Fast Dynamic Cuts, Distances and Effective Resistances via Vertex Sparsifiers

Li Chen, *Georgia Tech*
Gramoz Goranci, *University of Toronto*
Monika Henzinger, *University of Vienna*
Richard Peng, *Georgia Tech*
Thatchaphol Saranurak, *Toyota Technological Insititute*

***Abstract*—We present a general framework of designing efficient dynamic approximate algorithms for optimization problems 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 $\tilde{O}(n^{2/3})$[1] 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.

***Keywords*-dynamic graph algorithms; maximum flow; minimum cut; shortest paths; effective resistances; approximate**

See https://arxiv.org/pdf/2005.02368.pdf for the full version of this paper.

I. 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 [1], [2], and databases [3], [4]. 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 [5]–[7]. The directed version, fully dynamic transitive closure, has seen even less progress [8]–[10], and is one of the best reflections of the difficulties of designing dynamic graph algorithms, especially in practice [11].

Over the past decade, dynamic graph algorithms and their lower bounds have been studied extensively. These results led to a significantly improved understanding 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 [12]–[15]. 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 [16]–[23], 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 [24]–[26]. For example, the st-maximum flow problem asks for the maximum number of edge-disjoint paths between a pair of vertices [27], while electrical flow minimizes a congestion measure related to the sums of squares of the flow values along edges [28]. 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 [29]. Such tools were recently used to give the first sublinear time data structures for maintaining $(1+\epsilon)$-approximate electrical flows and effective resistances, which raised the optimistic possibility that all flow-related problems can be maintained with $(1+\epsilon)$-approximation factors in subpolynomial time [30].

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, distances and effective resistance, with a focus on obtaining the best approximation ratios possible, but with

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

2575-8454/20/$31.00 ©2020 IEEE
DOI 10.1109/FOCS46700.2020.00109

sub-linear time per update/query.

Theorem I.1. *Given an undirected, weighted graph $G = (V, E)$, there is a fully dynamic randomized algorithm that maintains for every pair of nodes u and v, an estimate $\delta(u, v)$ that approximates the maximum-flow/minimum-cut between u and v in G up to a factor of $\tilde{O}(\log n)$ while supporting edge insertions/deletions of edges and queries in $\tilde{O}(n^{2/3})$ amortized time against an oblivious adversary, and $\tilde{O}(m^{3/4})$ amortized time against an adaptive adversary.*

This result constitutes the first non-trivial algorithm for the dynamic all-pair maximum flow problem in general graphs. We obtain a similar result for fully-dynamic approximate distance oracles.

Theorem I.2. *Given an undirected, weighted graph $G = (V, E)$, there is a fully dynamic randomized algorithm that maintains for every pair of nodes u and v, an estimate $\delta(u, v)$ that approximates the shortest path between u and v in G up to a factor of $\tilde{O}(\log n)$ while supporting insertions/deletions of edges and queries in $\tilde{O}(n^{2/3+o(1)})$ amortized time against an oblivious adversary.*

Our algorithm extends to the closely-related minimum transshipment problem, and we defer details about this extension to the full version of the paper.

If we restrict to the insertions-only setting, we obtain a *deterministic* algorithm and improve both the approximation ratio and the running time.

Theorem I.3. *Given an undirected, weighted graph $G = (V, E)$ and any two integer parameters $t, r \geq 1$, there is an incremental deterministic algorithm that maintains for every pair of nodes u and v, an estimate $\delta(u, v)$ that approximates the shortest path between u and v in G up to a factor of $(2r-1)^t$ while supporting insertions of edges and queries in $O(m^{1/(t+1)} n^{t/r} \log^{2t+2} n)$ worst-case time.*

Specifically, when setting $r := t^2$ to be some large constant, we get an approximation ratio of $(2t)^{2t}$ and roughly $O(m^{\frac{1}{t^2}})$ time per operation.

In the last application of our online, dynamic framework, we achieve a polynomial speed-up over the state-of-the-art algorithm [30] for the dynamic all-pair effective resistances problem.

Theorem I.4. *Given an undirected, weighted graph $G = (V, E)$ and and error parameter $\epsilon > 0$, there is an fully dynamic randomized algorithm that maintains for every pair of nodes u and v, an estimate $\delta(u, v)$ that approximates the effective resistance between u and v in G up to a factor of $(1+\epsilon)$ while supporting insertions/deletions of edges and queries in $\tilde{O}(n^{2/3+o(1)} \epsilon^{-O(1)})$ amortized time per operation.*

In each of the above cases, our obtained approximation ratios match up to constants the current best known approximation ratios of oracles versions of these problems on static graphs, namely tree flow sparsifiers/oblivious routings [31], distance oracles [32], and static computations of effective resistances [33]. 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 the one in Theorem I.3 are randomized, and their guarantees are only provable against an oblivious adversary (except the second algorithm in Theorem I.1 and the one in Theorem I.3), 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 are given in advance. In other words, the algorithm needs to output information about the graphs at various points in this given update sequence. 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.

Theorem I.5. *Given an undirected, weighted graph $G = (V, E)$ and any parameter $t \geq 1$, there is an* offline *fully dynamic algorithm that maintains for every pair of nodes u and v, an estimate $\delta(u, v)$ that approximates the maximum-flow/minimum-cut between u and v in G up to a factor of $O(\log^{4t} n)$ and supports any sequence of m operations in $\tilde{O}(m \cdot m^{1/(t+1)})$ total time.*

Theorem I.6. *Given an undirected, weighted graph $G = (V, E)$ and any two parameters $r, t \geq 1$, there is an* offline *fully dynamic algorithm that maintains for every pair of nodes u and v, an estimate $\delta(u, v)$ that approximates the shortest path between u and v in G up to a factor of $(2r-1)^t$ and supports any sequence of m operations in $\tilde{O}(m \cdot m^{1/(t+1)} n^{2/r})$ total time.*

Although the offline setting is weaker than the standard dynamic setting, it is interesting for two reasons. First, offline algorithms are used to obtain fast static algorithms (e.g. [34], [35]). Second, many conditional lower bounds (e.g. [12], [14], [15]) 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 specific graph properties) the sequence of updates is also known in advance.

A. Related work

Previous work on Dynamic Flows/Cuts and Shortest Paths. Despite the fact that all pair max-flow/min-cut is one of the cornerstone problems in 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. Concretely, one can maintain a dynamic cut-sparsifier (against an oblivious adversary) of size $\tilde{O}(n)$ due to [36] with $\tilde{O}(1)$ update time, and whenever a query is asked, we execute the fastest static approximation algorithms on the sparsifier in $\tilde{O}(n)$ time (for example, [37] for $(1+\epsilon)$-approximate max flow, [38] for $(1+\epsilon)$-approximate multi-commodity concurrent flow, and [39] for $O(\sqrt{\log n})$-approximate sparsest cuts). To the best of our knowledge, there is no previous algorithm that achieves $o(n)$ update and query time, even when restricting to amortized guarantees on the running times.

Perhaps the closest work to this paper is the dynamic algorithm due to Cheung at al. [40] for explicitly maintaining the values of all-pairs min-cuts in $\tilde{O}(m^2)$ update time. For the s-t max flow problem, where s and t are predetermined, there is an incremental algorithm with $O(n)$ amortized update time [41]. If we restrict to bipartite graphs with a certain specific structure, there is a $(1+\epsilon)$-approximation fully dynamic algorithm [36] with polylogarithmic worst-case update time. From the lower bound perspective, Dahlgaard [15] 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 sub-linear 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 [42], 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 [43] showed an exact fully-dynamic algorithm with $\tilde{O}(n^{5/6})$ update and query time. Recently, Goranci, Henzinger, and Thorup [44] designed an exact incremental algorithm with $O(\log^3 n \log\log^2 n)$ update time and $O(1)$ query time. The update time has been further improved to $O(\log n)$ by Ghaffari et al. [45].

The dynamic shortest-path problem is one of the central problems in dynamic graph algorithms and has been extensively studied in the literature. For unweighted, undirected graphs, Abraham, Chechik and Talwar [20] devised a dynamic algorithm using $\tilde{O}(\sqrt{m}n^{1/k})$ expected amortized update time, $O(k^2\rho^2)$ query time while approximating pairwise distances up to a factor of $2^{O(k\rho)}$, where $k \geq 2$ is an integer parameter and $\rho = 1 + \lceil \frac{\log n^{1-1/k}}{\log(m/n^{1-1/k})} \rceil$. In comparison, our algorithm from Theorem I.2 applies to weighted graphs but achieves worse approximation and running time guarantees.

Previous Work on 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 graph 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. [46]. This work has proven useful for several fundamental problems including dynamic minimum spanning forest and different variants of edge/vertex connectivity (e.g. [5]–[7], [42]). Edge sparsification has also been a key technique in dynamic shortest paths problems (e.g. [47] maintains distances on top of spanners, [48], [49] replace "dense parts" of graphs with sparser graphs). Recently, there are works that study edge sparsification for matching problems [50]–[52]. In fact, the core component of several dynamic matching algorithms is only to maintain such sparsifiers [50], [51].

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 [53]–[55] 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 [56], [57] but none of these works extend to general graphs. Several previous *offline* dynamic algorithms exploit vertex sparsification for maintaining minimum spanning forests [58], small edge/vertex connectivity [59], and effective resistance [60].

Recent Work on Dynamic Vertex Sparsification. Very recently, Goranci et al. [61] designed a fully dynamic algorithm for maintaining a tree flow sparsifier based on a new notion of expander decompositions. One of their applications is a fully dynamic algorithm for all-pair maximum-flows/minimum-cuts. Their algorithm is deterministic, has $n^{o(1)}$ worst-case update time and $O(\log^{1/6} n)$ query time, but can only guarantee an approximation ratio of $2^{O(\log^{5/6} n)} = n^{o(1)}$. Our algorithms from Theorems I.1 guarantees a much better approximation ratio of $\tilde{O}(\log n)$. However, our update and query times are slower and are randomized.

Concurrent to our work there have also been several recent developments on utilizing vertex sparsifiers to maintain c-edge connectivity for small values of c [59], [62]–[64].

II. Technical Overview

A. Incremental Vertex Sparsifiers

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* [65]–[70], 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 its generalization and 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[2]. For example, $\mathcal{P}(u, v, G)$ can be thought of as the distance between u and v in G. Let $T \subseteq V$ be a set of nodes in G called *terminals*. Given a parameter $\alpha \geq 1$, an *α-vertex sparsifier* of G w.r.t. T is a graph $H_T = (V', E')$ such that 1) $V' \supseteq T, |V'| \approx |T|$ and 2) $\mathcal{P}(u, v, H_T) \approx_\alpha \mathcal{P}(u, v, G)$ for all $u, v \in T$. That is, the number of vertices in H_T is close to the size of T and H_T approximately preserves the property $\mathcal{P}$ between pair-wise terminal vertices 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:

- PREPROCESS(G, α): preprocess the graph G,
- 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.

We now show that such an efficient sparsifier almost readily leads to a simple two-level incremental algorithm for maintaining the pair-wise property $\mathcal{P}$. Specifically, given an initial graph $G = (V, E)$ and an approximation parameter $\alpha \geq 1$, assume we want to design an (approximate) 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 PREPROCESS(G, α) 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. Next, we describe the implementation of insertions and queries. Upon insertion of a new edge $e = (u, v)$ in G, we invoke ADDTERMINAL(u) and ADDTERMINAL(v), and add e to H_T. For answering (s, t) queries, we invoke ADDTERMINAL(s) and 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.

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

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 $O(|E|f(n)/\beta) = O(mf(n)/\beta)$ and (2) the cost for adding the endpoints of β edges as terminals, which is $O(\beta g(n))$. By construction, $|T| = O(\beta)$ at any time, resulting in H_T being of size $O(\beta g(n))$ (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 trade-off

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

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

One challenge we face to extend the above approach to a *multi-level* incremental algorithm is the large flexibility allowed in the ADDTERMINAL operation. More concretely, given a sparsifier H_T w.r.t. T, whenever a new terminal is added to T and H_T is updated to $H_{T'}$, the implementation of this operation could potentially lead to vertices and/or edges being deleted from or inserted 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. We achieve this by modifying the operation of adding terminals in the definition of α-IVS as follows:

- ADDTERMINAL(u): let T' be $T \cup \{u\}$ and update H_T to $H_{T'}$ such that
  - $H_{T'}$ 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 can be thought of as the number of changes needed to construct the new sparsifier $H_{T'}$ from the old sparsifier H_T, i.e., $|H_{T'} \setminus H_T|$. This naturally leads to extending the definition of efficient $(\alpha, f(n), g(n))$-IVS by incorporating the function $r(n)$ such that $|H_{T'} \setminus H_T| \leq r(n)$. While it is straightforward to bound the recourse by the time needed to support the addition of terminals, i.e., $r(n) \leq g(n)$, 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 below.

Theorem II.1. *Let $G = (V, E)$ be a graph, and for any $u, v \in V$, let $\mathcal{P}(u, v, G)$ be a graph property between u and v in G. Let $f(n), g(n), r(n), h(n) \geq 1$ be functions, $\alpha \geq 1$ be an approximation parameter, $t \geq 1$ be the depth of the data structure, and let $\gamma, \mu_0, \mu_1, \ldots, \mu_t$ 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 $\delta(u, v)$ that approximates $\mathcal{P}(u, v, G)$ up to a factor of α^t, with worst-case update time of

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

and worst-case query time of

$$O\left(tu(n) + \mu_t r(\mu_t)h(\mu_t)\right),$$

where $c < 3$ is an universal constant.

We demonstrate the applicability of the above theorem to the incremental all-pair shortest path problem (see Theorem I.3) by showing that an efficient IVS can be constructed using a deterministic variant of the distance oracle due to Thorup and Zwick [32]. At a high level, the oracle preprocessing works as follows: (i) it constructs a hierarchy of carefully chosen centers and (ii) for each vertex $u \in V$, it finds the closest center to u at every level of the hierarchy and defines the *bunch* $B(u)$ to be the union over these centers. 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 implies an efficient $(2r-1, \tilde{O}(n^{1/r}), \tilde{O}(n^{1/r}), \tilde{O}(n^{1/r}))$-IVS for G, where $r \geq 1$. The correctness of our data-structure crucially exploits the fact that $\cup_{u\in T} B(u)$ is an α-vertex (distance) sparsifier of G w.r.t. T.

B. Fully Dynamic Vertex Sparsifiers

To extend our algorithmic framework to fully dynamic graphs, we start by modifying the definition of IVS as follows: i) invoke the old AddTerminal operation, where we did not require that $H_{T'} \supseteq H_T$ (as now we can support both insertions and deletions of edges), and ii) augment the data-structure with an operation that allows deleting edges from the maintained vertex sparsifier:

- Delete(e): delete e from G and update H_T to be an α-vertex sparsifier of G with respect to T.

We call this extended data-structure notion an *α-fully dynamic vertex sparsifier* (DVS) of G. Similarly to the above, we can define an efficient version of this definition which incorporates the recourse, i.e., the number of edge insertions or deletions to the old sparsifier for obtaining the one. Performing recursive invocations of the α-DVS leads to a generic meta theorem similar to Theorem II.1 for fully dynamic algorithms, which is deferred to the full version of the paper.

Our notion of α-DVS is inspired by the work of Durfee et al. [30], where an approximate Schur complement (which can be viewed an effective resistance vertex sparsifier) is maintained while supporting the addition of vertices to the set of terminals. However, their work ignored the notion of recourse, which in turn limited their algorithm to only a two-level hierarchy. As we will shortly discuss, the bound on recourse will play a critical role in improving the dynamic algorithm for maintaining effective resistances.

In the following, we start by discussing the application of our framework to dynamic (approximate) maximum-flows/minimum-cuts and shortest paths, both of which employ only a two-level hierarchical scheme due to the large recourse incurred by the implementation of their α-DVS operations.

C. Fully Dynamic Maximum-Flows/Minimum-Cuts

In the static setting, Madry [71] presented a general method for designing fast approximation algorithms for cut-based graph optimization problems. His work built on the cut-based graph decomposition of Räcke [31], which was developed in the context of constructing competitive oblivious routing schemes. The novel ingredient in Madry's construction was the notion of j-trees: a connected graph $H = (V_H, E_H)$ is a *j-tree* if it is a union of (i) a subgraph H' (referred to as the *core*) of H induced by a vertex set $V'_H \subseteq V_H$ with $|V'_H| \leq j$ and (ii) a forest F (referred to as the *envelope*) whose each connected component has exactly one vertex in V'_H, where $j \geq 1$. These graphs can be thought of as generalizations of trees (e.g., a 1-tree is a tree) and the main observation is that the core of a j-tree is a smaller graph than H itself and at the same time captures the non-trivial cut structure of the graph H. Madry's framework employed the multiplicative weights update (MWU) method to embed any general graph G into a convex combination of j-trees and achieved a significant speedup in the running-time over Räcke's tree-based embedding while still being able to approximate the cut structure of graph up to a poly-logarithmic factor.

The main technical contribution behind our dynamic maximum-flows/minimum-cuts result (Theorem I.1) is an algorithm for maintaining a j-tree-based embedding of a graph under the operation of adding vertices to the core of the j-trees. An important observation towards designing such an algorithm is that such an embedding can, in some sense, be viewed as a generalized version of the vertex sparsifier notion; instead of being a single graph, the decomposition is a small collection of vertex sparsifiers, where vertices in the cores correspond to the terminal vertices, that *together* approximate the cut-structure of a graph. Here, it is important to note that while j-trees are large graphs, it suffices to only

look for cuts in their cores whose size is much smaller by definition. In this way, our algorithmic task can be thought of as designing an efficient α-DVS.

At a high level, our dynamic j-tree based decomposition proceeds as follows. First, we construct such a decomposition following Madry's framework and then sample a logarithmic number of j-trees from it. The construction of a single j-tree H essentially follows the same idea from previous works: i) construct a low-stretch spanning tree T of G, ii) remove the tree edges that experienced a large congestion when embedding G to T and declare their endpoints to be terminals/core vertices, and (iii) re-route the impacted tree and non-tree edges. One crucial difference is that we add more vertices to the core of H so that every vertex that does not belong to a core can reach a core vertex within $O(n/j)$ steps. This does not affect the embedding quality of H and plays a pivotal role for obtaining an efficient implementation of the AddTerminal operation. Second, we maintain a dynamic cut sparsifier [36] for every core in the decomposition, which is critical for obtaining a faster query time. Finally, to declare a vertex a terminal, i.e., move a non-core vertex u to the core of H, we need to maintain an efficient representation of the tree paths used for embedding G into H. This is due to the fact that paths which contain u must now be shortcut at u, which in turn leads to re-routing many edges in the embedding. We achieve this by employing dynamic tree data structures, carefully bounding the number of changes incurred by this operation and amortizing the latter over a certain number of operations. Further implementation details can be found in Section III.

We remark that the edge updates and queries reduce to the AddTerminal operation. Specifically, upon insertion/deletion of an edge $e = (u, v)$, we declare both of the endpoints terminals, move them to the core of each H, and then directly insert/delete the edge from the core. Similarly, when a query is asked about the maximum-flow/minimum-cut between any vertex pair (s, t), we declare them terminals in each H, compute the minimum cut from scratch in the core of each H and report the minimum one among these cuts.

D. Fully Dynamic Shortest Paths

Our fully dynamic (approximate) shortest path algorithm (Theorem I.2) proceeds almost identically to the one involving flows. Inspired by Madry's framework, we start by introducing *metric* j-tree based decompositions; given a graph G, this decomposition is a convex combination $\{(\lambda_i, H_i)\}_i$ of j-trees such that (i) the distances in H_i dominate the ones in G and (ii) the expected distances in H, where H is sampled from the corresponding distribution, is within some factor from the distances in G. To construct a single metric j-tree, we find a low-stretch spanning tree of G [72], determine non-tree edges whose stretch in T is large, add these edges to T and declare their endpoints to be the core vertices. Similar to Madry's framework, we can then apply the MWU method to efficiently construct the decomposition.

The dynamic part of our data-structure remains almost the same, except that we maintain dynamic spanners [73] instead of dynamic cut sparsifiers in the core of each j-tree. Further details are deferred to the full version of this paper.

E. Fully Dynamic Effective Resistances

The dynamic effective resistances algorithm of Durfee et al. [30] is based on maintaining dynamic spectral vertex sparsifiers (also known as approximate Schur complements) under the addition of terminals. The crux of their approach was the observation that spectral vertex sparsifiers can be approximated by a collection of random walks. This was then further extended by carefully choosing the set of terminal so that the length of these walks is sub-linear in the size of the graph, which in turn allowed for both faster preprocessing and update time.

In this paper, we essentially take the same approach, with the main difference being that we also record vertices that appear in a large number of random walks and add them as terminals during the preprocessing. This ensures that non-terminal vertices appear in a much smaller number of walks, so their potential future addition to the terminal set does not affect much the collection of random walks. The latter gives that the recourse is small and thus allows us to use the recursive invocations of the fully dynamic meta theorem to improve their running time by a polynomial factor, as stated in Theorem I.4.

In each of the above applications, 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 queries[3], 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.

III. Fully Dynamic All-Pair Maximum Flows/Minimum Cuts

In this section, we show a dynamic algorithm that allows querying an estimate of the st-max flow for any pair s, t. In what follows, we will omit several proofs due to space constraints and refer the read to the full version of the paper.

Theorem III.1. *Given a graph $G = (V, E, c)$ with polynomial bounded weights, there is a dynamic data structure maintaining G subject to the following operations:*

[3]The 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

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 $s-t$ maximum flow/minimum cut value of G in $\tilde{O}(m^{2/3})$-time w.h.p.. If requested, return S in $O(|S|)$ time.*

We can replace m by n in the update and query time by running the algorithm on top of a dynamic sparsifier (see Lemma III.14), which in turn implies the algorithm that works against an oblivious adversary in Theorem I.1. The algorithm against an adaptive adversary is included in the full version of the paper.

A. Some Known Tools

In this section, we present known concepts and tools that we will exploit in our dynamic algorithms.

1) Cut, Flow, and Embedding: First, we recall notions of cuts and (multicommodity) flow.

Definition III.2. *Given a graph $G = (V, E, c)$, a cut C is any proper subset of V, i.e., $\emptyset \neq C \subset V$. Define $E(C) \coloneqq E(C, V \setminus C)$ and $c(C) \coloneqq \sum_{e \in E(C)} c(e)$.*

Definition III.3. [71] *Given a graph $G = (V, E, c)$, $f = (f^1, \ldots, f^k) \in \mathbb{R}^{E \times k}$ is a* multicommodity flow *if f^i is an $s_i t_i$-flow. Define $|f(e)| \coloneqq \sum_{i=1}^{k} |f^i(e)|$ as the total flow crossing the edge $e \in E$. A multicommodity flow f is feasible if for every edge e, we have $|f(e)| \leq c(e)$. A* single-commodity flow *f is a multicommodity flow with $k = 1$.*

Next, we recall the notion of graph embeddings.

Definition III.4. [71] *Let $G = (V, E, c)$ and $H = (V, E_H, c_H)$ be two graphs sharing the same vertex set. For every edge $e = (u, v) \in E$, let f^e be the flow that routes $c(e)$ amount of flow from u to v in H. Then $f \coloneqq (f^e \mid e \in E) \in \mathbb{R}^{E_H \times E}$, the collection of f^e for every edge $e \in E$, is an* embedding *of G into H.*

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

Definition III.5. [71] *Let $G = (V, E, c)$ and $H = (V, E_H, c_H)$ be graphs over the same vertex set and let $t \geq 1$. We say G is t-*embeddable *into H, denoted by $G \preceq_t H$, if there is an embedding f of G into H such that for every $e_H \in E_H$, $|f(e_H)| \leq t c_H(e_H)$. When $t = 1$, we ignore the subscript and say that G is embeddable into H, i.e., $G \preceq H$.*

We now define the notion cut sparsifiers.

Definition III.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* $(1+\epsilon)$-cut-sparsifier *of G (abbr. $H \approx_\epsilon G$) if for every $S \subseteq V$,*

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

We also present some structural results about the value of maximum flow/minimum cut between mutually embeddable graphs.

Corollary III.7. *For a given $t \geq 1$, let $G = (V, E, c)$ and $H = (V, E_H, c_H)$ be two graphs defined on the same vertex set such that $G \preceq H$ and $H \preceq_t G$. For any cut $S \subseteq V$, we have*

$$c(S) \leq c_H(S) \leq t c(C).$$

2) J-trees as Vertex Sparsifiers: We start by recall the notion of j-trees introduced by Madry [71].

Definition III.8. [71] *Given $j \geq 1$, a graph $H = (V_H, E_H, c_H)$ 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. For $S \subseteq C$, let $F(S)$ be the union of $F(u)$ for all u in S.*

Note that j trees are much simpler objects than general graphs because all but j vertices are contained in a forest, and the computing graph properties on forests is usually less challenging.

Next, we define the notion of (k, ρ, j)-decomposition for a given graph G. Roughly speaking, it can be thought of as a k-sized family of j-trees that approximate the cut/flow structure of G.

Definition III.9. [71] *A family of graphs $\{G_i\}_{i=1}^k$ is a (k, ρ, j)-decomposition of a graph $G = (V, E, c)$ if*

1) *G_i is a j-tree, for all $i = 1, \ldots, k$.*
2) *$G \preceq G_i$, for all $i = 1, \ldots, k$,*
3) *$\sum_i G_i \preceq_{k \cdot \rho} G$.*

Madry designed an algorithm that efficiently computes (k, ρ, j)-decomposition for G. We summarize his result in the following lemma.

Lemma III.10. [71] *Given any graph $G = (V, E, c)$ with polynomially bounded weights and a parameter $k \geq 1$, there is an $O(km \log^4 n)$-time algorithm that computes a*

$$\left(k, \tilde{O}(\log n), O\left(\frac{m \log^3 n}{k}\right)\right)\text{-decomposition of } G.$$

If $\{G_i\}_{i=1}^k$ is the family of graphs corresponding to this decomposition, then the weight ratio of each G_i is $O(mU)$, where U denotes the weight ratio of G. Moreover, if we sample G_i with probability $1/k$, for any fixed cut S, then the size of this cut in G_i is at most 2ρ times the size of the cut in G with probability at least $\frac{1}{2}$.

Unfortunately, we can not afford to maintain this many j-trees. To address this, we make use of the lemma below,

which shows that it is sufficient to maintain only a logarithmic number of such graphs.

Lemma III.11. [71] *Let $G = (V, E, c)$ be a graph with polynomially bounded weights, $k = \Omega(\log n)$ and $\{G_i\}_{i=1}^{k}$ be a (k, ρ, j)-decomposition of G. By sampling $O(\log n)$ graphs from $\{G_i\}_{i=1}^{k}$, every minimum cut between any two vertices is preserved up to a 2ρ-factor with high probability.*

Lemma III.11 implies that optimal pairwise minimum cuts can be preserved using $O(\log n)$-many j-trees up to a $O(\rho)$-factor with high probability. In our application, 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 prove the correctness of this approach, we define the concept of a *core cut*.

Definition III.12. *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*

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

In other words, we extend the cut S by including trees in the envelope rooted at vertex in S. The next lemma shows that it suffices to compute the minimum cuts in the core.

Lemma III.13. *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*

$$c_H(\text{ext}(S \cap C)) \leq c_H(S).$$

In other words, to find a minimum cut separating core vertices in H, it suffice to check only cuts in the core and then construct the core cut. To make this dynamic, we just need to support the operation of adding terminals as defined in the local sparsifiers paper [74], and in Chapter 6 of [29].

We next review algorithms for dynamically maintaining cut sparsifiers and approximating minimum cuts.

Lemma III.14. [36] *Given a graph $G = (V, E, c)$ with polynomially bounded weights, there is an algorithm that maintains a $(1+\epsilon)$-cut sparsifier H of G w.h.p. while support insertions/deletions of edges in $O(\log^6 n/\epsilon^2)$ amortized update time. The weight ratio of H is $O(nU)$, where U denotes the weight ratio of G. Moreover, we maintain a partition of H into $k = O(\log^3 n\epsilon^{-2} \log U)$ disjoint forests $T_1, \ldots, T_k$ where each vertex keeps the set of its neighbors in each forest T_i. After each edge insertion/deletion in G, at most one edge change occurs in each forest T_i.*

Lemma III.15. [37] (rephrased) *Given a graph $G = (V, E, c)$ with polynomially bounded weights, and a source/sink pair (s, t) with $s, t \in V$, there is an algorithm $\tilde{O}(m)$ that approximates the minimum cut between s and t up to a factor of 2. The algorithm can also report the corresponding cut with linear overhead in its size.*

B. The Main Lemma: Dynamic j-trees

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 III.11, we maintain only $O(\log n)$ of these j-trees instead of k. For every st-cut query, we ran the algorithm from lemma III.15 [37] on these $O(\log n)$ core graphs.

To dynamically maintain these core graphs, dynamic cut sparsifiers from lemma III.14 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 in the following lemma.

Lemma III.16. *Let $j \geq 1$ and $G = (V, E, c)$ be a graph with polynomially bounded weights and a j-tree H of G such that $H \preceq G \preceq_\alpha H$. Let $n = |V|, m = |E|$. We can dynamically maintain a $O(j)$-tree $\tilde{H}$ such that $\tilde{H} \preceq G \preceq_{O(\alpha)} \tilde{H}$ while supporting up to j operations of the following form:*

1) INITIALIZE(G)*: Build the data structure for maintaining H in $O(\frac{mn}{j} \log n)$-time.*
2) ADDTERMINAL(u)*: Move vertex u to the core of H in $O(\frac{mn}{j^2} \log^6 n)$ amortized time.*
3) INSERT(u, v, c)*: Insert the edge (u, v) to G in $O(\frac{mn}{j^2} \log^6 n)$ amortized time.*
4) DELETE(e)*: Delete the edge e from G in $O(\frac{mn}{j^2} \log^6 n)$ amortized 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, at any time, the core $C(\widetilde{H})$ is sparse, i.e., it has $O(j \log^4 j)$ edges.

The rest of this section is for proving III.16.

1) Tree-terminal path and edge moving: To prove Lemma III.16, we have to open the black box of j-tree construction. It creates a graph by first select a specific spanning tree/forest and then routes off-tree edges by tree paths and set of edges restricted on a small subset of vertices. To better understand and formalize the construction, we introduce some notations about spanning forests and trees.

Definition III.17. *Let F be a forest and let $C \subseteq V(F)$ be a subset of vertices. Consider the following steps: i) add at most $|C|$ vertices to C so that every pairwise lowest common ancestor is in C, ii) iteratively remove degree-1 vertices from $V(F) \setminus C$ until no such vertices remain, and iii) for each path with endpoints in C and no internal vertices in C, replace the whole path with a single edge. We call the resulting forest the* skeleton tree *of F with respect to C, and denote it by $S(F, C)$.*

Definition III.18. *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 III.19. *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, denoted by $T_{C,F}(u)$, as the only vertex of C in the component containing u in $T \setminus F$.*

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

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

We also define e's tree-representative path *as*

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

Next we review some notation from [75], which defined the so-called tree-portal paths. Portals are terminals in our terminology.

Definition III.20. *Let $G = (V, E)$ be a graph, T be a spanning forest of G and C be a subset of vertices called terminals. For any two 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*

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

We also define e's tree-terminal path *as*

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

2) Initializing a j-Tree: In this subsection, we review the static construction of a j-tree. To be able to exploit such a construction in the dynamic setting, we slightly modify the construction from [71]. Formal details about these modifications are presented in the full version of the paper.

At a high level, 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 $T \setminus F$. 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 $\mathrm{Repr}_{T,C,F}(e)$ for F or $\mathrm{Move}_{T,C}(e)$ 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 two tree paths concatenated by an edge in the core.

The sets T, C and F are computed as follows. 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) [76] 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 $\mathrm{Move}_{T,C}(e)$. 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 [71], such edge moving does not guarantee a j-tree.

Identical to [71], 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 [71] is that we add more terminal vertices (C) and route off-tree edges after we determine C. An argument from [77] shows that the more terminal we add, the better the congestion approximation.

3) Structural arguments for j-tree maintenance: To prove Lemma III.16, one has to make sure adding terminals does not increase the congestion. The argument is formalized as the following lemma.

Lemma III.21. *Let $G = (V, E, c)$ be a graph, T be a spanning forest of G, C be a subset of vertices and F be 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 $F \cup e_u$ is tree partition of T with respect to $C \cup u$, i.e., every component of $T \setminus (F \cup e_u)$ has exactly one vertex in C.*

Furthermore, let $H := \textsc{Route}(G, T, C, F)$. If $H \preceq_\alpha G$, then the graph $\overline{H} := \textsc{Route}(G, T, C \cup u, F \cup e_u) \preceq_\alpha G$.

To maintain the $O(j)$-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 III.22. *Let $G = (V, E, c)$ be a graph, T be a spanning forest of G, C be a subset of vertices and F be a tree partition of T with respect to C. Let $H := \textsc{Route}(G, T, C, F)$, 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 $H \preceq_\alpha G$ holds via the canonical j-tree embedding*, then both $(H \cup e) \preceq_\alpha (G \cup e)$ and $(H \setminus e) \preceq_\alpha (G \setminus e)$ holds.*

4) Proof sketch of Lemma III.16: Here, we only sketch the argument (see the full version of the paper for the full

proof). 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, $\widetilde{H}$. The reason for not applying sparsifier to the whole graph is because edges not in the core form a forest. And by Lemma III.13, 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 $\textsc{Route}(G, T, C, F)$. 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 $\textsc{Route}(G, T, C \cup u, F \cup e_u)$.

Thus, we maintain the following data structures:

1) A dynamic 2-cut sparsifier $\widetilde{C}(H)$ from Lemma III.14 for maintaining a sparsified core graph.
2) A *dynamic tree data structure* $D(T)$ (see the full version of the paper for the detailed guarantee) for finding such e_u. $D(T)$ is also used to find $\textsc{Repr}_{T,C,F}(e_u)$, 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 $\mathcal{W}$ as a collection of all such walks.
4) For every $x \in C$, maintain a set $P(x)$ consisting of walks in $\mathcal{W}$ ending up at x.
5) For every vertex $u \in V$, maintain a set $\texttt{RI}(u)$ consisting of walks in $\mathcal{W}$ containing u.

The last three data structure is for maintaining $\textsc{Move}_{T,C}(e), e \notin T$ with C increasing.

C. Put everything together

Given the key lemma, Lemma III.16, we can now proof the main theorem, Theorem III.1.

Proof of Theorem III.1: Let j be some parameter determined later and $k = \Theta(\frac{m \log U \log^2 n}{j})$. Initialization of the data structure is summarized as Algorithm 1. First apply Lemma III.10 to acquire a $(k, \tilde{O}(\log n), \Theta(j))$-decomposition of G, say $\{G_i\}_{i=1}^k$. Then we apply Lemma III.11 to sample $t = O(\log n)$ of them, say $\{G_i\}_{i=1}^t$. For each of G_i, we incur Lemma III.16 to build data structures for dynamical operations. Let $\{D_i\}_{i=1}^t$ be the data structures for each of $\{D_i\}_{i=1}^t$. 2-approximated dynamic cut sparsifers from Lemma III.14 is also built for the cores of $\{D_i\}_{i=1}^t$. Note that each D_i supports up to j operations, we rebuild $\{G_i\}_{i=1}^k$ and $\{G_i\}_{i=1}^t$ every j operations. To deal with the query $\text{mincut}(s, t)$, we run the algorithm from Lemma III.15 on each sparsified core of $\{D_i\}_{i=1}^t$. The running time is $\tilde{O}(tj) = \tilde{O}(j)$. Among results, the one with the smallest cut value is returned. The correctness comes from Lemma III.11 and Lemma III.13 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 $\{D_i\}_{i=1}^t$ 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 III.16, 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 $\{G_i\}_{i=1}^t$ and cut sparsifiers for cores. By charging the cost among j operations, the runtime cost charged with each operation is

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

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

Algorithm 1: Initialize(G)

$n := |V(G)|$, $m := |E(G)|$, $j := m^{2/3}$, $k := \Theta(\frac{m \log U \log^2 n}{j})$, $t := \Theta(\log n)$.
Let $\{G_i\}_{i=1}^k$ be a $(k, \tilde{O}(\log n), \Theta(j))$-decomposition of G by Lemma III.10.
Sample t graphs with repetition from G, say, $\{G_i\}_{i=1}^t$.
for $i = 1, \ldots, t$ **do**
    $D_i :=$ Initialize(G, G_i) by Lemma III.16.
return $\{D_i\}_{i=1}^t$

References

[1] K. M. Borgwardt, H.-P. Kriegel, and P. Wackersreuther, "Pattern mining in frequent dynamic subgraphs," in *International Conference on Data Mining (ICDM)*, 2006, pp. 818–822. 1

[2] A. Paranjape, A. R. Benson, and J. Leskovec, "Motifs in temporal networks," in *International Conference on Web Search and Data Mining (WSDM)*, 2017, pp. 601–610. 1

[3] R. Angles and C. Gutierrez, "Survey of graph database models," *ACM Computing Surveys (CSUR)*, vol. 40, no. 1, pp. 1–39, 2008. 1

[4] I. Robinson, J. Webber, and E. Eifrem, *Graph databases*. "O'Reilly Media, Inc.", 2013. 1

[5] D. Nanongkai and T. Saranurak, "Dynamic spanning forest with worst-case update time: adaptive, las vegas, and o(n$^{1/2 - \epsilon}$)-time," in *Symposium on Theory of Computing (STOC)*, 2017, pp. 1122–1129. 1, 3

[6] C. Wulff-Nilsen, "Fully-dynamic minimum spanning forest with improved worst-case update time," in *Symposium on Theory of Computing (STOC)*, 2017, pp. 1130–1143. 1, 3

[7] D. Nanongkai, T. Saranurak, and C. Wulff-Nilsen, "Dynamic minimum spanning forest with subpolynomial worst-case update time," in *Symposium on Foundations of Computer Science (FOCS)*, 2017, pp. 950–961. 1, 3

[8] P. Sankowski, "Dynamic transitive closure via dynamic matrix inverse," in *Symposium on Foundations of Computer Science (FOCS)*, 2004, pp. 509–517. 1

[9] L. Roditty and U. Zwick, "Improved dynamic reachability algorithms for directed graphs," *SIAM Journal on Computing*, vol. 37, no. 5, pp. 1455–1471, 2008. 1

[10] J. van den Brand, D. Nanongkai, and T. Saranurak, "Dynamic matrix inverse: Improved algorithms and matching conditional lower bounds," in *Symposium on Foundations of Computer Science (FOCS)*, 2019, pp. 456–480. 1

[11] K. Hanauer, M. Henzinger, and C. Schulz, "Faster fully dynamic transitive closure in practice," *CoRR*, vol. abs/2002.00813, 2020. 1

[12] A. Abboud and V. V. Williams, "Popular conjectures imply strong lower bounds for dynamic problems," in *Symposium on Foundations of Computer Science (FOCS)*, 2014, pp. 434–443. 1, 2

[13] M. Henzinger, S. Krinninger, D. Nanongkai, and T. Saranurak, "Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture," in *Symposium on Theory of Computing (STOC)*, 2015, pp. 21–30. 1

[14] A. Abboud and S. Dahlgaard, "Popular conjectures as a barrier for dynamic planar graph algorithms," in *Symposium on Foundations of Computer Science (FOCS)*, 2016, pp. 477–486. 1, 2

[15] S. Dahlgaard, "On the hardness of partially dynamic graph problems and connections to diameter," in *International Colloquium on Automata, Languages, and Programming (ICALP)*, 2016, pp. 48:1–48:14. 1, 2, 3

[16] P. Sankowski, "Subquadratic algorithm for dynamic shortest distances," in *International Conference on Computing and Combinatorics (COCOON)*, ser. Lecture Notes in Computer Science, vol. 3595, 2005, pp. 461–470. 1

[17] M. Thorup, "Worst-case update times for fully-dynamic all-pairs shortest paths," in *Symposium on Theory of Computing (STOC)*, 2005, pp. 112–119. 1

[18] A. Bernstein, "Fully dynamic (2 + epsilon) approximate all-pairs shortest paths with fast query and close to linear update time," in *Symposium on Foundations of Computer Science (FOCS)*, 2009, pp. 693–702. 1

[19] L. Roditty and U. Zwick, "Dynamic approximate all-pairs shortest paths in undirected graphs," *SIAM J. Comput.*, vol. 41, no. 3, pp. 670–683, 2012. 1

[20] I. Abraham, S. Chechik, and K. Talwar, "Fully dynamic all-pairs shortest paths: Breaking the o(n) barrier," in *APPROX-RANDOM*, ser. LIPIcs, vol. 28, 2014, pp. 1–16. 1, 3

[21] A. Bernstein, "Maintaining shortest paths under deletions in weighted directed graphs," *SIAM J. Comput.*, vol. 45, no. 2, pp. 548–574, 2016. 1

[22] I. Abraham, S. Chechik, and S. Krinninger, "Fully dynamic all-pairs shortest paths with worst-case update-time revisited," in *Symposium on Discrete Algorithms (SODA)*, 2017, pp. 440–452. 1

[23] S. Chechik, "Near-optimal approximate decremental all pairs shortest paths," in *Symposium on Foundations of Computer Science (FOCS)*, 2018, pp. 170–181. 1

[24] J. Han, J. Pei, and M. Kamber, *Data mining: concepts and techniques*. Elsevier, 2011. 1

[25] Y. Boykov, O. Veksler, and R. Zabih, "Fast approximate energy minimization via graph cuts," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 23, no. 11, pp. 1222–1239, 2001. 1

[26] X. J. Zhu, "Semi-supervised learning literature survey," University of Wisconsin-Madison Department of Computer Sciences, Tech. Rep., 2005. 1

[27] A. V. Goldberg and R. E. Tarjan, "Efficient maximum flow algorithms," *Commun. ACM*, vol. 57, no. 8, pp. 82–89, 2014. 1

[28] P. G. Doyle and J. L. Snell, *Random Walks and Electric Networks*, ser. Carus Mathematical Monographs. Mathematical Association of America, 1984, vol. 22. 1

[29] G. Goranci, "Dynamic graph algorithms and graph sparsification: New techniques and connections," Ph.D. dissertation, Universitat Wien, 2019, available at: http://arxiv.org/abs/1909.06413. 1, 8

[30] D. Durfee, Y. Gao, G. Goranci, and R. Peng, "Fully dynamic spectral vertex sparsifiers and applications," in *Symposium on Theory of Computing (STOC)*, 2019, pp. 914–925. 1, 2, 5, 6

[31] H. Räcke, "Optimal hierarchical decompositions for congestion minimization in networks," in *Symposium on Theory of Computing (STOC)*, 2008, p. 255–264. 2, 5

[32] M. Thorup and U. Zwick, "Approximate distance oracles," *J. ACM*, vol. 52, no. 1, pp. 1–24, 2005. 2, 5

[33] D. Spielman and N. Srivastava, "Graph sparsification by effective resistances," *SIAM Journal on Computing*, vol. 40, no. 6, pp. 1913–1926, 2011. 2

[34] K. Bringmann, M. Künnemann, and A. Nusser, "Fréchet distance under translation: Conditional hardness and an algorithm via offline dynamic grid reachability," in *Symposium on Discrete Algorithms (SODA)*, 2019, pp. 2902–2921. 2

[35] H. Li, S. Patterson, Y. Yi, and Z. Zhang, "Maximizing the number of spanning trees in a connected graph," *IEEE Transactions on Information Theory*, vol. 66, no. 2, pp. 1248–1260, 2019. 2

[36] I. Abraham, D. Durfee, I. Koutis, S. Krinninger, and R. Peng, "On fully dynamic graph sparsifiers," in *Symposium on Foundations of Computer Science (FOCS)*, 2016, pp. 335–344. 3, 6, 8

[37] R. Peng, "Approximate undirected maximum flows in $O(m\text{polylog}(n))$ time," in *Symposium on Discrete Algorithms (SODA)*, 2016, pp. 1862–1867. 3, 8

[38] J. Sherman, "Area-convexity, l_∞ regularization, and undirected multicommodity flow," in *Symposium on Theory of Computing (STOC)*, 2017, pp. 452–460. 3

[39] ——, "Breaking the multicommodity flow barrier for o(vlog n)-approximations to sparsest cut," in *Symposium on Foundations of Computer Science (FOCS)*, 2009, pp. 363–372. 3

[40] H. Y. Cheung, T. C. Kwok, and L. C. Lau, "Fast matrix rank algorithms and applications," *J. ACM*, vol. 60, no. 5, pp. 31:1–31:25, 2013. 3

[41] M. Gupta and S. Khan, "Simple dynamic algorithms for maximal independent set and other problems," *CoRR*, vol. abs/1804.01823, 2018. 3

[42] M. Thorup, "Fully-dynamic min-cut," *Combinatorica*, vol. 27, no. 1, pp. 91–127, 2007. 3

[43] J. Lacki and P. Sankowski, "Min-cuts and shortest cycles in planar graphs in o(n loglogn) time," in *European Symposium on Algorithms (ESA)*, vol. 6942, 2011, pp. 155–166. 3

[44] G. Goranci, M. Henzinger, and M. Thorup, "Incremental

exact min-cut in polylogarithmic amortized update time," *ACM Trans. Algorithms*, vol. 14, no. 2, pp. 17:1–17:21, 2018. 3

[45] M. Ghaffari, K. Nowicki, and M. Thorup, "Faster algorithms for edge connectivity via random 2-out contractions," in *Symposium on Discrete Algorithms (SODA)*, 2020, pp. 1260–1279. 3

[46] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig, "Sparsification - a technique for speeding up dynamic graph algorithms," *J. ACM*, vol. 44, no. 5, pp. 669–696, 1997. 3

[47] A. Bernstein and L. Roditty, "Improved dynamic algorithms for maintaining approximate shortest paths under deletions," in *Symposium on Discrete Algorithms (SODA)*, 2011, pp. 1355–1365. 3

[48] A. Bernstein and S. Chechik, "Deterministic decremental single source shortest paths: beyond the o(mn) bound," in *Symposium on Theory of Computing (STOC)*, 2016, pp. 389–397. 3

[49] ——, "Deterministic partially dynamic single source shortest paths for sparse graphs," in *Symposium on Discrete Algorithms (SODA)*, 2017, pp. 453–469. 3

[50] A. Bernstein and C. Stein, "Fully dynamic matching in bipartite graphs," in *International Colloquium on Automata, Languages, and Programming (ICALP)*, 2015, pp. 167–179. 3

[51] ——, "Faster fully dynamic matchings with small approximation ratios," in *Symposium on Discrete Algorithms (SODA)*, 2016, pp. 692–711. 3

[52] S. Solomon, "Local algorithms for bounded degree sparsifiers in sparse graphs," in *Innovations in Theoretical Computer Science Conference (ITCS)*, 2018, pp. 52:1–52:19. 3

[53] D. Eppstein, Z. Galil, G. F. Italiano, and T. H. Spencer, "Separator based sparsification. i. planary testing and minimum spanning trees," *J. Comput. Syst. Sci.*, vol. 52, no. 1, pp. 3–27, 1996. 3

[54] ——, "Separator-based sparsification II: edge and vertex connectivity," *SIAM J. Comput.*, vol. 28, no. 1, pp. 341–381, 1998. 3

[55] J. Fakcharoenphol and S. Rao, "Planar graphs, negative weight edges, shortest paths, near linear time," in *Symposium on Foundations of Computer Science (FOCS)*, 2001, pp. 232–241. 3

[56] G. Goranci, M. Henzinger, and P. Peng, "The power of vertex sparsifiers in dynamic graph algorithms," in *European Symposium on Algorithms (ESA)*, ser. LIPIcs, vol. 87, 2017, pp. 45:1–45:14. 3

[57] ——, "Dynamic effective resistances and approximate schur complement on separable graphs," in *European Symposium on Algorithms (ESA)*, ser. LIPIcs, vol. 112, 2018, pp. 40:1–40:15. 3

[58] D. Eppstein, "Offline algorithms for dynamic minimum spanning tree problems," in *Algorithms and Data Structures Symposium (WADS)*, ser. Lecture Notes in Computer Science, vol. 519, 1991, pp. 392–399. 3

[59] R. Peng, B. Sandlund, and D. D. Sleator, "Optimal offline dynamic 2, 3-edge/vertex connectivity," in *Algorithms and Data Structures Symposium (WADS)*, 2019, pp. 553–565. 3

[60] H. Li, S. Patterson, Y. Yi, and Z. Zhang, "Maximizing the number of spanning trees in a connected graph," *IEEE Transactions on Information Theory*, vol. 66, no. 2, pp. 1248–1260, 2019. 3

[61] G. Goranci, H. Räcke, T. Saranurak, and Z. Tan, "The expander hierarchy and its applications to dynamic graph algorithms," 2020. 3

[62] P. Chalermsook, S. Das, B. Laekhanukit, and D. Vaz, "Mimicking networks parameterized by connectivity," *CoRR*, vol. abs/1910.10665, 2019. 3

[63] Y. P. Liu, R. Peng, and M. Sellke, "Vertex sparsifiers for c-edge connectivity," *CoRR*, vol. abs/1910.10359, 2019. 3

[64] W. Jin and X. Sun, "Fully dynamic c-edge connectivity in subpolynomial time," *CoRR*, vol. abs/2004.07650, 2020. 3

[65] A. Moitra, "Approximation algorithms for multicommodity-type problems with guarantees independent of the graph size," in *Symposium on Foundations of Computer Science (FOCS)*, 2009, pp. 3–12. 4

[66] F. T. Leighton and A. Moitra, "Extensions and limits to vertex sparsification," in *Symposium on Theory of Computing (STOC)*, 2010, pp. 47–56. 4

[67] M. Charikar, T. Leighton, S. Li, and A. Moitra, "Vertex sparsifiers and abstract rounding algorithms," in *Symposium on Foundations of Computer Science (FOCS)*, 2010, pp. 265–274. 4

[68] J. Chuzhoy, "On vertex sparsifiers with steiner nodes," in *Symposium on Theory of Computing (STOC)*, 2012, pp. 673–688. 4

[69] K. Makarychev and Y. Makarychev, "Metric extension operators, vertex sparsifiers and lipschitz extendability," in *Symposium on Foundations of Computer Science (FOCS)*, 2010, pp. 255–264. 4

[70] M. Englert, A. Gupta, R. Krauthgamer, H. Räcke, I. Talgam-Cohen, and K. Talwar, "Vertex sparsifiers: New results from old techniques," *SIAM J. Comput.*, vol. 43, no. 4, pp. 1239–1262, 2014, announced at APPROX'10. 4

[71] A. Madry, "Fast approximation algorithms for cut-based problems in undirected graphs," in *Symposium on Foundations of Computer Science (FOCS)*, 2010, pp. 245–254. 5, 7, 8, 9

[72] I. Abraham and O. Neiman, "Using petal-decompositions to build a low stretch spanning tree," *SIAM J. Comput.*, vol. 48, no. 2, pp. 227–248, 2019. 6

[73] S. Forster and G. Goranci, "Dynamic low-stretch trees via dynamic low-diameter decompositions," in *Symposium on Theory of Computing (STOC)*, 2019, pp. 377–388. 6

[74] G. Goranci, M. Henzinger, and T. Saranurak, "Fast incremental algorithms via local sparsifiers," 2018. 8

[75] R. Kyng, R. Peng, S. Sachdeva, and D. Wang, "Flows in almost linear time via adaptive preconditioning," in *Symposium on Theory of Computing (STOC)*, 2019, pp. 902–913. 9

[76] I. Abraham, Y. Bartal, and O. Neiman, "Nearly tight low stretch spanning trees," in *Symposium on Foundations of Computer Science (FOCS)*, 2008, pp. 781–790. 9

[77] M. Ghaffari, A. Karrenbauer, F. Kuhn, C. Lenzen, and B. Patt-Shamir, "Near-optimal distributed maximum flow," *SIAM J. Comput.*, vol. 47, no. 6, pp. 2078–2117, 2018. 9