_1, \dots, \mu_\ell$ with $\mu_0 = m$ be parameters associated with the running time. Assume the following properties are satisfied*

1. G admits an $(\alpha, f(n), g(n), r(n))$ -efficient IVS
2. The property $\mathcal{P}(u, v, G)$ can be computed in $O(mh(n))$ time in a static graph with m edges and n vertices.

Then there is an incremental (approximate) dynamic algorithm that maintains for every pair of nodes u and v , an estimate $\delta(u, v)$, such that

$$\mathcal{P}(u, v, G) \leq \delta(u, v) \leq \alpha^\ell \cdot \mathcal{P}(u, v, G), \quad (1)$$

with worst-case update time of

$$T_u = O\left(\sum_{i=1}^{\ell} \left(\frac{\sum_{j=i}^{\ell} \mu_{j-1} f(\mu_{j-1})}{\mu_i} + g(\mu_{i-1})\right) \prod_{j=0}^{i-1} r(\mu_j) c^{i-1}\right),$$

and worst-case query time of

$$T_q = O(\ell T_u + \mu_\ell r(\mu_\ell) h(\mu_\ell)),$$

where $c < 3$ is an universal constant.

We declare this section to prove the Theorem 2.4

Data Structure. Consider some integer parameter $\ell \geq 1$ and parameters $\mu_0 \geq \dots \geq \mu_\ell$, with $\mu_0 = m$. Our data structure maintains

1. a hierarchy of graphs $\{G_i\}_{0 \leq i \leq \ell}$,
2. a hierarchy of terminal sets $\{T_i\}_{1 \leq i \leq \ell}$, each associated with the parameters $\{\mu_i\}_{1 \leq i \leq \ell}$, and
3. a hierarchy of $(\alpha, f(n), g(n), r(n))$ -efficient IVSs $\{\mathcal{D}_i\}_{1 \leq i \leq \ell}$, each associated with the graph G_{i-1} and the terminal set T_i .

The data structure is initialized recursively. First, initialize $T_i \leftarrow \emptyset$ for $1 \leq i \leq \ell$ and $G_0 \leftarrow G$. For $1 \leq i \leq \ell$, construct an $(\alpha, f(n), g(n), r(n))$ -efficient IVS, \mathcal{D}_i , of G_{i-1} with respect to terminal set T_i and set G_i be the sparsifier maintained by \mathcal{D}_i . That is, every G_i is an α -vertex sparsifier of G_{i-1} . By the transitivity, we know G_ℓ is an α^ℓ -vertex sparsifier of G_ℓ . Thus, we compute the estimate $\delta(u, v)$ in $G_{\ell-1}$, which should be the smallest graph we have.

To deal with edge insertion, we add both endpoints of the new edge to the terminal set of every IVS \mathcal{D}_i . Note that an edge insertion in \mathcal{D}_i produces $O(r(\mu_{i-1}))$ more insertions in the next level of IVS. Also, to bound the sparsifier size, which is related to the size of the terminal set, we have to rebuild \mathcal{D}_i whenever \mathcal{D}_i has handled μ_i updates since the last rebuild. A counter c_i is used for keeping track of the number of updates handled by \mathcal{D}_i . The rebuild cost is amortized over μ_i updates. To get worst-case cost, we use the standard reduction of creating copies of the data structure in the background.

Algorithm 2: REBUILD(i)

```
1 for  $j \in \{i, \dots, \ell\}$  do
2   Remove vertices from  $T_j$  except the newest  $\mu_j$  vertices
3    $\mathcal{D}_j$ .INITIALIZE( $G_{j-1}, T_j$ )
4   Set  $G_j$  to be the vertex sparsifier maintained by  $\mathcal{D}_j$ 
5   Set  $c_j \leftarrow 0$ 
```

Algorithm 3: INSERT(e)

```
1 for  $i \in \{0, 1, \dots, \ell\}$  do
2   Set  $E_{i+1} \leftarrow \phi$ 
3   for  $f \in E_i$  do
4     Perform  $\mathcal{D}_{i+1}$ .INSERT( $f$ ) and add the changes of  $G_{i+1}$  to  $E_{i+1}$ 
5     Set  $c_{i+1} \leftarrow c_{i+1} + 1$ 
6     if  $c_{i+1} \geq 2\mu_{i+1}$  then
7       REBUILD( $i + 1$ )
8       Break the loop
```

Handling Insertions. Consider the insertion of edge $e = uv$ in G , which is G_0 with an α -IVS \mathcal{D}_1 . We insert e to \mathcal{D}_1 via Corollary 2.2. Each edge insertion to G_i handled by \mathcal{D}_{i+1} creates $O(r(\mu_i))$ edge insertions in the resulting vertex sparsifier, G_{i+1} . Thus one edge insertion in G creates $O(c^i \prod_{j=0}^{i-1} r(\mu_j))$ edge insertions in G_i .

To bound the size of the terminal sets, we rebuild each \mathcal{D}_i every $2\mu_i$ updates to the graph G_{i-1} and also rebuild every $\mathcal{D}_j, i < j$ that gets affected. When rebuilding some \mathcal{D}_i , we rebuild it with respect to the terminal set T_i containing the latest-added μ_i vertices.

This algorithm is depicted in Algorithm 3.

Handling Queries. To answer the query for the approximate property $\mathcal{P}(s, t, G)$ between any pair of vertices s and t in G we proceed as follows. We first add s and t to every terminal set of $\{\mathcal{D}_i\}$. The algorithm for adding terminal in our hierarchical data structure is presented in Algorithm 4.

Then we run the algorithm from Theorem 2.4 Part 2 on G_ℓ , which is maintained by IVS \mathcal{D}_ℓ , to calculate the property $\mathcal{P}(s, t, G_\ell)$ between s and t in G_ℓ . The value $\mathcal{P}(s, t, G_\ell)$ is returned as an estimate to $\mathcal{P}(s, t, G)$. This algorithm is depicted in Algorithm 5.

Correctness. Let G be the current graph throughout the execution of the algorithm. Via induction, we know every data structure \mathcal{D}_i in the hierarchy is an α -IVS of G_{i-1} with respect to the terminal set T_i . Hence, G_i , maintained by \mathcal{D}_i , is an α -vertex sparsifier of G_{i-1} . Therefore via transitivity of vertex sparsifier, we know G_ℓ is an α^ℓ -vertex sparsifier of G_0 , which is the current graph G .

Thus, we have

$$\mathcal{P}(s, t, G) \leq \mathcal{P}(s, t, G_{\ell-1}) = \delta_{\mathcal{D}_\ell}(s, t) \leq \alpha^\ell \mathcal{P}(s, t, G).$$

Therefore the approximation claim in Theorem 2.4 is proved.

Running Time. We first study the update time of our data structure. Since rebuild is incurred every μ_i operations for the data structure H_i , we can charge the rebuild cost among μ_i operations. Note

Algorithm 4: ADDTERMINAL(u)

```
1 Set  $E_0 \leftarrow \phi$ 
2 for  $i \in \{0, 1, \dots, \ell - 1\}$  do
3   Set  $E_{i+1} \leftarrow \phi$ 
4   for  $f \in E_i$  do
5     Perform  $\mathcal{D}_{i+1}.\text{INSERT}(f)$  and add the changes of  $G_{i+1}$  to  $E_{i+1}$ 
6     Set  $c_{i+1} \leftarrow c_{i+1} + 1$ 
7     if  $c_{i+1} \geq 2\mu_{i+1}$  then
8       REBUILD( $i + 1$ )
9       Set  $E_{i+1} \leftarrow \phi$ 
10      Break the loop
11   Perform  $\mathcal{D}_{i+1}.\text{ADDTERMINAL}(u)$  and add the changes of  $G_{i+1}$  to  $E_{i+1}$ 
12   Set  $c_{i+1} \leftarrow c_{i+1} + 1$ 
13   if  $c_{i+1} \geq 2\mu_{i+1}$  then
14     REBUILD( $i + 1$ )
15     Set  $E_{i+1} \leftarrow \phi$ 
```

Algorithm 5: QUERY(s, t)

```
1 ADDTERMINAL( $s$ )
2 ADDTERMINAL( $t$ )
3 Run static algorithm on  $G_\ell$  to compute  $\mathcal{P}(s, t, G_\ell)$ 
4 return  $\mathcal{P}(s, t, G_\ell)$ 
```

that \mathcal{D}_i is an α -IVS of G_{i-1} , which is a graph on μ_{i-1} -vertices. By the size bound in Definition 5, rebuild cost is $O(\mu_{i-1}r(\mu_{i-1})f(\mu_{i-1}))$. Also notice all $\mathcal{D}_j, i < j \leq \ell$ are also rebuilt, each has rebuild cost $O(\mu_{j-1}r(\mu_{j-1})f(\mu_{j-1}))$. By amortizing the rebuild cost, we know the time \mathcal{D}_i spent on either ADDTERMINAL or INSERT is:

$$O\left(\frac{\sum_{j=i}^{\ell} \mu_{j-1}f(\mu_{j-1})}{\mu_i} + g(\mu_{i-1})\right).$$

Since an update in \mathcal{D}_i creates $\leq cr(\mu_i)$, c being some positive universal constant, updates in G_i , which is handled by the data structure in the next level, \mathcal{D}_{i+1} , we have to incorporate such quantity into the analysis. Note that when rebuilding \mathcal{D}_i , all $\mathcal{D}_j, i < j$ are also rebuilt. Hence no recourse is made for rebuilding. We can now analyze the number of updates handled by \mathcal{D}_i when 1 edge insertion happens in G . By simple induction, we know there will be $\leq \prod_{j=1}^{i-1} cr(\mu_j)$ updates in G_{i-1} . Thus for the data structure in i -th level, \mathcal{D}_i , there will be at most $\prod_{j=1}^{i-1} cr(\mu_j)$ updates. Combining these 2 quantities, we can bound the amortized update time of our data structure:

$$\begin{aligned} T_u &= O\left(\sum_{i=1}^{\ell} \left(\frac{\sum_{j=i}^{\ell} \mu_{j-1}f(\mu_{j-1})}{\mu_i} + g(\mu_{i-1})\right) \prod_{j=1}^{i-1} cr(\mu_j)\right) \\ &= O\left(\sum_{i=1}^{\ell} \left(\frac{\sum_{j=i}^{\ell} \mu_{j-1}f(\mu_{j-1})}{\mu_i} + g(\mu_{i-1})\right) \prod_{j=1}^{i-1} r(\mu_j)c^{i-1}\right). \end{aligned}$$

We next study the query time of our data-structure. When answering a query, we add s and t to each layer of the terminal set. When adding terminals to each layer of data structure \mathcal{D}_i , edge changes also propagate to lower levels. As for the analysis of update time, we have to take the recourse into account. The time can be bounded by $O(\ell T_u)$ where T_u is the update time of our data structure.

Then we compute $\mathcal{P}(s, t, G_\ell)$ in G_ℓ , which has $\mu_\ell r(\mu_\ell)$ edges as guaranteed by the definition of IVS. Since we have an $O(mh(n))$ algorithm for computing $\mathcal{P}(s, t)$ in an m -edge n -vertex graph, $\mathcal{P}(s, t, G_\ell)$ can be computed in $\mu_\ell r(\mu_\ell)h(\mu_\ell)$ time. Combining these 2 bounds, we can bound the query time by:

$$T_q = O(\ell T_u + \mu_\ell r(\mu_\ell)h(\mu_\ell)).$$

Lemma 2.5. *Let $\{\mu_i\}_{0 \leq i \leq \ell}$ be a family of parameters with $\mu_0 = m$. Also, all parameters regarding efficient IVS, $f(n)$, $g(n)$, $r(n)$, and $h(n)$, are of order $n^{o(1)}$. If we set*

$$\mu_i = m^{1-i/(\ell+1)}, \text{ where } 1 \leq i \leq \ell,$$

then the update time is

$$T_u = O\left(\sum_{i=1}^{\ell} \left(\frac{\sum_{j=i}^{\ell} \mu_{j-1} f(\mu_{j-1})}{\mu_i} + g(\mu_{i-1})\right) \prod_{j=1}^{i-1} r(\mu_j) c^{i-1}\right) = O\left(\ell c^\ell m^{1/(\ell+1)+o(\ell)}\right), \quad (2)$$

and the query time is

$$T_q = O(\ell T_u + \mu_\ell r(\mu_\ell)h(\mu_\ell)) = O\left(\ell^2 c^\ell m^{1/(\ell+1)+o(\ell)}\right). \quad (3)$$

Proof. Plug in the choice of μ_i , we have

$$\frac{\sum_{j=i}^{\ell} \mu_{j-1}}{\mu_i} = m^{i/(\ell+1)-1} \left(\sum_{j=i}^{\ell} 1 - m^{(j-1)/(\ell+1)}\right) = \sum_{j=i}^{\ell} m^{(i-j+1)/(\ell+1)} \leq 2m^{1/(\ell+1)}.$$

Since $r(n), c \geq 1$ and $r(n)$ is an increasing function, $\prod_{j=1}^{i-1} r(\mu_j) c^{i-1} = O(r(n)^\ell c^\ell) = O(m^{o(\ell)} c^\ell)$ holds. Combining these 2 inequalities, we can bound the update time by

$$\begin{aligned} T_u &= O\left(\sum_{i=1}^{\ell} \left(\frac{\sum_{j=i}^{\ell} \mu_{j-1} f(\mu_{j-1})}{\mu_i} + g(\mu_{i-1})\right) \prod_{j=1}^{i-1} r(\mu_j) c^{i-1}\right) \\ &= O\left(\sum_{i=1}^{\ell} \left(m^{1/(\ell+1)} f(n) + g(n)\right) r(n)^\ell c^\ell\right) \\ &= O\left(\ell c^\ell m^{1/(\ell+1)+o(\ell)}\right). \end{aligned}$$

The bound for query time is straightforward from the definition of μ_ℓ . □

3 Incremental All-Pairs Shortest Paths

In this section we show how to use Theorem 2.4 to design an online incremental algorithm for the approximate All-Pair Shortest-Paths problem with fast worst-case update and query time. Concretely, we will show that the assumption (1) in Theorem 2.4 is satisfied with certain parameters for shortest paths. Note that (2) follows immediately with $h(n) = \tilde{O}(1)$ by any $\tilde{O}(m)$ time single-pair shortest path algorithm. We have the following theorem.

Theorem 3.1. *Let $G = (V, E)$ be an undirected, weighted graph and c be some positive constant. For every $\ell, r \geq 1$, there is an incremental deterministic All-Pair Shortest-Paths algorithm that maintains for every pair of nodes u and v , an estimate $\delta(u, v)$ that approximates the shortest path distance between u and v up to a factor of $(2r - 1)^\ell$ while supporting updates and queries in $O(\ell c^{\ell-1} r^\ell m^{1/(\ell+1)} n^{\ell/r} \log^{\ell-1} n)$ and $O(\ell^2 c^{\ell-1} r^\ell m^{1/(\ell+1)} n^{\ell/r} \log^{\ell-1} n)$ worst-case time, respectively.*

Corollary 3.2. *Let $G = (V, E)$ be an undirected, weighted graph. For every $r \geq 1$, there is an incremental deterministic All-Pair Shortest-Paths algorithm that maintains for every pair of nodes u and v , an estimate $\delta(u, v)$ that approximates the shortest path distance between u and v up to a factor of $(2r - 1)$ while supporting updates and queries in $O(m^{1/2} n^{1/r})$ and $O(m^{1/2} n^{1/r})$ worst-case time, respectively.*

Proof. The argument holds trivially by replacing ℓ by 1 in Theorem 3.1. □

We start by introducing the usual definitions of sparsifiers and vertex sparsifiers for distances. Let $G = (V, E)$ be an undirected, weighted graph with a *terminal set* $T \subseteq V$. For $u, v \in V$, let $\text{dist}_G(u, v)$ denote the length of the shortest path between u and v in G .

Definition 3.3 (Sparsifiers for Distances). *Let $G = (V, E)$ be an undirected, weighted graph, and let $\alpha \geq 1$ be a stretch parameter. A graph $H = (V', E')$ with $V \subseteq V'$ is an α -distance sparsifier of G iff for all $u, v \in V$, $\text{dist}_G(u, v) \leq \text{dist}_H(u, v) \leq \alpha \cdot \text{dist}_G(u, v)$.*

Definition 3.4 (Vertex Sparsifiers for Distances). *Let $G = (V, E)$ be an undirected, weighted graph with a terminal set $T \subseteq V$, and let $\alpha \geq 1$ be a stretch parameter. A graph $H = (V', E')$ with $T \subseteq V'$ is an α -(vertex) distance sparsifier of G with respect to T iff for all $u, v \in T$, $\text{dist}_G(u, v) \leq \text{dist}_H(u, v) \leq \alpha \cdot \text{dist}_G(u, v)$.*

In the remainder of this section, we will show that the distance property in graphs admits an efficient incremental sparsifier oracle with desirable guarantees.

Lemma 3.5 (Efficient ISO's for Distances). *Given an undirected, weighted graph $G = (V, E)$, and a parameter $r \geq 1$, there is a deterministic algorithm that constructs an $(2r-1, O(n^{1/r} \log^2 n), O(n^{1/r} \log^2 n), O(n^{1/r} \log^2 n))$ -efficient (distance) IVS of G .*

We achieve this by showing a deterministic variant of the distance oracle due to Thorup and Zwick [TZ05]. While our construction closely follows the ideas presented in the deterministic oracle due to Roditty, Thorup and Zwick [RTZ05], we note that their work only bounds the *total* size of the oracle, which is not sufficient for our purposes. Similar ideas have been employed by Lacki et al. [LOP⁺15] for constructing bipartite emulators and by Abraham et al. [ACT14] for designing approximation algorithms for the fully-dynamic APSP problem.

We start by reviewing the randomized algorithm for APSP due to Thorup and Zwick [TZ05] (which is depicted in Figure 6), and then derandomize that algorithm and show how it can be used to solve the above problem.

1. Set $A = V$ and $A_r = \emptyset$, and for $1 \leq i \leq r-1$ obtain A_i by picking each node from A_{i-1} independently, with probability $n^{-1/r}$.
2. For each $1 \leq i < r$, and for each vertex $v \in V$, find the vertex $p_i(v) \in A_i$ (also known as the i -th *pivot*) that minimizes the distance to v , i.e.,

$$p_i(v) := \arg \min_{u \in A_i} \text{dist}_G(u, v),$$

and its corresponding distance value

$$\text{dist}_G(A_i, v) := \min\{\delta(w, v) \mid w \in A_i\} = \text{dist}_G(v, p_i(v)).$$

Algorithm 6: HIERARCHYCONSTRUCT(G, r)

```
1  $A_0 \leftarrow V; A_r \leftarrow \emptyset$ 
2 for  $i \leftarrow 1$  to  $r - 1$  do
3    $A_i \leftarrow \text{SAMPLE}(A_{i-1}, |V|^{-1/r})$ 
4 for every  $v \in V$  do
5   for  $i \leftarrow 0$  to  $r - 1$  do
6     Let  $\text{dist}_G(A_i, v) \leftarrow \min\{\text{dist}_G(w, v) \mid w \in A_i\}$ 
7     Let  $p_i(v) \in A_i$  be such that  $\text{dist}_G(p_i(v), v) = \text{dist}_G(A_i, v)$ 
8    $\text{dist}_G(A_r, v) \leftarrow \infty$ 
9   Let  $B(v) \leftarrow \cup_{i=0}^{r-1} \{w \in A_i \setminus A_{i+1} \mid \text{dist}_G(w, v) < \text{dist}_G(A_{i+1}, v)\}$ 
```

3. For each vertex $v \in V$, define the *bunch* $B(v) = \cup_{i=0}^{r-1} B_i(v)$, where

$$B_i(v) := \{w \in A_i \setminus A_{i+1} \mid \text{dist}_G(w, v) < \text{dist}_G(A_{i+1}, v)\}.$$

Thorup and Zwick [TZ05] showed that using the hierarchy of sets $(A_i)_{0 \leq i \leq r}$ chosen as above, the expected size of a bunch $\mathbb{E}[|B(v)|]$ is $O(rn^{1/r})$, for each vertex $v \in V$. We note that the only place where their construction uses randomization is when building the hierarchy of sets (the **for** loop in Step 2 in Figure 6). Therefore, to derandomize their algorithm it suffices to design a deterministic algorithm that efficiently computes a hierarchy of set $(A_i)_{0 \leq i \leq r}$ such that $|B(v)| \leq \tilde{O}(rn^{1/r})$, for each $v \in V$ (note that compared to the randomized construction, we are content with additional poly-log factors on the size of the bunches).

We present a deterministic algorithm for computing the hierarchy of sets that closely follows the ideas presented in the deterministic construction of Roditty, Thorup, and Zwick [RTZ05]. The main two ingredients of the algorithm are the *hitting set* problem, and the *source detection* problem. For the sake of completeness, we next review their definitions and properties.

Definition 3.6 (Hitting set). *Let U be a set of elements, and let $\mathbb{S} = \{S_1, \dots, S_p\}$ be a collection of subsets of U . We say that T is a hitting set of U with respect to \mathbb{S} if $T \subseteq U$, and T has a non-empty intersection with every set of \mathbb{S} , i.e., $T \cap S_i \neq \emptyset$ for every $1 \leq i \leq p$.*

It is known that computing a hitting set of minimum size is an NP-hard problem. In our setting however, it is sufficient to compute approximate hitting sets. Since our goal is to design a deterministic algorithm, one way to deterministically compute such sets is using a variant of the well-known greedy approximation algorithm: (1) Form the set T by repeatedly adding to T elements of U that ‘hit’ as many ‘unhit’ sets as possible, until only $|U|/s$ sets are unhit, where $|S_i| \geq s$ for each $1 \leq i \leq p$; (2) add an element from each one of the unhit sets to T . The lemma below shows that this algorithm finds a reasonably sized hitting set in time linear in the size of U the collection \mathbb{S} .

Lemma 3.7. *Let U be a set of size u and let $\mathbb{S} = \{S_1, \dots, S_p\}$ be the collection of subset of U , each of size at least s , where $s \leq p$. Then the above deterministic greedy algorithm runs in $O(u + ps)$ time and finds a hitting set T of U with respect to \mathbb{S} , whose size is bounded by $|T| = (u/s)(1 + \ln p)$.*

Note that the size of this hitting set is within $O(\log n)$ of the optimum size since in the worst case T has size at least u/s .

Algorithm 7: DETHIERARHCY(G, r)

Data: Undirected, weighted graph $G = (V, E)$, parameter $r \geq 1$

Result: Hierarchy of sets $(A_i)_{0 \leq i \leq r}$

```
1  $q = \lceil n^{1/r}(1 + \ln n) \rceil$ 
2  $A_0 \leftarrow V$ ;  $A_r \leftarrow \emptyset$ 
3 for  $i \leftarrow 0$  to  $r - 2$  do
4   Compute  $A_i(v, q, G)$  for each  $v \in V$  using the source detection algorithm (Lemma 3.9)
5   Let  $\{A_i(v, q, G)\}_{v \in V}$  be the resulting collection of sets
6   Compute a hitting set  $A_{i+1} \subseteq A_i$  with respect to  $\{A_i(v, q, G)\}_{v \in V}$  (Lemma 3.7)
7 return  $(A_i)_{0 \leq i \leq r}$ 
```

Definition 3.8 (Source Detection). *Let $G = (V, E)$ be an undirected, weighted graph, let $U \subseteq V$ be an arbitrary set of sources of size u , and let q be a parameter with $1 \leq q \leq u$. For every $v \in V$, we let $U(v, q, G)$ be the set of the q vertices of U that are closest to v in G .*

Roditty, Thorup, and Zwick [RTZ05] showed that the set $U(v, q, G)$ can be computed using q single-source shortest path computations. We review their result in the lemma below.

Lemma 3.9 ([RTZ05]). *For every $v \in V$, the set $U(v, q, G)$ can be computed in time $O(qm \log n)$.*

Our algorithm for constructing the hierarchy of sets $(A_i)_{0 \leq i \leq r}$, depicted in Figure 7, is as follows. Initially, we set $A_0 = V$ and $A_r = \emptyset$. To construct the set A_{i+1} , given the set A_i , for $0 \leq i \leq r - 2$, we first find the set $A_i(v, q, G)$, where $q = \tilde{O}(n^{1/r})$, using the source detection algorithm from Lemma 3.9. Then we observe that the collection of sets $\{A_i(v, q, G)\}_{v \in V}$ can be viewed as an instance of the minimum hitting set problem over the set (universe) A_i , i.e., we want to find a set $A_{i+1} \subseteq A_i$ of minimum size such that each set $A_i(v, q, G)$ in the collection contains at least one node of A_{i+1} . We construct A_{i+1} by invoking the deterministic greedy algorithm from Lemma 3.7, which produces a hitting set whose size is within $O(\log n)$ of the optimum one. We next prove the constructed hierarchy produces bunches whose sizes are comparable to the randomized construction, and also show that our deterministic construction can be implemented efficiently.

Lemma 3.10. *Given an undirected, weighted graph $G = (V, E)$, and a parameter $r \geq 1$, Algorithm 7 computes deterministically, in $O(rmn^{1/r} \log n)$ time, a hierarchy of sets $(A_i)_{0 \leq i \leq r}$ such that for each $v \in V$,*

$$|B(v)| = O(rn^{1/r} \log n).$$

Proof. We start by showing the bound on the size of the bunches. To this end, we first prove by induction on i that $|A_i| \leq n^{1-i/r}$ for all $0 \leq i \leq r - 1$. For the base case, i.e., $i = 0$, the claim is true by construction since $A_0 = V$. We assume that $|A_i| \leq n^{1-i/r}$ for the induction hypothesis, and show that $|A_{i+1}| \leq n^{1-(i+1)/r}$ for the induction step. Note that by construction each set in the collection $\{A_i(v, q, G)\}_{v \in V}$ has size $q = \lceil n^{1/r}(1 + \ln n) \rceil \geq n^{1/r}(1 + \ln n)$. Invoking the greedy algorithm from Lemma 3.7, we get a hitting set $A_{i+1} \subseteq A_i$ of size at most

$$\left(\frac{|A_i|}{q} \right) (1 + \ln n) \leq \left(\frac{n^{1-i/r}}{n^{1/r}(1 + \ln n)} \right) (1 + \ln n) = n^{1-(i+1)/r}.$$

We next show that for each $v \in V$ and for each $0 \leq i \leq r - 1$, $|B_i(v)| \leq O(n^{1/p} \log n)$, which in turn implies the claimed bound on the size of vertex bunches. Note that it suffices to show that $B_i(v) \subseteq A_i(v, q, G)$ since then $|B_i(v)| \leq |A_i(v, q, G)| \leq n^{1/p}(1 + \ln n) = O(n^{1/p} \log n)$. Recall that for $1 \leq i \leq r - 1$

$$B_i(v) = \{w \in A_i \setminus A_{i+1} \mid \text{dist}_G(w, v) < \text{dist}_G(A_{i+1}, v)\}$$

Algorithm 8: PREPROCESS($G, 2r - 1$)

- 1 Invoke HIERARCHYCONSTRUCT(G, r), where instead of Steps 1-3 invoke DETHIERARCHY(G, r)
 - 2 **for** each $v \in V$ **do**
 - 3 └ Store each $B(v)$, where $w \in B(v)$ holds $\text{dist}_G(v, w)$.
 - 4 Set $G_0 \leftarrow G$.
-

Algorithm 9: ADDTERMINAL(u)

- 1 Let H_T be the vertex sparsifier maintained w.r.t. T .
 - 2 Set $T \leftarrow T \cup \{u\}$.
 - 3 **for** every $v \in B(u)$ **do**
 - 4 └ Add (u, v) to $E(H_T)$ with weight $\text{dist}_{G_0}(v, u)$.
-

Now, by construction of A_{i+1} we have that $A_{i+1} \cap A_i(v, q, G) \neq \emptyset$, which implies that $B_i(v) \subseteq A_i(v, q, G)$ by the definition of $B_i(v)$.

We finally analyze the running time. For $0 \leq i \leq r-2$, consider the sequence of steps in the i -th iteration of the **for** loop in Figure 7. By Lemma 3.9, the time to construct the collection of sets $\{A_i(v, q, G)\}_{v \in V}$ is $O(mn^{1/r} \log n)$. Furthermore, since the size of each set in this collection is at least $q = O(n^{1/r} \log n)$, Lemma 3.7 guarantees that the greedy algorithm for computing a hitting set A_{i+1} takes $O(n^{1+1/r} \log n)$ time. Combining the above bounds, we get that the total time for the i -th iteration is $O(mn^{1/r} \log n)$. Since there are at most r iterations, we conclude that the running time of the algorithm is $O(rmn^{1/r} \log n)$. \square

We next show to implement the two operations of the efficient (distance) ISO from Lemma 3.5, namely PREPROCESS() and ADDTERMINAL().

In the preprocessing phase, depicted in Figure 8, given the graph G and the stretch parameter $(2r - 1)$, we first invoke HIERARCHYCONSTRUCT(G, r) in Figure 6, where Steps 1-3 are replaced by the deterministic algorithm for computing the hierarchy of sets DETHIERARCHY(G, r). Note that this modification ensures that our preprocessing algorithm is deterministic. Next, for each vertex $v \in V$, we store its bunch $B(v)$ in a balanced binary search tree, where each vertex $w \in B(v)$ has as key the value $\text{dist}_{G_0}(w, v)$ (this step could have been implemented differently, but as we will shortly see, it is useful in the subsequent applications of our algorithm), where G_0 is the graph G at the moment of preprocessing.

We next describe how to implement the ADDTERMINAL operation, depicted in Figure 9. Let T be the set of queried terminals. The main idea to construct a vertex distance sparsifier H_T of G with respect to T is to exploit the bunches that we stored in the preprocessing step. More concretely, let H_T be an initially empty graph. For each vertex $v \in T$, and every vertex in its bunch $u \in B(v)$, we add to H_T the edge (u, v) with weight $\text{dist}_G(u, v)$. Finally, we return H_T as a (vertex) distance sparsifier of G with respect to T .

Next we formally analyze the running time for the above operations and shows the correctness for the update operation.

Proof of Lemma 3.5. We first argue about the correctness. To show that the resulting graph H_T is indeed a vertex distance sparsifier with respect to T , we briefly review the update algorithm in the construction of Thorup and Zwick [TZ05], and show that this immediately applies to our graph setting.

Let $u, v \in T$ by any two terminals. The algorithm uses the variables w and i , and starts by setting $w \leftarrow u$, and $i \leftarrow 0$. Then it repeatedly increments the value of i , swaps u and v , and sets $w \leftarrow p_i(u) \in B(u)$, until $w \in B(v)$. Finally, it returns a distance estimate $\delta(u, v) = \text{dist}_G(w, u) + \text{dist}_G(w, v)$. Observe that $w = p_i(u) \in B(u)$ for some $0 \leq i \leq r - 1$ and $w \in B(v)$. By construction of our vertex sparsifier H_T , note

that the edges (w, u) and (w, v) , and their corresponding weights, $\text{dist}_G(w, u)$ and $\text{dist}_G(w, v)$, are added to H_T . Thus, there must exist a path between u and v in H_T whose stretch is at most the stretch of the distance estimate $\delta(u, v)$. Since in [TZ05] it was shown that for every $u, v \in T$,

$$\text{dist}_G(u, v) \leq \delta(u, v) \leq (2r - 1) \text{dist}_G(u, v),$$

we immediately get that

$$\text{dist}_G(u, v) \leq \text{dist}_{H_T}(u, v) \leq (2r - 1) \text{dist}_G(u, v).$$

We finally analyze the running time for both operations. First, note that by Lemma 3.10, the deterministic algorithm for constructing the hierarchy of sets $\text{DETHIERARCHY}(G, r)$ runs in $O(rmn^{1/r} \log n)$ time. Moreover, Thorup and Zwick [TZ05] showed that given a hierarchy of sets, the bunches for all vertices in G can be computed in $O(rmn^{1/r} \log n)$ time. Combining these two bounds we get that the operation $\text{PREPROCESS}(G, r)$ runs in $O(rmn^{1/r} \log n)$ time. For the running time of $\text{ADDTERMINAL}(u)$, recall that H_T adds only edges between u and its bunch, $B(u)$. Since the size of each individual vertex bunch is bounded by $O(rn^{1/r} \log n)$ (Lemma 3.10), the time for adding edges adjacent to u is bounded by $O(|B(u)|) = O(rn^{1/r} \log n)$. This also bounds the recourse of adding u to the terminal set. \square

We finally prove the main result of this section, i.e., Theorem 3.1.

Proof of Theorem 3.1. Let $G = (V, E)$ be a graph and consider an $(2r-1, O(rn^{1/r} \log n), O(rn^{1/r} \log^2 n), O(rn^{1/r} \log n))$ -efficient (distance) IVS H of G (Lemma 3.5). Recall that given any pair of vertices s, t in G , one can compute shortest path between s and t in $O(m \log n)$ time. Thus, plugging the parameters $\alpha = 2r - 1$, $f(n) = O(rn^{1/r} \log n)$, $g(n) = O(rn^{1/r} \log^2 n)$, $r(n) = O(rn^{1/r} \log n)$ and $h(n) = O(\log n)$ in Theorem 2.4 and choosing the running time parameters as in Lemma 2.5, we get an incremental algorithm such that for any pair of vertices u and v reports a query estimate $\delta(u, v)$ that approximates the shortest path distance between u and v up to a $(2r - 1)^\ell$ factor, and handles update operations in worst-case time of

$$O\left(\ell c^{\ell-1} r^\ell m^{1/(\ell+1)} n^{\ell/r} \log^{\ell-1} n\right)$$

and query operations in worst-case time of

$$O\left(\ell^2 c^{\ell-1} r^\ell m^{1/(\ell+1)} n^{\ell/r} \log^{\ell-1} n\right).$$

\square

4 Offline Dynamic Algorithms via Vertex Sparsifiers

In this section, we show how to use efficient vertex sparsifier constructions to design *offline* (approximate) dynamic algorithms for graph problems with certain properties while achieving fast amortized update and query time. To achieve this we use a framework that has been exploited for solving offline 3-connectivity [PSS19]. Our main contribution is to show that this generalizes to a much wider class of problems, leading to several interesting bounds which are not yet known in the *online* dynamic graph literature. Also, we show negative results on lower-bounds in dynamic problems.

We start by defining the model. We are given an undirected graph $G = (V, E)$ and an *offline* sequence of events or operations x_1, \dots, x_m , where x_i is either an edge update (insertion or deletion), or a query q_i which asks about some graph property in G at time i . The goal is to process this sequence of updates in G while spending total time proportional to $O(mf(m))$, where $f(m)$ is ideally some sub-linear function in m .

We next show that an analogue to Theorem 2.4 can also be obtained in the offline graph setting. Our algorithm makes use of the notion of vertex sparsifiers as well as their useful properties including transitivity and decomposability. In our construction we want graph properties that admit (1) fast algorithms for computing vertex sparsifiers and (2) guarantee that the size of such sparsifiers is reasonably small. We formalize these requirements in the following definition.

Definition 4.1. Let $G = (V, E)$ be a graph, with a terminal set $T \subseteq V$ and let $f(n), s(n) \geq 1$ be functions. We say that $(G', \alpha, f(n), s(n))$ is an α -efficient vertex sparsifier of G with respect to T iff G' is an α -vertex sparsifier of G , the time to construct G' is $O(m \cdot f(n))$, and the size of G' is $O(|T| \cdot s(n))$.

Theorem 4.2. Let $G = (V, E)$ be a graph, and for any $u, v \in V$, let $\mathcal{P}(u, v, G)$ be a solution to a minimization problem between u and v in G . Let $f(n), s(n), h(n) \geq 1$ be functions, $\alpha, \ell \geq 1$ be parameters associated with the approximation factor, and let $\beta_0, \beta_1, \dots, \beta_\ell$ with $\beta_0 = m$ be parameters associated with the running time. Assume the following properties are satisfied

1. G admits an efficient vertex sparsifier $(G', \alpha, f(n), s(n))$,
2. G' is transitive and decomposable,
3. The property $\mathcal{P}(u, v, G)$ can be computed in $O(mh(n))$ time in a graph with m edges and n vertices.

Then there is an offline (approximate) dynamic algorithm that maintains for every pair of nodes u and v , an estimate $\delta(u, v)$, such that

$$\mathcal{P}(u, v, G) \leq \delta(u, v) \leq \alpha^\ell \cdot \mathcal{P}(u, v, G). \quad (4)$$

The total time for processing a sequence of m operations is:

$$\tilde{O} \left(\beta_0 \left(\sum_{j=1}^{\ell} \left(\frac{\beta_{j-1}}{\beta_j} \right) f(n) + \beta_\ell h(n) \right) s(n) \right) \quad \text{where } \beta_0 = m. \quad (5)$$

Before describing the underlying data-structure upon which the above theorem builds, we reduce the arbitrary sequence of operations into a more structured one, and also build a particular view for the problem. These will allow us to greatly simplify the presentation.

Concretely, first we may assume that each edge is inserted and deleted exactly once during the sequence of operations. We achieve this by simply treating each edge instance as a new edge, i.e., we assume that each insertion of an edge $e = (u, v)$ inserts a new edge that is different from all previous instances of (u, v) .

Second, since we are given the entire sequence of operations, for each edge e we associate an interval $[i_e, d_e]$ which indicates the insertion and deletion time of e in the operation sequence. Furthermore, we denote by q_t the time when query q was asked in the operation sequence. Let $[1, m]$ denote the interval covering the entire event sequence. If we are interested in processing updates from a given interval $[r, s]$, we will define graphs that consists of two types of edges with respect to this interval:

1. *non-permanent edges*, which are edges affected by an event in this interval, i.e., $E_{[r, s]}^p = \{e \mid i_e \text{ or } d_e \in [r, s]\}$,
2. *permanant edges*, which are edges present throughout the entire interval, i.e., $E_{[r, s]}^{np} = \{e \mid i_e < r \leq s < d_e\}$.

Additionally, it will be useful to define the queried vertex pairs within the interval $[r, s]$: $Q_{[r, s]} = \{q \mid q_t \in [r, s]\}$.

Data Structure. We now describe a generic tree data-structure \mathcal{T} , which allows us to unify our framework and thus greatly simplify the presentation. This tree structure is obtained by hierarchically partitioning the operation sequence into smaller disjoint intervals. These intervals induce graphs that are suitable for applying vertex sparsifiers, which in turn allow us to process updates in a fast way, while paying some error in the accuracy of the query operations.

Consider some integer parameter $\ell \geq 1$ and parameters $\beta_0, \beta_1, \dots, \beta_\ell$ with $\beta_0 = m$. The tree \mathcal{T} has $\ell + 1$ levels, where each level i is associated with the parameter β_i , $i = 0, \dots, \ell$. Each node of the tree stores some interval from the event sequence. Formally, our decomposition tree T satisfies the following properties:

1. The root of the tree stores the interval $[1, m]$.
2. The intervals stored at nodes of same level are disjoint.
3. Each interval $[r, s]$ stored at a node in \mathcal{T} is associated with
 - a graph $G_{[r,s]} = (V, E_{[r,s]}^p)$,
 - a graph of *new permanent edges* $H_{[r,s]} = G_{[r,s]} \setminus G_{[q,t]}$, where $G_{q,t}$ is the parent of $G_{[r,s]}$ in \mathcal{T} (if any).
 - a set of *boundary vertices* $\partial_{[r,s]} = V(E_{[r,s]}^{np}) \cup V(Q_{[r,s]})$.
4. If $[r, s] \subseteq [q, t]$ then it holds that (a) $\partial_{[r,s]} \subseteq \partial_{[q,t]}$, and (b) $E_{[q,t]}^p \subseteq E_{[r,s]}^p$.
5. The length of the interval stored at a node at level i is β_i .
6. A node at level i has β_i/β_{i+1} children.
7. The number of nodes at level i is at most $O(\beta_0/\beta_i)$.

The lemma below shows that a decomposition tree can be constructed in time proportional to the length of the operation sequence times the height of the tree.

Lemma 4.3. *Let $G = (V, E)$ be a dynamic graph where the sequence of operations is revealed upfront. Then there is an algorithm that computes the decomposition tree \mathcal{T} in $O(\ell m)$ time, where m denotes the length of the operation sequence and ℓ is the height of the tree.*

Proof. Let \mathcal{T} be a tree with a single node (corresponding to its root) that stores the interval $[1, m]$. We augment \mathcal{T} in the following natural way: (a) We partition the interval $[1, m]$ into $\beta_0/\beta_1 = m/\beta_1$ disjoint intervals, each of length β_1 . (b) For each of these intervals we create a node in the tree \mathcal{T} , and connect each node with the root of \mathcal{T} , i.e., those nodes form the children of the root, and thus the nodes at level 1 of \mathcal{T} . (c) We recursively apply steps (a) and (b) to the newly generated nodes until we reach the $(\ell + 1)$ -st level of the tree.

By the construction above, it easily follows that the generated tree \mathcal{T} satisfies properties (1), (2), (4), (5), (6) and (7). Thus, it remains to show how to compute the quantities in (3). This can be achieved by (a) computing the intervals $[i_e, d_e]$, for every edge e in the sequence (note that this is possible because we assumed that every edge is inserted and deleted exactly once within the interval $[1, m]$), and (2) for each node in the tree, computing the sets $E_{[r,s]}^{np}$ and $E_{[r,s]}^p$.

For the running time, observe that computing the intervals $[i_e, d_e]$ takes $O(m)$ time. Having computed these intervals, we can level-wise compute the permanent and non-permanent edges for each node in that particular level. By disjointedness of the intervals, the amount of work we perform per level is $O(m)$. Since there are most $O(\ell)$ levels, it follows that the running time for constructing the decomposition tree is $O(\ell m)$. \square

Computing vertex sparsifiers in the hierarchy. We next show how to efficiently compute a vertex sparsifier $G'_{[r,s]}$ for each node $G_{[r,s]}$ from the decomposition tree \mathcal{T} . The main idea behind this algorithm is to leverage the sparsifier computed at the parent nodes as well as apply the efficient vertex sparsifiers from Theorem 4.2 Part 1. The procedure accomplishing this task for a single node of the tree \mathcal{T} is formally given in Algorithm 10. To compute the vertex sparsifier for every node, we simply apply it in a top-down fashion to the nodes of \mathcal{T} .

Algorithm 10: VERTEXSPARSIFY($G_{[r,s]}$)

```

1 if  $G_{[r,s]}$  is the root node then
2    $G''_{[r,s]} = G'_{[r,s]} \leftarrow (V, \emptyset)$ , i.e, the empty graph.
3 else
4   Let  $G_{[q,t]}$  be the parent of  $G_{[r,s]}$  in  $\mathcal{T}$ 
5    $G''_{[r,s]} \leftarrow (G'_{[q,t]} \cup H_{[r,s]})$ , where  $G'_{[q,t]}$  is an efficient vertex sparsifier of  $G_{[q,t]}$  with respect to
    $\partial_{[q,t]}$ 
6   Let  $G'_{[r,s]}$  be an  $\alpha$ -efficient vertex sparsifier of  $G''_{[r,s]}$  with respect to  $\partial_{[r,s]}$  (Theorem 4.2 Part 1)
7 return  $G'_{[r,s]}$ 

```

To argue about the usefulness of Algorithm 10, we need to bound the quality of sparsifiers produced at the nodes of \mathcal{T} . The lemma below show that the quality grows multiplicatively with the number of levels in \mathcal{T} .

Lemma 4.4. *Let $G_{[r,s]}$ be a node of \mathcal{T} at level $i \geq 0$. Then $G' = \text{VERTEXSPARSIFY}(G_{[r,s]})$ outputs an α^i -efficient vertex sparsifier of $G_{[r,s]}$ with respect to $\partial_{[r,s]}$.*

Proof. We proceed by induction on i . For the base case, i.e., $i = 0$, $G_{[1,m]}$ is the root node. Since $E_{[1,m]}^p = \emptyset$ by definition of permanent edges, we get that $G'_{[1,m]} = G_{[1,m]}$, i.e., $G_{[1,m]}$ is a sparsifier of itself.

Let $G_{[r,s]}$ be a node at level $i > 0$. Let $G_{[q,t]}$ be the parent of $G_{[r,s]}$ in \mathcal{T} , and let $G'_{[q,t]}$ be its cut sparsifier at level $(i - 1)$, as defined in Algorithm 10. By Property (4) of \mathcal{T} note that $E_{[q,t]}^p \subseteq E_{[r,s]}^p$ since $[r,s] \subseteq [q,t]$. Also recall that $H_{[r,s]} = G_{[r,s]} \setminus G_{[q,t]}$. By induction hypothesis, we know that $G'_{[q,t]}$ is an α^{i-1} -efficient vertex sparsifier of $G_{[q,t]}$ with respect to $\partial_{[q,t]}$. This together with the decomposability property in Theorem 4.2 Part 2 imply that that $G''_{[r,s]} = G'_{[q,t]} \cup (G_{[r,s]} \setminus G_{[q,t]})$ is an α^{i-1} -efficient vertex sparsifier of $G_{[q,t]} \cup (G_{[r,s]} \setminus G_{[q,t]}) = G_{[r,s]}$ with respect to $\partial_{[q,t]}$. Now, by Theorem 4.2 Part 1 we get that $G'_{[r,s]}$ is an α -efficient vertex sparsifier of $G''_{[r,s]}$ with respect to $\partial_{[r,s]}$. Since $\partial_{[r,s]} \subseteq \partial_{[q,t]}$, and applying the transitivity property (Theorem 4.2 Part 2) on $G'_{[r,s]}$ and $G''_{[r,s]}$, we get that $G'_{[r,s]}$ is an $\alpha^{i-1+1} = \alpha^i$ -efficient vertex sparsifier of $G_{[r,s]}$. \square

We now state a crucial property of the nodes in the decomposition tree \mathcal{T} , which allows us to get a reasonable bound on the running time for computing vertex sparsifiers for the nodes in \mathcal{T} .

Lemma 4.5. *Let $G_{[r,s]}$ be a node in the decomposition tree \mathcal{T} , and let $G_{[q,t]}$ be its parent. Then we have that the number of new permanent edges of $G_{[r,s]}$ is bounded by the number of non-permanent edges of its parent, i.e., $|E(H_{[r,s]})| \leq |E_{[q,t]}^{np}|$.*

Proof. If an edges in in $H_{[r,s]}$, then it is not in $E_{[r,s]}^p$, thus it is a non-permanent edge in $G_{[q,t]}$. \square

The lemma below gives a bound on the running time for computing vertex sparsifiers in \mathcal{T} .

Lemma 4.6. *The total running time for computing the vertex sparsifiers for each node in the decomposition tree T of height ℓ is bounded by*

$$\tilde{O}\left(\beta_0 \cdot \left(\sum_{j=1}^{\ell} \frac{\beta_{j-1}}{\beta_j} f(n)s(n)\right)\right), \quad \text{where } \beta_0 = m.$$

Proof. For $i \geq 1$, let $Y(i)$ be the total time for computing the vertex sparsifiers for all the nodes in \mathcal{T} up to (and including) level i . Furthermore, let $Z(i)$ be the total time for computing the vertex sparsifier of the nodes at level i in Y (and excluding other levels). We will show by induction on the number of levels i that $Y(i) = O\left(\beta_0 \cdot \left(\sum_{j=1}^i \frac{\beta_{j-1}}{\beta_j}\right) f(n)s(n)\right)$, which with $i = \ell$ implies the claim we want to prove.

For the base case, i.e., $i = 1$, consider any node $G_{[r,s]}$ at level 1 of \mathcal{T} . By construction of \mathcal{T} , $G_{[r,s]}$ contains at most $O(\beta_0)$ permanent edges. Furthermore, note that the parent of $G_{[r,s]}$ is the root node $G_{[1,m]}$, for which $G'_{[1,m]} = (V, \emptyset)$. Thus, by Theorem 4.2 Part 1 we get that the time to compute an efficient vertex sparsifier per node is $O(\beta_0 \cdot f(n))$. By Property (7) of \mathcal{T} , the number of nodes at level 1 is $O(\beta_0/\beta_1)$, implying that the total running time is $Y(1) = Z(1) = O\left(\beta_0 \left(\frac{\beta_0}{\beta_1}\right) f(n)\right) = O\left(\beta_0 \left(\frac{\beta_0}{\beta_1}\right) f(n)g(n)\right)$, as desired.

We next show the inductive step. Let $G_{[r,s]}$ be a node at level $i > 1$, and let $G_{[q,t]}$ be its parent. We want to bound the size of the intermediate graph $G''_{[r,s]} = (G'_{[q,t]} \cup H_{[r,s]})$, as defined in Algorithm 10, which in turn determines the running time for computing an efficient vertex sparsifier of $G_{[r,s]}$. To this end, first observe that Theorem 4.2 Part 1 implies that the size of sparsifier $G'_{[q,t]}$ of $G_{[q,t]}$ is bounded by

$$O(|\partial_{[q,t]}| \cdot s(n)) \leq |V(E_{[r,s]}^{np}) \cup V(Q_{[r,s]})| \cdot s(n) \leq O(\beta_{i-1} \cdot s(n)),$$

since the number of non-permanent edges and queries is proportional to the length of the interval being considered. Second, by Lemma 4.5, we also have that $|E(H_{[r,s]})| \leq |E_{[q,t]}^{np}| \leq O(\beta_{i-1})$, thus giving that $|G''_{[r,s]}| \leq O(\beta_{i-1} \cdot s(n))$. As Algorithm 10 runs CUTSPARSIFY on the graph $G''_{[r,s]}$, Theorem 4.2 Part 1 gives that the running time to compute an efficient vertex sparsifier for the node $G_{[r,s]}$ is $O(\beta_{i-1} \cdot f(n)s(n))$, and that its size is $O(\beta_{i-1} \cdot s(n))$. Combining this together with the fact that the number of nodes at level i is at most $O(\beta_0/\beta_i)$ (Property (7) of \mathcal{T}) imply that

$$Z(i) = O\left(\beta_0 \cdot \frac{\beta_{i-1}}{\beta_i} f(n)s(n)\right).$$

To complete the inductive step, note that by induction hypothesis,

$$Y(i-1) = O\left(\beta_0 \cdot \left(\sum_{j=1}^{i-1} \frac{\beta_{j-1}}{\beta_j}\right) f(n)s(n)\right).$$

Summing over this and the bound on $Z(i)$ we get

$$\begin{aligned} Y(i) &= Y(i-1) + Z(i) \\ &= O\left(\beta_0 \cdot \left(\sum_{j=1}^{i-1} \frac{\beta_{j-1}}{\beta_j}\right) f(n)s(n)\right) + O\left(\beta_0 \cdot \left(\frac{\beta_{i-1}}{\beta_i}\right) f(n)s(n)\right) \\ &= O\left(\beta_0 \cdot \left(\sum_{j=1}^i \frac{\beta_{j-1}}{\beta_j} f(n)s(n)\right)\right). \end{aligned}$$

□

Processing operations in the hierarchy. So far we have shown how to reduce the sequence of operations into smaller intervals in a hierarchical manner, while (approximately) preserving the properties of the edges and queries involved in the offline sequence. In what follows, we observe that for processing these events, it is sufficient to process the nodes (and their corresponding intervals) stored at the last level ℓ of the tree decomposition \mathcal{T} (note that this is possible because intervals at level ℓ form a partitioning of the event sequence $[1, m]$, and all vertex pairs within intervals that will be involved in edge updates or queries are preserved using vertex sparsifiers).

The algorithm for processing the updates is quite simple: for every node $G_{[r,s]}$ at level ℓ of \mathcal{T} , we process all operations in the interval consecutively: for each edge insertion or deletion we add or remove that suitable edges to $G'_{[r,s]}$, and for each query (x, y) we run on the vertex sparsifier $G'_{[r,s]}$ the static algorithm from Theorem 4.2 Part 3 to calculate the property $\mathcal{P}(x, y, G'_{[r,s]})$ between x and y in $G'_{[r,s]}$. (note that this is possible since $\partial_{[r,s]} \supseteq \{x, y\}$ by construction of \mathcal{T}).

We next analyze the total time for processing the sequence of events in the last level of \mathcal{T} .

Lemma 4.7. *The total time for processing the whole sequence of operations at level ℓ of the decomposition tree \mathcal{T} is $\tilde{O}(\beta_0 \beta_\ell \cdot s(n)h(n))$, where $\beta_0 = m$.*

Proof. As in the worst-case there can be at most $O(\beta_\ell)$ queries within the interval, and since the size of $G'_{[r,s]}$ is also bounded by $O(\beta_\ell s(n))$, by Theorem 4.2 Part 3 it follows that answering all the queries and processing the non-permanent edges within a single interval at level ℓ is bounded by $\tilde{O}(\beta_\ell^2 s(n)h(n))$. Combining this with the fact that the number of nodes at level ℓ is $O(\beta_0/\beta_\ell)$ (Property (7) of \mathcal{T}), we get that the total cost for processing the queries is $\tilde{O}(\beta_0 \beta_\ell \cdot s(n)h(n))$. \square

Combining Lemma 4.6 and Lemma 4.7 leads to an overall performance of

$$\tilde{O}\left(\beta_0 \left(\sum_{j=1}^{\ell} \left(\frac{\beta_{j-1}}{\beta_j}\right) f(n) + \beta_\ell h(n)\right) s(n)\right) \quad \text{where } \beta_0 = m,$$

which proves the claimed total update time in Theorem 4.2.

We finally prove the correctness of our algorithm. Concretely, we show that the estimate we return when processing any query (x, y) in the last level of the hierarchy approximates the property \mathcal{P} of the graph G up to an α^ℓ factor, thus proving the claimed estimate in Theorem 4.2.

To this end, let q_i be a query in the sequence of operations $[1, m]$. Since the intervals at level ℓ of \mathcal{T} form a partitioning of $[1, m]$, there must exist an interval $[r, s]$ that contains the query q_i . Let (x, y) be the queried vertex pair of q_i . By Lemma 4.4, we get that the graph $G'_{[r,s]}$ at level ℓ is an α^ℓ -vertex sparsifier of $G_{[r,s]}$ with respect to $\partial_{[r,s]}$. Since by construction $\partial_{[r,s]} \supseteq \{x, y\}$, we get that the $G'_{[r,s]}$ approximates the property $\mathcal{P}(x, y, G)$ of $G_{[r,s]}$ up to an α^ℓ factor. Finally, recall that we run the algorithm from Theorem 4.2 Part 3 on $G'_{[r,s]}$, thus worsening the approximation in the worst-case by at most a constant factor, which yields the claimed bound.

4.1 Applications to Offline Shortest Paths and Max-Flow

In this sub-section, we show how to use our general Theorem 4.2 to design offline dynamic algorithms for the approximate All Pair Shortest Paths and All Pair Max-Flow with reasonably small total update time.

We first consider shortest paths. Recall that our goal is to show that assumptions (1), (2) and (3) from Theorem 4.2 are satisfied with certain parameters for the shortest path measure. For (1) we make the following observation: given a graph G , a subset of terminals T , and a parameter $r \geq 1$, we can construct an *efficient* (vertex) distance sparsifier $(H_T, (2r - 1), \tilde{O}(n^{1/r}), \tilde{O}(n^{1/r}))$ by simply constructing an efficient incremental vertex sparsifier for G using Lemma 3.5 and add T to its terminal set. Also note that assumption

(2) is satisfied by the transitivity and decomposability of H , and finally recall that (3) follows by any $\tilde{O}(m)$ time single pair shortest path algorithm. These together imply the following result.

Theorem 4.8. *Let $G = (V, E)$ be an undirected, weighted graph. For every $r, \ell \geq 1$, there is an offline fully dynamic approximate All Pair Shortest Path algorithm that maintains for every pair of nodes u and v , a distance estimate $\delta(u, v)$ such that*

$$\text{dist}_G(u, v) \leq \delta(u, v) \leq (2r - 1)^\ell \text{dist}_G(u, v).$$

The total time for processing a sequence of m operations is

$$\tilde{O}(m \cdot m^{1/(\ell+1)} n^{2/r}).$$

We now proceed with max flow. Following essentially the same idea as with shortest paths, we need to show that assumptions (1), (2) and (3) from Theorem 4.2 are satisfied with certain parameters for the max flow measure. For (1) we show the existence of *efficient* (vertex) flow sparsifier ($H_T, O(\log^4 n), \tilde{O}(1), \tilde{O}(1)$) via the following lemma.

Theorem 4.9 ([RST14, Pen16]). *Given an undirected, weighted graph $G = (V, E)$, there is an $\tilde{O}(m)$ time randomized algorithm $\text{FLOWSPARSIFY}(G)$ that with high probability computes a tree-based flow sparsifier $H = (V', E')$ with $V \subseteq V'$ satisfying the following properties*

1. H is a bounded degree rooted tree,
2. H has quality $O(\log^4 n)$,
3. There is a one-to-one correspondence between the leaf nodes of H and the nodes in G ,
4. The height of H is at most $O(\log^2 n)$.

Proof. The original construction of Räcke et al. [RST14] produces a rooted tree H' which satisfies the above properties, except that H' has unbounded degree and the height of the tree is $O(\log n)$. Since we will exploit the bounded degree assumption in the subsequent applications of our data-structure, here we present a standard reduction from H' to a bounded degree tree H at the cost of increasing the height of the tree by a logarithmic factor.

Let H' be the rooted tree we described above. Let $u \in H'$ be an internal node of degree larger than 2 and let $C(u)$ be its children. We start by removing all edges between u and its children $C(u)$ from H' , and record all their corresponding edge weights. Next, we create a binary rooted tree \tilde{H} where the children $C(u)$ are the leaf nodes, i.e., $L(\tilde{H}) = C(u)$, and u is the root of \tilde{H} . To complete the construction of \tilde{H} we need to define its edge weights. To this end, for any subtree $R \subseteq \tilde{H}$, let $E(L(R))$ denote the set of edges incident to leaf nodes in R . We distinguish the following two cases. (1) If $e = (x, y) \in E(L(\tilde{H}))$, i.e., e is an edge incident to a leaf of \tilde{H} , and $x \in L(\tilde{H}) = C(u)$, we set $w_{\tilde{H}}(x, y) = w_{H'}(x, u)$. (2) If $e = (x, y) \notin E(L(\tilde{H}))$, then let \tilde{H}_x and \tilde{H}_y be the trees obtained after deleting the edge e from \tilde{H} . Furthermore, for any subtree $R \subseteq \tilde{H}$ define

$$w(R) := \sum_{e \in E(L(R))} w_{\tilde{H}}(e).$$

Finally, for $e = (x, y) \notin E(L(\tilde{H}))$ and $e \in \tilde{H}$ we set

$$w_{\tilde{H}}(x, y) = \min\{w(\tilde{H}_x), w(\tilde{H}_y)\}.$$

Note that the weight-sums $w(\tilde{H}_x)$ and $w(\tilde{H}_y)$ can be calculated since we first defined the weights for edges in $E(L(\tilde{H}))$. Also observe that H' remains a tree because we simply removed children of u (which could

be viewed as a star) and replaced this by another bounded degree tree \tilde{H} . We repeat the above process for every internal node of H' until H' becomes a bounded degree rooted tree, and denote by H the final resulting tree.

We claim that H has depth at most $O(\log^2 n)$. To see this, recall that the initial height of H' was $O(\log n)$, and every replacement of the star centered at a non-terminal with a bounded degree tree increases the height by an additive of $O(\log n)$. Summing over $O(\log n)$ levels gives the claimed bound.

Finally, it is easy to see that H is flow sparsifier of quality 1 for H' with respect to all leaf nodes of H' , which in turn correspond to the nodes of graph G . Thus, H is also a flow sparsifier for G with quality $O(\log^4 n)$. \square

We next show how to construct a vertex sparsifier w.r.t. a given terminal set T .

The construction involves 2 phases: (1) preprocessing, and (2) constructing vertex sparsifier. In the preprocessing phase, given a graph G , we simply invoke $\text{FLOWSPARSIFY}(G)$ from Theorem 4.9 and let H be the resulting tree-based sparsifier. The main idea for constructing a (vertex) flow sparsifier H_T of G with respect to T is to exploit the fact that H is a tree. Concretely, let H_T be an initially empty graph. For $v \in T$, let $H[v, r]$ be the path in H from v to the root r of H (since $v \in T \subseteq V$, recall that v is a leaf node of H by Property (3) in Theorem 4.9). For each $v \in T$, and every edge $e \in H[v, r]$, we add e with weight $w_H(e)$ to H_T . Finally, we return H_T as a (vertex) flow sparsifier of G with respect to T . The following lemma analyzes the running time for the above procedure and shows the correctness.

Lemma 4.10. *Given an undirected, weighted graph $G = (V, E)$, and a subset of vertices T , there is an algorithm that produce a (vertex) flow sparsifier H_T w.r.t. T in time $\tilde{O}(m)$. H_T has size $O(|T| \log^2 n)$ and quality $O(\log^4 n)$.*

In other words, there is an efficient (vertex) flow sparsifier $(H_T, O(\log^4 n), \tilde{O}(1), \tilde{O}(1))$.

Proof of Lemma 4.10. We first argue about the correctness of the output H_T . First, we show that H_T is 1-(vertex) flow sparsifier of H with respect to T . To see this, note that since H is a tree, every (multi-commodity) flow among any two leaf vertices (u, v) is routed according to the unique shortest path between u and v in H , denoted by $H[u, v]$. Since H_T is formed taking the union of the paths $H[v, r]$, for each $v \in T$, and $H[u, v] \subseteq (H[v, r] \cup H[u, r])$, it follows that $H[u, v]$ is also contained in H_T . Thus every flow that we can route in H among any two pairs in T , we can feasible route in H_T . For the other direction, observe that by construction $H_T \subseteq H$. Therefore, any flow among any two pairs in T that can be feasibly routed in H_T , can also be routed in H (this follows since H has more edges than H_T , and thus the routing in H has more flexibility). Combining the above we get that H_T is a quality 1-(vertex) flow sparsifier of H . Since H is flow sparsifier of G with quality $O(\log^4 n)$ (Property (2) in Theorem 4.9) and $T \subseteq V$, applying transitivity on H_T and H we get that H_T is a quality $O(\log^4 n)$ (vertex) flow sparsifier of G with respect to T .

We finally analyze the running time for both operations. Recall that the preprocessing phase is implemented by simply invoking $\text{FLOWSPARSIFY}(G)$. By Theorem 4.9, we know that the latter can be implemented in $\tilde{O}(m)$, which in turn bounds the running time of our preprocessing phase. For the running time of constructing vertex sparsifier, recall that H_T consists of the union over the paths $P(v, r, H)$, for each $v \in T$. Since the length of each such path is bounded by $O(\log^2 n)$ (Property (4) in Theorem 4.9), we get that the size of H_T is bounded by $O(|T| \log^2 n)$. Note that after having access to any leaf vertex v , the path $H[v, r]$ can be retrieved from H in time proportional to its length. This implies that the time to output H_T is also bounded by $O(|T| \log^2 n)$. \square

Also note that assumption (2) is satisfied by the transitivity and decomposability of H , and finally recall that (3) follows by employing the $\tilde{O}(m)$ time (approximate) (s, t) -maximum flow algorithm due to Peng [Pen16]. These together imply the following theorem.

Theorem 4.11. *Let $G = (V, E)$ be an undirected, weighted graph. For every $\ell \geq 1$, there is an offline fully dynamic approximate All Pairs Max Flow algorithm that maintains for every pair of nodes u and v , a flow estimate $\delta(u, v)$ such that*

$$\frac{1}{\tilde{O}(\log^{4\ell} n)} \max\text{-flow}_G(u, v) \leq \delta(u, v) \leq \max\text{-flow}_G(u, v).$$

The total time for processing a sequence of m operations is

$$\tilde{O}(m \cdot m^{1/(\ell+1)}).$$

4.2 Implications on Hardness of Approximate Dynamic Problems

4.2.1 Approximate max flow and cut sparsifiers

Assuming the OMv conjecture, Dahlgaard [Dah16] show that any incremental exact max flow algorithm on undirected graphs must have amortized update time at least $\Omega(n^{1-o(1)})$. However, the hardness of approximation is not known⁵:

Proposition 4.12. *There is no polynomial lower bound for dynamic $\omega(\text{polylog}(n))$ -approximate max flow in the offline setting.*

This follows directly from Theorem 4.11. Thus the important open problem is whether we can prove a hardness for dynamic $(1 + \epsilon)$ -approximate max flow algorithms on undirected graphs for a constant $\epsilon > 0$.

On the other hand, it is not known whether, given a set of k terminals, there is a $(1 + \epsilon)$ -approximate cut (vertex) sparsifier of size $\text{poly}(k, 1/\epsilon)$ or even $\text{poly}(k, 1/\epsilon, \log n)$. If a cut sparsifier can only contain terminals as nodes, then the approximation ratio must be at least $\Omega(\sqrt{\log k}/\log \log k)$ [MM10]. If we need an exact cut sparsifier, then the size must be at least $2^{\Omega(k)}$ [KR17].

In what follows we draw a connection between these two open problems; if there is a very efficient algorithm for the above cut sparsifier, then there cannot be a $\Omega(n^{1-o(1)})$ lower bound in the offline setting for the dynamic approximate max flow. Moreover, if the cut-sparsifier has size almost best possible, then there cannot be even a super-polylogarithmic lower bound. Concretely, we show the following.

Theorem 4.13. *If there is an algorithm that, given a undirected graph $G = (V, E)$ with m edges and a set $T \subset V$ of k terminals, constructs an $(1 + \epsilon)$ -approximate cut vertex sparsifier of size $s = \text{poly}(k, 1/\epsilon, \log n)$ in time $O(m \text{poly}(\log n, 1/\epsilon))$, there is an offline dynamic algorithm for maintaining $(1 + \epsilon')$ -approximate value of max flow with update time $u = O(n^{1-\gamma} \text{poly}(1/\epsilon'))$ for some constant $\gamma > 0$. Moreover, if the size of the sparsifier $s = k \cdot \text{poly}(1/\epsilon, \log n)$, then we obtain the update time of $u = O(\text{poly}(\log n, 1/\epsilon'))$. The dynamic algorithm is Monte Carlo randomized and it is correct with high probability.*

Proof. Let us assume ϵ' is a constant for simplicity. The proof generalizes easily when ϵ' is not a constant.

First, we only need to consider offline dynamic algorithms where the underlying graph has $m = \tilde{O}(n)$ edges at every time step and the length of the update sequences is n . This is because there is a dynamic algorithm by [ADK⁺16] that can maintain a cut sparsifier $H = (V, E')$ of a graph $G = (V, E)$ when the terminal set is V with $\tilde{O}(1)$ worst-case update. So we can work on H instead, and divide the update sequences into segments of length n . If we have an offline dynamic algorithm with update time u on average on each period, then the average update time is $\tilde{O}(u)$ over the whole sequence.

We set $\epsilon = \epsilon'/10 \log n$. Suppose that the sparsifier from the assumption has size only $s = k \cdot \text{poly}(1/\epsilon, \log n) = \tilde{O}(k)$. Then, we apply the same proof as in Theorem 4.11, except that the number of levels of the decomposition tree will be $\log n$ instead of $O(\sqrt{\log n})$. The quality of the cut-sparsifier at any level is at most

⁵However, on directed graphs, the hardness of approximation is known. This is because even dynamic reachability is hard under several conjectures[AW14, HKNS15].

$(1 + \epsilon)^{\log n} = (1 + \epsilon'/10 \log n)^{\log n} \leq (1 + \epsilon')$. The total running time will be $\tilde{O}(m^{1+\frac{1}{\log n+1}}) = \tilde{O}(n)$. The latter implies that update time on average is $O(\text{polylog}(n))$.

Assume that $s = k^c \cdot \text{poly}(1/\epsilon, \log n) = \tilde{O}(k^c)$ for some constant $c > 1$. Then, we can apply again the same proof from Theorem 4.11. By using only two levels of the decomposition tree, we can obtain an update time of $\tilde{O}(n^{1-\frac{1}{c+1}})$. Concretely, if we set $\beta_0 = m$ and $\beta_1 = m^{1/(c+1)}$ then the time for computing the decomposition tree is $\frac{\beta_0}{\beta_1} \cdot \tilde{O}(\beta_0) = \tilde{O}(n^{2-\frac{1}{c+1}})$. The total time for running approximate max flow on the cut-sparsifier in the second level at each step is $\beta_0 \cdot \tilde{O}(\beta_1^c) = \tilde{O}(n^{2-\frac{1}{c+1}})$. Thus it follows that the update time is $\tilde{O}(n^{1-\frac{1}{c+1}})$ on average. \square

4.2.2 Approximate distance oracles on general graphs

There are previous hardness results for approximation algorithms for dynamic shortest path problems (including single-pair, single-source and all-pairs problems) [HKNS15]. All such results show a very high lower bound, e.g. $\Omega(n^{1-\epsilon})$ or $\Omega(n^{1/2-\epsilon})$ time on an n -node graph. However, they hold only when the approximation factor is a small constant. It is open whether one can obtain weaker polynomial lower bounds for larger approximation factors. We show that it is impossible to show super-constant factor lower-bounds in several settings.

Proposition 4.14. *There is no polynomial lower bound for dynamic $\omega(1)$ -approximate distance oracles in the offline setting (and also in the online incremental setting).*

More formally, for any lower bound stating that $\omega(1)$ -approximate offline dynamic distance oracle algorithm on n -node graphs requires at least $u(n)$ update time or $q(n)$ query time, then we have $u(n) = n^{o(1)}$ and $q(n) = n^{o(1)}$. The same holds for online incremental algorithm with worst-case update time.

This follows directly from Theorems 3.1 and 4.8.

4.2.3 Approximate distance oracles on planar graphs

Similar to the situations above, assuming the APSP conjecture, Abboud and Dahlgaard [AD16] show that any offline fully dynamic algorithm for exact distance oracles on planar graph requires either update time or query time of $\Omega(n^{1/2-o(1)})$. We can still hope for a hardness result for $(1 + \epsilon)$ -approximate distance oracles, but this remains an important open problem in the field of dynamic algorithms.

Recall the definition of *distance approximating minors*, which are vertex distance sparsifiers that are required to be minors of the input graph. In the exact setting, Krauthgamer et al. [KNZ14] showed that any distance preserving minor with respect to k terminals, even when restricted to planar graphs, must have size $\Omega(k^2)$ size. Cheung et al. [CGH16] showed that for planar graphs there is a $(1 + \epsilon)$ -distance approximating minor of size $\tilde{O}(k^2 \epsilon^{-2})$. The natural question is whether there is a $(1 + \epsilon)$ -approximate *minor* distance sparsifier for k terminals that has size $k^{1.99} \cdot \text{poly}(1/\epsilon, \log n)$.

We again draw a connection between dynamic graph algorithms and vertex sparsifiers; if there is a very efficient algorithm for such distance sparsifiers, then we cannot extend the $\Omega(n^{1/2-o(1)})$ lower bound to the approximate setting. Moreover, if the sparsifier has the (almost) best possible size, then there cannot be even a super-polylogarithmic lower bound. More precisely, we show the following.

Theorem 4.15. *Let G be an undirected graph $G = (V, E)$ with m edges and a set $T \subset V$ of k terminals. If there is an algorithm that constructs a $(1 + \epsilon)$ -distance approximating minor of size $s = k^{2/(1+3\gamma)} \cdot \text{poly}(1/\epsilon, \log n)$, for some constant $0 < \gamma \leq 1/3$, in time $O(m \text{poly}(\log n, 1/\epsilon))$, then there is an offline dynamic $(1 + \epsilon')$ -approximate distance oracle algorithm for with update and query time $u = O(n^{1/2-\gamma/2})$. In fact, if the size of the sparsifier is $s = k \cdot \text{poly}(1/\epsilon', \log n)$, then we obtain an update and query time of $u = O(\text{poly}(\log n))$.*

The proof will be very similar to the one in Theorem 4.13 except that we need to be more careful about planarity. Thus we first prove the following useful lemma.

Lemma 4.16. *Each vertex sparsifier $G'_{[r_p, s_p]}$ corresponding to a node in our decomposition tree is planar.*

Proof. First, consider a sequence of $H_{[r_1, s_1]}, H_{[r_2, s_2]}, \dots, H_{[r_p, s_p]}$ corresponding to a path in the decomposition tree, where $H_{[r_1, s_1]}$ is a child of the root⁶, and $H_{[r_i, s_i]}$ is a parent of $H_{[r_{i+1}, s_{i+1}]}$. Observe that $\cup_{1 \leq i \leq p} H_{[r_i, s_i]} = G_{[r_p, s_p]}$ which is planar.

From Algorithm 10, we unfold the recursion and obtain that

$$G'_{[r_p, s_p]} = \text{VERTEXSPARSIFY}(\text{VERTEXSPARSIFY}(\dots) \cup H_{[r_{p-1}, s_{p-1}]} \cup H_{[r_p, s_p]}).$$

Note that we omit the second parameter of VERTEXSPARSIFY only for readability. We assume by induction $G'_{[r_{p-1}, s_{p-1}]} = \text{VERTEXSPARSIFY}(\text{VERTEXSPARSIFY}(\dots) \cup H_{[r_{p-1}, s_{p-1}]})$ is planar. We will prove that $G'_{[r_p, s_p]}$ is planar. To this end, observe that $G'_{[r_{p-1}, s_{p-1}]}$ is a minor of $\cup_{1 \leq i \leq p-1} H_{[r_i, s_i]}$. Next, we need the following observation.

Claim 4.17. *Let G_1 be a minor of G_2 . Let (u, v) be an edge such that $u, v \in V(G_1) \cap V(G_2)$, i.e., the endpoints are nodes of both G_1 and G_2 . Then, $G_1 \cup \{(u, v)\}$ is a minor of $G_2 \cup \{(u, v)\}$. In particular, if $G_2 \cup \{(u, v)\}$ is planar, then so is $G_1 \cup \{(u, v)\}$.*

We apply Claim 4.17 where $G_2 = \cup_{1 \leq i \leq p-1} H_{[r_i, s_i]}$ and $G_1 = G'_{[r_{p-1}, s_{p-1}]}$. As the endpoints of $H_{[r_i, s_i]}$ are in both G_1 and G_2 by construction and $G_2 \cup H_{[r_p, s_p]} = \cup_{1 \leq i \leq p} H_{[r_i, s_i]}$ is planar, then $G_1 \cup H_{[r_p, s_p]}$ is planar. Finally, $G'_{[r_p, s_p]} = \text{VERTEXSPARSIFY}(G_1 \cup H_{[r_p, s_p]})$ is a minor of $G_1 \cup H_{[r_p, s_p]}$, so $G'_{[r_p, s_p]}$ is planar. \square

Now, we prove Theorem 4.15.

Proof of Theorem 4.15. We first prove the case when $s = k \cdot \text{poly}(1/\epsilon, \log n)$. We again prove the theorem when ϵ' is a constant for simplicity. Set $\epsilon = \epsilon'/10 \log n$. We build the corresponding decomposition tree with $\log n$ levels. The quality of the sparsifier at any level is at most $(1 + \epsilon)^{\log n} = (1 + \epsilon'/10 \log n)^{\log n} \leq (1 + \epsilon')$. The total running time will be $\tilde{O}(m^{1 + \frac{1}{\log n + 1}}) = \tilde{O}(n)$ using the same argument as in Lemma ???. That is the update time on average is $O(\text{poly}(\log n))$.

For the case when $s = k^{2/(1+3\gamma)} \cdot \text{poly}(1/\epsilon, \log n)$, the proof is the same except the parameters need to be carefully chosen. Set $\epsilon = \epsilon'\gamma/2$. We choose $\beta_0 = m = O(n)$, $\beta_1 = n^{(1+\gamma)/2}$, and $\beta_{i+1} = n^{(1+\gamma-2\gamma i)/2}$ for $i \geq 0$. We get that there will be at most $1/\gamma$ levels in the decomposition tree and thus the quality at each level is at most

$$(1 + \epsilon)^{1/\gamma} \leq e^{\epsilon/\gamma} = e^{\epsilon'/2} \leq (1 + \epsilon')$$

because $(1 + x) \leq e^x$ for any x and $e^{x/2} \leq (1 + x)$ for $0 \leq x \leq 1$.

For each i , the total time to build the sparsifiers in level $i+1$ by running the algorithm sparsifier at level i is $n/\beta_{i+1} \cdot \tilde{O}(\beta_i^{2/(1+3\gamma)})$. This is because there are n/β_{i+1} many sparsifiers, and the algorithm is applied on a graph of size $\tilde{O}(\beta_i^{2/(1+3\gamma)})$. By direct calculation we have that

$$n/\beta_{i+1} \cdot \beta_i^{2/(1+3\gamma)} = n^{1-(1+\gamma-2i\gamma)/2 + \frac{(1+\gamma-2(i-1)\gamma)}{(1+3\gamma)}} \leq n^{1.5-\gamma/2}.$$

⁶Note that the graph $H_{[r, s]}$ is not defined at the root.

To see this, note that $2/(1 + 3\gamma) \geq 1$ and consider the following chain of inequalities:

$$\begin{aligned}
& \frac{(1 + \gamma - 2(i - 1)\gamma)}{(1 + 3\gamma)} - (1 + \gamma - 2i\gamma)/2 \\
& \leq \frac{1 + \gamma}{1 + 3\gamma} - (i - 1)\gamma - \frac{1 + \gamma}{2} + i\gamma \\
& \leq (1 - \gamma) + \gamma - 1/2 - \gamma/2 \\
& = 1/2 - \gamma/2.
\end{aligned}$$

It follows that the total time over all levels is $\frac{1}{\gamma} \cdot O(n^{1.5-\gamma/2})$, which in turn implies an average update time of $O(n^{0.5-\gamma/2})$. This completes the proof. \square

5 Fully-Dynamic Algorithms via Fully-Dynamic Vertex Sparsifier

In this section, we present a meta data-structure for dynamically maintain \mathcal{P} , some properties of the graph G . Like before, we also utilize the idea of α -vertex sparsifier. Unlike the incremental scheme, we introduce another parameter s specifying the edge-sparsity of the vertex sparsifier.

Let $G = (V, E)$ be a graph. An (α, s) -vertex sparsifier of G for \mathcal{P} w.r.t. the terminal set T is a graph H that is an α -vertex sparsifier of G for \mathcal{P} w.r.t. T with number of edges, $|E(H)|$, bounded by $O(|T|s)$.

Definition 5.1 (Fully-Dynamic Vertex Sparsifier). *Let $G = (V, E)$ be a graph, $T \subseteq V$ be the set of terminals, and α be a non-negative parameter. A data-structure \mathcal{D} is an (α, s) -Fully-Dynamic Vertex Sparsifier (abbr. (α, s) -DVS) of G if \mathcal{D} explicitly maintains an (α, s) -vertex sparsifier H_T of G with respect to T and supports following operations:*

1. *PREPROCESS(G, T): preprocess G in time t_p*
2. *ADDTERMINAL(u): let T' be $T \cup \{u\}$ and update H_T to $H_{T'}$ in time t_a such that i) $H_{T'}$ is an (α, s) -vertex sparsifier of G with respect to T' , and ii) the recourse, number of edge changes from H_T to $H_{T'}$, is at most r_a . This operation should return the set of edges inserted and deleted from H_T .*
3. *DELETE(e): delete e from G while maintaining H_T being an (α, s) -vertex sparsifier of G with respect to T in t_d time and r_d recourse in H_T .*

Also, such H_T have size $O(|T|s)$.

Given a (α, s) -DVS \mathcal{D} of G , we can support edge insertion by first adding 2 endpoints to the terminal set and then add the edge directly to the vertex sparsifier H_T . The correctness comes from the decomposability. To specify the cost, the following corollary is presented to formalized the approach.

Lemma 5.2. *Let $G = (V, E)$ be a graph and \mathcal{D} be an (α, s) -DVS of G . Let t_a, r_a be the time and recourse for \mathcal{D} to handle ADDTERMINAL operation. \mathcal{D} also maintains an (α, s) -vertex sparsifier H_T of G with respect to T subject to the following operation:*

- *INSERT(e): insert e to G while maintaining H_T being an (α, s) -vertex sparsifier of G with respect to T in $O(t_a)$ time and $O(r_a)$ recourse in H_T .*

Proof. The proof works similar to the one of Lemma 2.2. \square

To design a fully-dynamic data structure that support update and queries in sub-linear time, we will focus on building a fully-dynamic vertex sparsifier whose update time and recourse are sub-linear in n . This requirement is made precise in the following definition.

Definition 5.3 (Efficient DVS). Let $G = (V, E)$ be a graph, α be a non-negative parameter, and $s(n)$, $f(n, k)$, $g(n, k)$, $r(n, k) \geq 1$ be functions where n and k correspond to the maximum size of the vertex set and the terminal set respectively. We say that \mathcal{D} is an $(\alpha, s(k), f(n, k), g(n, k), r(n, k))$ -efficient DVS of G if \mathcal{D} is an $(\alpha, s(k))$ -DVS of G with Preprocessing time $t_p = O(m \cdot f(n, k))$, AddTerminal time $t_a = O(g(n, k))$, and recourse $r_a = O(r(n, k))$, and Delete time $t_d = O(g(n, k))$, and recourse $r_d = O(r(n, k))$.

Next we show how to use an efficient fully-dynamic vertex sparsifier to design an (approximate) fully-dynamic algorithms for problems with certain properties while achieving fast amortized update and query time.

Theorem 5.4. Let $G = (V, E)$ be a graph, and for any $u, v \in V$, let $\mathcal{P}(u, v, G)$ be a solution to a minimization problem between u and v in G . Let $s(k)$, $f(n, k)$, $g(n, k)$, $r(n, k)$, $h(n) \geq 1$ be functions with $s(k) \leq r(n, k)$, $\alpha \geq 1$ be the approximation factor, $\ell \geq 1$ be the depth of the data structure, and let $\mu_0, \mu_1, \dots, \mu_\ell$ with $\mu_0 = m$ be parameters associated with the running time. Assume the following properties are satisfied

1. G admits an $(\alpha, s(k), f(n, k), g(n, k), r(n, k))$ -efficient DVS
2. The property $\mathcal{P}(u, v, G)$ can be computed in $O(mh(n))$ time in a static graph with m edges and n vertices.

Then there is a (approximate) fully-dynamic algorithm that maintains for every pair of nodes u and v , an estimate $\delta(u, v)$, such that

$$\mathcal{P}(u, v, G) \leq \delta(u, v) \leq \alpha^\ell \cdot \mathcal{P}(u, v, G), \quad (6)$$

with amortized update time of

$$T_u = O\left(\sum_{i=1}^{\ell} c^{i-1} \prod_{j=1}^{i-1} r(\mu_j, \mu_{j-1}) \left(\frac{\mu_{i-1} s(\mu_{i-1}) f(\mu_{i-1}, \mu_i)}{\mu_i} + g(\mu_{i-1}, \mu_i)\right)\right),$$

and amortized query time of

$$T_q = O(\ell T_u + \mu_\ell s(\mu_\ell) h(\mu_\ell)),$$

where $c < 3$ is a universal constant.

We prove the theorem in the rest of the section.

Data Structure. Given some integer parameter $\ell \geq 1$, and parameters $m = \mu_0 \geq \dots, \mu_\ell$. Our data structure maintains

1. a hierarchy of graphs $\{G_i\}_{0 \leq i \leq \ell}$,
2. a hierarchy of terminal sets $\{T_i\}_{1 \leq i \leq \ell}$, each associated with the parameters $\{\mu_i\}_{1 \leq i \leq \ell}$, and
3. a hierarchy of $(\alpha, s(k), f(n, k), g(n, k), r(n, k))$ -efficient DVSs $\{\mathcal{D}_i\}_{1 \leq i \leq \ell}$, each associated with a graph G_{i-1} and the terminal set T_i .

The data structure is initialized recursively. First, initialize $T_i \leftarrow \phi$ for $1 \leq i \leq \ell$ and $G_0 \leftarrow G$. For $1 \leq i \leq \ell$, construct an $(\alpha, s(n), f(n, k), g(n, k), r(n, k))$ -efficient DVS \mathcal{D}_i of graph G_{i-1} w.r.t. the terminal set T_i and set G_i be the sparsifier maintained by \mathcal{D}_i . Hence, G_i is an α -vertex sparsifier of G_{i-1} w.r.t. T_i . By transitivity, we know G_ℓ is an α^ℓ -vertex sparsifier of G_0 , which is the input graph G .

When answering a query, we add both s and t to the terminal sets of every \mathcal{D}_i . Then compute $\mathcal{P}(s, t, G_\ell)$ and output the value as an estimate of $\mathcal{P}(s, t, G)$.

When dealing with a edge update in G , we first add both endpoints to the terminal set of \mathcal{D}_1 . Then we update the edge in G_1 , the vertex sparsifier maintained by \mathcal{D}_1 . Edge changes propagate down the hierarchy. To bound the size of each G_i , $1 \leq i \leq \ell$, rebuild \mathcal{D}_i for every $2\mu_i$ updates in G_{i-1} w.r.t. the most recently added μ_i terminal vertices.

The rebuild scheme is easier than the incremental case. When rebuilding \mathcal{D}_i , we treat it as a series of edge updates in G_i . Since \mathcal{D}_i 's are DVS, they can handle both edge insertions and deletion.

Running Time. We first study the update time of our data structure.

\mathcal{D}_i maintains the graph G_{i-1} , which has at most μ_{i-1} vertices, and it has at most μ_i terminal vertices due to rebuild. Hence \mathcal{D}_i spends $O(g(\mu_{i-1}, \mu_i))$ -time per operation of `ADDTERMINAL` or edge updates.

Since rebuild is incurred every $2\mu_i$ operations for the data structure \mathcal{D}_i , we can charge the rebuild cost among μ_i operations. Note that \mathcal{D}_i is an $(\alpha, s(n))$ -DVS of G_{i-1} , which is a graph with $O(\mu_{i-1})$ vertices and $O(\mu_{i-1}s(\mu_{i-1}))$ edges. By amortizing the rebuild cost, we know the time \mathcal{D}_i spent on either `ADDTERMINAL` or edge updates is:

$$O\left(\frac{\mu_{i-1}s(\mu_{i-1})f(\mu_{i-1}, \mu_i)}{\mu_i} + g(\mu_{i-1}, \mu_i)\right).$$

Since an update in \mathcal{D}_i creates $O(r(\mu_{i-1}, \mu_i))$ updates in G_i , which is handled by the data structure in the next level, \mathcal{D}_{i+1} , we have to incorporate such quantity in to the analysis. Also, we have to take the recourse from rebuild into account. Every rebuild creates $O(|E(G_i)|) = O(\mu_i s(\mu_i))$ updates to G_i . By amortizing it over μ_i updates in \mathcal{D}_i , each update in G_i has recourse $O(s(\mu_i) + r(\mu_{i-1}, \mu_i)) = O(r(\mu_{i-1}, \mu_i))$ since $s(n) \leq r(n, k)$. Thus one update in \mathcal{D}_i creates $\leq cr(\mu_{i-1}, \mu_i)$ updates in G_i for c being some universal positive constant. We can now analyze the amount of updates handled by \mathcal{D}_i when 1 edge update happens in G . By simple induction, we know there will be $\leq \prod_{j=1}^{i-1}(cr(\mu_j, \mu_{j+1}))$ updates in G_{i-1} . Thus for the data structure in i -th level, \mathcal{D}_i , there will be $\leq c^{i-1} \prod_{j=1}^{i-1} r(\mu_j, \mu_{j+1})$ updates. Combining these 2 quantities, we can bound the amortized update time of our data structure:

$$T_u = O\left(\sum_{i=1}^{\ell} c^{i-1} \prod_{j=1}^{i-1} r(\mu_j, \mu_{j+1}) \left(\frac{\mu_{i-1}s(\mu_{i-1})f(\mu_{i-1}, \mu_i)}{\mu_i} + g(\mu_{i-1}, \mu_i)\right)\right).$$

We next study the query time of our data-structure. When answering a query, we add s and t to each layer of the terminal set. When adding terminals to each layer of data structure \mathcal{D}_i , edge changes also propagate to lower levels. As for the analysis of update time, we have to take the recourse into account. The time can be bounded by $O(\ell T_u)$ where T_u is the update time of our data structure.

Then we compute $\mathcal{P}(s, t, G_\ell)$ in G_ℓ , which has $\mu_\ell s(\mu_\ell)$ edges as guaranteed by the definition of DVS. Since we have an $O(mh(n))$ algorithm for computing $\mathcal{P}(s, t, G)$ in an m -edge n -vertex graph G , $\mathcal{P}(s, t, G_\ell)$ can be computed in $\mu_\ell s(\mu_\ell)h(\mu_\ell)$ time. Combining these 2 bounds, we can bound the query time by:

$$T_q = O(\ell T_u + \mu_\ell s(\mu_\ell)h(\mu_\ell)).$$

Lemma 5.5. *Let $\{\mu_i\}_{0 \leq i \leq \ell}$ be a family of parameters with $\mu_0 = m$. Suppose $s(n) = n^{o(1)}$, $f(n, k) = O((n/k)^d)$, both $g(n, k)$ and $r(n, k)$ are of order $O((n/k)^e)$ for some positive constants $d + 1 \leq e$ and $h(n) = n^{o(1)}$. Let t be any positive constant, if we set*

$$\mu_i = m^{1-i/(\ell+t)}, \quad \text{where } 1 \leq i \leq \ell,$$

then the update time is

$$T_u = O\left(\sum_{i=1}^{\ell} c^{i-1} \prod_{j=1}^{i-1} r(\mu_j, \mu_{j+1}) \left(\frac{\mu_{i-1}s(\mu_{i-1})f(\mu_{i-1}, \mu_i)}{\mu_i} + g(\mu_{i-1}, \mu_i)\right)\right) = O\left(\ell c^\ell m^{e\ell/(\ell+t)} m^{o(1)/(\ell+t)}\right), \quad (7)$$

and the query time is

$$T_q = O(\ell T_u + \mu_\ell s(\mu_\ell)h(\mu_\ell)) = O\left(\max\{\ell^2 c^\ell m^{e\ell/(\ell+t)} m^{o(1)/(\ell+t)}, m^{(t+o(1))/(\ell+t)}\}\right). \quad (8)$$

Proof. Plug in the choice of μ_i , we have

$$\frac{\mu_{i-1}^s(\mu_{i-1})f(\mu_{i-1}, \mu_i)}{\mu_i} + g(\mu_{i-1}, \mu_i) = m^{(1+o(1)+d)/(\ell+t)} + m^{e/(\ell+t)} = O(m^{(o(1)+e)/(\ell+t)}).$$

Also, since $i \leq \ell$,

$$\begin{aligned} c^{i-1} \prod_{j=1}^{i-1} r(\mu_j, \mu_{j+1}) &\leq c^{\ell-1} \prod_{j=1}^{\ell-1} r(\mu_j, \mu_{j+1}) \\ &= O(c^{\ell-1} \prod_{j=1}^{\ell-1} m^{e/(\ell+t)}) \\ &= O(c^{\ell-1} m^{e(\ell-1)/(\ell+t)}) \end{aligned}$$

Combining these 2 inequalities, we can bound the update time by

$$\begin{aligned} T_u &= O\left(\sum_{i=1}^{\ell} c^{i-1} \prod_{j=1}^{i-1} r(\mu_j, \mu_{j+1}) \left(\frac{\mu_{i-1}^s(\mu_{i-1})f(\mu_{i-1}, \mu_i)}{\mu_i} + g(\mu_{i-1}, \mu_i)\right)\right) \\ &= O\left(\sum_{i=1}^{\ell} c^{\ell-1} m^{e(\ell-1)/(\ell+t)} m^{(o(1)+e)/(\ell+t)}\right) \\ &= O\left(\ell c^{\ell} m^{e\ell/(\ell+t)} m^{o(1)/(\ell+t)}\right) \end{aligned}$$

The bound for query time is straightforward from the definition of μ_ℓ . □

Corollary 5.6. *To asymptotically minimize the query time given in Lemma 5.5, we set*

$$t \leftarrow e\ell, \ell \leftarrow O(1).$$

We have update time

$$T_u = O\left(m^{(e+o(1))/(e+1)}\right) \tag{9}$$

and query time

$$T_q = O\left(m^{(e+o(1))/(e+1)}\right). \tag{10}$$

Corollary 5.7. *Under the same setting as Lemma 5.5 except $d + 1 > e$, we set*

$$t \leftarrow e\ell, \ell \leftarrow \sqrt{\log m}.$$

We can asymptotically minimize update time

$$T_u = O\left(m^{e/(e+1)+o(1)} \sqrt{\log m}\right) \tag{11}$$

and query time

$$T_q = O\left(m^{e/(e+1)+o(1)} \log m\right). \tag{12}$$

Proof. Under this setting, the only difference in the analysis of Lemma 5.5 is

$$\frac{\mu_{i-1}S(\mu_{i-1})f(\mu_{i-1}, \mu_i)}{\mu_i} + g(\mu_{i-1}, \mu_i) = m^{(1+o(1)+d)/(\ell+t)} + m^{e/(\ell+t)} = O(m^{(1+o(1)+d)/(\ell+t)}) = O(m^{o(1)}).$$

Thus,

$$\begin{aligned} T_u &= O\left(\sum_{i=1}^{\ell} c^{i-1} \prod_{j=1}^{i-1} r(\mu_j, \mu_{j+1}) \left(\frac{\mu_{i-1}S(\mu_{i-1})f(\mu_{i-1}, \mu_i)}{\mu_i} + g(\mu_{i-1}, \mu_i)\right)\right) \\ &= O\left(\sum_{i=1}^{\ell} c^{\ell-1} m^{e(\ell-1)/(\ell+t)} m^{o(1)}\right) \\ &= O\left(\ell c^{\ell} m^{e(\ell-1)/(\ell+e\ell)} m^{o(1)}\right) \\ &= O\left(\ell c^{\ell} m^{e/(e+1)} m^{o(1)}\right) \\ &= O\left(\sqrt{\log m} \cdot m^{e/(e+1)} m^{o(1)}\right) \end{aligned}$$

□

6 Fully-Dynamic All Pair Max-Flow/Min-Cut

In this section, we incorporate *Local Sparsifier* with tools developed previously to give a data structure for min cut query in a dynamic graph. The main theorem we prove is as follows.

Theorem 6.1. *Given a graph $G = (V, E, c)$ with weight ratio $U = \text{poly}(n)$. There is a dynamic data structure maintaining G subject to the following operations:*

1. *INSERT(u, v, c): Insert the edge (u, v) to G in amortized $O(m^{2/3} \log^7 n)$ -time.*
2. *DELETE(e): Delete the edge e from G in amortized $O(m^{2/3} \log^7 n)$ -time.*
3. *MINCUT(s, t): Output a $\tilde{O}(\log n)$ -approximation to the min- st -cut value of G in $\tilde{O}(m^{2/3})$ -time w.h.p.. The cut set S can be obtained on demand with linear overhead in $|S|$.*

Using the so-called *dynamic sparsifier*, we can speed-up Theorem 6.1 by a factor of $O(m/n)$. Which is significant for G being a dense graph originally.

Corollary 6.2. *Given a graph $G = (V, E, c)$ with weight ratio $U = \text{poly}(n)$. There is a dynamic data structure maintaining G subject to the following operations:*

1. *INSERT(u, v, c): Insert the edge (u, v) to G in amortized $O(n^{2/3} \log^{41/3} n)$ -time.*
2. *DELETE(e): Delete the edge e from G in amortized $O(n^{2/3} \log^{41/3} n)$ -time.*
3. *MINCUT(s, t): Output a $\tilde{O}(\log n)$ -approximation to the min- st -cut value of G in $\tilde{O}(n^{2/3})$ -time w.h.p.. The cut set S can be obtained on demand with linear overhead in $|S|$.*

6.1 Cut, Flow and L_∞ -Embeddability

In this section, we present concepts critical in building local sparsifier preserving cut/flow value.

Definition 6.3. Given a graph $G = (V, E, c)$, a cut C is any proper subset of V , i.e., $\emptyset \neq C \subset V$. Define $E(C) := E(C, V \setminus C)$ and $c(C) := \sum_{e \in E(C)} c(e)$.

Definition 6.4. [Mad10] Given a graph $G = (V, E, c)$, $\mathbf{f} = (f^1, \dots, f^k) \in \mathbb{R}^{E \times k}$ is multicommodity flow if f_i is a $s_i t_i$ -flow. Define $|\mathbf{f}(e)| := \sum_{i=1}^k |f^i(e)|$ as the total flow crossing the edge $e \in E$. A multicommodity flow \mathbf{f} is feasible if for every edge e , we have $|\mathbf{f}(e)| \leq c(e)$. Single-commodity flow \mathbf{f} is a multicommodity flow with $k = 1$.

We use the notion graph embedding for describing relation between flows in 2 graphs.

Definition 6.5. [Mad10] Given 2 graphs with same vertex set $G = (V, E, c), H = (V, E_H, c_H)$. For every edge $e = uv \in E$, f^e is a flow that routes $c(e)$ amount of flow from u to v in H . Then $\mathbf{f} := (f^e \mid e \in E) \in \mathbb{R}^{E_H \times E}$, the collection of f^e for every edge $e \in E$, is an embedding of G into H .

Use the notion of embedding, we can define embeddability between graphs over same vertex set.

Definition 6.6. [Mad10] Given $t \geq 1$, and 2 graphs with same vertex set $G = (V, E, c), H = (V, E_H, c_H)$. We say G is t -embeddable into H , denoted by $G \leq_t H$ if there is an embedding \mathbf{f} of G into H such that for every $e_H \in E_H$, $|\mathbf{f}(e_H)| \leq t \cdot c_H(e_H)$. For $t = 1$, we ignore the subscript and say G is embeddable into H and $G \leq H$.

Here we also define the notion of cut approximation between graphs.

Definition 6.7. Given a graph $G = (V, E, c)$ and $\epsilon \in (0, 1)$, we say a graph $H = (V, E_H \subseteq E, c_H)$ is a $(1 + \epsilon)$ -cut-sparsifier of G if $S \subseteq V$,

$$(1 - \epsilon)c(S) \leq c_H(S) \leq (1 + \epsilon)c(S).$$

Denoted by $H \sim_\epsilon G$.

Here we present some structural results about the value of max flow/min cut between mutually embeddable graphs.

Lemma 6.8. Given $t \geq 1$, and 2 graphs $G = (V, E, c), H = (V, E_H, c_H)$ on the same vertex set such that $G \leq_t H$. For any cut S , we have $c(S) \leq t \cdot c_H(S)$.

Proof. By the property of flow, $|\sum_{d \in E_H(S)} f^e(d)| = c(e)$ if $e \in E(S)$ and 0 otherwise. Therefore,

$$\begin{aligned} c(S) &= \sum_{e \in E(S)} c(e) = \sum_{e \in E} \left| \sum_{d \in E_H(S)} f^e(d) \right| \\ &\leq \sum_{e \in E} \sum_{d \in E_H(S)} |f^e(d)| \\ &= \sum_{d \in E_H(S)} \sum_{e \in E} |f^e(d)| \\ &= \sum_{d \in E_H(S)} |f(d)| \\ &\leq \sum_{d \in E_H(S)} t \cdot c_H(d) = t \cdot c_H(S). \end{aligned}$$

□

Corollary 6.9. Given $t \geq 1$, and 2 graphs $G = (V, E, c)$, $H = (V, E_H, c_H)$ on the same vertex set such that $G \leq H$ and $H \leq_t G$. For any cut $C \subseteq V$, we have

- $c(C) \leq c_H(C)$ and
- $c_H(C) \leq t \cdot c(C)$

Proof. It directly comes from Lemma 6.8. □

6.2 J -trees as Vertex Sparsifier

Here we introduce the notion of j -tree [Mad10]. Intuitively, j -tree is obtained from the original graph by contracting vertices. The resulting graph has at most j vertices but preserves the value of cuts between them. We deploy such a routine in reducing the number of vertices in the original graph. This helps the design of the data structure we need. Since we can answer queries by computing min cut values in a smaller graph.

Definition 6.10. [Mad10] Given $j \leq 1$, $H = (V_H, E_H, c_H)$ is a j -tree if it is a connected graph being a union of a core $C(H)$ which is a subgraph of H induced by some vertex set $C \subseteq V_H$ with $|C| \leq j$; and of an envelope $F(H)$ which is a forest on H with each component having exactly one vertex in the core $C(H)$. For any core vertex $u \in C$, define $F(u)$ to be the vertex set of the component containing u in the envelope. Also for $S \subseteq C$, $F(S)$ is the union of $F(u)$, $\forall u \in S$.

We are interested in such j -tree structure because of the following lemma. It suggests that we can approximate max flow/min cut within several simpler graphs and introduce a sampling scheme that can reduce the number of them to consider.

But first, we are going to define the notion of the congestion ρ -decomposition of a graph G .

Definition 6.11. [Mad10] A family of graphs $G_1 \dots G_k$ is a (k, ρ, j) -decomposition of a graph $G = (V, E, c)$ if

1. Each G_i is a j -tree, and there are at most k of them.
2. $G \leq G_i, \forall i$.
3. $\sum G_i \leq_{k \cdot \rho} G$.

Using this definition, we are able to state more clearer of the benefit of using j -trees.

Lemma 6.12. [Mad10] Given any graph $G = (V, E, c)$ with weight ratio U and $k \geq 1$, we can find in $O(km \log^4 n)$ -time a

$$\left(k, \rho := O(\log n \cdot \log \log n \cdot (\log \log \log n)^3), O\left(\log^2 n \frac{m \log U}{k}\right) \right)$$

-decomposition of $G, G_1 \dots G_k$. The weight ratio of each G_i is $O(mU)$. Moreover, if we sample G_i with probability $1/k$, for any fixed cut S , the size of this cut in G_i is at most 2ρ times the size of the cut in G with probability at least $\frac{1}{2}$.

To speed-up, we can not afford to maintain this many j -trees. The following lemma shows that we can maintain only $O(\log n)$ of them while preserving the approximation quality.

Lemma 6.13. Given any graph $G = (V, E, c)$ with weight ratio U and $k = \Omega(\log n)$ and a (k, ρ, j) -decomposition of G, G_1, \dots, G_k . By sampling $O(\log n)$ graphs from G_1, \dots, G_k , every min st cut is preserved up to a 2ρ -factor with high probability.

Proof. Let $C(s, t)$ be some minimum st -cut in G . Let $\mathcal{C} = \{C(s, t) \mid s, t \in \binom{V}{2}\}$. Clearly $|\mathcal{C}| \leq n^2$. By lemma 6.12 we know $c(C(s, t)) \leq c_{G_i}(C(s, t))$ and with probability at least 0.5, $c_{G_i}(C(s, t)) \leq 2\rho c(C(s, t))$ by sampling G_i uniformly. By sampling $t = d \log n$ of them, say G_1, \dots, G_t , we have with probability at least $1 - n^{-d}$ that:

$$\min_{i \in [t]} \{c_{G_i}(C(s, t))\} \leq 2\rho c(C(s, t)).$$

Taking the union bound over all cuts in \mathcal{C} which has at most n^2 of them, we have with probability $1 - n^{-d+2}$ that

$$\forall s, t \in \binom{V}{2}, c(C(s, t)) \leq \min_{i \in [t]} \{c_{G_i}(C(s, t))\} \leq 2\rho c(C(s, t)).$$

□

Lemma 6.13 says that optimal st -cut can be preserved using $O(\log n)$ -many j -trees up to a $O(\rho)$ -factor with high probability. In our usage, we compute cuts only in the core instead of the entire j -tree. Cuts in the core are then projected back to the j -tree. To justify the correctness in terms of minimum st -cut, we define the concept of *core cut*.

Definition 6.14. Let $j \geq 1$ and $H = (V, E_H, c_H)$ be some j -tree. Let $C_H = (C \subseteq V, E_C, c_C)$ be the core of H . Given any cut $S \subseteq C$ of C_H , the core cut of S with respect to C in H is

$$\Pi(S) := \bigcup_{u \in S} F(u).$$

That is, extend the cut S by including trees in the envelope rooted at vertex in S .

Now we present a lemma that justify computing min cuts in the core.

Lemma 6.15. Let $j \geq 1$ and $H = (V, E_H, c_H)$ be some j -tree. Let $C_H = (C \subseteq V, E_C, c_C)$ be the core of H . For any cut $S \subseteq V$ of H , we have

$$c_H(\Pi(S \cap C)) \leq c_H(S).$$

That is, to find minimum cut separating core vertices in H , it suffice to check only cuts in the core and then construct the core cut.

Proof. Let $E_F := E_H \setminus E_C$ be the edge set of the envelope. Note that

1. $E(S) \cap E_C = E(\Pi(S \cap C)) \cap E_C$. Crossing edges in the core are identical for both cuts.
2. $E(\Pi(S \cap C)) \cap E_F = \phi$. There is no crossing edges in the envelope for cut $\Pi(S \cap C)$.

Conclusively, we have

$$\begin{aligned} c_H(\Pi(S \cap C)) &= c_H(E(\Pi(S \cap C)) \cap E_C) + c_H(E(\Pi(S \cap C)) \cap E_F) \\ &= c_H(E(S) \cap E_C) + 0 \\ &\leq c_H(E(S) \cap E_C) + c_H(E(S) \cap E_F) = c_H(S). \end{aligned}$$

□

Then to make this dynamic, we just need to support the ‘add terminal’ operation as defined in the local sparsifiers paper [GHS18], and in Chapter 6 of [Gor19].

6.3 Dynamic Cut Sparsifier

Lemma 6.16. [ADK⁺16] *Given a graph $G = (V, E, c)$ with weight ratio U . There is an $(1 + \epsilon)$ -cut sparsifier H of G w.h.p., Such H supports the following operations:*

- *Insert(u, v, c): Insert the edge (u, v) to G in amortized $O(\log^5 n \epsilon^{-2} \log U)$ -time.*
- *Delete(e): Delete the edge e from G in amortized $O(\log^5 n \epsilon^{-2} \log U)$ -time.*

Such H is a subgraph of G with different weight and the weight ratio of H is $O(nU)$. Moreover, we maintain a partition of H into $k = O(\log^3 n \epsilon^{-2} \log U)$ disjoint forests T_1, \dots, T_k with each vertex keeps the set of its neighbors u in each forest T_i . After each edge insertion/deletion in G , at most 1 edge change occurs in each forest T_i .

By first deploying a dynamic cut sparsifier from Lemma 6.16, we can speed-up Theorem 6.1 by ripping of the dependency on m .

Proof of Corollary 6.2. First apply Lemma 6.16 to acquire a 2-approximation dynamic cut sparsifier H of G . Such H has $O(n \log^4 n)$ edges. Then we incur Theorem 6.1 to maintain the sparsifier H . For every edge update in G , by Lemma 6.16, it becomes $O(\log^3 n \log U) = O(\log^4 n)$ edge changes in H . Therefore, one single edge update can be handled in amortized time

$$O\left(\log^4 n \cdot (n \log^4 n)^{2/3} \log^7 n\right) = O\left(n^{2/3} \log^{41/3} n\right).$$

For every $\text{MinCut}(s, t)$ query, we incur the same query on the sparsifier H , which can be computed in $\tilde{O}((n \log^4 n)^{2/3}) = \tilde{O}(n^{2/3})$ -time. Since such H preserves cut/flow value up to a factor of 2, so the result is still within $\tilde{O}(\log n)$ approximation with high probability. \square

6.4 An $\tilde{O}(m)$ -time 2-Approximation Max Flow/Min Cut Solver

Here we present a near-linear time max flow algorithm from [Pen16].

Lemma 6.17. ([Pen16], rephrased) *Given a graph $G = (V, E, c)$ with weight ratio $U = \text{poly}(n)$ and source/sink pairs and t . We can compute a 2-approximation for value of the minimum cut between s and t in $O(m \log^{32} n \log^2 \log n) = \tilde{O}(m)$ -time. The cut set S can be obtained on demand with linear overhead in $|S|$.*

6.5 The main theorem

In this section, we give details in building a dynamic data structure for approximately computing minimum st -cuts in a dynamic graph. The high-level idea is to build $(k, \rho, j := O(m^{2/3}))$ -decomposition of the original graph and dynamically maintain them. By lemma 6.13, we maintain only $O(\log n)$ of these j -trees instead of k . For every st -cut query, we ran the algorithm from lemma 6.17 [Pen16] on these $O(\log n)$ core graphs.

To dynamically maintain these core graphs, dynamic cut sparsifiers from lemma 6.16 are used. In addition to that, we also present a data structure for maintaining j -tree under the operation of adding a vertex to the core.

To prove the theorem, we need a dynamic data structure for maintaining j -trees. The tools are formalized as the following lemma.

Lemma 6.18. *Given $j \geq 1$ and a graph $G = (V, E, c)$ with weight ratio $U = \text{poly}(n)$ and a j -tree H of G such that $H \leq G \leq_\alpha H$. Let $n = |V|, m = |E|$. We can dynamically maintain a $O(j)$ -tree \tilde{H} such that $\tilde{H} \leq G \leq_{O(\alpha)} \tilde{H}$ under up to j of following operations:*

1. *INITIALIZE*(G): Build data structures for maintaining H in $O(\frac{mn}{j} \log n)$ -time.
2. *ADDTERMINAL*(u): Move vertex u to the core of H . Such operation can be done in amortized $O(\frac{mn}{j^2} \log^6 n)$ -time.
3. *INSERT*(u, v, c): Insert the edge (u, v) to G in amortized $O(\frac{mn}{j^2} \log^6 n)$ -time.
4. *DELETE*(e): Delete the edge e from G in amortized $O(\frac{mn}{j^2} \log^6 n)$ -time.

The total number of edge change in the core is $O(\frac{mn}{j})$. Hence the amortized number of edge changes per operation is $O(\frac{mn}{j^2})$. Also, $C(\tilde{H})$ (core of \tilde{H}) is sparse, i.e., it has $O(j \log^4 j)$ edges.

6.6 Tree-terminal path and edge moving

To prove Lemma 6.18, we have to open the black box of j -tree construction. It creates a graph by first select a proper spanning tree/forest and then route off-tree edges by tree paths and set of edges restricted on a small subset of vertices. To well-understand and formalize the construction, we introduce some notations about spanning forests and trees.

Definition 6.19. Given a forest F and a subset of vertices $C \subseteq V(F)$. Add at most $|C|$ vertices to C so that every pairwise lowest common ancestor is in C . Then iteratively remove vertices from $V(F) \setminus C$ of degree 1 until no such vertices remain. For each path with endpoints in C and no internal vertices in C , replace the whole path with a single edge. We define the resulting forest as Skeleton Tree of F with respect to C , denoted by $S(F, C)$.

Definition 6.20. Given a forest F , define $F[u, v]$ as the unique uv -path in F if they are connected. Given any edge $e = uv$, we use F_e to denote the path $F[u, v]$.

Definition 6.21. Given a forest T and a subset of vertices C , a subset of edges $F \subseteq T$ is a tree partition of T with respect to C if every component of $T \setminus F$ has exactly one vertex in C . For every vertex u in T , we define u 's representative with respect to a tree partition F and C as the only vertex of C in the component containing u in $T \setminus F$. Denoted as $T_{C,F}(u)$.

For any edge $e = uv \in T$, we define e 's tree-representative moving as

$$\text{Repr}_{T,C,F}(e = uv) := \begin{cases} e, & \text{for } e \in T \setminus F \\ T_{C,F}(u)T_{C,F}(v), & \text{for } e \in F \end{cases}.$$

Use this $\text{Repr}_{T,C,F}$, we define e 's tree-representative path as

$$Q_{T,C,F}(e = uv) := \begin{cases} e, & \text{for } e \in T \setminus F \\ T[u, T_{C,F}(u)] + \text{Repr}_{T,C,F}(e) + T[T_{C,F}(v), v], & \text{for } e \in F \end{cases}.$$

Here we introduce notations from [KPSW19], which defines the so-called tree-portal paths. Portals are terminals in our terminology.

Definition 6.22. Given a graph $G = (V, E)$, a spanning forest T and a subset of vertices C (terminals). For any 2 vertices $u, v \in V$, define $T_C(u, v)$ to be the vertex in C closest to u in $T[u, v]$. If no such vertex exists, $T_C(u, v) := \perp$.

For any edge $e = uv \in E \setminus T$, first, we can orient arbitrarily. Then e 's tree-terminal edge moving can be defined as

$$\text{Move}_{T,C}(e = uv) := \begin{cases} uv, & \text{for } T_C(u, v) = \perp \\ T_C(u, v)T_C(v, u), & \text{otherwise} \end{cases}.$$

Use this $\text{Move}_{T,C}$, we can define e 's tree-terminal path as

$$P_{T,C}(e = uv) := \begin{cases} T[u, v] + \text{Move}_{T,C}(e), & \text{for } T_C(u, v) = \perp \\ T[u, T_C(u, v)] + \text{Move}_{T,C}(e) + T[T_C(v, u), v], & \text{otherwise} \end{cases}.$$

6.7 Initializing a J -Tree

In this subsection, we review the static construction of a j -tree. It is summarized in Algorithm 13. For the dynamical purpose, we slightly modify the construction from [Mad10].

Briefly, the procedure first computes (1) T , some spanning tree of G , (2) C , an $O(j)$ -sized subset of vertices, and (3) F , a subset of edges in T . Then a $O(j)$ -tree, H , is constructed by moving endpoints into C for edges not in forest $T \setminus F$. The construction is summarized in Algorithm 12. Hence, it is easy to see that the core graph of H is $H[C]$, the subgraph induced by C . Such moving is defined using either $\text{Repr}_{T,C,F}(e)$ for F or $\text{Move}_{T,C}(e)$ otherwise.

Such edge moving corresponds to an embedding of G into H . Each edge of G is routed in H using either one tree path or 2 tree paths concatenated by an edge in the core.

T , C and F are computed by Algorithm 11. First, we try to embed G into some spanning tree T of G by routing each edge e of G using the unique tree path T_e . Heuristically, to minimize the congestion incurred in each tree edge, a low stretch spanning tree (LSST) [ABN08] is used as T . LSST guarantees low "total" congestion on tree edges.

But to ensure low congestion on "every edge", we remove tree edges with the highest congestion (relative to its capacity) and route impacted edges alternatively. The removed tree edges are collected as set F and endpoints of them are collected as set C . Ideally, we move every edge not in T using $\text{Move}_{T,C}(e)$. And for data structural purposes, we add $O(j)$ more vertices to C to make sure we route each edge using a short tree path, i.e., of size $O(n/j)$. As discussed in [Mad10], such edge moving does not guarantee a j -tree.

Identical to [Mad10], we add vertices in $S(T, C)$, skeleton tree of C , to C . And add $O(|C|)$ more edges to F so that F is a tree partition of T with respect to the new C .

The main difference from [Mad10] is that we add more terminal vertices (C) and route off-tree edges after we determine C . An argument from [GKK⁺18] shows that the more terminal we add, the better the congestion approximation.

6.8 Data Structure for dynamical maintenance

Here we present the data structure for maintaining a j -tree, i.e., proving Lemma 6.18.

6.8.1 Structural arguments for j -tree maintenance

To prove Lemma 6.18, one has to make sure adding terminals does not increase the congestion. The argument is formalized as the following lemma:

Lemma 6.23. *Given a graph $G = (V, E, c)$, a spanning forest T of G , a subset of vertices C and F , a tree partition of T with respect to C . For any vertex $u \in V \setminus C$, there is an edge $e_u \in T \setminus F$ such that $F + e_u$ is tree partition of T with respect to $C + u$ (every component of $T \setminus (F + e_u)$ has exactly one vertex in C).*

Furthermore, let $H := \text{ROUTE}(G, T, C, F)$. If $H \leq_\alpha G$, then the graph $\overline{H} := \text{ROUTE}(G, T, C + u, F + e_u) \leq_\alpha G$.

Algorithm 11: COMPUTETCF($G = (V, E, c), j, l : E \rightarrow \mathbb{R}_{\geq 0}$)

- 1 Compute a spanning tree T of G with average stretch $\tilde{O}(\log n)$ with respect to l in $\tilde{O}(m)$ -time by [ABN08].
 - 2 For each edge $e = uv \in E$, let f^e be the uv -flow in T that routes $c(e)$ units of flow along T_e . Let \mathbf{f} be the collection of f^e for every $e \in E$. \mathbf{f} is therefore an embedding of G into T . $|\mathbf{f}| \in \mathbb{R}^{E(T)}$, the vector of amount of flow crossing each edge of T , can be computed in $\tilde{O}(m)$ -time.
 - 3 For $e \in T$, define its relative loading, $rload(e) := |\mathbf{f}(e)|/c(e)$.
 - 4 Decompose $E(T)$ into $O(\log n)$ subsets $F_i, i \in \{1, \dots, \lceil \log \|f\|_\infty + 1 \rceil\}$, for $e \in F_i$ if $rload(e) \in (R/2^i, R/2^{i-1}]$ where $R = \max_{e \in T} rload(e)$.
 - 5 Let i_0 be the minimal index such that $|F_{i_0}| = \Omega(j/\log n)$. Define $F = \bigcup_{i=1}^{i_0} F_i$. Note that $|F| \leq j$ and contains edges with the largest relative load.
 - 6 Define C consists of terminal vertices and all endpoints of edges of F .
 - 7 Add $O(j)$ more vertices to C such that every path of length $O(n/j)$ on T contains at least one vertex in C .
 - 8 Add vertices appeared in $S(T, C)$ to C as well.
 - 9 For every adjacent $uv \in E(S(T, C))$, add the edge $e \in T[u, v]$ with largest $rload(e)$ to F . Note that endpoints of such an edge are not added to C .
 - 10 By the construction of F , we know every tree in $T \setminus F$ contains exactly one vertex in C .
 - 11 **return** (T, C, F)
-

Algorithm 12: ROUTE($G = (V, E, c), T, C, F$)

- 1 $E_{\text{tree}} := \{\text{Repr}_{T, C, F}(e) \mid e \in T\}$.
 - 2 $E_{\text{off-tree}} := \{\text{Move}_{T, C}(e) \mid e \notin T\}$.
 - 3 Define an embedding \mathbf{f} of G into H as follows.
 - 4 For every $e \in T$, f^e routes $c(e)$ units through path $Q_{T, C, F}(e)$.
 - 5 For every $e \notin T$, f^e routes $c(e)$ units through path $P_{T, C}(e)$.
 - 6 Define $c_H(e) := |\mathbf{f}(e)|$, the amount of flow crossing e in the embedding.
 - 7 $H := (V, E_H := E_{\text{tree}} \cup E_{\text{off-tree}}, c_H)$.
 - 8 Remove self-loops from H .
 - 9 **return** H .
// H is a $|C|$ -tree with core $C(H) = H[C]$, the subgraph induced by subset of vertices C .
// The embedding \mathbf{f} of G into H is referred as the canonical j -tree embedding.
-

Algorithm 13: JTREE($G = (V, E, c), j, l : E \rightarrow \mathbb{R}_{\geq 0}$)

- 1 $(T, C, F) := \text{COMPUTETCF}(G, j, l)$.
 - 2 $H := \text{ROUTE}(G, T, C, F)$.
 - 3 **return** H .
-

Proof. Let $x = T_{C,F}(u)$, the only vertex in C in u 's component in $T \setminus F$. Since edges in $T \setminus F$ appear in both G and H , let e_u be the edge with minimum $c_H(e)$ in $T[u, x]$.

Clearly, $F + e_u$ is tree partition of T with respect to $C + u$. Since we delete one edge in u 's component which is a tree, it is split into 2 components such that x and u are in different components.

Observation from [GKK⁺18] that adding more vertices into C does not increase congestion immediately gives us that, $\overline{H} := \text{ROUTE}(G, T, C + u, F + e_u) \leq_\alpha G$. \square

To maintain $O(j)$ -tree H under dynamic edge updates in G , we first add both endpoints of the updating edge to the terminal and then perform the edge update in $C(H)$, core of H , directly. One has to make sure such behavior does not increase the congestion when routing G in H . The following lemma gives such promise:

Lemma 6.24. *Given a graph $G = (V, E, c)$, a spanning forest T of G , a subset of vertices C and F , a tree partition of T with respect to C . Let $H := \text{ROUTE}(G, T, C, F)$, and $e = uv$ be any edge with $u, v \in C$ (e might not be in G) with capacity c_e . First note that $G[C] \subseteq H[C]$.*

If $H \leq_\alpha G$ holds via the canonical j -tree embedding, then both $(H + e) \leq_\alpha (G + e)$ and $(H - e) \leq_\alpha (G - e)$ holds.

Proof. Let \mathbf{f} be the canonical j -tree embedding of G into H . Let \mathbf{f}^+ be an embedding of $G + e$ into $H + e$ defined by routing $e \in (G + e)$ using $e \in (H + e)$ and routing other edges using the one defined in \mathbf{f} . To bound congestion incurred in $H + e$ using \mathbf{f}^+ , observe that $|\mathbf{f}^+(e_H)| = |\mathbf{f}(e_H)|$ for any $e_H \in H$ and $|\mathbf{f}^+(e)| = c_e$. The observation comes from the fact routing $e \in (G + e)$ only affects $e \in (H + e)$. Thus, $(H + e) \leq_\alpha (G + e)$ holds. If $e \in G$, observe that \mathbf{f} routes e using only its counterpart in H . It is because \mathbf{f} routes e using the path $P_{T,C,F}(e)$ containing only $e \in H$. Thus, define \mathbf{f}^- , an embedding of $G - e$ into $H - e$, by routing any other edge than e via \mathbf{f} . For any edge $e_H \in H - e$, we have $|\mathbf{f}^-(e_H)| = |\mathbf{f}(e_H)|$. Therefore, no edge has congestion increased and $(H - e) \leq_\alpha (G - e)$ holds. \square

By the above 2 lemmas, we can guarantee low congestion if we maintain the j -tree correctly.

We need the following dynamic tree data structure.

Lemma 6.25. *Given a rooted forest T edge weight $w : E(T) \rightarrow \mathbb{R}$, there is a deterministic data structure $D(T)$ supports following operations in $O(\log n)$ amortized time.*

1. $\text{root}(u)$: Return the root of the tree containing u .
2. $\text{makeRoot}(u)$: Make u as the root of the tree containing it.
3. $\text{pathMax}(u, v)$: Return the edge with maximum weight in the unique uv path. Or $-\infty$ if u, v are not connected in T .
4. $\text{cut}(e)$: Remove the edge e from T .
5. $\text{link}(u, v, c)$: Add a new edge $e = uv$ with weight c . It is guaranteed that no cycle is formed after adding this new edge.

Lemma 6.26. *Given a rooted forest T and a subset of vertices $C \subseteq V(T)$, there is a deterministic data structure $S(T)$ maintaining $S(T, C)$, the skeleton tree of T with respect to C , under following operations in $O(\log n)$ amortized time.*

1. $\text{AddC}(u)$: Return the set $V(S(F, C \cup \{u\})) \setminus V(S(F, C))$, which has size at most 2. Then add u to C .
2. $\text{Neighbor}(u \in C)$: Return the neighboring vertices of u in $S(F, C)$.

6.8.2 Proof sketch of Lemma 6.18

Intuitively, we maintain the j -tree H by mimicking the static procedure. To make the resulting graph sparse, a dynamic cut sparsifier is used for the core. Worth noticing, we maintain both the whole j -tree H and the one with sparsified core, \tilde{H} . The reason for not applying sparsifier to the whole graph is because edges not in the core form a forest. And by Lemma 6.15, we only care cuts in the core graph.

The most important part of our data structure is to support the `ADDTERMINAL` operation. Initially, the j -tree H is constructed by `ROUTE`(G, T, C, F). When adding some vertex u to C , we have to (1) find the edge e_u in $T \setminus F$ and (2) update H as `ROUTE`($G, T, C + u, F + e_u$).

Thus, We maintain the following data structures:

1. A dynamic 2-cut sparsifier $\tilde{C}(H)$ from Lemma 6.16 for maintaining a sparsified core graph.
2. A dynamic tree data structure $D(T)$ from Lemma 6.25 for finding such e_u . $D(T)$ is also used to find $\text{REPR}_{T,C,F}(e_u)$, the corresponding edge of e_u in the core.
3. For every off-tree edge $e = uv$, maintain both $T[u, T_{uv}(C)]$ and $T[T_{vu}(C), v]$ walks using doubly linked list. Maintain \mathscr{W} as a collection of all such walks.
4. For every $x \in C$, maintain a set $P(x)$ consisting of walks in \mathscr{W} ending up at x .
5. For every vertex $u \in V$, maintain a set $\text{RI}(u)$ consisting of walks in \mathscr{W} containing u .

The last 3 data structure is for maintaining $\text{MOVE}_{T,C}(e)$, $e \notin T$ with C increasing.

6.8.3 Formal proof of Lemma 6.18

Proof of Lemma 6.18. We may assume the j -tree, H , is constructed using the static procedure stated previously. Recall that T is the low-stretch spanning tree of G . C is the set of terminal (vertices in the core) of H . F is the set of tree edges chopped off. The procedure for `ADDTERMINAL`(u) is presented in Algorithm 15.

Algorithm 14: `INITIALIZE`(G, H)

- 1 Let T be the low stretch spanning tree used in constructing H .
 - 2 Initialize $D(T)$ for T from Lemma 6.25
 - 3 Initialize $S(T)$ for T from Lemma 6.26
 - 4 Initialize a dynamic cut sparsifier from Lemma 6.16 for $C(H)$, say $\tilde{C}(H)$.
 - 5 Let C be the initial terminal set, i.e., $V(C(H))$.
 - 6 $\mathscr{W} := \phi$.
 - 7 $\text{RI} := \phi$.
 - 8 $P := \phi$.
 - 9 **for** $e = uv \in E(G) \setminus E(T)$ **do**
 - 10 Let w_u be the $T[u, T_{uv}(C)]$ walk.
 - 11 Add w_u to $\text{RI}(a)$ for every $a \in V(w_u)$.
 - 12 **if** $T_{uv}(C)$ exists **then**
 - 13 Add w_u to $P(T_{uv}(C))$.
 - 14 Add w_u to \mathscr{W} .
 - 15 Same for the $w_v := T[T_{vu}(C), v]$ walk.
 - 16 **return** ($D(T), S(T), \tilde{C}(H), \mathscr{W}, \text{RI}, P$)
-

Algorithm 15: ADDTERMINAL(u)

```
1 Let  $(D(T), S(T), \tilde{C}(H), \mathscr{W}, \text{RI}, P)$  be the data structures defined in Algorithm 14
2  $E^+ := \phi, E^- := \phi$ ; //  $E^+$  and  $E^-$  are the sets of edges to be inserted or deleted in  $C(H)$ 
   respectively
3  $x := D(T).\text{root}(u)$ ; //  $x$  is the vertex in  $C$  in  $u$ 's component in  $T \setminus F$ .
4  $e_u := D(T).\text{pathMax}(t, u)$ ; //  $e_u$  is the edge with highest  $c_H(e)$  in  $T[x, u]$ 
5  $D(T).\text{cut}(e_u)$ 
6  $D(T).\text{makeRoot}(u)$ .
   // Update  $C := C + u$  and  $F := F + e_u$ .
7 Add  $(x, u, c_H(e_u))$  to  $E^+$ .
8 Add edges in  $T$  incident to  $u$ 's component in  $T \setminus F$  to  $E^-$ .
9 For edges visited in previous step, add them to  $E^+$  by replacing 1 endpoint  $x$  to  $u$ .
   // Maintain  $\text{Move}_{T,C}(e)$  for edges  $e \notin T$ .
10 for  $W \in \mathscr{W}$  such that  $u \in W$  do
11   | Add the core edge for  $W$  to  $E^-$ .
12   | Shortcut  $W$  at  $u$ .
13   | Add the core edge for the shortened  $W$  to  $E^+$ .
   // Update edges in the core.
14 for  $e \in E^+$  do
15   |  $\tilde{C}(H).\text{Insert}(e)$ 
16 for  $e \in E^-$  do
17   |  $\tilde{C}(H).\text{Delete}(e)$ 
18 return  $(D(T), S(T), \tilde{C}(H), \mathscr{W}, \text{RI}, P)$ 
```

When adding a vertex u to the terminal set C , we first have to find a tree edge e_u via $D(T)$. Then we update both C , the terminal set, and F , tree partition with respect to new C . As shown in Lemma 6.23, finding such e_u reduces to a path query in a dynamic tree. After that, we have to update H to $\text{ROUTE}(G, T, C, F)$. $\text{ROUTE}(G, T, C, F)$ maps edges in T using $\text{REPR}_{T,C,F}$ and $\text{MOVE}_{T,C}$ otherwise. For edges in T having different $\text{REPR}_{T,C,F}(e)$, they corresponds to edges incident to u 's component in $T \setminus (F + e_u)$. This step can be made efficient by only move the smaller part out and relabel if necessary. This ensures a $O(|T| \log n) = O(n \log n)$ total time complexity. By amortizing them across j operations, this step has amortized $O((n/j) \log n)$ -time. To update $\text{MOVE}_{T,C}(e)$ for edges $e = vw \notin T$, we explicitly maintain both $T[v, T_C(v, w)]$ and $T[T_C(w, v), w]$ walks. Observe that $T_{C+u}(v, w)$ is either $T_C(v, w)$ or u depending on whether $u \in T[v, T_C(v, w)]$. This observation tells us that $T[v, T_C(v, w)]$ only gets shorter with prefix unchanged. When adding u to the terminal, we simply find all $T[v, T_C(v, w)]$ walks containing u and shortcut them at u . Using doubly linked list and pointers, we can find these walks and shortcut them with $O(1)$ overhead. The total time complexity on maintaining $\text{MOVE}_{T,C}(e)$ can be bounded by the total length of $T[v, T_C(v, w)]$ walks. From the construction of JTREE , we know every such $T[v, T_C(v, w)]$ has length $O(n/j)$. Hence the total time complexity is $O(mn/j)$, and amortized time complexity per operation is $O(mn/j^2)$. Note that each of these edge change incurs a $O(\log^6 n)$ for edge updates in $\tilde{C}(H)$, sparsifier of the core graph. Combining above bounds, we know $\text{ADDTERMINAL}(u)$ has amortized time complexity of

$$O\left(\log^6 n \left(\frac{mn}{j^2} + \frac{n}{j} \log n\right)\right) = O\left(\frac{mn}{j^2} \log^6 n\right).$$

For $\text{INSERT}(u, v, c)$ and $\text{DELETE}(e)$, we first add both endpoints to terminal as presented in Algorithm 16

Algorithm 16: INSERT(u, v, c)

- 1 AddTerminal(u)
 - 2 AddTerminal(v)
 - 3 $\tilde{C}(H)$.Insert(u, v, c)
-

Algorithm 17: DELETE($e = uv$)

- 1 AddTerminal(u)
 - 2 AddTerminal(v)
 - 3 $\tilde{C}(H)$.Delete($u, v, c(e)$)
-

and Algorithm 17. Then we directly insert/delete the interested edge from the core graph. The time complexity is occupied by the cost of adding terminal vertex. Since we correctly maintain a unsparsifier $O(j)$ -tree H , we have $H \leq G \leq_{\alpha} H$ by Lemma 6.23 and Lemma 6.24. Also, by the decomposability of cut approximation and the core of \tilde{H} is a 2-cut sparsifier, we have $H \leq G \leq_{O(\alpha)} H$. \square

6.9 Put everything together

Proof of Theorem 6.1. Let j be some parameter determined later and $k = \Theta(\frac{m \log U \log^2 n}{j})$. Initialization of

Algorithm 18: INITIALIZE(G)

- 1 $n := |V(G)|, m := |E(G)|, j := m^{2/3}, k := \Theta(\frac{m \log U \log^2 n}{j}), t := \Theta(\log n)$.
 - 2 Let $\mathbf{G} = \{G_1, \dots, G_k\}$ be a $(k, \tilde{O}(\log n), \Theta(j))$ -decomposition of G by Lemma 6.12.
 - 3 Sample t graphs with repetition from \mathbf{G} , say, G_1, \dots, G_t .
 - 4 **for** $i = 1, \dots, t$ **do**
 - 5 $D_i := \text{Initialize}(G, G_i)$ by Lemma 6.18.
 - 6 **return** $\mathbf{D} := \{D_1, \dots, D_t\}$
-

the data structure is summarized as Algorithm 18. First apply Lemma 6.12 to acquire a $(k, \tilde{O}(\log n), \Theta(j))$ -decomposition of G , say G_1, \dots, G_k . Then we apply Lemma 6.13 to sample $t = O(\log n)$ of them, say G_1, \dots, G_t . For each of G_i , we incur Lemma 6.18 to build data structures for dynamical operations. Let D_1, \dots, D_t be the data structures for each of G_1, \dots, G_t . 2-approximated dynamic cut sparsifiers from Lemma 6.16 is also built for the cores of D_1, \dots, D_t . Note that each D_i supports up to j operations, we rebuild G_1, \dots, G_k and D_1, \dots, D_t every j operations. To deal with the query mincut(s, t), we run the algorithm from Lemma 6.17 on each sparsified core of D_1, \dots, D_t . The running time is $\tilde{O}(t \times j) = \tilde{O}(j)$. Among results, the one with the smallest cut value is returned. The correctness comes from Lemma 6.13 and Lemma 6.15 with high probability. The quality of the result is within $\tilde{O}(\log n)$ -factor with the optimal solution. For edge updates, we propagate them to D_1, \dots, D_t in amortized time $O(t \cdot \frac{mn}{j^2} \log^6 n) = O(\frac{mn}{j^2} \log^7 n)$. As guaranteed by Lemma 6.18, each operation corresponds to $O(\frac{mn}{j^2})$ changes to the core. Each of the edge change is handled by the cut sparsifier in $O(\log^6 n)$ -time. So the update time is

$$O\left(\frac{mn}{j^2} \log^2 n + t \cdot \frac{mn}{j^2} \log^6 n\right) = O\left(\frac{mn}{j^2} \log^7 n\right).$$

The cost for rebuild consists of 2 parts, $O(km \log m)$ -time for building decomposition of G and $O(tm \log^6 n)$ -time for initializing D_1, \dots, D_t and cut sparsifiers for cores. By charging the cost among j operations, the

runtime cost charged with each operation is

$$O\left(\frac{km \log n + tm \log^6 n}{j}\right) = O\left(\frac{m \log U \log^2 n \cdot m \log n}{j^2}\right) = O\left(\frac{m^2}{j^2} \log^4 n\right).$$

To balance the query cost and update cost, j is set to $m^{2/3}$. So time complexity per operation is now $\tilde{O}(m^{2/3})$. \square

6.10 Dynamic Max-flow Against an Adaptive Adversary

We next show how to modify our j -tree based data-structure to obtain a randomized dynamic algorithm that works against an adaptive adversary.

Theorem 6.27. *Given a graph $G = (V, E, c)$ with polynomially bounded capacities, there is a dynamic data structure that maintains G against an adaptive adversary subject to the following operations:*

1. *INSERT(u, v, c): Insert the edge (u, v) to G in $\tilde{O}(m^{3/4})$ amortized time.*
2. *DELETE(e): Delete the edge e from G in $\tilde{O}(m^{3/4})$ amortized time.*
3. *MINCUT(s, t): Output an $\tilde{O}(\log n)$ -approximation to the st -min-cut value of G in $O(m^{3/4})$ time w.h.p. The cut set S can be obtained on demand with linear overhead in $|S|$.*

Our previous construction used the oblivious adversary assumption in two places. (1) First, when building a decomposition of the original graph into $O(j)$ -trees, we sampled only a logarithmic number of them during the preprocessing phase and dynamically maintained these sampled graphs. Note that an adaptive adversary could use the query operation to reveal information about the random bits used by our algorithm and which graphs we sampled, and this is why we needed to assume that adversary is oblivious. To circumvent this assumption, we instead maintain all the $O(j)$ -trees in the decomposition and sample a small number of them only when handling queries. (2) Second, the dynamic cut sparsifier from Lemma 6.16 works only against an oblivious adversary, so we need a dynamic cut sparsifier that works against an adaptive adversary. In fact, because we explicitly compute a sparsifier on the core vertices, it suffices to have a data structure that outputs the sparsifier in time proportional to the number of core vertices. This allows to use fresh random bits when sampling a sparsifier during the query operation. Such an algorithm can be inferred from previous literature by combining expander decomposition based construction of graph sparsifiers [ST11] with recent works on decremental maintenances of expanders [NSW17, SW19]. Slightly more formally, given an n -vertex graph G , using the pruning procedure from [SW19], we can maintain an expander decomposition under edge deletions and recurse on the edges between expander clusters. To handle edge insertions, we employ a well-known reduction from decremental to full-dynamic algorithms (see e.g., Lemma 4.17 from [ADK⁺16]), which in turn leads to a fully-dynamic algorithm for maintaining a hierarchy of expander decompositions. Since cut/spectral sparsifiers are decomposable, and constructing them on expanders amounts to sampling $O(\log n \epsilon^{-2})$ random edges per vertex [ST11, PS14], it follows that constructing a sparsifier from the current hierarchy of expanders can be done in $\tilde{O}(n \epsilon^{-2})$ time. The above idea is explicitly implemented in the recent work by Bernstein et al. [BvdBG⁺20] and we formally state their result below.

Lemma 6.28. [BvdBG⁺20, Theorem 10.5] *There exists a fully dynamic algorithm that maintains for any weighted graph with an $(1 + \epsilon)$ -approximate cut sparsifier against an adaptive adversary. The algorithm's pre-processing time is bounded by $O(m)$, amortized update time is $\tilde{O}(1)$ and query time is $\tilde{O}(n \epsilon^{-3} \log U)$. The query operation returns an $(1 + \epsilon)$ -approximate cut sparsifier of G .*

We now explain the necessary modifications to our data-structure. Similarly to Algorithm 18, given a graph G , we compute a $(k, \tilde{O}(\log n), \Theta(j))$ -decomposition $\mathbf{G} = \{G_1, \dots, G_k\}$ of G using Lemma 6.12, where $k = \Theta\left(\frac{m \log^2 n \log U}{j}\right)$. We maintain each G_i from \mathbf{G} using the data-structure D_i from Lemma 6.18, where for each core in G_i we maintain an adaptive dynamic cut sparsifier using Lemma 6.28 (recall that previously we sampled $O(\log n)$ G_i 's from \mathbf{G} and maintained a dynamic sparsifier against an *oblivious* adversary for each of them). These data-structures are rebuilt from scratch every j operations. Upon receiving an edge insertion or deletion, we pass the corresponding update to each D_i . When receiving an st -min cut query, we first add s and t to the core of each G_i and then sample G_1, \dots, G_t with repetition from \mathbf{G} , where $t = \Theta(\log n)$. For each $i = 1, \dots, t$, we construct a cut sparsifier for the core of G_i using the query operation from Lemma 6.28. On each sparsified core of G_i 's we compute an st -min cut from scratch and then return the smallest value among those min cuts as an estimate. Note that sampling $\Theta(\log n)$ graphs whenever we receive a query ensures that the adversary cannot learn anything useful about our algorithm.

Proof of Lemma 6.27. The correctness proof is exactly the same as in Theorem 6.1. We next study the running time. The preprocessing cost consists of (1) the cost for computing the decomposition \mathbf{G} and (2) and the cost for initializing the data-structure D_1, \dots, D_k . By Lemma 6.12, (1) is bounded by $\tilde{O}(km)$ while (2) is bounded by $\tilde{O}(kmn/j)$ by Lemma 6.18. Since we rebuild our data-structure from scratch every j operations, the cost of the rebuild charged to each operation is

$$\tilde{O}\left(\frac{km + kmn/j}{j}\right) = \tilde{O}\left(\left(\frac{m^2}{j^2} + \frac{m^2 n}{j^3}\right) \log U\right) = \tilde{O}\left(\frac{m^3 \log U}{j^3}\right),$$

where the last inequality uses that $j \leq m$.

Next, by Lemma 6.18, the amortized time to support an edge insertion or deletion in D_i is $\tilde{O}(mn/j^2)$. Since we maintain k different D_i 's, it follows that the amortized time per edge insertion or deletion is bounded by

$$\tilde{O}\left(k \cdot \frac{mn}{j^2}\right) = \tilde{O}\left(\frac{m^2 n \log U}{j^3}\right).$$

Combining the above bounds, it follows that the amortized update time is $\tilde{O}\left(\frac{m^3 \log U}{j^3}\right)$.

Up to a logarithmic factor, the query cost is dominated by (1) the time to construct a cut sparsifier for the core and (2) the time to compute an st -min cut on a graph of size $\tilde{O}(j)$. As both can be implemented in $\tilde{O}(j)$ time, it follows that the query time is also $\tilde{O}(j)$. To balance the update and query time, we set $j = m^{3/4}$, which proves the lemma. \square

7 Fully-Dynamic All-Pairs Shortest Paths

In this section, we once again demonstrate the power of *Fully-Dynamic Vertex Sparsifier* by designing a $\tilde{O}(\log n)$ -approximate dynamic APSP oracle with sublinear update and query time. The main result of this section is formalized as the following theorem:

Theorem 7.1. *Given a graph $G = (V, E, l)$, we have a fully-dynamic data structure that maintains all pair distance up to $\tilde{O}(\log n)$ -factor and supports following operations:*

1. *INSERT(u, v, c): Insert the edge (u, v) to G in amortized $O(m^{2/3} \log^4 n)$ -time.*
2. *DELETE(e): Delete the edge e from G in amortized $O(m^{2/3} \log^4 n)$ -time.*
3. *DISTANCE(s, t): Output a $\tilde{O}(\log n)$ -approximation to the st -distance value of G in $O(m^{2/3+o(1)})$ -time w.h.p..*

Apply the dynamic spanner from [FG19] on the input graph G , we can reduce the number of edges from m to $O(n^{1+o(1)})$ while preserving distance up to $O(1)$ -factor.

Corollary 7.2. *Given a graph $G = (V, E, l)$, we have a fully-dynamic data structure that maintains all pair distance up to $\tilde{O}(\log n)$ -factor and supports following operations:*

1. *INSERT(u, v, c): Insert the edge (u, v) to G in amortized $O(n^{2/3+o(1)})$ -time.*
2. *DELETE(e): Delete the edge e from G in amortized $O(n^{2/3+o(1)})$ -time.*
3. *DISTANCE(s, t): Output a $\tilde{O}(\log n)$ -approximation to the st -distance value of G in $O(n^{2/3+o(1)})$ -time w.h.p..*

7.1 Path, Distance and L_1 -embeddability

In this subsection, we present concepts regarding preserving distance.

Definition 7.3. *Given 2 graphs on the same vertex set $G = (V, E, l), H = (V, E_H, l_H)$. A routing of H into G is a set of paths $\mathcal{P} = \{P^e \text{ an } uv\text{-path in } H \mid e = uv \in E\}$. We say G is t -routable in H (or H t -routes G), denoted by $H \leq_t^1 G$, if for every edge $e \in G$, $l_H(P^e) \leq t \cdot l(e)$ holds. The subscript is often omitted when $t = 1$.*

When the routing \mathcal{P} is clear in the context, we often write $l_H(e)$ to denote $l_H(P^e)$.

Lemma 7.4. *Given 2 graphs on the same vertex set $G = (V, E, l), H = (V, E_H, l_H)$. If G is t -routable in H , $H \leq_\beta^1 G$, then $d_H(s, t) \leq \beta \cdot d_G(s, t), \forall s, t \in V$.*

Corollary 7.5. *If $G \leq^1 H \leq_\beta^1 G$, we have*

$$d_G(s, t) \leq d_H(s, t) \leq \beta \cdot d_G(s, t), \forall s, t \in V.$$

Corollary 7.6. *If $H \subseteq G$, we have $G \leq^1 H$.*

7.2 Metric- J -Trees as Vertex Sparsifier

Here we introduce the notion of *Metric- J -Tree*, whose name adopts from j -tree in [Mad10]. Given a graph, a *Metric- J -Tree* is built from a spanning tree with additional $O(j)$ edges. In a graph of this family, distance computation can be speed up by transforming such graph into one with much less vertices. This property is exactly what we need for vertex sparsifier.

Definition 7.7. *Given a graph $G = (V, E, l)$, positive integer j , a subset of edges F of size at most j , and a spanning tree T of G . The Metric- j -tree of G with respect to T and F , denoted by $J_G^1(T, F)$, is a subgraph of G with vertex set V and edge set*

$$E_H = E(T) \cup F.$$

That is, $J_G^1(T, F)$ keeps only spanning tree T and edges in F . Also, when we speak of routing of $J_G^1(T, F)$ into G , we route edge $e = uv$ not in F or T via T_e , the unique uv -path in T . And route other edges using their identical counterpart.

Such structure is interested because of the following theorem. It suggest that we can approximate distance within several $J_G^1(T, F)$. First, we define metric-decomposition of G .

Definition 7.8. *Given a graph $G = (V, E, l)$ and a family of graphs \mathcal{G} . A collection of graphs H_1, \dots, H_k and k real numbers $\lambda_1, \dots, \lambda_k$ is a (k, ρ, \mathcal{G}) -metric-decomposition of G if*

1. $\lambda_i \geq 0, \forall i$ and $\sum_i \lambda_i = 1$.
2. $H_i \in \mathcal{G}, \forall i$.
3. $G \leq^1 H_i, \forall i$.
4. $\sum_i \lambda_i H_i \leq^1_\rho G$.

When \mathcal{G} is the family of all metric- j -tree of G , we denote (k, ρ, \mathcal{G}) -metric-decomposition of G by (k, ρ, j) -metric-decomposition of G .

Such decomposition can be computed via MWU-like method, just like [Mad10]. It is formalized as the following:

Theorem 7.9. *Let $G = (V, E, l)$ be a graph with weight ratio $U = \text{poly}(n)$. In $O(km \log^4 n)$ -time, we can find a*

$$\left(k, \alpha = \tilde{O}(\log n), O\left(\frac{m \log^3 n}{k}\right) \right)$$

-metric-decomposition of G . Or conversely, given positive integer j , we can compute a $(O((m/j) \log^3 n), \tilde{O}(\log n), j)$ -metric-decomposition of G in $O((m^2/j) \log^7 n)$ time.

The proof of Theorem 7.9 uses same MWU-like approach as [Mad10]. Briefly, we compute one component of the decomposition one at a time. In addition to edge length, we also incur edge weight on the graph. Intuitively, such edge weight regularizes stretch of over-stretched edges in the current decomposition.

We formally define stretch and how this edge weight interact with the graph. For a given edge weight function \mathbf{w} on G , and a graph $H = (V, E_H, l_H)$ that routes G . Define the *volume* $\mathbf{w}(H)$ of H (with respect to \mathbf{w}) to be $\mathbf{w}(H) := \sum_{e \in G} \mathbf{w}(e) l_H(e)$. Also, for any edge $e \in G$, define the *stretch of e in H* to be $\eta_H(e) := \frac{l_H(e)}{l(e)}$ and denote by $\eta(H)$ the maximum value of $\eta_H(e)$, i.e. $\eta(H) := \max_{e \in G} \eta_H(e)$. Furthermore, define a set $\psi(H)$ as

$$\psi(H) := \{e \in G \mid 0.5\eta(H) \leq \eta_H(e)\}.$$

Intuitively, the set $\psi(H)$ contains edges of G that suffer stretch at least as the half of the maximum stretch.

Given these definitions, the MWU method is formally stated as the following with proof deferred in the Appendix B.1:

Lemma 7.10. *Let $\alpha \geq \log m$ and a family of graphs \mathcal{G} such that for any edge weight function \mathbf{w} on G , we can find in $O(f(m))$ time a subgraph $H_{\mathbf{w}} = (V, E_{H_{\mathbf{w}}}, l_{H_{\mathbf{w}}})$ of G that belongs to \mathcal{G} and:*

1. $\mathbf{w}(H_{\mathbf{w}}) \leq \alpha \mathbf{w}(G)$,
2. $G \leq^1 H_{\mathbf{w}}$, and
3. $|\psi(H_{\mathbf{w}})| \geq \frac{4\alpha m}{k}$

then a $(k, 2\alpha, \mathcal{G})$ -metric-decomposition of G can be computed in $O(k \cdot f(m))$ time.

Next we will discuss how to construct $J_G^1(T, F)$ given edge weight function \mathbf{w} that satisfies all 3 conditions Theorem 7.10 needs. First we present the following lemma that derives from low stretch spanning tree construction in [ABN08].

Lemma 7.11. *Let $G = (V, E)$ be a graph with non-negative edge length l and edge weight w . We can find a spanning tree T of G (with edge length l) s.t.*

$$l_{uv} \leq d_T(u, v), \forall e = uv$$

and

$$\sum_{e=uv} d_T(u, v) w_e \leq \alpha \sum_{e=uv} l_e w_e$$

for some $\alpha = O(\log n \log \log n) = \tilde{O}(\log n)$. Such tree T can be computed in $\tilde{O}(m)$ -time.

From now on, we use α to denote the ratio in Lemma 7.11.

Given a graph $G = (V, E, l)$, positive integer k for sparsity of *metric-decomposition*, and a edge weight function w . Also let $U = \text{poly}(n)$ be the weight ratio of G . The construction works like the following

1. Compute LSST T using Lemma 7.11 with respect to l and w . Route G using T , i.e., route each edge $e = uv$ using the unique uv -path in T .
2. For every edge $e = uv \in G$, compute its stretch, $\eta(e) = d_T(u, v)/l_e$, in $O(m)$ time. Observe that $\eta(e) \leq mU$.
3. Partition edges of G into $\log(mU)$ sets $F_1, F_2, \dots, F_{\log(mU)}$, with $F_i := \{e \in G \mid 2^{i-1} \leq \eta(e) < 2^i\}$. Define $F_{\geq i} = \bigcup_{j=i}^{\log(mU)} F_j$. Add edges with $\eta(e) < 1$ to F_1 .
4. Find smallest $j^* \leq \log(mU)$ such that

$$|F_{\geq j^*}| \leq \frac{4(2\alpha + 1)m \log mU}{k}, |F_{\geq j^*-1}| > \frac{4(2\alpha + 1)m \log mU}{k}$$

If no such j^* exists, i.e., $|F_{\log(mU)}| > 4(2\alpha + 1)m \log mU/k$, let $F = \phi$ and output $H_w = J_G^1(T, \phi) = T$.

5. By Pigeonhole principle, there is some $\bar{j}, j^* \leq \bar{j} \leq \log(mU)$ such that

$$|F_{\bar{j}-1}| \geq \frac{4(2\alpha + 1)m}{k}.$$

6. Let F be the set $F_{\geq \bar{j}}$ and output $H_w = J_G^1(T, F)$.

For condition B.1 of Lemma 7.10, we know $w(T) \leq \alpha w(G)$. Since T is a subgraph of H_w , $d_{H_w}(u, v) \leq d_T(u, v)$, $\forall u, v$ holds for every pair of vertex u, v . We have $w(H_w) \leq w(T) \leq \alpha w(G)$.

For condition 2 of Theorem 7.10, $G \leq^1 H_w$ holds trivially since H_w is a subgraph of G .

For condition 3 of Theorem 7.10, if the procedure ends at Step 4 and outputs G , we know $\psi(H_w) \supseteq F_{\log(mU)}$ and $|F_{\log(mU)}| > 4(2\alpha + 1)m \log mU/k$. Thus,

$$|\psi(H_w)| \geq |F_{\log(mU)}| > \frac{4(2\alpha + 1)m \log mU}{k} \geq \frac{4\alpha m}{k}$$

Otherwise, we observe that in $J_G^1(T, F_{\geq \bar{j}})$, edges in $F_{\geq \bar{j}}$ have stretch 1 and therefore $\psi(H_w) \supseteq F_{\bar{j}-1}$ and $|F_{\bar{j}-1}| \geq 4(2\alpha + 1)m/k$.

Also notice that the set F has size $\leq 4(2\alpha + 1)m \log mU/k = O(m \log^3 n/k)$.

The above procedure works in $\tilde{O}(m)$ -time which is occupied by the LSST construction from Lemma 7.11. It is summarized as the following lemma:

Lemma 7.12. Given a graph $G = (V, E, w)$, positive integer k , and edge weight function w . We can compute a spanning tree T and a subset of edges F in $\tilde{O}(m)$ time such that

1. $w(J_G^1(T, F)) \leq \alpha w(G)$,
2. $G \leq^1 J_G^1(T, F)$,
3. $|\psi(J_G^1(T, F))| \geq \frac{4\alpha m}{k}$, and
4. $|F| = O(\frac{m \log^3 n}{k})$.

Proof of Theorem 7.9. It comes directly from Lemma 7.10 and Lemma 7.12. □

7.3 Transform $J_G^1(T, F)$ into a vertex sparsifier

In this section, we show how to construct vertex sparsifier, H_C , that preserve distance within interested terminal set C . Let H_C be an empty graph with vertex set C initially. The construction works as follows:

1. Given $J_G^1(T, F)$, add endpoints of edges in F to C .
2. Add vertices in $S(T, C)$, degree 3 vertices in the Steiner tree of C in T , to C as well.
3. Since vertices of $S(T, C)$ are in C as well, for every edge $e = uv \in S(T, C)$, add edge uv with length $l(T_e)$ to H_C . That is, edge uv corresponds to the unique tree uv -path T .
4. For any edge $e \notin T$, add $\text{MOVE}_{T, C}(e)$ to with length $l(P_{T, C}(e))$ to H_C .

Suppose T, F, C are given, denote the construction of H_C by $\text{ROUTE}^1(G, T, C, F)$.

It works almost the same as the vertex sparsifier construction for the dynamic max flow problem.

Such H_C is a preserve terminal-wise distance in $J_G^1(T, F)$, which is formalized and proved in the following lemma:

Lemma 7.13.

$$\forall s, t \in C, d_{H_C}(s, t) \leq d_{J_G^1(T, F)}(s, t).$$

Proof. Let P^* be the shortest st -path in $J_G^1(T, F)$. We break P^* into maximal segments of paths, P_1, \dots, P_k , such that each of them intersects with C only at endpoints. By construction, each P_i is either i) a tree path, or ii) an edge comes from F . Both possibilities has their mapped edge in H_C . Let e_1, \dots, e_k be corresponding edges of P_1, \dots, P_k . Clearly, e_1, \dots, e_k forms a st -path in H_C . hence

$$d_{J_G^1(T, F)}(s, t) = \sum_{i=1}^k l_{J_G^1(T, F)}(P_i) = \sum_{i=1}^k l_{H_C}(e_i) \geq d_{H_C}(s, t).$$

□

If $J_G^1(T, F)$ preserve distance of G for vertex set C within factor of t , so does H_C .

Corollary 7.14. If $J_G^1(T, F) \leq_t^1 G$,

$$\forall s, t \in C, d_{H_C}(s, t) \leq t \cdot d_G(s, t).$$

By dynamically maintain this construction, we have the following dynamic data structure:

Lemma 7.15. Given a graph $G = (V, E, w)$, positive integer j , spanning tree T and a subset of edges F of size $O(j)$. Suppose $J_G^1(T, F) \leq_t^1 G$ for some $t > 0$, we can maintain a vertex sparsifier \tilde{H} that maintains terminal-wise distance up-to t -factor that supports up to $O(j)$ of following operations:

1. *INITIALIZE*(G, T, F): Build data structures for maintaining H in $O((mn/j) \log n)$ -time.
2. *ADDTERMINAL*(u): Add u to the terminal set of $C = C_G(T, F)$. Such operation can be done in amortized $O((mn/j^2) \log^3 n)$ -time.
3. *INSERT*(u, v, c): Insert the edge (u, v) to G in amortized $O((mn/j^2) \log^3 n)$ -time.
4. *DELETE*(e): Delete the edge e from G in amortized $O((mn/j^2) \log^3 n)$ -time.

The total number of edge changes in \tilde{H} is $O(mn/j)$, hence each operation has amortized recourse of $O(mn/j^2)$. Also, \tilde{H} has $O(n^{1+o(1)})$ edges.

7.4 Dynamic Metric J -Tree

In this section, we present tools and arguments that help us prove Lemma 7.15.

7.4.1 Structural arguments

To prove Lemma 7.15, one has to make sure adding terminals does not increase the stretch. The argument is formalized as the following lemma:

Lemma 7.16. Given a graph $G = (V, E, l)$, a spanning forest T of G , a subset of vertices C and F , a subset of edges. Suppose there is no branch vertex in T with respect to C , i.e. $V(S(T, C)) = C$. Let u be a vertex. We have $V(S(T, C + u)) \supseteq C$ and $|V(S(T, C + u)) \setminus C| \leq 2$.

Furthermore, let $H := \text{ROUTE}^1(G, T, C, F)$. If $H \leq_\alpha G$, then the graph $\overline{H} := \text{ROUTE}^1(G, T, V(S(T, C + u)), F) \leq_\alpha G$.

Proof. Suppose we root T at u . Since $V(S(T, C)) = C$, then either i) all vertex in C lies in one subtree of T or ii) u lies in a path connecting 2 vertices of C . If i) happens, $V(S(T, C + u)) \setminus C$ has u and x , the lowest common ancestor of all vertices of C . If ii) happens, $V(S(T, C + u)) \setminus C = \{u\}$.

Observe that we route every edge of G in \overline{H} using a shorter path. Thus the stretch does not increase. \square

To maintain metric- $O(j)$ -tree H under dynamic edge updates in G , we first add both endpoints of the updating edge to the terminal and then perform the edge update in H_C , the vertex sparsifier constructed from H . One has to make sure such behavior does not increase the stretch when routing G in H . The following lemma gives such promise:

Lemma 7.17. Given a graph $G = (V, E, l)$, a spanning forest T of G , a subset of vertices C and F , a subset of edges. Let $H := \text{ROUTE}^1(G, T, C, F)$, and $e = uv$ be any edge with $u, v \in C$ (e might not be in G) with length l_e . First note that $G[C] \subseteq H[C]$.

If $H \leq_\alpha^1 G$, then both $(H + e) \leq_\alpha^1 (G + e)$ and $(H - e) \leq_\alpha^1 (G - e)$ holds.

Proof. For $(H + e) \leq_\alpha^1 (G + e)$, $H + e$ can route edges in G via the routing that implements $H \leq_\alpha^1 G$. For newly added edge e , $H + e$ route it using the e .

Since H routes G by tree-terminal path, edges not in $H[C]$ are routed without using $e \in H[C]$. For $(H - e) \leq_\alpha^1 (G - e)$, all edges of $G - e$ can be routed by $H - e$ using the old routing. \square

7.4.2 Data structure toolbox

To keep the resulting vertex sparsifier has small number of edges, we use the dynamic spanner data structure from [FG19].

Lemma 7.18 ([FG19]). *Given a graph $G = (V, E, c)$ with weight ratio $U = \text{poly}(n)$, there is a randomized fully dynamic data structure on maintaining a spanner of G with stretch $(1 + \epsilon)(2k - 1)$ and expected size $O(n^{1+1/k} \log^2 n \epsilon^{-1})$ with expected amortized update time $O(k \log^3 n)$.*

7.4.3 Proof of Lemma 7.15

Proof of Lemma 7.15. The data structure is almost the same as Lemma 6.18. Except we use Lemma ?? to update edges in \tilde{H} that corresponds to edge changes from $S(T, C)$ to $S(T, C + u)$. Also, instead of dynamic cut sparsifier, we use dynamic spanner from Lemma 7.18 to reduce the number of edges in the resulting vertex sparsifier. \square

7.5 Put everything together

From Theorem 7.9, we know with probability 0.5, some G_i sampled from distribution defined by λ_i 's preserve distance up to 4α -factor. Given this, our dynamic data structures first sampled $t = O(\log n)$ of them, say, G_1, \dots, G_t . Given any query s, t , we have

1. $d_G(s, t) \leq d_{G_i}(s, t), \forall s, t$ deterministically, and
2. $\min_{1 \leq i \leq t} d_{G_i}(s, t) \leq 4\alpha d_G(s, t)$ with high probability.

In order to compute $d_{G_i}(s, t)$ efficiently, we have to reduce both number of edges and vertices of G_i 's.

Proof of Theorem 7.1. Let $j = m^{2/3}, k = \Theta((m \log^3 n)/j)$. Just like Theorem 6.1 for dynamic max flow, we first apply Theorem 7.9 to acquire a $(k, \tilde{O}(\log n), \Theta(j))$ -metric-decomposition of G , say, $\{(\lambda_i, H_i)\}_{1 \leq i \leq k}$. Then we sample $t = O(\log n)$ of them, say, H_1, \dots, H_t with probability $\Pr(\text{draw } H_j) = \lambda_j$.

For each of $H_i, i \leq t$, we incur Lemma 7.15 to construct a data structure, say D_i , that support dynamic operations.

Every j operations, we rebuild the whole thing from scratch. Rebuild takes $O(km \log^n) = O((m^2 \log^4 n)/j)$ -time. We charge the rebuild cost to these j operations, each of them now is charged with $O((m^2 \log^4 n)/j^2) = O(m^{2/3} \log^4 n)$ -time. Each edge update, we propagate them to these t data structures. Each of the D_i can handle edge update in amortized $O((mn \log^3 n)/j^2) = O(m^{2/3} \log^3 n)$ -time. Since there are $t = O(\log n)$ of them, each edge update can be handled in amortized $O(m^{2/3} \log^4 n)$ -time.

Given a query s, t , we compute st -distance in each of the t vertex sparsifiers of size $O(j^{1+o(1)}) = O(m^{2/3+o(1)})$. With high probability, $\min_{1 \leq i \leq t} d_{D_i}(s, t) \leq \tilde{O}(\log n) d_G(s, t)$. We output the minimum of these t distances. Since we can compute exact distance in $O(|E| + |V| \log |V|)$ -time, the st query can be handled in $O(t \times m^{2/3+o(1)}) = O(m^{2/3+o(1)})$ -time. \square

8 Fully-Dynamic Effective Resistance

In this section, we utilize the local sparsifier construction from [DGGP19]. By apply the construction recursively, we can speed-up the data structure. It is formalized as the following theorem.

Theorem 8.1. *Given a positive integer d , error term $\epsilon \in (0, 1)$, and a graph $G = (V, E, c)$ with weight ratio $U = \text{poly}(n)$. There is a dynamic data structure maintaining G subject to the following operations:*

1. $\text{INSERT}(u, v, c)$: Insert the edge (u, v) to G in amortized $O(n^{2/3+1/(3d+3)}\epsilon^{-(2d+4)} \log^{2d+11} n)$ -time.
2. $\text{DELETE}(e)$: Delete the edge e from G in amortized $O(n^{2/3+1/(3d+3)}\epsilon^{-(2d+4)} \log^{2d+11} n)$ -time.
3. $\text{ER}(s, t)$: Output a $(1 + 2d\epsilon)$ -approximation to $\mathcal{R}_{\text{eff}}^G(s, t)$ in $O\left(n^{2/3+1/(3d+3)}\epsilon^{-(2d+2)} \log^{9d+10} n\right)$ -time.

All of above guarantees hold with high probability.

8.1 Effective Resistance, Random Walks and L_2 -Embeddability

In this subsection, we define notions related to the *Laplacian* of a graph. Notions and properties of Laplacians are critical in building the desired data structure. Boldface is used to indicate that the variable is a vector. We use χ_i to denote the vector with i -th coordinate being 1 and 0 elsewhere. Also define $\chi_{i,j} = \chi_i - \chi_j$.

Definition 8.2. Given a graph $G = (V, E, c)$. Let $\mathbf{B} \in \mathbb{R}^{V \times E}$ be the edge-incidence matrix of G , i.e., $\mathbf{B}_e = \chi_{u,v}$, $\forall e = uv \in E$ with arbitrary orientation. Let $\mathbf{C} \in \mathbb{R}^{E \times E}$ be the diagonal matrix with $C_{e,e} = c(e)$, $\forall e \in E$. The Laplacian $\mathbf{L}_G \in \mathbb{R}^{V \times V}$ of G is defined as

$$\mathbf{L}_G := \mathbf{B}^\top \mathbf{C} \mathbf{B}.$$

That is, $\mathbf{L}_G(u, u) = \sum_{e \in E} c(e)$, is the weighted degree of u . And $\mathbf{L}_G(u, v) = -\sum_{e=uv \in E} c(e)$, the minus sum of edge weights between u, v .

For edge weight always being non-negative in our discussion, we define the Laplacian norm with respect to x by

$$\|x\|_{\mathbf{L}_G} := \sqrt{x^\top \mathbf{L}_G x}.$$

Also, Moore-Penrose pseudo-inverse of \mathbf{L}_G is defined as \mathbf{L}_G^\dagger . Using this definition, we can define *effective resistance* between 2 vertices.

Definition 8.3. Given $G = (V, E, c)$, the effective resistance between 2 vertices u and v is defined as

$$\mathcal{R}_{\text{eff}}^G(u, v) := \chi_{u,v}^\top \mathbf{L}_G^\dagger \chi_{u,v}.$$

Laplacian system solver is used to compute effective resistance. As computing $\mathbf{L}_G^\dagger \chi_{u,v}$ is essentially solving for x under $\mathbf{L}_G x = \chi_{u,v}$. Therefore, $\mathcal{R}_{\text{eff}}^G(u, v) = x_u - x_v$. The fastest solver is due to [CKM⁺14] which formalized as follows:

Lemma 8.4. [CKM⁺14] Given a graph $G = (V, E, c)$, $b = \mathbf{L}_G x^*$ and error term $\epsilon > 0$. There is an algorithm that finds x w.h.p. such that

$$\|x^* - x\|_{\mathbf{L}_G} \leq \epsilon \|x^*\|_{\mathbf{L}_G}$$

in $O(m\sqrt{\log n} \log \frac{1}{\epsilon} \cdot (\log \log n)^{3+\delta})$ -time for any constant δ .

Using this fast solver, effective resistance between vertices can be computed efficiently.

Corollary 8.5. Given a graph $G = (V, E, c)$, $u, v \in V$ and error term $\epsilon > 0$. We can compute a value ϕ w.h.p. such that

$$(1 - \epsilon)\mathcal{R}_{\text{eff}}^G(u, v) \leq \phi \leq (1 + \epsilon)\mathcal{R}_{\text{eff}}^G(u, v)$$

in $\tilde{O}(m)$ -time.

Definition 8.6. Given a graph $G = (V, E, c)$ and $\epsilon \in (0, 1)$, we say a graph $H = (V, E_H \subseteq E, c_H)$ is a $(1 + \epsilon)$ -spectral-sparsifier of G if $\forall x \in \mathbb{R}^V$,

$$(1 - \epsilon)x^\top \mathbf{L}_G x \leq x^\top \mathbf{L}_H x \leq (1 + \epsilon)x^\top \mathbf{L}_G x.$$

Denoted by $H \approx_\epsilon G$.

Fact 8.7. $H \approx_\epsilon G$ implies $H \sim_\epsilon G$.

To speed up, we would like to compute effective resistance in the sparsified graph instead of the original one. The following fact supports motivation.

Fact 8.8. If $H \approx_\epsilon G$, $\forall u, v \in V$ we have

$$(1 - \epsilon)\mathcal{R}_{\text{eff}}^G(u, v) \leq \mathcal{R}_{\text{eff}}^H(u, v) \leq (1 + \epsilon)\mathcal{R}_{\text{eff}}^G(u, v).$$

For a graph $G = (V, E, c)$ with non-negative weight, we can define *random walk* in G using the distribution proportional to edge weight. That is, given a walk u_0, u_1, \dots, u_k , the probability

$$\Pr_G(u_{k+1} = v \mid u_0, \dots, u_k) = \frac{c(u_k v)}{c(u_k)}.$$

Alternatively, by fixing a starting vertex u_0 , a walk u_0, \dots, u_k is sampled with probability

$$\Pr_G(u_0, \dots, u_k) = \prod_{i=0}^{k-1} \frac{c(u_i u_{i+1})}{c(u_i)}.$$

8.2 Dynamic Spectral Sparsifier

Lemma 8.9. [ADK⁺ 16] Given a graph $G = (V, E, c)$ with weight ratio U . There is an $(1 + \epsilon)$ -spectral sparsifier H of G w.h.p.. Such H supports the following operations:

- **Insert(u, v, c):** Insert the edge (u, v) to G in amortized $O(\log^9 n \epsilon^{-2})$ -time.
- **Delete(e):** Delete the edge e from G in amortized $O(\log^9 n \epsilon^{-2})$ -time.

The weight ratio of H is $O(nU)$. Moreover, the size of H is $O(n \log^9 n \epsilon^{-2})$.

8.3 Schur Complement as Vertex Sparsifier

To design an efficient data structure for computing effective resistance, we need a smaller graph preserving desired information.

Definition 8.10. Given a graph $G = (V, E, c)$ and $C \subseteq V$. Write $D = V \setminus C$. We write the Laplacian of G as

$$\mathbf{L} = \begin{bmatrix} \mathbf{L}_{[C, C]} & \mathbf{L}_{[C, D]} \\ \mathbf{L}_{[D, C]} & \mathbf{L}_{[D, D]} \end{bmatrix}.$$

The Schur Complement of G onto C , denoted by $\text{SC}(G, C)$, is the matrix obtained after performing Gaussian Elimination on variables corresponds to D . The closed form is given by

$$\text{SC}(G, C) = \mathbf{L}_{[C, C]} - \mathbf{L}_{[C, D]} \mathbf{L}_{[D, D]}^{-1} \mathbf{L}_{[D, C]}.$$

Fact 8.11. Given $C_1 \subseteq C_2 \subseteq V$. We have $\text{SC}(G, C_1) = \text{SC}(\text{SC}(G, C_2), C_1)$.

Fact 8.12. $\text{SC}(G, C)$ is a Laplacian matrix of some graph with vertex set C .

Using this fact, we abuse the notation by using $\text{SC}(G, C)$ to denote both the Laplacian and the corresponding graph. An important property about $\text{SC}(G, C)$ is that it preserves effective resistance.

Fact 8.13.

$$\forall u, v \in C, \mathcal{R}_{\text{eff}}^G(u, v) = \mathcal{R}_{\text{eff}}^{\text{SC}(G, C)}(u, v).$$

If we can efficiently maintain the Schur Complement for some C small enough together with edge sparsification scheme, we can compute effective resistance in a much smaller graph using Lemma 8.4.

Building the $\text{SC}(G, C)$ naively is a sequential process, which does not fit in our dynamic data structure paradigm. In [DGGP19], they provide an alternative way of constructing $\text{SC}(G, C)$ approximately using random walks. The following lemma justifies this approach.

Lemma 8.14. [DPPR17] *Given any graph $G = (V, E, c)$ and a subset of vertices $C \subseteq V$. Given any walk $w = v_0, \dots, v_l$, we say w is terminal-free if $w \cap T = \{v_0, v_l\}$. The Schur complement $\text{SC}(G, C)$ is given as an union over all multi-edges corresponding to terminal-free walks v_0, \dots, v_l with weight*

$$c(v_0 v_l) \prod_{i=1}^{l-1} \frac{c(v_i v_{i+1})}{c(v_i)}.$$

8.4 Dynamic Schur Complement

In this section, we introduce a dynamic data structure for maintaining $\text{SC}(G, C)$ [DGGP19]. We adapt this data structure for faster effective resistance computation in the dynamic graph. To design a faster data structure, we deploy a recursive routine and slightly modify the tool. The result is stated as the following lemma:

Lemma 8.15. [DGGP19] *Given $\beta \in (0, 1)$, error term $\epsilon \in (0, 1)$, a graph $G = (V, E, c)$ with weight ratio U , and a terminal set $T \subseteq V$ with $|T| = \beta m$. There is a dynamic data structure \mathcal{D} that maintains $\tilde{H} \approx_{\epsilon} \text{SC}(G, C)$ for some $T \subseteq C$ with $|C| = \Theta(m\beta)$. Such \tilde{H} is a subgraph of $\text{SC}(G, C)$ with $O(m\beta\epsilon^{-2} \log^9 n)$ edges and weight ratio $O()$. Such data structure supports up to $O(m\beta)$ of the following operations:*

1. *INITIALIZE(G, T, β): Initialize \mathcal{D} in $O(m\beta^{-4}\epsilon^{-4} \log^4 n)$ time.*
2. *INSERT(u, v, w): Insert an new edge uv with weight w to G in amortized $O(\beta^{-2}\epsilon^{-2} \log^3 n)$ time.*
3. *DELETE(e): Delete the edge e from G in amortized $O(\beta^{-2}\epsilon^{-2} \log^3 n)$ time.*
4. *ADDTERMINAL(u): Move vertex u to T in amortized $O(\beta^{-2}\epsilon^{-2} \log^3 n)$ time.*

\mathcal{D} maintains $O(m\epsilon^{-2} \log n)$ random walks each with $O(\beta^{-1} \log n)$ distinct vertices. For every vertex $u \notin C$, total occurrence of u in these random walks is $O(\beta^{-2}\epsilon^{-2} \log^2 n)$. The total number of edge change in H is $O(m\beta^{-1}\epsilon^{-2} \log^2 n)$. All the above bounds hold with high probability.

Here we briefly outlined the process for Initialize(G, T, β) and point out a small modification from the original construction in [DGGP19].

1. Initialize C with T . For each edge uv , add both u, v to C with probability β . Let H be an empty graph.
2. For each edge $e = uv$, sample 2 random walks w_u, w_v starting from u and v respectively. Such random walk is generated until either a vertex in C is hit or $\Theta(\beta^{-1} \log n)$ distinct vertices is visited. For both walks hit vertices $a, b \in C$, Let r be the resistance between ab in the path $w_u + e + w_v$. An edge ab with weight $1/r$ is added to H .

3. Repeat above step for $\rho := \epsilon^{-2} \log n$ times and scale edge weights in H by $1/\rho$. Therefore, there are $O(m\epsilon^{-2} \log n)$ walks generated and their total size is $O(m\beta^{-1}\epsilon^{-2} \log^2 n)$. Balanced binary search trees are used to maintain these walks. Also, a reverse index for each vertex to locate the position in every walk containing it.
4. Identify the top βm vertices with most occurrence in these $O(m\epsilon^{-2} \log n)$ walks, add them to C as well. Meanwhile, shortcut those walks involving these vertices. By a Markov-type argument, vertex not in C has occurrence at most $O(\beta^{-2}\epsilon^{-2} \log^2 n)$.
5. Incur a dynamic $(1 + \epsilon)$ -spectral sparsifier from Lemma 8.9 on H . The resulting sparse graph \tilde{H} is the desired approximator for $\text{SC}(G, C)$.

The only difference from [DGGP19] is the 4th step, which is crucial for efficiency. As this step can be viewed as performing $\beta m \text{AddTerminal}(u)$ operations, which does not affect the approximation guarantee.

8.5 The Main Result

In this section, we prove the main theorem for dynamic effective resistance. The high-level idea is to utilize the Fact 8.11 and build layers of Schur Complements. The effective resistance computation is performed in the last layer, which has the smallest size.

Proof of Theorem 8.1. Let $\beta = n^{-1/(3d+3)}$. First we incur a dynamic $(1 + \epsilon)$ -spectral-sparsifier on G . Let G_0 be the sparsified G . Define a chain of graphs G_1, G_2, \dots, G_d , where G_{i+1} is the sparsified Schur Complement on $\beta|E(G_i)|$ vertices, i.e. $G_{i+1} \approx_\epsilon \text{SC}(G_i, C_i)$ for some C_i of size $\beta|E(G_i)|$. Each G_{i+1} is maintained using Lemma 8.15 on G_i with β and error term ϵ . Also, we need to rebuild G_{i+1} every $\beta|E(G_i)|$ steps. For every edge updates, it is propagated down to G_d . For $\text{ER}(s, t)$ query, both s, t are added to the terminal set of every G_i s. And then we incur Lemma 8.4 to compute $\mathcal{R}_{\text{eff}}^{G_d}(s, t)$, which gives a $(1 + \epsilon)^d = 1 + O(d\epsilon)$ -approximation. Now we analyze the running time for this data structure. Let $n_i = |V(G_i)|$ and $m_i = |E(G_i)|$, it is clear that $S := \log^9 n \epsilon^{-2} = \frac{m_i}{n_i}, \forall i$. Also, one edge update or terminal add in G_i creates amortized $\gamma := \beta^{-2}\epsilon^{-2} \log^{-2} n$ changes to G_{i+1} . For a single G_i , the rebuild cost is spread across βm_i operations. That is,

$$\frac{m_i \beta^{-4} \epsilon^{-4} \log^4 n}{\beta m_i} = \beta^{-5} \epsilon^{-4} \log^4 n$$

which dominates the actual time complexity for each dynamic operations. Also, every rebuild creates m_i changes to G_{i+1} . By distributing them across βm_i operations before next rebuild, every change in G_i creates amortized β^{-1} number of changes to G_{i+1} . But $\beta^{-1} = o(\gamma)$, therefore we can still bound the changes in G_{i+1} per change in G_i by $O(\gamma)$. For every change in G , it creates $S \cdot \gamma^{i-1}$ changes to G_i . And each change costs $\beta^{-5} \epsilon^{-4} \log^4 n$ to handle. So for every change, the amortized time can be expressed as

$$\begin{aligned} O\left(\sum_{i=1}^d S \cdot \gamma^{i-1} \cdot \beta^{-5} \epsilon^{-4} \log^4 n\right) &= O\left(\beta^{-(2d+3)} \epsilon^{-(2d+4)} \log^{2d+11} n\right) \\ &= O\left(n^{2/3+1/(3d+3)} \epsilon^{-(2d+4)} \log^{2d+11} n\right). \end{aligned}$$

And for $\text{ER}(s, t)$ query, Lemma 8.4 is ran on G_d , which has $m_d = nS(\beta S)^d$ edges. So each query, we can bound the time by

$$\begin{aligned} O(m_d \log n) &= O\left(n\beta^d \epsilon^{-(2d+2)} \log^{9d+10} n\right) \\ &= O\left(n^{2/3+1/(3d+3)} \epsilon^{-(2d+2)} \log^{9d+10} n\right). \end{aligned}$$

□

A Proof of Lemma 7.11

To prove Lemma 7.11, we use the following algorithm from [ABN08] that computes a low-stretch-spanning tree in $O(m \log n)$ -time.

Lemma A.1 ([ABN08]). *Given a graph $G = (V, E, l)$, there is an algorithm that computes a spanning tree T such that*

$$\frac{1}{|E|} \sum_{e=uv \in E} \frac{d_T(u, v)}{l(e)} \leq \alpha,$$

where $\alpha = O(\log n \log \log n) = \tilde{O}(\log n)$. The algorithm runs in $O(m \log n)$ -time.

Proof of Lemma 7.11. In order to use Lemma A.1, we have to convert the graph into the one without edge weights. For every $e \in E$, define

$$r(e) = 1 + \left\lfloor \frac{l(e)w(e)|E|}{\mathbf{w}(G)} \right\rfloor,$$

and create a graph \bar{G} identical to G except we add $r(e) - 1$ more parallel edges for every edge $e \in E$. Then we apply Lemma A.1 on \bar{G} and return the resulting spanning tree T . Such T satisfies

$$\sum_{e=uv \in E} \frac{d_T(u, v)r(e)}{l(e)} \leq \alpha |E(\bar{G})|$$

First, we show the running time by bound the size of \bar{G} . Note that

$$|E(\bar{G})| = \sum_{e \in E} r(e) \leq \sum_{e \in E} \left(1 + \frac{l(e)w(e)|E|}{\mathbf{w}(G)} \right) \leq |E| + |E| = 2|E|.$$

Thus, the algorithm runs in $O(m \log n)$ -time.

Next, we show the approximation guarantee. Observe that for every edge e ,

$$r(e) \geq \frac{l(e)w(e)|E|}{\mathbf{w}(G)} \geq \frac{l(e)w(e)}{\mathbf{w}(G)} \frac{\sum_{f \in E} r(f)}{2}.$$

Plug in this lower bound and we have

$$\alpha \sum_{e \in E} r(e) \geq \sum_{e=uv \in E} \frac{d_T(u, v)r(e)}{l(e)} \geq \sum_{e=uv \in E} \frac{d_T(u, v)w(e) \sum_{f \in E} r(f)}{2\mathbf{w}(G)},$$

and hence,

$$2\alpha\mathbf{w}(G) \geq \sum_{e=uv \in E} d_T(u, v)w(e).$$

Such T satisfies the requirement of this lemma. □

B Multiplicative-Weight-Update methods

B.1 Proof of Theorem 7.10

Lemma 7.10. *Let $\alpha \geq \log m$ and a family of graphs \mathcal{G} such that for any edge weight function \mathbf{w} on G , we can find in $O(f(m))$ time a subgraph $H_{\mathbf{w}} = (V, E_{H_{\mathbf{w}}}, l_{H_{\mathbf{w}}})$ of G that belongs to \mathcal{G} and:*

1. $w(H_w) \leq \alpha w(G)$,
2. $G \leq^1 H_w$, and
3. $|\psi(H_w)| \geq \frac{4\alpha m}{k}$

then a $(k, 2\alpha, \mathcal{G})$ -metric-decomposition of G can be computed in $O(k \cdot f(m))$ time.

Let $G = (V, E, l)$ be a graph. Let $\{H_i\}_i$ be the set of graphs in \mathcal{G} such that $G \leq^1 H_i \leq^1 G$. Introduce a coefficient λ_i for each H_i . These λ_i 's are initially zero and in the end only a small number of them will become nonzero. For a given graph $H_i = (V, E_i, l_i)$ that t_i -routes G , recall the definition of $\eta_{H_i}(e) = l_{H_i}(e)/l(e), \forall e \in G$, stretch of e routed by H_i .

Following approach from [Mad10] and [R08], let M be an $|E| \times N$ matrix, N being the cardinality of $\{H_i\}_i$, with $M_{e,i} = \eta_{H_i}(e)$. Let $\lambda = (\lambda_1, \dots, \lambda_N)$ be a vector corresponding to a convex combination of $\{H_i\}_i$. If $\max M\lambda \leq \alpha$ for some $\alpha > 0$, we have

$$\begin{aligned} \forall e \in E, \sum_i \lambda_i \eta_{H_i}(e) &\leq \alpha, \text{ and thus} \\ \forall e \in E, \sum_i \lambda_i l_{H_i}(e) &\leq \alpha l(e) \end{aligned}$$

Consider the following constrains:

$$\begin{aligned} \text{lmax}(M\lambda) &\leq 3\alpha \\ \sum_i \lambda_i &= 1 \\ \lambda_i &\geq 0, \forall i \end{aligned}$$

where $\text{lmax}(\mathbf{x}) = \ln \sum_{e \in E} \exp(x_e) \geq \max_{e \in E} x_e$. Any valid λ would corresponds to a $(3\alpha, \mathcal{G})$ -distance-decomposition of G for $\alpha \geq \ln m$.

Here we present some known fact about lmax function:

Fact B.1.

$$\forall \mathbf{x} \geq 0, \max_{e \in E} x_e \leq \text{lmax}(\mathbf{x}) \leq \max_{e \in E} x_e + \ln m$$

Fact B.2. Define

$$\text{partial}_e(\mathbf{x}) := \frac{\partial \text{lmax}(\mathbf{x})}{\partial x_e} = \frac{\exp(x_e)}{\sum_f \exp(x_f)}.$$

We have

$$\forall \mathbf{x}, \epsilon \geq 0, \epsilon \leq 1, \text{lmax}(\mathbf{x} + \epsilon) \leq \text{lmax}(\mathbf{x}) + 2 \sum_e \epsilon_e \text{partial}_e(\mathbf{x})$$

Fact B.3. Define

$$\text{partial}_i(\lambda) := \frac{\partial \text{lmax}(M\lambda)}{\partial \lambda_i} = \sum_e \eta_{H_i}(e) \cdot \text{partial}_e(M\lambda) = \sum_e \frac{l_{H_i}(e)}{l(e)} \cdot \text{partial}_e(M\lambda).$$

Let $\mathbf{1}_i$ be the all zero vector except i -th coordinate being 1. Recall $\eta(H_i)$ being the maximum stretch when H_i routes G . For any $0 \leq \delta_i \leq 1/\eta(H_i)$, we have

$$\text{lmax}(M(\lambda + \delta_i \mathbf{1}_i)) \leq \text{lmax}(M\lambda) + 2\delta_i \text{partial}_i(\lambda)$$

Proof of Theorem 7.10. The vector λ is found as follows. Starting with $\lambda = 0$. As long as $\sum_i \lambda_i < 1$, we define edge weights \mathbf{w} with $w(e) = \text{partial}_e(M\lambda)/l(e)$. By condition , we can compute $H_{i(\mathbf{w})}$ such that

$$\mathbf{w}(H_{i(\mathbf{w})}) = \sum_e l_{H_{i(\mathbf{w})}}(e) \frac{\text{partial}_e(M\lambda)}{l(e)} \leq \alpha \sum_e l(e) \frac{\text{partial}_e(M\lambda)}{l(e)} = \alpha \mathbf{w}(G).$$

Next we increase $\lambda_{i(\mathbf{w})}$ by $\min\{1/\eta(H_{i(\mathbf{w})}), 1 - \sum_i \lambda_i\}$.

Let λ be the resulting solution. We have to make sure $\text{lmax}(M\lambda) \leq 3\alpha$. First, we observe that

$$\mathbf{w}(H_{i(\mathbf{w})}) = \sum_e l_{H_{i(\mathbf{w})}}(e) \frac{\text{partial}_e(M\lambda)}{l(e)} = \text{partial}_i(\lambda),$$

and

$$\mathbf{w}(G) = \sum_e l(e) \frac{\text{partial}_e(M\lambda)}{l(e)} = \sum_e \text{partial}_e(M\lambda) = 1.$$

Therefore, we have $\text{partial}_i(\lambda) = \mathbf{w}(H_{i(\mathbf{w})}) \leq \alpha \mathbf{w}(G) = \alpha$.

By Fact B.3 and $\lambda_{i(\mathbf{w})} \leq 1/\eta(H_{i(\mathbf{w})})$, at each iteration, we have

$$\begin{aligned} \text{lmax}(M\lambda) &\leq \text{lmax}(M\mathbf{0}) + 2 \sum_i \lambda_i \mathbf{w}(H_{i(\mathbf{w})}) \\ &\leq \text{lmax}(M\mathbf{0}) + 2 \sum_i \lambda_i \alpha \\ &\leq \ln m + 2\alpha \leq 3\alpha \end{aligned}$$

Next, we have to show an upper-bound on the number of iterations. Define the potential function $\Phi(\lambda) := \sum_e \sum_i \lambda_i \eta_{H_i}(e)$. Initially, $\Phi(\lambda) = \Phi(\mathbf{0}) = 0$. The potential is only increasing throughout the algorithm. At the end, we have $\Phi(\lambda) := \sum_e \sum_i \lambda_i \eta_{H_i}(e) \leq 3\alpha m$ since $\sum_i \lambda_i \eta_{H_i}(e) \leq 3\alpha, \forall e$. Observe every time we update λ , $\Phi(\lambda)$ increases by $|\psi(H_{i(\mathbf{w})})|/2 \geq 2\alpha m/k$, since for every $e \in \psi(H_{i(\mathbf{w})})$,

$$\lambda_{i(\mathbf{w})} \eta_{H_{i(\mathbf{w})}}(e) \geq \frac{\eta_{H_{i(\mathbf{w})}}(e)}{\eta(H_{i(\mathbf{w})})} \geq \frac{1}{2}$$

Therefore, by Condition 3 on lower-bounding the size of $|\psi(H_{i(\mathbf{w})})|$, we have at most $1.5k$ iterations and the theorem follows. \square

References

- [ABN08] I. Abraham, Y. Bartal, and O. Neiman. Nearly tight low stretch spanning trees. In *2008 49th Annual IEEE Symposium on Foundations of Computer Science*, pages 781–790, Oct 2008. [37](#), [46](#), [54](#)
- [ACK17] Ittai Abraham, Shiri Chechik, and Sebastian Krininger. Fully dynamic all-pairs shortest paths with worst-case update-time revisited. In *SODA*, pages 440–452. SIAM, 2017. [2](#)
- [ACT14] Ittai Abraham, Shiri Chechik, and Kunal Talwar. Fully dynamic all-pairs shortest paths: Breaking the $o(n)$ barrier. In *APPROX-RANDOM*, volume 28 of *LIPICs*, pages 1–16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2014. [2](#), [12](#)

- [AD16] Amir Abboud and Søren Dahlgaard. Popular conjectures as a barrier for dynamic planar graph algorithms. In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9–11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*, pages 477–486, 2016. [2](#), [3](#), [25](#)
- [ADK⁺16] Ittai Abraham, David Durfee, Ioannis Koutis, Sebastian Krinninger, and Richard Peng. On fully dynamic graph sparsifiers. In *FOCS*, pages 335–344. IEEE Computer Society, 2016. [3](#), [24](#), [34](#), [43](#), [52](#)
- [AG08] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1–39, 2008. [2](#)
- [AW14] Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18–21, 2014*, pages 434–443, 2014. [2](#), [3](#), [24](#)
- [BC16] Aaron Bernstein and Shiri Chechik. Deterministic decremental single source shortest paths: beyond the $o(mn)$ bound. In *STOC*, pages 389–397. ACM, 2016. [4](#)
- [BC17] Aaron Bernstein and Shiri Chechik. Deterministic partially dynamic single source shortest paths for sparse graphs. In *SODA*, pages 453–469. SIAM, 2017. [4](#)
- [Ber09] Aaron Bernstein. Fully dynamic $(2 + \epsilon)$ approximate all-pairs shortest paths with fast query and close to linear update time. In *FOCS*, pages 693–702. IEEE Computer Society, 2009. [2](#)
- [Ber16] Aaron Bernstein. Maintaining shortest paths under deletions in weighted directed graphs. *SIAM J. Comput.*, 45(2):548–574, 2016. [2](#)
- [BKN19] Karl Bringmann, Marvin Künnemann, and André Nusser. Fréchet distance under translation: Conditional hardness and an algorithm via offline dynamic grid reachability. In *Symposium on Discrete Algorithms (SODA)*, pages 2902–2921, 2019. [3](#)
- [BKW06] Karsten M Borgwardt, Hans-Peter Kriegel, and Peter Wackersreuther. Pattern mining in frequent dynamic subgraphs. In *Sixth International Conference on Data Mining (ICDM’06)*, pages 818–822. IEEE, 2006. [2](#)
- [BR11] Aaron Bernstein and Liam Roditty. Improved dynamic algorithms for maintaining approximate shortest paths under deletions. In *SODA*, pages 1355–1365. SIAM, 2011. [4](#)
- [BS15] Aaron Bernstein and Cliff Stein. Fully dynamic matching in bipartite graphs. In *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6–10, 2015, Proceedings, Part I*, pages 167–179, 2015. [4](#)
- [BS16] Aaron Bernstein and Cliff Stein. Faster fully dynamic matchings with small approximation ratios. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10–12, 2016*, pages 692–711, 2016. [4](#)
- [BvdBG⁺20] Aaron Bernstein, Jan van den Brand, Maximilian Probst Gutenberg, Danupon Nanongkai, Thatchaphol Saranurak, Aaron Sidford, and He Sun. Fully-dynamic graph sparsifiers against an adaptive adversary. *CoRR*, abs/2004.08432, 2020. [43](#)

- [BVZ01] Yuri Boykov, Olga Veksler, and Ramin Zabih. Fast approximate energy minimization via graph cuts. *IEEE Transactions on pattern analysis and machine intelligence*, 23(11):1222–1239, 2001. [2](#)
- [CDLV19] Parinya Chalermsook, Syamantak Das, Bundit Laekhanukit, and Daniel Vaz. Mimicking networks parameterized by connectivity. *CoRR*, abs/1910.10665, 2019. [4](#)
- [CGH16] Yun Kuen Cheung, Gramoz Goranci, and Monika Henzinger. Graph minors for preserving terminal distances approximately - Lower and Upper Bounds. In *International Colloquium on Automata Languages and Programming (ICALP)*, pages 131:1–131:14, 2016. [25](#)
- [Che18] Shiri Chechik. Near-optimal approximate decremental all pairs shortest paths. In *FOCS*, 2018. [2](#)
- [Chu12] Julia Chuzhoy. On vertex sparsifiers with steiner nodes. In *Symposium on Theory of Computing (STOC)*, pages 673–688, 2012. [4](#)
- [CKL13] Ho Yee Cheung, Tsz Chiu Kwok, and Lap Chi Lau. Fast matrix rank algorithms and applications. *J. ACM*, 60(5):31:1–31:25, 2013. [3](#)
- [CKM⁺14] Michael B. Cohen, Rasmus Kyng, Gary L. Miller, Jakub W. Pachocki, Richard Peng, Anup B. Rao, and Shen Chen Xu. Solving sdd linear systems in nearly $m \log^{1/2} n$ time. In *Proceedings of the Forty-Sixth Annual ACM Symposium on Theory of Computing, STOC '14*, page 343–352, New York, NY, USA, 2014. Association for Computing Machinery. [51](#)
- [CLLM10] Moses Charikar, Tom Leighton, Shi Li, and Ankur Moitra. Vertex sparsifiers and abstract rounding algorithms. In *Symposium on Foundations of Computer Science (FOCS)*, pages 265–274, 2010. [4](#)
- [Dah16] Søren Dahlgaard. On the hardness of partially dynamic graph problems and connections to diameter. In *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*, pages 48:1–48:14, 2016. [2](#), [3](#), [24](#)
- [DGGP19] David Durfee, Yu Gao, Gramoz Goranci, and Richard Peng. Fully dynamic spectral vertex sparsifiers and applications. In *Symposium on Theory of Computing (STOC)*, pages 914–925, 2019. [2](#), [6](#), [50](#), [52](#), [53](#)
- [DPPR17] D. Durfee, J. Peebles, R. Peng, and A. B. Rao. Determinant-preserving sparsification of sddm matrices with applications to counting and sampling spanning trees. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 926–937, Oct 2017. [52](#)
- [DS84] Peter G. Doyle and J. Laurie Snell. *Random Walks and Electric Networks*, volume 22 of *Carus Mathematical Monographs*. Mathematical Association of America, 1984. [2](#)
- [EGIN97] David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. Sparsification - a technique for speeding up dynamic graph algorithms. *J. ACM*, 44(5):669–696, 1997. [4](#)
- [EGIS96] David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Thomas H. Spencer. Separator based sparsification. i. planary testing and minimum spanning trees. *J. Comput. Syst. Sci.*, 52(1):3–27, 1996. [4](#)
- [EGIS98] David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Thomas H. Spencer. Separator-based sparsification II: edge and vertex connectivity. *SIAM J. Comput.*, 28(1):341–381, 1998. [4](#)

- [EGK⁺14] Matthias Englert, Anupam Gupta, Robert Krauthgamer, Harald Räcke, Inbal Talgam-Cohen, and Kunal Talwar. Vertex sparsifiers: New results from old techniques. *SIAM J. Comput.*, 43(4):1239–1262, 2014. Announced at APPROX’10. 4
- [Epp91] David Eppstein. Offline algorithms for dynamic minimum spanning tree problems. In *WADS*, volume 519 of *Lecture Notes in Computer Science*, pages 392–399. Springer, 1991. 4
- [FG19] Sebastian Forster and Gramoz 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, page 377–388, New York, NY, USA, 2019. Association for Computing Machinery. 44, 49
- [FR01] Jittat Fakcharoenphol and Satish Rao. Planar graphs, negative weight edges, shortest paths, near linear time. In *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*, pages 232–241, 2001. 4
- [GHP17] Gramoz Goranci, Monika Henzinger, and Pan Peng. The power of vertex sparsifiers in dynamic graph algorithms. In *ESA*, volume 87 of *LIPICs*, pages 45:1–45:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. 4
- [GHP18] Gramoz Goranci, Monika Henzinger, and Pan Peng. Dynamic effective resistances and approximate schur complement on separable graphs. In *ESA*, volume 112 of *LIPICs*, pages 40:1–40:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018. 4
- [GHS18] Gramoz Goranci, Monika Henzinger, and Thatchaphol Saranurak. Fast incremental algorithms via local sparsifiers. 2018. 34
- [GHT18] Gramoz Goranci, Monika Henzinger, and Mikkel Thorup. Incremental exact min-cut in poly-logarithmic amortized update time. *ACM Trans. Algorithms*, 14(2):17:1–17:21, 2018. 4
- [GK18] Manoj Gupta and Shahbaz Khan. Simple dynamic algorithms for maximal independent set and other problems. *CoRR*, abs/1804.01823, 2018. 3
- [GKK⁺18] Mohsen Ghaffari, Andreas Karrenbauer, Fabian Kuhn, Christoph Lenzen, and Boaz Patt-Shamir. Near-optimal distributed maximum flow. *SIAM J. Comput.*, 47(6):2078–2117, 2018. 37, 38
- [Gor19] Gramoz Goranci. *Dynamic Graph Algorithms and Graph Sparsification: New Techniques and Connections*. PhD thesis, Universitat Wien, 2019. Available at: <http://arxiv.org/abs/1909.06413>. 2, 34
- [GRST20] Gramoz Goranci, Harald Räcke, Thatchaphol Saranurak, and Zihan Tan. The expander hierarchy and its applications to dynamic graph algorithms, 2020. 4
- [GT14] Andrew V. Goldberg and Robert Endre Tarjan. Efficient maximum flow algorithms. *Commun. ACM*, 57(8):82–89, 2014. 2
- [HHS20] Kathrin Hanauer, Monika Henzinger, and Christian Schulz. Faster fully dynamic transitive closure in practice. *CoRR*, abs/2002.00813, 2020. 2
- [HKNS15] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *STOC*, pages 21–30. ACM, 2015. 2, 24, 25

- [HPK11] Jiawei Han, Jian Pei, and Micheline Kamber. *Data mining: concepts and techniques*. Elsevier, 2011. [2](#)
- [JS20] Wenyu Jin and Xiaorui Sun. Fully dynamic c -edge connectivity in subpolynomial time. *CoRR*, abs/2004.07650, 2020. [4](#)
- [KNZ14] Robert Krauthgamer, Huy L. Nguyen, and Tamar Zondiner. Preserving terminal distances using minors. *SIAM J. Discrete Math.*, 28(1):127–141, 2014. [25](#)
- [KPSW19] Rasmus Kyng, Richard Peng, Sushant Sachdeva, and Di Wang. Flows in almost linear time via adaptive preconditioning. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2019, New York, NY, USA, 2019. Association for Computing Machinery. [36](#)
- [KR17] Robert Krauthgamer and Inbal Rika. Refined vertex sparsifiers of planar graphs. *CoRR*, abs/1702.05951, 2017. [24](#)
- [LM10] Frank Thomson Leighton and Ankur Moitra. Extensions and limits to vertex sparsification. In *Symposium on Theory of Computing (STOC)*, pages 47–56, 2010. [4](#)
- [LOP⁺15] Jakub Lacki, Jakub Ocwieja, Marcin Pilipczuk, Piotr Sankowski, and Anna Zych. The power of dynamic distance oracles: Efficient dynamic algorithms for the steiner tree. In *Symposium on Theory of Computing (STOC)*, pages 11–20, 2015. [12](#)
- [LPS19] Yang P. Liu, Richard Peng, and Mark Sellke. Vertex sparsifiers for c -edge connectivity. *CoRR*, abs/1910.10359, 2019. [4](#)
- [LPYZ18a] Huan Li, Stacy Patterson, Yuhao Yi, and Zhongzhi Zhang. Maximizing the number of spanning trees in a connected graph. *CoRR*, abs/1804.02785, 2018. [3](#)
- [LPYZ18b] Huan Li, Stacy Patterson, Yuhao Yi, and Zhongzhi Zhang. Maximizing the number of spanning trees in a connected graph. *CoRR*, abs/1804.02785, 2018. [4](#)
- [LS11] Jakub Lacki and Piotr Sankowski. Min-cuts and shortest cycles in planar graphs in $o(n \log \log n)$ time. In Camil Demetrescu and Magnús M. Halldórsson, editors, *Proc. of the Annual European Symposium (ESA)*, volume 6942, pages 155–166, 2011. [4](#)
- [Mad10] Aleksander Madry. Fast approximation algorithms for cut-based problems in undirected graphs. In *FOCS*, pages 245–254. IEEE Computer Society, 2010. [6](#), [31](#), [32](#), [33](#), [37](#), [45](#), [46](#), [56](#)
- [MM10] Konstantin Makarychev and Yury Makarychev. Metric extension operators, vertex sparsifiers and lipschitz extendability. In *Symposium on Foundations of Computer Science (FOCS)*, pages 255–264, 2010. [4](#), [24](#)
- [Moi09] Ankur Moitra. Approximation algorithms for multicommodity-type problems with guarantees independent of the graph size. In *FOCS*, pages 3–12. IEEE Computer Society, 2009. [4](#)
- [NS17] Danupon Nanongkai and Thatchaphol Saranurak. Dynamic spanning forest with worst-case update time: adaptive, las vegas, and $o(n^{1/2 - \epsilon})$ -time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2017, Montreal, QC, Canada, June 19-23, 2017, pages 1122–1129, 2017. [2](#), [4](#)

- [NSW17] Danupon Nanongkai, Thatchaphol Saranurak, and Christian Wulff-Nilsen. Dynamic minimum spanning forest with subpolynomial worst-case update time. In *FOCS*, pages 950–961. IEEE Computer Society, 2017. [2](#), [4](#), [43](#)
- [PBL17] Ashwin Paranjape, Austin R Benson, and Jure Leskovec. Motifs in temporal networks. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, pages 601–610, 2017. [2](#)
- [Pen16] Richard Peng. Approximate undirected maximum flows in $O(m\text{polylog}(n))$ time. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 1862–1867, 2016. [3](#), [22](#), [23](#), [35](#)
- [PS14] Richard Peng and Daniel A. Spielman. An efficient parallel solver for SDD linear systems. In David B. Shmoys, editor, *Symposium on Theory of Computing (STOC)*, pages 333–342. ACM, 2014. [43](#)
- [PSS19] Richard Peng, Bryce Sandlund, and Daniel Dominic Sleator. Optimal offline dynamic 2, 3-edge/vertex connectivity. In *Algorithms and Data Structures Symposium (WADS)*, pages 553–565, 2019. [4](#), [16](#)
- [R08] Harald Räcke. Optimal hierarchical decompositions for congestion minimization in networks. In *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing, STOC '08*, page 255–264, New York, NY, USA, 2008. Association for Computing Machinery. [3](#), [6](#), [56](#)
- [RST14] Harald Räcke, Chintan Shah, and Hanjo Täubig. Computing cut-based hierarchical decompositions in almost linear time. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, pages 227–238, 2014. [22](#)
- [RTZ05] Liam Roditty, Mikkel Thorup, and Uri Zwick. Deterministic constructions of approximate distance oracles and spanners. In *Automata, Languages and Programming, 32nd International Colloquium, ICALP 2005, Lisbon, Portugal, July 11-15, 2005, Proceedings*, pages 261–272, 2005. [12](#), [13](#)
- [RWE13] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph databases*. " O'Reilly Media, Inc.", 2013. [2](#)
- [RZ08] Liam Roditty and Uri Zwick. Improved dynamic reachability algorithms for directed graphs. *SIAM Journal on Computing*, 37(5):1455–1471, 2008. [2](#)
- [RZ12] Liam Roditty and Uri Zwick. Dynamic approximate all-pairs shortest paths in undirected graphs. *SIAM J. Comput.*, 41(3):670–683, 2012. [2](#)
- [San04] Piotr Sankowski. Dynamic transitive closure via dynamic matrix inverse (extended abstract). In *45th Symposium on Foundations of Computer Science (FOCS 2004), 17-19 October 2004, Rome, Italy, Proceedings*, pages 509–517, 2004. [2](#)
- [San05] Piotr Sankowski. Subquadratic algorithm for dynamic shortest distances. In *COCOON*, volume 3595 of *Lecture Notes in Computer Science*, pages 461–470. Springer, 2005. [2](#)
- [She09] Jonah Sherman. Breaking the multicommodity flow barrier for $o(\text{vlog } n)$ -approximations to sparsest cut. In *FOCS*, pages 363–372. IEEE Computer Society, 2009. [3](#)

- [She17] Jonah Sherman. Area-convexity, l_∞ regularization, and undirected multicommodity flow. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 452–460, 2017. 3
- [Sol18] Shay Solomon. Local algorithms for bounded degree sparsifiers in sparse graphs. In *9th Innovations in Theoretical Computer Science Conference, ITCS 2018, January 11-14, 2018, Cambridge, MA, USA*, pages 52:1–52:19, 2018. 4
- [SS11] D. Spielman and N. Srivastava. Graph sparsification by effective resistances. *SIAM Journal on Computing*, 40(6):1913–1926, 2011. Available at <http://arxiv.org/abs/0803.0929>. 3
- [ST11] Daniel A. Spielman and Shang-Hua Teng. Spectral sparsification of graphs. *SIAM J. Comput.*, 40(4):981–1025, 2011. 43
- [SW19] Thatchaphol Saranurak and Di Wang. Expander decomposition and pruning: Faster, stronger, and simpler. In Timothy M. Chan, editor, *Symposium on Discrete Algorithms (SODA)*, pages 2616–2635. SIAM, 2019. 43
- [Tho05] Mikkel Thorup. Worst-case update times for fully-dynamic all-pairs shortest paths. In *STOC*, pages 112–119. ACM, 2005. 2
- [Tho07] Mikkel Thorup. Fully-dynamic min-cut. *Combinatorica*, 27(1):91–127, 2007. 4
- [TZ05] Mikkel Thorup and Uri Zwick. Approximate distance oracles. *J. ACM*, 52(1):1–24, 2005. 3, 6, 12, 15, 16
- [vdBNS19] Jan van den Brand, Danupon Nanongkai, and Thatchaphol Saranurak. Dynamic matrix inverse: Improved algorithms and matching conditional lower bounds. In *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 456–480, 2019. Available at: <https://arxiv.org/abs/1905.05067>. 2
- [Wul17] Christian Wulff-Nilsen. Fully-dynamic minimum spanning forest with improved worst-case update time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 1130–1143, 2017. 2, 4
- [Zhu05] Xiaojin Jerry Zhu. Semi-supervised learning literature survey. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 2005. 2