A New Algorithm for Decremental Single-Source Shortest Paths with Applications to Vertex-Capacitated Flow and Cut Problems

Julia Chuzhoy*
Toyota Technological Institute at Chicago
Chicago, IL, U.S.A.
cjulia@ttic.edu

Sanjeev Khanna†
University of Pennsylvania
Philadelphia, PA, U.S.A.
sanjeev@cis.upenn.edu

ABSTRACT

We study the vertex decremental Single-Source Shortest Paths (SSSP) problem: given an undirected graph $G = (V, E)$ with lengths $\ell(e) \geq 1$ on its edges that undergoes vertex deletions, and a source vertex s, we need to support (approximate) shortest-path queries in G: given a vertex v, return a path connecting s to v, whose length is at most $(1 + \epsilon)$ times the length of the shortest such path, where ϵ is a given accuracy parameter. The problem has many applications, for example to flow and cut problems in vertex-capacitated graphs.

Decremental SSSP is a fundamental problem in dynamic algorithms that has been studied extensively, especially in the more standard edge-decremental setting, where the input graph G undergoes edge deletions. The classical algorithm of Even and Shiloach supports exact shortest-path queries in $O(mn)$ total update time. A series of recent results have improved this bound to $O(m^{1+o(1)} \log L)$, where L is the largest length of any edge. However, these improved results are randomized algorithms that assume an *oblivious* adversary. To go beyond the oblivious adversary restriction, recently, Bernstein, and Bernstein and Chechik designed deterministic algorithms for the problem, with total update time $\tilde{O}(n^2 \log L)$, that by definition work against an adaptive adversary. Unfortunately, their algorithms introduce a new limitation, namely, they can only return the approximate length of a shortest path, and not the path itself. Many applications of the decremental SSSP problem, including the ones considered in this paper, crucially require both that the algorithm returns the approximate shortest paths themselves and not just their lengths, and that it works against an adaptive adversary.

Our main result is a randomized algorithm for vertex decremental SSSP with total expected update time $O(n^{2+o(1)} \log L)$, that responds to each shortest-path query in $\tilde{O}(n \log L)$ time in expectation, returning a $(1 + \epsilon)$-approximate shortest path. The algorithm works against an adaptive adversary. The main technical ingredient of our algorithm is an $\tilde{O}(|E(G)| + n^{1+o(1)})$-time algorithm to compute a *core decomposition* of a given dense graph G, which allows us to compute short paths between pairs of query vertices in G efficiently.

We use our result for vertex-decremental SSSP to obtain $(1 + \epsilon)$-approximation algorithms for maximum s-t flow and minimum s-t cut in vertex-capacitated graphs, in expected time $n^{2+o(1)}$, and an $O(\log^4 n)$-approximation algorithm for the vertex version of the sparsest cut problem with expected running time $n^{2+o(1)}$. These results improve upon the previous best known algorithms for these problems in the regime where $m = \omega(n^{1.5+o(1)})$.

CCS CONCEPTS

• **Theory of computation → Network flows**; **Shortest paths**; **Dynamic graph algorithms**.

KEYWORDS

Decremental single-source shortest paths; sparsest cut; vertex-capacitated graphs.

ACM Reference Format:
Julia Chuzhoy and Sanjeev Khanna. 2019. A New Algorithm for Decremental Single-Source Shortest Paths with Applications to Vertex-Capacitated Flow and Cut Problems. In *Proceedings of the 51st Annual ACM SIGACT Symposium on the Theory of Computing (STOC '19), June 23–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3313276.3316320

1 INTRODUCTION

We consider the *vertex*-decremental Single-Source Shortest Paths (SSSP) problem in edge-weighted undirected graphs, and its applications to several cut and flow problems in vertex-capacitated graphs. In the vertex-decremental SSSP, we are given an undirected graph G with lengths $\ell(e) \geq 1$ on its edges, and a source vertex s. The goal is to support (approximate) shortest-path queries from the source vertex s, as the graph G undergoes a sequence of online adversarial vertex deletions. We consider two types of queries: in a path-query, we are given a query vertex v, and the goal is to return a path connecting s to v, whose length is at most $(1 + \epsilon)$ times the length of the shortest such path, where ϵ is the given accuracy parameter. In a dist-query, given a vertex v, we need to report an (approximate) distance from s to v. We will use the term *exact* path-query when the algorithm needs to report the shortest s-v path, and *approximate* path-query when a $(1 + \epsilon)$-approximate shortest s-t path is sufficient. We will similarly use the terms of exact and approximate dist-query. We also distinguish between an *oblivious adversary* setting, where the sequence of vertex deletions is fixed in advance, and *adaptive adversary*, where each vertex in the

*Part of the work was done while the author was a Weston visiting professor in the Department of Computer Science and Applied Mathematics, Weizmann Institute. Supported in part by NSF grant CCF-1616584.
†Supported in part by the National Science Foundation grant CCF-1617851.

STOC '19, June 23–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6705-9/19/06…$15.00
https://doi.org/10.1145/3313276.3316320

deletion sequence may depend on the responses of the algorithm to previous queries.

A closely related variation of this problem, that has been studied extensively, is the *edge*-decremental SSSP problem, where the graph G undergoes edge deletions and not vertex deletions. Edge-decremental SSSP captures the vertex-decremental version as a special case, and has a long history with many significant developments just in the past few years. We start by briefly reviewing the work on edge-decremental SSSP, focusing primarily on undirected graphs. The two parameters of interest are the *total update time*, defined as the total time spent by the algorithm on maintaining its data structures over the entire sequence of deletions, and *query time*, defined as the time needed to respond to a single path-query or dist-query. A classic result of Even and Shiloach [10, 11, 18] gives an algorithm that supports exact path-query and dist-query with only $O(mn)$ total update time over all edge deletions, with $O(1)$ query time for dist-query and $O(n)$ query time for path-query. While the $O(mn)$ update time represents a significant improvement over the naive algorithm that simply recomputes a shortest path tree after each edge deletion, it is far from the near-linear total update time results that are known for many other decremental problems in undirected graphs. It remained an open problem for nearly 3 decades to improve upon the update time of the algorithm. Roditty and Zwick [28] highlighted a fundamental obstacle to getting past the $O(mn)$ time barrier using combinatorial approaches, even for unweighted undirected graphs, by showing that the long-standing problem of designing fast combinatorial algorithms for Boolean matrix multiplication can be reduced to the exact edge-decremental SSSP. Furthermore, in a subsequent work, Henzinger *et al.* [17] showed that, assuming online Boolean matrix-vector multiplication conjecture, the $O(mn)$ time barrier for exact edge-decremental SSSP holds even for arbitrary algorithms for the problem. The obstacles identified by these conditional results, however, only apply to supporting **exact** dist-query. Essentially all subsequent work on edge-decremental SSSP has thus focused on the task of supporting approximate path-query and dist-query. In the informal discussion below we assume that the accuracy parameter ϵ is a constant and ignore the dependence of the time bounds on it.

Bernstein and Roditty [6] made the first major progress in breaking the $O(mn)$ update time barrier, by showing an algorithm that supports approximate dist-query in undirected unweighted graphs with $n^{2+o(1)}$ total update time, and $O(1)$ query time. Subsequently, Henzinger, Krinninger, and Nanogkai [15] improved this update time to $O(n^{1.8+o(1)} + m^{1+o(1)})$, and shortly afterwards, the same authors [14] extended it to arbitrary edge lengths and improved it further to an essentially optimal total update time of $O(m^{1+o(1)} \log L)$ where L is the largest length of an edge. All three algorithms are randomized, and moreover, they assume that the edge deletion sequence is given by an oblivious adversary. In particular, for these results to apply, the deletion sequence cannot depend on the responses to queries previously returned by the algorithm. For many applications, it is crucial that the algorithm can handle an *adaptive* adversary, and support path-query. For instance, fast approximation schemes for computing a maximum multicommodity flow in a graph (see, for instance, [12, 13]) rely on a subroutine that can identify an approximate shortest s-t path under suitably chosen edge lengths, and pushing flow along such a path. The edge lengths are then updated for precisely the edges lying on the path; such updates can be modeled by the deletion of edges or vertices on the path. Thus, the edges that are deleted at any step strongly depend on the responses to the approximate path queries from previous steps. Moreover, these applications require that we obtain the actual approximate shortest paths themselves, and not just approximate distances.

The goal of eliminating the oblivious adversary restriction initiated a search for deterministic edge-decremental SSSP algorithms, which, by definition, can handle adaptive deletion sequences. Bernstein and Chechik [4] gave the first deterministic algorithm to break the $O(mn)$ total update time barrier. Their algorithm achieves a total update time of $\tilde{O}(n^2)$ and an $O(1)$ query time for approximate dist-query. In a subsequent work [5], they improved this bound further for sparse graphs, obtaining a total update time of $\tilde{O}(n^{5/4}\sqrt{m}) = \tilde{O}(mn^{3/4})$, keeping the query time of $O(1)$ for approximate dist-query. Both these results required that the underlying graph is undirected and *unweighted*, that is, all edge lengths are unit. In a further progress, Bernstein [3] extended these results to edge-weighted undirected graphs, obtaining a total update time of $\tilde{O}(n^2 \log L)$, where L is the largest edge length, with query time of $O(1)$ for approximate dist-query. While all these results successfully eliminated the oblivious adversary restriction required by the previous works that achieved $o(mn)$ total update time, their approach introduced another limitation: as noted in [3], all three results only support approximate dist-query, but not approximate path-query.

At a high level, the approach used in these results is based on partitioning the edges of the underlying graph into a *light* sub-graph, where the average vertex degree is small and a *heavy* sub-graph, where the degree of each vertex is high, say at least τ. Any shortest s-v path can be decomposed into segments that alternately traverse through the light and the heavy graph. The segments traversing through the light graph are explicitly maintained using the approach of Even and Shiloach [10, 11, 18], exploiting the fact that the edge density is low in the light graph. The segments traversing through the heavy graph, on the other hand, are not maintained explicitly. Instead, it is observed that any shortest s-v path may contain at most $O(n/\tau)$ edges from the heavy graph, so they do not contribute much to the path length. This implicit guarantee on the total length of segments traversing the heavy graph suffices for obtaining an estimate on the shortest path length by only maintaining shortest paths in the light graph. However, it leaves open the task of finding these segments themselves.

Our main technical contribution is to design an algorithm that allows us to support approximate path-query against an adaptive adversary, by explicitly maintaining short paths in the heavy graph. Specifically, we design an algorithm that, given a pair of vertices u, u', that belong to the same connected component C of the heavy graph, returns a short path connecting u to u' in C, where the length of the path is close to the implicit bound that was used in [3, 4].

Formally, assume that we are given a **simple** undirected graph G with a source vertex s and lengths $\ell(e) > 0$ on edges $e \in E(G)$, that undergoes **vertex** deletions. Throughout the algorithm, for every pair u, v of vertices, the distance $\text{dist}(u, v)$ between them is

the length of the shortest path from u to v in the current graph G, using the edge lengths $\ell(e)$. We also assume that we are given an error parameter $0 < \epsilon < 1$. We design an algorithm that supports approximate single-source shortest-path queries, denoted by path-query(v). The query receives as input a vertex v, and returns a path connecting s to v in the current graph G, if such a path exists, such that the length of the path is at most $(1+\epsilon)\,\mathrm{dist}(s,v)$. Our main result for SSSP is summarized in the following theorem.

Theorem 1.1. *There is a randomized algorithm, that, given a parameter $0 < \epsilon < 1$, and a simple undirected n-vertex graph G with lengths $\ell(e) > 0$ on edges $e \in E(G)$, together with a special source vertex $s \in V(G)$, such that G undergoes vertex deletions, supports queries* path-query(v). *For each query* path-query(v), *the algorithm returns a path from s to v in G, if such a path exists, whose length is at most $(1+\epsilon)\,\mathrm{dist}(s,v)$. The algorithm works against an* **adaptive** *adversary. The total expected update time of the algorithm is $O\left(\frac{n^{2+o(1)}\cdot\log^2(1/\epsilon)\cdot\log L}{\epsilon^2}\right)$, where L is the ratio of largest to smallest edge length $\ell(e)$, and each query is answered in expected time $O(n\cdot\mathrm{poly}\log n\cdot\log L\cdot\log(1/\epsilon))$.*

We emphasize that the algorithm is Las Vegas: that is, it always returns a path with the required properties, but its running time is only bounded in expectation. The adversary is allowed to view the complete state of the algorithm, that is, the contents of all its data structures.

One of the main technical contributions of our algorithm is a *core decomposition* of dense graphs. Suppose we are given an n-vertex graph G, such that every vertex in G has degree at least h, where h is sufficiently large, say $h \geq n^{1/\log\log n}$. Informally, a *core* K is an expander-like sub-graph of G, such that every vertex of K has at least $h^{1-o(1)}$ neighbors in K. The "expander-like" properties of the core ensure that, even after $h^{1-o(1)}$ vertex deletions, given any pair u, u' vertices of K, we can efficiently find a short path connecting u to u' in K (the length of the path depends on the balancing of various parameters in our algorithm, and is $n^{o(1)}$). A *core decomposition* of G consists of a collection $K_1,\ldots,K_r$ of disjoint cores in G, such that $r \leq n/h^{1-o(1)}$. Additionally, if we denote by U the set of vertices of G that do not belong to any core, then we require that it is an h-universal set: that is, even after $h^{1-o(1)}$ vertices are deleted from G, every surviving vertex of U can reach one of the cores through a path of length $O(\log n)$. We show a randomized algorithm that with high probability computes a valid core decomposition in a given graph G in time $\tilde{O}(|E(G)| + n^{1+o(1)})$.

While the result above leaves open the question if a similar algorithm can also be obtained for edge-decremental SSSP, for many cut and flow problems on vertex-capacitated graphs, the vertex-decremental SSSP suffices as a building block. We describe next some of these applications. We note here that the idea of using dynamic graph data structures to speed up cut and flow computations is not new. In particular, Madry [24] systematically explored this idea for the maximum multicommodity flow and the concurrent flow problems, significantly improving the previous best known results for them.

Our first application shows that there is an $O(n^{2+o(1)})$-time algorithm for computing approximate maximum s-t flow and minimum s-t cut in vertex-capacitated undirected graphs. For approximate maximum s-t flow problem in edge-capacitated undirected graphs, a sequence of remarkable developments incorporating ideas from continuous optimization to speed-up maximum flow computation has culminated in an $\tilde{O}(m/\epsilon^2)$-time algorithm for computing a $(1+\epsilon)$-approximate flow [8, 20, 22, 27, 29]. We refer the reader to [26] for an excellent survey of these developments. However, no analogous results are known for maximum flow in vertex-capacitated undirected graphs. The main technique for solving the vertex-capacitated version appears to be via standard reduction to the edge-capacitated directed case, and relying on fast algorithms for maximum s-t flow problem in edge-capacitated directed graphs. Two recent breakthrough results for exact maximum s-t flow in edge-capacitated directed graphs include an $\tilde{O}(m\sqrt{n}\log^{O(1)}C)$ time algorithm by Lee and Sidford [23], and an $\tilde{O}(m^{10/7}\log C)$ time algorithm by Madry [25]; here C denotes the largest integer edge capacity. The two bounds are incomparable: the former bound is preferable for dense graphs, and the latter for sparse graphs. For approximate maximum s-t flow problem in edge-capacitated directed graphs, approaches based on the primal-dual framework [12, 13] (or equivalently, the multiplicative weights update method [1]) can be used to compute a $(1+\epsilon)$-approximate s-t flow in $f(n,m,\epsilon)O(m/\epsilon^2)$ time where $f(n,m,\epsilon)$ denotes the time needed to compute a $(1+\epsilon)$-approximate shortest path from s to t. Our approach is based on this connection between approximate shortest path computations and approximate flows, and we obtain the following results.

Theorem 1.2. *There is a randomized algorithm, that, given a simple undirected graph $G=(V,E)$ with capacities $c(v) \geq 0$ on its vertices, a source s, a sink t, and an accuracy parameter $\epsilon \in (0,1]$, computes a $(1+\epsilon)$-approximate maximum s-t flow, and a $(1+\epsilon)$-approximate minimum vertex s-t cut in $O(n^{2+o(1)}/\mathrm{poly}(\epsilon))$ expected time.*

Our proof closely follows the analysis of the primal-dual approach for maximum multicommodity flow problem as presented in [12, 13]; this algorithm simultaneously outputs an approximate s-t flow and an approximate fractional s-t cut. The main primitive needed for this framework is the ability to compute a $(1+\epsilon)$-approximate shortest source-sink paths in a vertex-weighted graph that is undergoing weight increases. We show that Theorem 1.1 can be used to implement these dynamic approximate shortest path computations in $O(n^{2+o(1)}/\mathrm{poly}(\epsilon))$ total expected time. The fractional s-t cut solution can be rounded in $O(m)$ time by using the standard random threshold rounding. The runtime obtained in Theorem 1.2 outperforms previously known bounds in the regime of $m = \omega(n^{1.5+o(1)})$.

Our second application is a new algorithm for approximating vertex sparsest cut in undirected graphs. A *vertex cut* in a graph G is a partition (A,X,B) of its vertices, so that there is no edge from A to B (where A or B may be empty). The *sparsity* of the cut is $\frac{|X|}{\min\{|A|,|B|\}+|X|}$. In the vertex sparsest cut problem, the goal is to compute a vertex cut of minimum sparsity. We prove the following theorem.

Theorem 1.3. *There is a randomized algorithm, that, given a simple undirected graph $G=(V,E)$, computes an $O(\log^4 n)$-approximation to the vertex sparsest cut problem in $O(n^{2+o(1)})$ expected time.*

To establish the above result, it suffices to design an algorithm that runs in $O(n^{2+o(1)})$ expected time, and for any target value α, either finds a vertex cut of sparsity $O(\alpha)$ or certifies that the sparsity of any vertex cut is $\Omega(\alpha/\log^4 n)$. We design such an algorithm by using the cut-matching game of Khandekar, Rao, and Vazirani [21]. The game proceeds in rounds where in each round a bipartition of vertices is given, and the goal is to find a routing from one set to the other with vertex congestion at most $1/\alpha$. This is essentially the vertex-capacitated s-t flow problem, and we can use ideas similar to the one in Theorem 1.2 to solve it. If every round of the cut-matching game can be successfully completed, then we have successfully embedded an expander that certifies that vertex sparsity is $\Omega(\alpha/\log^4 n)$. On the other hand, if any round of the game fails, then we show that we can output a vertex cut of sparsity at most $O(\alpha)$. The run-time of this approach is governed by the time needed to solve the vertex-capacitated maximum s-t flow problem, and we utilize Theorem 1.2 to implement this step in $O(n^{2+o(1)})$ expected time. Alternatively, one can implement the vertex-capacitated maximum s-t flow step using the algorithms for computing maximum s-t flow in edge-capacitated directed graphs in $\tilde{O}(m\sqrt{n})$ time in dense graphs [23], or in $\tilde{O}(m^{10/7})$ time in sparse graphs [25]. Thus an identical approximation guarantee to the one established in Theorem 1.3 can be obtained in $\tilde{O}(\min\{m\sqrt{n}, m^{10/7}\})$ time using previously known results [23, 25]. Another approach for vertex sparsest cut problem is to use the primal-dual framework of Arora and Kale [2] who achieve an $O(\sqrt{\log n})$ approximation to the directed sparsest cut problem in $\tilde{O}(m^{1.5} + n^{2+\epsilon})$ time and an $O(\log n)$-approximation in $\tilde{O}(m^{1.5})$ time. Since directed sparsest cut captures vertex sparsest cut in undirected graphs as a special case, these guarantees also hold for the vertex sparsest cut problem. As before, the runtime obtained in Theorem 1.3 starts to outperform previously known bounds in the regime of $m = \omega(n^{1.5+o(1)})$, albeit achieving a worse approximation ratio than the one achieved in [2].

Subsequent Work. In a follow-up work, Chuzhoy and Saranurak [9] have extended our results to edge-decremental SSSP, obtaining total expected update time $\tilde{O}(n^2 \log L/\epsilon^2)$. They also obtain the first approximate algorithm for edge-decremental All-Pairs Shortest Paths in unweighted undirected graphs with adaptive adversary, whose running time is $n^{o(3)}$.

Organization. We start with an overview of our techniques in Section 2, and preliminaries in Section 3. The proof of Theorem 1.1 is provided in Section 4. Due to lack of space, some of the proofs, including those of Theorems 1.2 and 1.3, are deferred to the full version of the paper.

2 OVERVIEW OF THE PROOF OF THEOREM 1.1

We now provide an overview of our main result, namely, the proof of Theorem 1.1. This informal overview is mostly aimed to convey the intuition; in order to simplify the discussion, the values of some of the parameters and bounds in this overview are given imprecisely. As much of the previous work in this area, our results use the classical Even-Shiloach trees [10, 11, 18] as a building block. Given a graph G with integral edge lengths, that is subject to edge deletions, a source vertex s, and a distance bound D, the Even-Shiloach Tree data structure, that we denote by ES-Tree(G, s, D), maintains a shortest-path tree T of G, rooted at s, up to distance D. In other words, a vertex $v \in V(G)$ belongs to T iff $\text{dist}_G(s, v) \leq D$, and for each such vertex v, $\text{dist}_T(s, v) = \text{dist}_G(s, v)$. The total update time of the algorithm is $O(|E(G)| \cdot D \cdot \log n)$. In addition to maintaining the shortest-path tree T, the data structure stores, with every vertex $v \in V(T)$, the value $\text{dist}_G(s, v)$.

At a high level, our algorithm follows the framework of [3, 4]. Using standard techniques, we can reduce the problem to a setting where we are given a parameter $D = \Theta(n/\epsilon)$, and we only need to correctly respond to path-query(v) if $D \leq \text{dist}(s, v) \leq 4D$; otherwise we can return an arbitrary path, or no path at all. Let us assume first for simplicity that all edges in the graph G have unit length. In [3, 4], the algorithm proceeds by selecting a threshold $\tau \approx \frac{n}{\epsilon D}$, and splitting the graph G into two subgraphs, a sparse graph G^L, called the *light graph*, and a dense graph G^H, called the *heavy graph*. In order to do so, we say that a vertex $v \in V(G)$ is *heavy* if the degree of v, $d(v) \geq \tau$, and it is *light* otherwise. Graph G^L contains all vertices of G and all edges $e = (u, v)$, such that at least one of u, v is a light vertex; notice that $|E(G^L)| \leq n\tau \leq O(n^2/\epsilon D)$. Graph G^H contains all heavy vertices of G, and all edges connecting them. The algorithm also maintains the *extended light graph* $\hat{G}^L$, that is obtained from G^L, by adding, for every connected component C of G^H, a vertex v_C to $\hat{G}^L$, and connecting it to every heavy vertex $u \in C$ with an edge of weight 1/2. So, in a sense, in $\hat{G}^L$, we create "shortcuts" between the heavy vertices that lie in the same connected component of G^H. The crux of the algorithm consists of two observations: (i) for every vertex v, $\text{dist}_G(s, v) \approx \text{dist}_{\hat{G}^L}(s, v)$; and (ii) since graph $\hat{G}^L$ is sparse, we can maintain, for every vertex $v \in G$ with $\text{dist}_G(s, v) \leq 4D$, the distances $\text{dist}_{\hat{G}^L}(s, v)$ in total update time $O(n^2/\epsilon)$. In order to see the latter, observe that $|E(\hat{G}^L)| \leq |E(G^L)| + O(n) \leq O(n^2/\epsilon D)$. We can use the data structure ES-Tree$(\hat{G}^L, s, D)$, with total update time $O(|E(\hat{G}^L)| \cdot D \cdot \log n) = O(n^2 \log n/\epsilon)$ (in fact, the threshold τ was chosen to ensure that this bound holds). In order to establish (i), observe that graph $\hat{G}^L$ is obtained from graph G, by "shortcutting" the edges of G^H, and so it is not hard to see that $\text{dist}_{\hat{G}^L}(s, v) \leq \text{dist}_G(s, v)$ for all $v \in V(G)$. The main claim is that $\text{dist}_G(s, v) \leq (1 + \epsilon)\,\text{dist}_{G^L}(s, v)$, and in particular that for any path P in $\hat{G}^L$ connecting the source s to some vertex $v \in V(G)$, there is a path P' in G connecting s to v, such that the length of P' is at most the length of P plus ϵD. Assuming this is true, it is easy to verify that for every vertex v with $D \leq \text{dist}(s, v) \leq 4D$, $\text{dist}_G(s, v) \leq \text{dist}_{\hat{G}^L}(s, v)(1 + \epsilon)$, and so it is sufficient for the algorithm to report, as an answer to a query dist-query(v), the value $\text{dist}_{\hat{G}^L}(s, v)$, which is stored in ES-Tree$(\hat{G}^L, s, D)$. Consider now some path P in $\hat{G}^L$, and let C be any connected component of G^H, such that $v_C \in P$. Let u, u' be the vertices of the original graph G appearing immediately before and immediately after v_C in P. Let $Q(u, u')$ be the shortest path connecting u to u' in the heavy graph G^H. As every vertex in G^H is heavy, the length of $Q(u, u')$ is bounded by $4|V(C)|/\tau$: indeed, assume that $Q(u, u') = (u = u_0, u_1, \ldots, u_r = u')$, and let $S = \{u_i \mid i = 1 \mod 4\}$ be a subset of vertices on $Q(u, u')$. Then for every pair u_i, u_j of distinct vertices in S, the set of their neighbors must be disjoint (or we could shorten the path $Q(u, u')$ by

connecting u_i to u_j through their common neighbor). Since every vertex in G^H has at least τ neighbors, $|S| \leq |V(C)|/\tau$, and so $Q(u,u')$ may contain at most $4|V(C)|/\tau$ vertices. Once we replace each such vertex v_C on path P with a path connecting the corresponding pair u,u' of vertices in the original graph, the length of P increases by at most $\sum_{C:v_C\in P} 4|V(C)|/\tau \leq 4n/\tau = O(\epsilon D)$. This argument allows the algorithms of [3, 4] to maintain approximate distances from the source s to every vertex of G, by simply maintaining the data structure ES-Tree$(\hat{G}^L, s, D)$. However, in order to recover the **path** connecting s to the given query vertex v in G, we should be able to compute all required paths in the heavy graph G^H. Specifically, we need an algorithm that allows us to answer queries path-query(u,u',C): given a connected component C of G^H, and a pair u,u' of vertices of C, return a path connecting u to u' in C, whose length is at most $O(|V(C)|/\tau)$. The main contribution of this work is an algorithm that allows us to do so, when the input graph G is subject to **vertex** deletions. (We note that for technical reasons, the value τ in our algorithm is somewhat higher than in the algorithms of [3, 4], which translates to somewhat higher running time $O(n^{2+o(1)}\log^2(1/\epsilon)/\epsilon^2)$, where $o(1) = \Theta(1/\log\log n)$).

A first attempt at a solution. For simplicity of exposition, let us assume that all vertices in the heavy graph G^H have approximately the same degree (say between h and $2h$, where $h \geq \tau$ is large enough, so, for example, $h \geq n^{1/\log\log n}$), so the number of edges in G^H is $O(hn)$. Using the same argument as before, for every connected component C in G^H, and every pair $u,u' \in V(C)$ of its vertices, there is a path connecting u to u' in C, of length $O(|V(C)|/h)$; we will attempt to return paths whose lengths are bounded by this value in response to queries. A tempting simple solution to this problem is the following: for every connected component C of G^H, select an arbitrary vertex $s(C)$ to be its source, and maintain the ES-Tree$(C, s(C), D(C))$ data structure, for the distance bound $D(C) \approx |V(C)|/h$. Such a tree can be maintained in total time $\tilde{O}(|E(C)|\cdot|V(C)|/h)$, and so, across all components of G^H, the total update time is $\tilde{O}(|E(G^H)|n/h) = \tilde{O}(n^2)$. Whenever a query path-query(u,u',C) arrives, we simply concatenate the path connecting u to $s(C)$ and the path connecting u' to $s(C)$ in the tree ES-Tree$(C, s(C), D(C))$; using the same argument as before, it is easy to show that the resulting path is guaranteed to be sufficiently short. Unfortunately, this solution does not seem to work in the case where the adversary is adaptive, since the adversary may repeatedly delete edges incident to $s(C)$ until it becomes isolated (in the vertex-decremental setting, neigbhors of $s(C)$ are repeatedly deleted). In order to get around this issue, Henzinger et al. [16] introduced *moving* ES-trees; but unfortunately the running time in their approach is prohibitive for us.

Solution: core decomposition. A natural approach to overcome this difficulty is to create, in every connected component C of G^H a "super-source", that would be difficult to disconnect from the rest of the component C. This motivates the notion of *cores* that we introduce. Recall that for the sake of exposition, we have assumed that the degrees of all vertices in G^H are between h and $2h$, where $h \geq n^{1/\log\log n}$. Intuitively, a core is a highly-connected graph. For example, a good core could be an expander graph K, such that every vertex $v \in V(H)$ has many neighbors in K (say, at least $h/n^{o(1)}$). If we use a suitable notion of expander, this would ensure that, even after a relatively long sequence of vertex deletions (say up to $h/n^{o(1)}$), every pair of vertices in K has a short path connecting them. Intuitively, we would like to use the core as the "super-source" of the ES-Tree structure. Unfortunately, the bad scenario described above may happen again, and the adversary can iteratively delete vertices in order to isolate the core from the remainder of the graph. To overcome this difficulty, we use the notion of *core decomposition*. A core decomposition is simply a collection of disjoint cores in G^H, but it has an additional important property: If U is the set of all vertices of G^H that do not lie in any of the cores, then it must be an *h-universal set*: namely, after a sequence of up to $h/n^{o(1)}$ deletions of vertices from G^H, each remaining vertex of U should be able to reach one of the cores using a short path (say, of length at most $\text{poly}\log n$). Our algorithm then uses the cores as the "super-source", in the following sense. We construct a new graph $\tilde{G}$, by starting from G^H and contracting every core K into a super-node $z(K)$. We also add a new source vertex s, that connects to each resulting super-node. Our algorithm then maintains ES-Tree$(\tilde{G}, s, \text{poly}\log n)$, that allows us to quickly recover a short path connecting any given vertex in G^H to some core. The main technical contribution of this paper is an algorithm that computes a core decomposition in time $\tilde{O}(|E(G^H)| + n^{1+o(1)})$. Before we discuss the core decomposition, we quickly summarize how our algorithm processes shortest-path queries, and provide a high-level analysis of the total update time.

Responding to queries. Recall that in path-query(u,u',C), we are given a connected component C of G^H, and a pair u,u' of its vertices. Our goal is to return a path connecting u to u', whose length is at most $O(|V(C)|/\tau)$; in fact we will return a path of length at most $O(|V(C)|/h)$. Recall that for every core K, we require that every vertex $v \in K$ has at least $h/n^{o(1)}$ neighbors in K, and that all cores in the decomposition are disjoint. Therefore, the total number of cores contained in C is at most $|V(C)|n^{o(1)}/h$. We will maintain a simple spanning forest data structure in graph G^H that allows us, given a pair u,u' of vertices that belong to the same connected component C of G^H, to compute an arbitrary path P connecting u to u' in C. Next, we *label* every vertex w of P with some core K: if w belongs to a core K, then the label of w is K; otherwise, the label of w is any core K, such that w can reach K via a short path (of length $\text{poly}\log n$). The labeling is performed by exploiting the ES-Tree$(\tilde{G}, s, \text{poly}\log n)$ data structure described above. Once we obtain a label for every vertex on the path P, we "shortcut" the path through the cores: if two non-consecutive vertices of P have the same label K, then we delete all vertices lying on P between these two vertices, and connect these two vertices via the core K. As the number of cores in C is at most $|V(C)|n^{o(1)}/h$, eventually we obtain a path connecting u to u', whose length is $|V(C)|n^{o(1)}\,\text{poly}\log n/h = |V(C)|n^{o(1)}/h$, as required.

Running time analysis. As already mentioned, our algorithm for computing the core decomposition takes time $\tilde{O}(|E(G)|+n^{1+o(1)}) = O(n^{1+o(1)}h)$; it seems unlikely that one can hope to obtain an algorithm whose running time is less than $\Theta(|E(G^H)|) = \Theta(nh)$. Our core decomposition remains "functional" for roughly $h/n^{o(1)}$ iterations, that is, as long as fewer than $h/n^{o(1)}$ vertices are deleted. Once we delete $h/n^{o(1)}$ vertices from the graph, we are no longer guaranteed that pairs of vertices within the same core have short paths connecting them (in fact they may become disconnected),

and we are no longer guaranteed that the vertices of U can reach the cores via short paths. Therefore, we partition our algorithm into phases, where every phase consists of the deletion of up to $h/n^{o(1)}$ vertices. Once $h/n^{o(1)}$ vertices are deleted, we recompute the core decomposition, the graph $\tilde{G}$, and the ES-Tree$(\tilde{G}, s, \text{poly}\log n)$ data structure that we maintain. Note that, since G has n vertices, the number of phases is bounded by $n^{1+o(1)}/h$, and recall that we spend $O(n^{1+o(1)}h)$ time per phase to recompute the core decomposition. Therefore, the total update time of the algorithm is $O(n^{2+o(1)})$ (we have ignored multiplicative factors that depend on ϵ).

Why our algorithm only handles vertex deletions. As mentioned above, it is unlikely that we can compute a core decomposition in less than $\Theta(|E(G^H)|) = \Theta(nh)$ time. If our goal is an algorithm for SSSP with total update time of $O(n^{2+o(1)})$, then we can afford at most $O(n^{1+o(1)}/h)$ computations of the core decomposition. If we allow edge deletions, this means that a phase may include up to roughly $h^2/n^{o(1)}$ edge deletions, since $|E(G^H)| = \Theta(nh)$. Since the degrees of the vertices are between h and $2h$, the cores cannot handle that many edge deletions, as they can cause an expander graph to become disconnected, or some vertices of U may no longer have short paths connecting them to the cores. However, in the vertex-deletion model, we only need to tolerate the deletion of up to roughly $h/n^{o(1)}$ vertices per phase, which we are able to accommodate, as the degrees of all vertices are at least h.

The core decomposition. The main technical ingredient of our algorithm is the core decomposition. In the vertex-deletion model, it is natural to define a core K as a *vertex expander*: that is, for every vertex-cut (X, Y, Z) in K (so no edges connect X to Z in K), $|Y| \geq \min\{|X|, |Z|\}/n^{o(1)}$ must hold. Additionally, as mentioned above, we require that every vertex in K has at least $h/n^{o(1)}$ neighbors that lie in K. Unfortunately, these requirements appear too difficult to fulfill. For instance, a natural way to construct a core-decomposition is to iteratively decompose the graph G^H into connected clusters, by computing, in every current cluster R, a sparse vertex cut (X, Y, Z), and then replacing R with two new graphs: $R[X \cup Y]$ and $R[Y \cup Z]$. We can continue this process, until every resulting graph is a vertex expander. Unfortunately, this process does not ensure that the resulting cores are disjoint, or that every vertex in a core has many neighbors that also belong to the core. Moreover, even if all pairs of vertices within a given core K have short paths connecting them, it is not clear how to recover such paths, unless we are willing to spend $O(|E(K)|)$ time on each query. Therefore, we define the cores somewhat differently, by using the notion of a *core structure*. Intuitively, a core structure consists of two sets of vertices: set K of vertices – the core itself, and an additional set $U(K)$ of at most $|K|$ vertices, called the *extension of the core*. We will ensure that all core-sets K are disjoint, but the extension sets may be shared between the cores. Additionally, we are given a sub-graph G^K of G^H, whose vertex set is $K \cup U(K)$. We will ensure that all such sub-graphs are "almost" disjoint in their edges, in the sense that every edge of G^H may only belong to at most $O(\log n)$ such graphs, as this will be important in the final bound on the running time. Finally, the core structure also contains a *witness graph* W^K - a sparse graph, that is a $1/n^{o(1)}$-expander (in the usual edge-expansion sense), whose vertex set includes every vertex of K, and possibly some additional vertices from $U(K)$. We also compute an *embedding* of W^K into G^K, where each edge $e = (u, v) \in E(W^K)$ is mapped to some path P_e in G^K, connecting u to v, such that all such paths P_e are relatively short, and they cause low vertex-congestion in G^K. The witness graph W^K and its embedding into G^K allow us to quickly recover short paths connecting pairs of vertices in the core K.

One of the main building blocks of our core decomposition algorithm is an algorithm that, given a subgraph H of G, either computes a sparse and almost balanced vertex-cut in H, or returns a core containing most vertices of H. The algorithm attempts to embed an expander into H via the cut-matching game of [21]. If it fails, then we obtain a sparse and almost balanced vertex-cut in H. Otherwise, we embed a graph W into H, that is with high probability an expander. Graph W then serves as the witness graph for the resulting core. The cut-matching game is the only randomized part of our algorithm. If it fails (which happens with low probability), then one of the queries to the heavy graph may return a path whose length is higher than the required threshold (that is known to the algorithm). In this case, we simply recompute all our data structures from scratch. This ensures that our algorithm always returns a correct approximate response to path-query, with the claimed expected running time, and is able to handle an adaptive adversary.

Handling arbitrary vertex degrees. Recall that so far we have assumed that all vertices in G^H have similar degrees. This was necessary because, if some vertices of G^H have low degrees (say d), but $|E(G^H)|$ is high (say $O(nh)$ for some $h \gg d$), then we would be forced to recompute the core decomposition very often, every time that roughly d vertices are deleted, while each such computation takes at least $\Theta(n^{1+o(1)}h)$ time, resulting in a total running time that is too high. To overcome this difficulty, we partition the heavy graph G^H into graphs $\Lambda_1, \ldots, \Lambda_r$ that we call *layers*, where for each $1 \leq i \leq r$, all vertices in graph Λ_i have degree at least h_i, while $|E(\Lambda_i)| \leq n^{1+o(1)}h_i$. We ensure that $h_1 \geq h_2 \geq \ldots, \geq h_r$, and that these values are geometrically decreasing. We maintain a core decomposition for each such graph Λ_i separately. For all $1 \leq i \leq r$, roughly every $h_i/n^{o(1)}$ vertex deletions, we recompute the layers $\Lambda_i, \ldots, \Lambda_r$, and their corresponding core decompositions.

Handling arbitrary edge lengths. So far we have assumed that all edge lengths are unit. When the edge lengths are no longer the same, we need to use the approach of [3]. We partition all edges into classes, where class i contains all edges whose length is between 2^i and 2^{i+1}. Unfortunately, we can no longer use the same threshold τ for the definition of the heavy and the light graph for all edge lengths. This is since we are only guaranteed that, whenever two vertices u, u' belong to the same connected component C of G^H, there is a path containing at most $|V(C)|/\tau$ edges connecting u to u' in C. But as some edges may now have large length, the actual length of this path may be too high. Following [3], we need to define different thresholds τ_i for each edge class i, where roughly $\tau_i = \tau \cdot 2^i$, for the original threshold τ. This means that graph $\hat{G}^L$ may now become denser, as it may contain many edges from classes i where i is large. We use the Weight-Sensitive-Even-Shiloach data structure of [3] in order to handle $\hat{G}^L$.

3 PRELIMINARIES

We follow standard graph theoretic-notation. All graphs in this paper are undirected, unless explicitly said otherwise. Graphs may

have parallel edges, except for simple graphs, that cannot have them. Given a graph $G = (V, E)$ and two disjoint subsets A, B of its vertices, we denote by $E_G(A, B)$ the set of all edges with one endpoint in A and another in B, and by $E_G(A)$ the set of all edges with both endpoints in A. We also denote by $\text{out}_G(A)$ the set of all edges with exactly one endpoint in A. We may omit the subscript G when clear from context. Given a subset $S \subseteq V$ of vertices of G, we denote by $G[S]$ the sub-graph of G induced by S.

Decremental Connectivity/Spanning Forest. We use the results of [19], who provide a deterministic data structure, that we denote by CONN-SF(G), that, given an n-vertex unweighted undirected graph G, that is subject to edge deletions, maintains a spanning forest of G, with total running time $O((m+n)\log^2 n)$, where $n = |V(G)|$ and $m = |E(G)|$. Moreover, the data structure supports connectivity queries: given a pair u, v of vertices of G, return "yes" if u and v are connected in G, and "no" otherwise. The running time to respond to each such query is $O(\log n/\log\log n)$; we denote by $\text{conn}(G, u, u')$ the connectivity query for u and u' in the data structure. Since the data structure maintains a spanning forest for G, we can also use it to respond to a query $\text{path}(G, u, v)$: given two vertices u and v in G, return any simple path connecting u to v in G if such a path exists, and return $\emptyset$ otherwise. If u and v belong to the same connected component C of G, then the running time of the query is $O(|V(C)|)$.

Even-Shiloach Trees [10, 11, 18]. Suppose we are given a graph $G = (V, E)$ with integral lengths $\ell(e) \geq 1$ on its edges $e \in E$, a source s, and a distance bound $D \geq 1$. Even-Shiloach Tree (ES-Tree) is a deterministic data structure that maintains, for every vertex v with $\text{dist}(s, v) \leq D$, the distance $\text{dist}(s, v)$, under the deletion of edges from G. Moreover, it maintains a shortest-path tree from vertex s, that includes all vertices v with $\text{dist}(s, v) \leq D$. We denote the corresponding data structure by ES-Tree(G, s, D). The total running time of the algorithm, including the initialization and all edge deletions, is $O(m \cdot D \log n)$, where $m = |E|$.

Low-Degree Pruning Procedure. Procedure Deg-Prune(H, d) takes as input a simple graph H and an integer d, and computes a partition (J_1, J_2) of $V(H)$, as follows: start with $J_1 = \emptyset$ and $J_2 = V(H)$. While there is a vertex $v \in J_2$, such that fewer than d neighbors of v lie in J_2, move v from J_2 to J_1. The procedure can be implemented to run in time $O(|E(H)| + |V(H)|)$. Moreover, it is not hard to show that the total number of edges incident to the vertices of J_1 is $O(d|J_1|)$, and that for any other partition (J', J'') of $V(H)$, such that the degree of every vertex in $H[J'']$ is at least d, $J'' \subseteq J_2$ must hold. The proof is deferred to the full version of the paper.

4 DECREMENTAL SINGLE-SOURCE SHORTEST PATHS

This section contains the proof of Theorem 1.1, with some details deferred to the full version of the paper. Using standard arguments, it suffices to prove the following theorem.

Theorem 4.1. *There is a randomized algorithm, that, given parameters $0 < \epsilon < 1$ and $D > 0$, and a simple undirected n-vertex graph G with lengths $\ell(e) > 0$ on edges $e \in E(G)$, together with a special source vertex $s \in V(G)$, such that G undergoes vertex deletions, supports queries $\text{path-query}_D(v)$. For each query $\text{path-query}_D(v)$, the algorithm returns a path from s to v in G, of length is at most $(1+\epsilon)\,\text{dist}(s, v)$, if $D \leq \text{dist}(s, v) \leq 2D$; otherwise, it either returns an arbitrary path connecting s to v, or correctly establishes that $\text{dist}(s, v) > 2D$. The algorithm works against an adaptive adversary. The total expected running time is $O\left(\frac{n^{2+o(1)}\cdot\log^2(1/\epsilon)}{\epsilon^2}\right)$, and each query is answered in expected time $O(n \operatorname{poly}\log n \log(1/\epsilon))$.*

We now focus on the proof of Theorem 4.1. Throughout the proof, we denote by G the current graph, obtained from the input graph after the current sequence of vertex deletions, and n is the number of vertices present in G at the beginning of the algorithm. A standard transformation can be used to ensure that $D = \lceil 4n/\epsilon \rceil$, and that all edge lengths are integers between 1 and $2D$. At a very high level, our proof follows the algorithm of [3]. We partition all edges of G into $\lambda = \lfloor \log(4D) \rfloor$ classes $E_1, \ldots, E_\lambda$, where for $1 \leq i \leq \lambda$, edge e belongs to E_i iff $2^i \leq \ell(e) < 2^{i+1}$. Next, for each $1 \leq i \leq \lambda$, we define a threshold value τ_i. For technical reasons, these values are somewhat different from those used in [3]. In order to define τ_i, we need to introduce a number of parameters that we will use throughout the algorithm. First, we let $\alpha^* = 1/2^{3\sqrt{\log n}}$ – this will be the expansion parameter for the cores. The second parameter is $\ell^* = \frac{16c^*\log^{12} n}{\alpha^*}$, for some large enough constant c^*. This parameter will serve as an upper bound on the lengths of paths between pairs of vertices in a core. Observe that $\ell^* = 2^{O(\sqrt{\log n})}$. Our third main parameter is $\Delta = 128c^*\log^{20} n/\alpha^* = 2^{O(\sqrt{\log n})} = n^{o(1)}$. This parameter will be used in order to partition the algorithm into phases. Lastly, For each $1 \leq i \leq \lambda$, we let τ_i be the maximum between $4n^{2/\log\log n}$ and $\frac{n}{\epsilon D}\cdot 2^{20}\cdot\ell^*\cdot\Delta\cdot\log^4 n\cdot\lambda\cdot 2^i$. Notice that $\tau_i = \max\left\{n^{o(1)}, \frac{n^{1+o(1)}\cdot 2^i\cdot\log D}{\epsilon D}\right\}$.

Bernstein [3] used the threshold values τ_i in order to partition the edges of G into two subsets, which are then used to define two graphs: a light graph and a heavy graph. We proceed somewhat differently. First, it would be more convenient for us to define a separate heavy graph for each edge class, though we still keep a single light graph. Second, our process of partitioning the edges between the heavy graphs and the light graph is somewhat different from that in [3]. However, we still ensure that for each $1 \leq i \leq \lambda$, the light graph may contain at most $n\tau_i$ edges of E_i throughout the algorithm; this is a key property that the algorithm of [3] exploits.

Fix an index $1 \leq i \leq \lambda$, and let G_i be the sub-graph of G induced by the edges in E_i. We run Procedure Deg-Prune on graph G_i and degree threshold $d = \tau_i$. Recall that the procedure computes a partition (J', J'') of $V(G_i)$, by starting with $J' = \emptyset$ and $J'' = V(G_i)$, and then iteratively moving from J'' to J' vertices v whose degree in $G_i[J'']$ is less than d. The procedure can be implemented to run in time $O(|E_i| + n)$. We say that the vertices of J' are *light for class i*, and the vertices of J'' are *heavy for class i*. We now define the graph G_i^H – the heavy graph for class i, as follows. The set of vertices of G_i^H contains all vertices that are heavy for class i. The set of edges contains all edges of E_i whose **both** endpoints are heavy for class i. We also define a light graph G_i^L for class i, though we will not use it directly. Its vertex set is $V(G)$, and the set of edges contains all edges of E_i that do not belong to graph G_i^H. Clearly, every edge of G_i^L is incident to at least one vertex that is light for class i, and it is easy to verify that $|E(G_i^L)| \leq n\tau_i$. As the algorithm progresses and vertices are deleted from G, some vertices that are heavy for class i may

become light for it (this happens when a vertex v that is currently heavy for class i has fewer than τ_i neighbors that are also heavy for class i). Once a vertex v becomes light for class i, every edge in G_i^H that is incident to v is removed from G_i^H and added to G_i^L, and v is deleted from G_i^H. In particular, $E(G_i^H)$ and $E(G_i^L)$ always define a partition of the current set E_i of edges. Moreover, it is easy to verify that the total number of edges that are ever present in G_i^L is bounded by $n\tau_i$, and that, throughout the algorithm, every vertex of G_i^H has degree at least τ_i in G_i^H. The main technical contribution of this paper is the next theorem, that allows us to deal with the heavy graphs.

Theorem 4.2. *There is a randomized algorithm, that, for a fixed index $1 \leq i \leq \lambda$, supports queries* path-query(u, v, C)*: given two vertices u and v that belong to the same connected component C of graph G_i^H, return a path, connecting u to v in C, that contains at most $2^{13}\frac{|V(C)|}{\tau_i} \cdot \Delta \cdot \ell^* \cdot \log^4 n$ edges. The total expected update time of the algorithm is $O(n^{2+o(1)})$, and each query* path-query(u, v, C) *is processed in expected time $O(|V(C)| \log^4 n)$. The algorithm works against an adaptive adversary.*

With the above theorem in hand, a simple modification of the approach used in [3] suffices to obtain Theorem 4.1. We defer these details to the final version of the paper and focus on the proof of Theorem 4.2 in the remainder of this section. In order to simplify the notation, we will denote the graph G_i^H by G^* from now on. We also denote $\tau = \tau_i$, and we will use the fact that $\tau \geq 4n^{2/\log\log n}$, and that every vertex in G^* has degree at least τ. The central notions that we use in our proof are those of a *core structure* and a *core decomposition*. Our algorithm will break the graph G^* into sub-graphs and will compute a core decomposition in each such subgraph. In every subgraph that we will consider, the degrees of all vertices are at least $n^{1/\log\log n}$. In the following subsections we define core structures and a core decomposition and develop the technical machinery that we need to construct and maintain them.

4.1 Core Structures and Cores

In this subsection, we define core structures and cores, that play a central role in our algorithm. We also establish some of their properties, and provide an algorithm that computes short paths between a given pair of vertices of a core.

Throughout this subsection, we will assume that we are given some graph, that we denote by $\hat{G}$, that is a subgraph of our original n-vertex graph G. Therefore, throughout this subsection, we assume that $|V(\hat{G})| \leq n$. We also assume that we are given a parameter $h > n^{1/\log\log n}$, and that every vertex in $\hat{G}$ has degree at least h.

Definition. *Given a graph $\hat{G}$ with $|V(\hat{G})| \leq n$, a* core structure *$\mathcal{K}$ in $\hat{G}$ consists of the following four ingredients: (i) two disjoint vertex sets: a set $K \neq \emptyset$ of vertices, that we refer to as the core itself, and a set $U(K)$ of at most $|K|$ vertices, that we call the* extension *of K; (ii) a connected subgraph $\hat{G}^K \subseteq \hat{G}[K \cup U(K)]$, with $V(\hat{G}^K) = K \cup U(K)$; (iii) a graph W^K, that we refer to as the* witness graph *for K, with $K \subseteq V(W^K) \subseteq K \cup U(K)$, such that the maximum vertex degree of W^K is at most $\log^3 n$; and (iv) for every edge $e = (x, y) \in E(W^K)$, a path $P(e)$ in $\hat{G}^K$, that connects x to y, such that every path in set $\Psi(W^K) = \left\{P(e) \mid e \in E(W^K)\right\}$ contains at most $c^* \log^8 n$ vertices (here c^* is the constant that appears in the definition of ℓ^*); and every vertex of $\hat{G}^K$ participates in at most $c^* \log^{19} n$ paths of $\Psi(W^K)$. If, additionally, W^K is an α^*-expander, then we say that $\mathcal{K} = (K, U(K), G^K, W^K)$ is a* perfect core structure.

We call the set $\Psi(W^K) = \left\{P(e) \mid e \in E(W^K)\right\}$ of paths the *embedding of W^K into $\hat{G}^K$*, and we view this embedding as part of the witness graph W^K.

Definition. *We say that a core structre $\mathcal{K} = (K, U(K), G^K, W^K)$ is an h-core structure iff every vertex $v \in K$ of the core has degree at least $h/(32 \log n)$ in G^K. A perfect core structure with this property is called a* perfect h-core structure.

The following theorem allows us to compute short paths between pairs of vertices within a core. The proof is deferred to the full version of the paper.

Theorem 4.3. *There is a deterministic algorithm, that, given a graph $\hat{G}$ with $|V(\hat{G})| \leq n$, undergoing at most h/Δ vertex deletions, and an h-core structure $\mathcal{K} = (K, U(K), \hat{G}^K, W^K)$ in $\hat{G}$, supports queries* core-path(u, v) *for pairs $u, v \in K$ of vertices. Given such a query, the algorithm either returns a path from u to v in the current graph $\hat{G}^K$, of length at most ℓ^*, or correctly determines that $\mathcal{K}$ is not a perfect core structure, (that is, W^K is not an α^*-expander). The total running time of the algorithm is $O(|E(\hat{G}^K)| \operatorname{poly}\log n)$, and the total time to process each query is $O(\ell^* + |K| \log^3 n)$.*

We emphasize that, if the core structure $\mathcal{K}$ that serves as input to Theorem 4.3 is a perfect h-core structure, then the algorithm is guaranteed to return a path from u to v of length at most ℓ^* in the current graph $\hat{G}^K$.

4.2 Core Decomposition

In addition to core structures, our second main tool is a core decomposition. In this subsection we define core decompositions and we state a theorem that allows us to compute them. Before we define a core decomposition, we need to define an h-universal set of vertices.

Definition. *Suppose we are given a subgraph $\hat{G} \subseteq G$, and a set S of its vertices. Let J be another subset of vertices of $\hat{G}$. We say that J is an h-*universal set with respect to *S iff for every vertex $u \in J$ and for every subset R of at most h/Δ vertices of $\hat{G} \setminus \{u\}$, there is a path in $\hat{G}[S \cup J] \setminus R$, connecting u to a vertex of S, whose length is at most $\log n$.*

Finally, we are ready to define a core decomposition.

Definition. *An h-*core decomposition *of a graph $\hat{G}$ with $|V(\hat{G})| \leq n$ is a collection $\mathcal{F} = \left\{(K_i, U(K_i), \hat{G}^{K_i}, W^{K_i})\right\}_{i=1}^{r}$ of h-core structures in $\hat{G}$, such that $K_1, \ldots, K_r$ are mutually disjoint (but a vertex $v \in V(\hat{G})$ may belong to a number of extension sets $U(K_i)$, in addition to belonging to some core K_j), and every edge of $\hat{G}$ participates in at most $\log n$ graphs $\hat{G}^{K_1}, \ldots, \hat{G}^{K_r}$. Additionally, if we denote $\tilde{K} = \bigcup_{i=1}^{r} K_i$ and $J = V(\hat{G}) \setminus \tilde{K}$, then set J is h-universal with respect to $\tilde{K}$. We say that this decomposition is a* perfect h-core decomposition *iff every core structure in $\mathcal{F}$ is a perfect h-core structure.*

The main building block of our algorithm is the following theorem, whose proof is deferred to full version of the paper.

Theorem 4.4. *There is a randomized algorithm, that, given a sub-graph $\hat{G} \subseteq G$ and a parameter $h \geq n^{1/\log\log n}$, such that every vertex of $\hat{G}$ has degree at least h in $\hat{G}$, computes an h-core decomposition of $\hat{G}$. Moreover, with high probability, the resulting core decomposition is perfect. The running time of the algorithm is $O((|E(\hat{G})| + |V(\hat{G})|^{1+o(1)})\operatorname{poly}\log n)$.*

4.3 Completing the Proof of Theorem 4.2

We use the parameter Δ defined in previous subsections; recall that $\Delta = 128c^* \log^{20} n/\alpha^* = 2^{O(\sqrt{\log n})} = n^{o(1)}$. We start with a high-level intuition to motivate our next steps. Consider the graph $G^* = G_i^H$, and let d be its average vertex degree. For simplicity, assume that $d = \Delta^j$ for some integer j. Let us additionally assume, for now, that the degree of every vertex in G^* is at least $h = \Delta^{j-1}$ (this may not be true in general). We can then compute an h-core decomposition $\mathcal{F}$ of G^* using Theorem 4.4. Note that, as long as we delete fewer than h/Δ vertices from G^*, the current core decomposition remains functional, in the following sense: for every core structure $\mathcal{K} = (K, U(K), (G^*)^K, W^K) \in \mathcal{F}$, for every pair $u, v \in K$ of vertices in the core that were not deleted yet, we can use Theorem 4.3 to compute a path of length at most ℓ^* between u and v; and for every vertex w of G^* that does not lie in any core K, there is a path of length at most $\log n$ connecting it to some core, from the definition of the h-universal set. Both these properties are exploited by our algorithm in order to respond to queries path-query. Note that computing the core decomposition takes time $O((|E(G^*)| + n^{1+o(1)})\operatorname{poly}\log n) = O(n^{1+o(1)}\Delta^j)$, and the total time required to maintain the data structures from Theorem 4.3 for every core is also bounded by this amount, since every edge of G^* may belong to at most $\log n$ graphs $(G^*)^K$, where K is a core in the decomposition. We can partition the algorithm into phases, where in every phase, $h/\Delta = \Delta^{j-2}$ vertices are deleted from G^*. Once a phase ends, we recompute the core decomposition. Since the number of phases is bounded by n/Δ^{j-2}, and the total running time within each phase is $O(n^{1+o(1)}\Delta^j)$, the total running time of the algorithm would be at most $n^{2+o(1)}$, as required. The main difficulty with this approach is that some vertices of G^* may have degrees that are much smaller than the average vertex degree. Even though we could still compute the core decomposition, we are only guaranteed that it remains functional for a much smaller number of iterations – the number that is close to the smallest vertex degree in G^*. We would then need to recompute the core decomposition too often, resulting in a high running time.

In order to overcome this difficulty, we partition the vertices of G^* into "layers". Let z_1 be the smallest integer, such that the maximum vertex degree in G^* is less than Δ^{z_1}, and let z_2 be the largest integer, such that $\Delta^{z_2} < \tau/(32\log n)$. Let $r = z_1 - z_2$, so $r \leq \log n$. We emphasize that the values z_1, z_2 and r are only computed once at the beginning of the algorithm and do not change as vertices are deleted from G^*. We will split the graph G^* into r layers, by defining sub-graphs $\tilde{\Lambda}_1, \ldots, \tilde{\Lambda}_r$ of G^*, that are disjoint in their vertices. For each $0 \leq j \leq r$, we use a parameter $h_j = \Delta^{z_1 - j}$, so that $h_0 = \Delta^{z_1}$ upper-bounds the maximum vertex degree in G^*, $h_r = \Delta^{z_2} < \tau/(32\log n)$, and for $1 < j \leq r$, $h_j = h_{j-1}/\Delta$. We will ensure that for each $1 \leq j \leq r$, graph $\tilde{\Lambda}_j$ contains at most nh_{j-1} edges, and that every vertex in $\tilde{\Lambda}_j$ has degree at least h_j. Additionally, for each $1 < j \leq r$, we will define a set D_j of *discarded vertices*: intuitively, these are vertices v, such that v does not belong to layers $1, \ldots, j-1$, but almost all neighbors of v do. We need to remove these vertices since otherwise the average vertex degree in subsequent layers may fall below τ, even while some high-degree vertices may still remain. For each $1 \leq j \leq r$, we also define a graph Λ_j, which is the sub-graph of G^* induced by all vertices of $\tilde{\Lambda}_j, \ldots, \tilde{\Lambda}_r$ and of $D_{j+1}, \ldots, D_r$ (for consistency, we set $D_1 = \emptyset$). For each $1 \leq j \leq r$, roughly every h_j/Δ iterations (that is, when h_j/Δ vertices are deleted), our algorithm will recompute the graphs $\tilde{\Lambda}_j, \ldots, \tilde{\Lambda}_r$, the corresponding sets $D_{j+1}, \ldots, D_r$ of vertices, and the $h_{j'}$-core decomposition of each graph $\tilde{\Lambda}_{j'}$, for all $j \leq j' \leq r$. This is done using Procedure Constr-Layers(Λ_j), that is formally defined in Figure 1. This procedure is also used at the beginning of the algorithm, with $\Lambda_1 = G^*$, to compute the initial partition into layers. Note that some layers may be empty. Note also that, from our choice of parameters, $h_r \geq \tau/(32\Delta\log n) \geq n^{1/\log\log n}$, since $\tau \geq 4n^{2/\log\log n}$, $\Delta = 2^{O(\sqrt{\log n})}$, and n is large enough. This ensures that we can apply Theorem 4.4 to each graph $\tilde{\Lambda}_j$.

When a vertex is deleted from the original graph G, we will use procedure Del-Vertex(G^*, v) that we describe later, in order to update our data structures. As the result of this deletion, some vertices may stop being heavy for class i, and will need in turn be deleted from G^*. Procedure Del-Vertex(G^*, v) will iteratively delete all such vertices from the current graph, and then procedure Constr-Layers may be triggered as needed, as part of Procedure Del-Vertex. When we say that some invariant holds throughout the algorithm, we mean that it holds between the different calls to Del-Vertex(G^*, v), and it may not necessarily hold during the execution of this procedure.

Note that the running time of Procedure Constr-Layers(Λ_j, j), excluding the recursive calls to the same procedure with graph Λ_{j+1}, is $O(n^{1+o(1)}h_j)$. Since the values h_j form a geometrically decreasing sequence, the total running time of Procedure Constr-Layers(Λ_j, j), including all recursive calls is also bounded by $O(n^{1+o(1)}h_j)$. We will invoke this procedure at most $n\Delta/h_j$ times over the course of the algorithm – roughly every h_j/Δ vertex deletions. Therefore, in total, all calls to Procedure Constr-Layers will take time $O(n^{2+o(1)})$.

Throughout the algorithm, for each $1 \leq j \leq r$, we denote by K_j^* the set of all vertices that lie in the cores of $\mathcal{F}_j$, that is, $K_j^* = \bigcup_{(K, U(K), \tilde{\Lambda}_j^K, W^k) \in \mathcal{F}_j} K$, and by $\overline{K}_j^*$ the set of the remaining vertices of $\tilde{\Lambda}_j$; recall that these vertices form an h-universal set in $\tilde{\Lambda}_j$ with respect to K_j^*.

We prove that, throughout the algorithm, for all $1 < j \leq r$, for every vertex $v \in D_j$, there is a path P of length at most $j\log n$, connecting v to a vertex of $\bigcup_{j'<j} K_{j'}^*$, such that every vertex of P lies in $\left(\bigcup_{j'<j} \tilde{\Lambda}_{j'}\right) \cup \left(\bigcup_{j' \leq j} D_{j'}\right)$. Thus each discarded vertex can reach a core vertex via a short path. The details are deferred to the full version of the paper.

Data Structures. Our algorithm maintains the following data structures.

First, we maintain the connectivity/spanning data structure CONN-SF(G^*) for the graph G^*. Recall that the total time required

Procedure Constr-Layers(Λ_j, j)

Input: an integer $1 \leq j \leq r$ and a vertex-induced subgraph $\Lambda_j \subseteq G^*$ containing at most $\Delta n h_j$ edges, such that the degree of every vertex in Λ_j is at least h_r.

(1) If $j = r$, then set $\tilde{\Lambda}_r = \Lambda_r$; Compute the h_r-core decomposition $\mathcal{F}_r$ of $\tilde{\Lambda}_r$ in time $O((|E(\tilde{\Lambda}_r)| + n^{1+o(1)}) \operatorname{poly}\log n) = O(\Delta n h_r + n^{1+o(1)}) = O(n^{1+o(1)} h_r)$ and terminate the algorithm.
From now on we assume that $j < r$.
(2) Run Procedure Deg-Prune(Λ_j, h_j) to partition $V(\Lambda_j)$ into two subsets, J_1, J_2, in time $O(|E(\Lambda_j)| + |V(\Lambda_j)|) = O(\Delta n h_j)$.
(3) Set $\tilde{\Lambda}_j = G^*[J_2]$; observe that every vertex in $\tilde{\Lambda}_j$ has degree at least h_j and $|E(\tilde{\Lambda}_j)| \leq \Delta n h_j$.
(4) Compute the h_j-core decomposition of $\tilde{\Lambda}_j$ in time $\tilde{O}(|E(\tilde{\Lambda}_j)| + n^{1+o(1)}) = O(n^{1+o(1)} h_j)$. Denote by $\mathcal{F}_j$ the resulting set of core structures.
(5) Temporarily set $\Lambda_{j+1} = G^*[J_1]$. Observe that Λ_{j+1} has at most $n h_j = n\Delta h_{j+1}$ edges.
(6) Run Procedure Deg-Prune(Λ_{j+1}, h_r), to compute a partition (R', R'') of $V(\Lambda_{j+1})$, so that every vertex of R'' has at least h_r neighbors in R'', in time $O(|E(\tilde{\Lambda}_{j+1})| + |V(\tilde{\Lambda}_{j+1})|) \leq O(n h_j)$.
(7) Set $D_{j+1} = R'$ and delete all vertices of R' from Λ_{j+1}.
(8) Run Constr-Layers($\Lambda_{j+1}, j+1$).

Figure 1: Procedure Constr-Layers

to maintain this data structure under edge deletions is $\tilde{O}(|E(G^*)| + n) = \tilde{O}(m+n)$, where $m = |E(G)|$ is the total number of edges in the original input graph G. Recall that the data structure can process queries of the form path(G^*, u, v): given two vertices u and v in G^*, return any simple path connecting u to v in G^* if such a path exists, and return $\emptyset$ otherwise. If u and v belong to the same connected component C of G^*, then this query can be processed in time $O(|V(C)|)$.

For every level $1 \leq j \leq r$, we maintain the graphs Λ_j and $\tilde{\Lambda}_j$, together with the h_j-core decomposition $\mathcal{F}_j$ of $\tilde{\Lambda}_j$, and the set D_j of discarded vertices. As already discussed, all these are recomputed at most $n\Delta/h_j$ times over the course of the algorithm, by calling procedure Constr-Layers(Λ_j, j). Each call to the procedure requires running time $n^{1+o(1)} h_j$, and so overall, the running time spent on executing the procedure Constr-Layers(Λ_j, j), over the course of the algorithm, for all $1 \leq j \leq r$, is at most $n^{2+o(1)}$.

For every pair $1 \leq j < j' \leq r$ of indices, for every vertex $v \in \tilde{\Lambda}_{j'} \cup D_{j'}$, we maintain a set $L_j(v)$ of all neighbors of v that lie in $\tilde{\Lambda}_j \cup D_j$. In order to maintain these sets, every time procedure Constr-Layers(Λ_j, j) is called, we construct the sets $L_j(v)$ of vertices for all $v \in \Lambda_{j+1}$. This can be done in time $O(|E(\Lambda_j)|) = n^{1+o(1)} h_j$, without increasing the asymptotic running time of the procedure. Additionally, whenever a vertex $u \in \tilde{\Lambda}_j \cup D_j$ is deleted, we will update the lists $L_j(v)$ of all its neighbors $v \in \Lambda_{j+1}$.

For every level $1 \leq j \leq r$, and every core structure $\mathcal{K} = (K, U(K), \tilde{\Lambda}_j^K, W^K) \in \mathcal{F}_j$, we maintain the data structure from Theorem 4.3, that supports queries core-path(u, v) for pairs $u, v \in K$ of vertices. The total running time required to maintain this data structure for $\mathcal{K}$ is $O(|E(\tilde{\Lambda}_j^K)| \operatorname{poly}\log n)$. Since the core decomposition of $\tilde{\Lambda}_j$ ensures that every edge of $\tilde{\Lambda}_j$ belongs to at most $\log n$ graphs $\tilde{\Lambda}_j^K$, where K is a core from the decomposition, the total time required to maintain this data structure for all cores in $\mathcal{F}_j$ is at most $|E(\tilde{\Lambda}_j^K)| \operatorname{poly}\log n = \tilde{O}(n\Delta h_j) = O(n^{1+o(1)} h_j)$. The core decomposition for $\tilde{\Lambda}_j$ is computed at most $n\Delta/h_j$ times over the course of the algorithm, and for each such new core decomposition, we may spend up to $O(n^{1+o(1)} h_j)$ time maintaining its cores. Therefore, the total time spent on maintaining all cores, across all levels $1 \leq j \leq r$, is at most $O(n^{2+o(1)} \Delta \log n) = O(n^{2+o(1)})$.

For every level $1 \leq j \leq r$, we maintain a counter $N(j)$, for the number of vertices that were deleted from G^* since the last call to Constr-Layers(Λ_j, j).

Finally, we need to maintain data structures that allow us to find short paths from the vertices of $\overline{K}_j^*$ to the vertices of K_j^* for all $1 \leq j \leq r$, and from the vertices of D_j to the vertices of $\bigcup_{j'<j} K_j^*$. Let us fix a level $1 \leq j \leq r$.

First, we construct a new graph H_j, obtained from graph $\tilde{\Lambda}_j$, as follows. Let $\mathcal{K}_1, \ldots, \mathcal{K}_z \in \mathcal{F}_j$ be the core structures that are currently in the core decomposition, and let $K_1, \ldots, K_z$ be their corresponding cores. Starting from graph $\tilde{\Lambda}_j$, we contract every core K_y into a vertex $v(K_y)$. We then add a source vertex s, and connect it to each such new vertex $v(K_y)$. All edges have unit length. The resulting graph is denoted by H_j. We maintain an Even-Shiloach tree for H_j, from the source vertex s, up to distance $(\log n + 1)$: ES-Tree$(H_j, s, (\log n + 1))$. The total time required to maintain this tree is $O(|E(H_j)| \log^2 n) = O(n\Delta h_j \log^2 n)$. Graph H_j and the tree ES-Tree$(H_j, s, (\log n + 1))$ will be recomputed at most $n\Delta/h_j$ times – every time that the procedure Constr-Layers(Λ_j, j) is called. Therefore, the total time needed to maintain all these trees throughout the algorithm is $O(n^{2+o(1)})$.

Lastly, we construct a new graph H'_j, as follows. We start with $G^*[D_j]$, and add a source s, that connects with an edge to every vertex $v \in D_j$ that has a neighbor in $\bigcup_{j'<j}(D_{j'} \cup \tilde{\Lambda}_{j'})$. We maintain an Even-Shiloach tree of H'_j, from the source vertex s, up to distance $(\log n + 1)$: ES-Tree$(H'_j, s, (\log n + 1))$. The total time required to maintain this tree is $O(|E(H'_j)| \log n) = O(n\Delta h_j \log n)$. Graph H'_j and the tree ES-Tree$(H'_j, s, (\log n + 1))$ will be recomputed at most $n\Delta/h_{j-1} = n/h_j$ times – every time that the procedure Constr-Layers($\Lambda_{j-1}, j-1$) is called. Therefore, the total time needed to maintain all these trees throughout the algorithm is $O(n^{2+o(1)})$.

Vertex Deletion. We now describe an update procedure when a vertex v is deleted from the graph G. First, if $v \notin G^*$, then there is nothing to be done. Otherwise, we will maintain a set Q of vertices to be deleted, that is initialized to $Q = \{v\}$. While $Q \neq \emptyset$, we let u be any vertex in Q. We delete u from G^*, updating the connectivity data structure CONN-SF(G^*), and from all graphs $\Lambda_j, \tilde{\Lambda}_j, H_j, H'_j$, to which u belongs. We also update the affected Even-Shiloach trees for H_j and H'_j for all j. For every neighbor u' of u in G^*, we decrease $d(u')$ by 1. If $d(u') < \tau$ but $u' \notin Q$, we add u' to Q. Otherwise, if u belonged to $\tilde{\Lambda}_j \cup D_j$, and $u' \in \Lambda_{j+1}$, for some $1 \leq j \leq r$, then we remove u from $L_j(u')$. We also update the counters $N(j)$ with

the number of deleted vertices. Once $Q = \emptyset$, we check whether we need to call procedure Constr-Layers(Λ_j, j) for any index j. In order to do so, for every $1 \leq j \leq r$, we check whether $N(j) \geq h_j/\Delta$. If this is true for any j, we select the smallest such j, and run the procedure Constr-Layers(Λ_j, j). We also set the counters $N(j')$ for all $j' \geq j$ to 0. We have already accounted for the time needed to maintain all our data structures. Additional running time required by the vertex deletion procedure is bounded by the sum of degrees of all vertices deleted from G^* plus $O(\log n)$, and so the total time incurred by the vertex deletion procedure over the course of the algorithm is $O(|E(G^*)| \log n)$. Overall, the total running time of the whole algorithm is $n^{2+o(1)}$. It now remains to describe an algorithm for responding to queries.

Responding to Queries. Suppose we are given path-query(u, u', C), where C is some connected component of G^*, and $u, u' \in V(C)$. Our goal is to return a path connecting u to u' in C, of length at most $2^{13}|V(C)| \cdot \Delta \cdot \ell^* \cdot \log^4 n/\tau$, in expected time $O(|V(C)| \log^4 n)$. Our first step is to compute a simple path P connecting u to u' in C, by calling Procedure path(G^*, u, u') in the connectivity data structure CONN-SF(G^*). This query can be processed in time $O(|V(C)|)$. We denote this path by $(u_1, u_2, \ldots, u_z)$, where $u_1 = u$ and $u_z = u'$.

Let $\mathcal{R}$ be the collection of all core sets K, whose corresponding core structure $\mathcal{K}$ lies in $\bigcup_{j=1}^{r} \mathcal{F}_j$. We let $\mathcal{R}' \subseteq \mathcal{R}$ be the set of cores K that are contained in C. In our next step, we compute, for every vertex $u_a \in V(P)$, a core $K(a) \in \mathcal{R}'$, and path P_a of length at most $\log^2 n$ in C, connecting u_a to some vertex of $K(a)$, in time $O(|V(C)| \log^2 n)$, by exploiting the data structures H_j and H'_j for $1 \leq j \leq r$; we defer the details to the full version of the paper.

For every vertex $u_a \in V(P)$, we *label* u_a with the corresponding core $K(a)$. Our next step is to "shortcut" the path P, by an algorithm, that, in time $O(|V(C)|)$, computes a sequence $Q = (q_1, q_2, \ldots, q_{z'})$ of vertices on path P, such that $q_1 = u$; $q_{z'} = u'$; and for every consecutive pair q_a, q_{a+1}, either there is an edge between $q_a, q_{a'}$ in G^*, or these two vertices have the same label. Moreover, the algorithm ensures that every label K may appear at most twice in Q, as a label of two consecutive vertices, and the length of Q is at most $2^{13}|V(C)|\Delta \log^2 n/\tau$.

Finally, we turn Q into a path in G^*, by iteratively performing the following process. Let q_a, q_{a+1} be a pair of consecutive vertices on Q, such that there is no edge connecting q_a to q_{a+1} in G^*. Then both q_a and q_{a+1} have the same label, that we denote by K, and we have stored to paths: path $P(q_a)$, connecting q_a to some vertex $q'_a \in K$, and path $P(q_{a+1})$, connecting q_{a+1} to some vertex $q'_{a+1} \in K$. The lengths of both paths are at most $\log^2 n$. Assume that the core structure $\mathcal{K}$ corresponding to K lies in $\mathcal{F}_j$. We then run the algorithm from Theorem 4.3 on $\mathcal{K}, q'_a$ and q'_{a+1}. If the outcome of this algorithm is a path Q_a, of length at most ℓ^*, connecting q'_a to q'_{a+1} in the current graph $\tilde{\Lambda}_j^K$, then we insert the concatenation of the paths $P(q_a), Q_a, P(q_{a+1})$ between q_a and q_{a+1} into Q, and continue to the next iteration. Otherwise, the algorithm correctly establishes that the core structure $\mathcal{K}$ is not perfect, that is, W^K is not an α^*-expander. Since our core decomposition algorithm ensures that with high probability every core structure it computes is perfect, the probability that this happens is at most $1/n^c$ for some large constant c. In this case, we run Procedure Constr-Layers($\Lambda_1, 1$) and restart the algorithm for computing the path connecting u to u' in C from scratch. The running time in this case is bounded by $O(n^{2+o(1)})$, but, since the probability of this event is at most $1/n^c$, the expected running time in this case remains $O(\ell^* + |K| \log^3 n)$.

We now assume that every time Theorem 4.3 is called, a path connecting the two corresponding vertices q'_a and q'_{a+1} is returned. Once we process every consecutive pair q_a, q_{a+1} of vertices on Q that have no edge connecting them in G^*, we obtain a path connecting u to u' in C. The length of the path is bounded by $|Q|(\ell^* + \log^2 n)$, where $|Q|$ is the length of the original sequence Q, so $|Q| \leq 2^{13}|V(C)|\Delta \log^2 n/\tau$. Therefore, the final length of the path that we obtain is at most $2^{13}|V(C)|\ell^*\Delta \log^4 n/\tau$. We now bound the total expected running time of the last step. We invoke Theorem 4.3 at most once for every core K that serves as a label of a vertex on Q, and each such call takes expected time $O(\ell^* + |K| \log^3 n)$. Recall that for all $1 \leq j \leq r$, for all core structures $\mathcal{K} \in \mathcal{F}_j$, their corresponding cores are vertex-disjoint. Therefore, the total running time of this step is bounded by $O(\ell^*|\boldsymbol{Q}| + r|V(C)| \log^3 n) = O(\ell^*\Delta|V(C)| \log^2 n/\tau) + O(|V(C)| \log^4 n) = O(|V(C)| \log^4 n)$, as $\tau \geq \ell^*\Delta$.

ACKNOWLEDGEMENTS

We are grateful to Chandra Chekuri for pointing us to some closely related prior work, and to Shiri Chechik for sharing with us an early version of her paper [7].

REFERENCES

[1] Sanjeev Arora, Elad Hazan, and Satyen Kale. The multiplicative weights update method: a meta-algorithm and applications. *Theory of Computing*, 8(1):121–164, 2012.

[2] Sanjeev Arora and Satyen Kale. A combinatorial, primal-dual approach to semidefinite programs. *J. ACM*, 63(2):12:1–12:35, 2016.

[3] Aaron Bernstein. Deterministic partially dynamic single source shortest paths in weighted graphs. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 80. Schloss Dagstuhl-Leibniz-Center for Computer Science, 2017.

[4] Aaron Bernstein and Shiri Chechik. Deterministic decremental single source shortest paths: beyond the O(mn) bound. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, pages 389–397. ACM, 2016.

[5] Aaron Bernstein and Shiri Chechik. Deterministic partially dynamic single source shortest paths for sparse graphs. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 453–469. SIAM, 2017.

[6] Aaron Bernstein and Liam Roditty. Improved dynamic algorithms for maintaining approximate shortest paths under deletions. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2011, San Francisco, California, USA, January 23-25, 2011*, pages 1355–1365, 2011.

[7] Shiri Chechik. Near-optimal approximate decremental all pairs shortest paths. In *Proc. of the IEEE 59th Annual Symposium on Foundations of Computer Science*, 2018.

[8] Paul Christiano, Jonathan A. Kelner, Aleksander Madry, Daniel A. Spielman, and Shang-Hua Teng. Electrical flows, Laplacian systems, and faster approximation of maximum flow in undirected graphs. In *Proceedings of the 43rd ACM Symposium on Theory of Computing, STOC 2011, San Jose, CA, USA, 6-8 June 2011*, pages 273–282, 2011.

[9] Julia Chuzhoy and Thatchaphol Saranurak. On dynamic shortest paths with adaptive adversary. Unpublished manuscript.

[10] Yefim Dinitz. Dinitz' algorithm: The original version and Even's version. In *Theoretical computer science*, pages 218–240. Springer, 2006.

[11] Shimon Even and Yossi Shiloach. An on-line edge-deletion problem. *Journal of the ACM (JACM)*, 28(1):1–4, 1981.

[12] Lisa Fleischer. Approximating fractional multicommodity flow independent of the number of commodities. *SIAM J. Discrete Math.*, 13(4):505–520, 2000.

[13] Naveen Garg and Jochen Könemann. Faster and simpler algorithms for multicommodity flow and other fractional packing problems. In *39th Annual Symposium on Foundations of Computer Science, FOCS '98, November 8-11, 1998, Palo Alto, California, USA*, pages 300–309, 1998.

[14] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Decremental single-source shortest paths on undirected graphs in near-linear total update time. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 146–155, 2014.

[15] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. A subquadratic-time algorithm for decremental single-source shortest paths. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, pages 1053–1072, 2014.

[16] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Dynamic approximate all-pairs shortest paths: Breaking the O(mn) barrier and derandomization. *SIAM Journal on Computing*, 45(3):947–1006, 2016.

[17] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 21–30, 2015.

[18] Monika Rauch Henzinger and Valerie King. Fully dynamic biconnectivity and transitive closure. In *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on*, pages 664–672. IEEE, 1995.

[19] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, July 2001.

[20] Jonathan A. Kelner, Yin Tat Lee, Lorenzo Orecchia, and Aaron Sidford. An almost-linear-time algorithm for approximate max flow in undirected graphs, and its multicommodity generalizations. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, pages 217–226, 2014.

[21] Rohit Khandekar, Satish Rao, and Umesh Vazirani. Graph partitioning using single commodity flows. *Journal of the ACM (JACM)*, 56(4):19, 2009.

[22] Yin Tat Lee, Satish Rao, and Nikhil Srivastava. A new approach to computing maximum flows using electrical flows. In *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, pages 755–764, 2013.

[23] Yin Tat Lee and Aaron Sidford. Path finding methods for linear programming: Solving linear programs in õ(vrank) iterations and faster algorithms for maximum flow. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 424–433, 2014.

[24] Aleksander Madry. Faster approximation schemes for fractional multicommodity flow problems via dynamic graph algorithms. In *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010*, pages 121–130, 2010.

[25] Aleksander Madry. Computing maximum flow with augmenting electrical flows. In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*, pages 593–602, 2016.

[26] Aleksander Madry. Gradients and flows: Continuous optimization approaches to the maximum flow problem. In *Proceedings of the International Congress of Mathematicians 2018 (ICM 2018)*. WORLD SCIENTIFIC, 2018.

[27] 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.

[28] Liam Roditty and Uri Zwick. On dynamic shortest paths problems. *Algorithmica*, 61(2):389–401, 2011.

[29] Jonah Sherman. Nearly maximum flows in nearly linear time. In *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, 26-29 October, 2013, Berkeley, CA, USA*, pages 263–269, 2013.