New Algorithms and Hardness for Incremental Single-Source Shortest Paths in Directed Graphs*

Maximilian Probst Gutenberg
University of Copenhagen
Copenhagen, Denmark
maximilian.probst@outlook.com

Virginia Vassilevska Williams
MIT
USA

Nicole Wein
MIT
USA

ABSTRACT

In the dynamic Single-Source Shortest Paths (SSSP) problem, we are given a graph $G = (V, E)$ subject to edge insertions and deletions and a source vertex $s \in V$, and the goal is to maintain the distance $d(s, t)$ for all $t \in V$.

Fine-grained complexity has provided strong lower bounds for exact partially dynamic SSSP and approximate fully dynamic SSSP [ESA'04, FOCS'14, STOC'15]. Thus much focus has been directed towards finding efficient partially dynamic $(1 + \epsilon)$-approximate SSSP algorithms [STOC'14, ICALP'15, SODA'14, FOCS'14, STOC'16, SODA'17, ICALP'17, ICALP'19, STOC'19, SODA'20, SODA'20]. Despite this rich literature, for directed graphs there are no known deterministic algorithms for $(1 + \epsilon)$-approximate dynamic SSSP that perform better than the classic ES-tree [JACM'81]. We present the first such algorithm.

We present a *deterministic* data structure for incremental SSSP in weighted directed graphs with total update time $\tilde{O}(n^2 \log W/\epsilon^{O(1)})$ which is near-optimal for very dense graphs; here W is the ratio of the largest weight in the graph to the smallest. Our algorithm also improves over the best known partially dynamic *randomized* algorithm for directed SSSP by Henzinger et al. [STOC'14, ICALP'15] if $m = \omega(n^{1.1})$.

Complementing our algorithm, we provide improved conditional lower bounds. Henzinger et al. [STOC'15] showed that under the OMv Hypothesis, the partially dynamic exact s-t Shortest Path problem in undirected graphs requires amortized update or query time $m^{1/2-o(1)}$, given polynomial preprocessing time. Under a new hypothesis about finding Cliques, we improve the update and query lower bound for algorithms with polynomial preprocessing time to $m^{0.626-o(1)}$. Further, under the k-Cycle hypothesis, we show that any partially dynamic SSSP algorithm with $O(m^{2-\epsilon})$ preprocessing time requires amortized update or query time $m^{1-o(1)}$, which is essentially optimal. All previous conditional lower bounds that come close to our bound [ESA'04,FOCS'14] only held for "combinatorial" algorithms, while our new lower bound does not make such restrictions.

CCS CONCEPTS

• **Theory of computation → Dynamic graph algorithms**; *Data structures design and analysis.*

KEYWORDS

dynamic algorithm, shortest path, single source shortest path, conditional lower bound

ACM Reference Format:
Maximilian Probst Gutenberg, Virginia Vassilevska Williams, and Nicole Wein. 2020. New Algorithms and Hardness for Incremental Single-Source Shortest Paths in Directed Graphs. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing (STOC '20), June 22–26, 2020, Chicago, IL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3357713.3384236

1 INTRODUCTION

A dynamic graph G is a sequence of graphs $G_0, G_1, \ldots, G_t$ such that G_0 is the *initial* graph that is subsequently undergoing *edge updates* such that every two consecutive versions G_i and G_{i+1} of the dynamic graph G differ only in one edge (or the weight of one edge). If the sequence of update operations consists only of edge deletions and weight increases, we say that G is a *decremental* graph and if the update operations are restricted to edge insertions and weight decreases, we say that G is *incremental.* In either case, we say that G is *partially dynamic* and if the update sequence is mixed we say G is *fully dynamic.*

In the study of dynamic graph algorithms, we are concerned with maintaining properties of G efficiently. More precisely, we are concerned with designing a data structure that supports *update* and *query* operations such that after the i^{th} edge update is processed, an adversary can query properties of G_i.

We consider the problem of (approximate) Single-Source Shortest Paths (SSSP) in a partially dynamic graph G. In this problem, a dedicated source vertex $s \in V$ is given on initialization and the query operation takes as input any vertex $t \in V$ and outputs the (approximate) shortest-path distance estimate $\hat{d}(s, t)$ from s to t in the current version of G. We say that distance estimates have *stretch* $\alpha \geq 1$, if the algorithm guarantees that $d(s, t) \leq \hat{d}(s, t) \leq \alpha d(s, t)$ is satisfied for every distance estimate where $d(s, t)$ denotes the distance from s to t in the current version of G.

When proving lower bounds for partially dynamic SSSP, we also consider a potentially easier problem, s-t Shortest Path (s-t SP), thus

*The first author was visiting Massachusetts Institute of Technology, Massachusetts, US, when the work was done and is supported by Thorup's Investigator Grant from the Villum Foundation under Grant No. 16582 and by STIBOFONDEN's IT Travel Grant for PhD Students. The second author is supported by an NSF CAREER Award, NSF Grants CCF-1528078 and CCF-1514339, a BSF Grant BSF:2012338, a Sloan Research Fellowship and a Google faculty fellowship. The third author is supported by an NSF Graduate Fellowship and NSF Grant CCF-1514339.

STOC '20, June 22–26, 2020, Chicago, IL, USA

ACM ISBN 978-1-4503-6979-4/20/06…$15.00
https://doi.org/10.1145/3357713.3384236

obtaining stronger lower bounds. In s-t SP, one wants to maintain a shortest path from s to t for some fixed s and t.

1.1 Motivation

Partially dynamic SSSP is a well-motivated problem with wide-ranging applications:

- Partially dynamic data structures are often used as internal data structures to solve the fully dynamic version of the problem (see for example [42, 46, 51] for applications of partially dynamic SSSP) which in turn can be used to maintain properties of real-world graphs undergoing changes.
- Partially dynamic SSSP is often employed as internal data structure for related problems such as maintaining the diameter in partially dynamic graphs [12, 25] or matchings in incremental bipartite graphs [20].
- Many static algorithms use partially dynamic algorithms as a subroutine. For example, incremental All-Pairs Shortest Paths can be used to construct light spanners [10] and greedy spanners. Moreover, a recent line of research shows that many flow problems can be reduced to decremental SSSP, and recent progress has already led to faster algorithms for multi-commodity flow [49], vertex-capacitated flow, and sparsest vertex-cut [27].

1.2 Prior Work

In this section we discuss prior work directly related to our results. We use $\tilde{O}$ notation to suppress factors of $\log n$ and assume constant ϵ in the approximation. For a more detailed discussion of prior results, we refer the reader to the full version of the article.

Let m be the maximum number of edges and let n be the maximum number of vertices[1] in any version of the dynamic input graph G. If G is weighted, we denote by W the aspect ratio of the graph, which is the largest weight divided by the smallest weight in the graph. For partially dynamic algorithms, we follow the convention of stating the *total update time* rather than the time for each individual update. Unless otherwise stated, queries take worst-case constant time.

Algorithms for partially dynamic directed SSSP. For directed graphs, the classic ES-tree data structure by Even and Shiloach [53] and its later extensions by Henzinger and King [45] initiated the field, with total update time $O(mnW)$ for exact incremental/decremental directed SSSP. Using an edge rounding technique [15, 16, 28, 49, 50, 58], the ES-tree can further handle edge weights more efficiently, giving an $\tilde{O}(mn \log W)$ time algorithm for incremental/decremental $(1+\epsilon)$-approximate directed SSSP. This result has been improved to total update time $mn^{9/10+o(1)} \log W$ by the breakthrough results of Henzinger, Forster, and Nanongkai [40, 41]. Their algorithm is Monte Carlo and works against an oblivious adversary (an adversary that fixes the entire graph sequence of updates in advance). Whilst presented only in the decremental setting, this algorithm appears to extend to the incremental setting.

Very recently, Probst and Wulff-Nilsen [37] improved upon this result and presented a randomized data structure for decremental directed SSSP against an oblivious adversary with total update time $\tilde{O}(\min\{mn^{3/4} \log W, m^{3/4}n^{5/4} \log W\})$. They also give a Las Vegas algorithm with total update time $\tilde{O}(m^{3/4}n^{5/4} \log W)$ that works against an adaptive adversary. They also get slightly improved bounds for unweighted decremental graphs. We point out, however, that their data structure cannot return approximate shortest paths to the adversary as it would reveal the random choices. Further, unlike the data structure by Henzinger et al., their approach cannot be extended to the incremental setting since it relies heavily on finding efficient separators and on maintaining the topological order of vertices in the graph.

In summary, all known partially dynamic algorithms for directed graphs that are faster than ES-trees are *randomized*, and their *amortized* update time even for $m \sim n^2$ insertions is at least some polynomial. Moreover, it is unclear how to extend the result from [37] to the incremental setting. This is in stark contrast to the undirected setting where the currently best bounds for decremental graphs [18, 38, 39] extend straight-forwardly to incremental graphs. This is reminiscent of the connectivity problem, which in undirected incremental graphs is almost trivial while the problem of maintaining strong-connectivity in directed graphs is solved to near-optimality in decremental graphs [21] but still not fully understood in incremental graphs [14, 17, 22]. We believe that understanding strong-connectivity might be a preliminary for understanding single-source shortest paths in directed graphs.

Lower bounds. Conditional lower bounds for partially dynamic SSSP were first studied by Roditty and Zwick [51]. They showed that in the weighted setting, APSP can be reduced to partially dynamic SSSP with $O(n^2)$ updates and queries, thus implying that the amortized query/update time must be $n^{1-o(1)}$, unless APSP can be solved in truly subcubic time (i.e. $n^{3-\epsilon}$ for constant $\epsilon > 0$).

For unweighted SSSP in the partially dynamic setting, there is a weaker lower bound [51]: Under the Boolean Matrix Multiplication (BMM) hypothesis, any combinatorial incremental/decremental algorithm for unweighted SSSP requires amortized $n^{1-o(1)}$ update/query time. Abboud and Vassilevska Williams [3] modified the [51] construction to give stronger lower bounds even for the unweighted s-t SP problem: any combinatorial incremental/decremental algorithm for unweighted s-t SP requires either amortized $n^{1-o(1)}$ update time or $n^{2-o(1)}$ query time.

We point out that the unweighted SSSP lower bounds of [3, 51] are weak in two ways: (1) they are only for combinatorial algorithms, and (2) they hold only when the number of edges m is quadratic in the number of vertices, so in terms of m, the update lower bound is merely $m^{0.5-o(1)}$. Henzinger et al. [44] aimed to rectify (1). They introduced a very believable assumption, the OMv Hypothesis, which is believed to hold for arbitrary algorithms, not merely combinatorial ones. Henzinger et al. [44] showed that under the OMv Hypothesis, incremental/decremental s-t SP (in the word-RAM model) requires $m^{0.5-o(1)}$ amortized update time[2] or $m^{1-o(1)}$ query time, thus obtaining the same lower bounds as under the BMM Hypothesis, but now for not necessarily combinatorial algorithms.

[1] Some algorithms allow vertex updates, therefore the number of vertices might be due to change.

[2] Similar to the lower bounds based on BMM, the Henzinger et al. [44] lower bound on the update time is $n^{1-o(1)}$ but the number of edges in the construction is quadratic in n, so that in terms of m, the lower bound is $m^{0.5-o(1)}$.

1.3 Results

Our main result is a new elegant algorithm for the incremental SSSP problem in weighted digraphs.

Theorem 1.1. *There is a deterministic algorithm that given a weighted digraph $G = (V, E)$ subject to Δ edge insertions and weight decreases, a vertex $s \in V$, and $\epsilon > 0$, maintains for every vertex v an estimate $\hat{d}(v)$ such that after every update $d(s,v) \leq \hat{d}(v) \leq (1+\epsilon)d(s,v)$, and runs in total time $\tilde{O}(n^2 \log W/\epsilon^{2.5} + \Delta)$. Path queries are answered take time linear in the number of path edges.*

Our result is the first deterministic partially dynamic directed SSSP algorithm to improve over the long-standing $O(mn)$ time bound achieved by the ES-tree [53]. Our result is essentially optimal for very dense graphs, and is the first algorithm with essentially optimal update time for any density in directed graphs. Furthermore, our algorithm further improves on the *randomized* $mn^{0.9+o(1)} \log W$ time algorithm of Henzinger et al. [40] if $m = \omega(n^{1.1})$ (their paper presents only in the decremental setting, but it appears to extend to the incremental setting as well).

A further strength of our algorithm is that in addition to returning distance estimates, it can also return the corresponding approximate shortest paths, i.e. it is path-reporting. Previously known path-reporting dynamic SSSP algorithms in the directed setting except for the ES-tree are randomized against an oblivious adversary. In this setting, we in fact also improve for undirected graphs where the only path-reporting deterministic (or adaptive) algorithms are the $\tilde{O}(m^{7/4}n^{1/4})$ update time algorithm by Henzinger, Forster and Nanongkai [43], and the randomized algorithm for decremental graphs by by Chuzhoy and Khanna [27] which has update time $n^{2+o(1)}$ and which even more recently derandomized [26]. However, the latter algorithm works only in the restricted setting of vertex deletions and requires $n^{1+o(1)}$ query time.

Our second contribution includes several new fine-grained lower bounds for the partially dynamic SSSP and s-t-SP problems in unweighted undirected graphs. The only known conditional lower bounds for partially dynamic SSSP and s-t-SP in unweighted graphs give an update time lower bound of $m^{0.5-o(1)}$. While the ES-tree data structure does achieve an $O(\sqrt{m})$ amortized update/query time upper bound whenever $m = \Theta(n^2)$, this upper bound does not improve for lower sparsities. This motivates the following question:

Is partially dynamic SSSP solvable with amortized update/query time $O(\sqrt{m})$ for all sparsities m?

Our work answers this question with the tools of fine-grained complexity. Our first result is based on the following k-Cycle hypothesis (see [11, 48]).

Hypothesis 1.2 (k-Cycle Hypothesis). *In the word-RAM model with $O(\log m)$ bit words, for any constant $\epsilon > 0$, there exists a constant integer k, so that there is no $O(m^{2-\epsilon})$ time algorithm that can detect a k-cycle in an m-edge graph.*

Our first result says that under the k-Cycle Hypothesis, if the preprocessing time of a partially dynamic s-t-SP algorithm is subquadratic $O(m^{2-\epsilon})$ for $\epsilon > 0$, then in fact the algorithm cannot achieve truly sublinear, $O(m^{1-\epsilon'})$ amortized update and query time for any $\epsilon' > 0$. This is a quadratic improvement over the previous known lower bounds, and it is also tight, as trivial recomputation achieves amortized update/query time $O(m)$.

Theorem 1.3. *Under Hypothesis 1.2, there can be no constant $\epsilon > 0$ such that partially dynamic s-t SP in undirected graphs can be solved with $O(m^{2-\epsilon})$ preprocessing time, and $O(m^{1-\epsilon})$ update* and *query time, for all graph sparsities m.*

A consequence of the proof of Theorem 1.3 above is that (under Hypothesis 1.2) the $O(mn)$ total update time achieved by ES-trees is essentially optimal, also when m is close to linear in n. Recall that the OMv lower bound only showed this for $m = \Theta(n^2)$. While the above lower bound is tight, it only holds for truly subquadratic preprocessing time. Recall that the only known lower bound for arbitrary polynomial preprocessing time is the $m^{0.5-o(1)}$ bound under OMv.

We first develop an intricate reduction that shows that an efficient enough partially dynamic s-t SP algorithm can be used to solve the 4-Clique problem. Then we define an *online* version of 4-Clique, similar to OMv that is plausibly hard even for arbitrary polynomial time preprocessing. We show that if 4-Clique requires $n^{c-o(1)}$ time for some c, then any algorithm for partially dynamic s-t SP with $O(m^{c/2-\epsilon})$ preprocessing time for some $\epsilon > 0$, must have update or query time at least $m^{(c-2)/2-o(1)}$. 4-Clique is known to be solvable in $O(n^{3.252})$ time [33], and if the matrix multiplication exponent ω is > 2, the best running time for 4-clique would still be truly supercubic. Thus, the update time in our conditional lower bound, $m^{(c-2)/2-o(1)}$ is polynomially better than $m^{0.5-o(1)}$, as long as $\omega > 2$. Recent results [4, 5] show that the known techniques for matrix multiplication cannot show that ω is less than 2.16.

While the connection between clique detection and s-t SP is interesting in its own right, it does not resolve the limitation on the preprocessing time of our previous lower bound. To fix this, we introduce an online version of 4-Clique, generalizing the OMv (actually the related OuMv [44]) problem:

Definition 1.4 (OMv3 problem). *In the OMv3 problem, we are given an $n \times n$ Boolean matrix A that can be preprocessed and then n queries consisting of three length n Boolean vectors u, v, w have to be answered online by outputting the Boolean value*

$$\bigvee_{i,j,k} (u_i \wedge v_j \wedge w_k \wedge A[i,j] \wedge A[j,k] \wedge A[k,i]).$$

One can think of u, v, w as giving the neighbors of an incoming vertex q in the three partitions of a tripartite graph, and then the Boolean value just answers whether q would be part of a 4-Clique if it were added to the graph. This is the natural extension of Henzinger et al.'s OuMv problem. OMv3 is easy to solve in $O(n^{\omega})$ time per query by computing whether the neighborhood defined by u, v, w contains a triangle. We hypothesize that there is no better algorithm, even if one is to preprocess the matrix in arbitrary polynomial time:

Hypothesis 1.5 (OMv3 Hypothesis). *Any algorithm solving OMv3 with polynomial preprocessing time needs $n^{\omega+1-o(1)}$ total time to solve OMv3 in the word-RAM model with $O(\log n)$ bit words.*

Using this Hypothesis, using essentially the same reduction as from 4-Clique to s-t SP, we obtain plausible conditional lower bounds for arbitrary polynomial preprocessing time and polynomially higher than $m^{0.5-o(1)}$ update/query time lower bound, improving the prior known results.

Theorem 1.6. *In the word-RAM model with $O(\log m)$ bit words, under Hypothesis 1.5, any incremental/decremental s-t Shortest Paths algorithm with polynomial preprocessing time needs $m^{(\omega-1)/2-o(1)}$ amortized update or query time. For the current value of ω, the update lower bound is $\Omega(m^{0.626})$.*

In terms of both m and n, Theorem 1.6 implies that when $m = O(n)$, partially dynamic s-t Shortest Paths with arbitrary polynomial preprocessing needs total time $mn^{(\omega-1)/2-o(1)}$. This is the best limitation to date that both allows for arbitrary polynomial preprocessing and also holds for sparse graphs. If one considers "combinatorial" algorithms (i.e. where $\omega = 3$), one gets that ES trees are essentially optimal again.

We refer the reader to section 5 for a more detailed discussion of our fine-grained results.

2 PRELIMINARIES

For a dynamic weighted directed graph $G = (V, E, w)$, we let G_i denote the i^{th} version of G but simply write G if the context is clear. We define n to be the number of vertices in G and we define m to be the maximum number of edges in any version of G, respectively. For each vertex $v \in V$, we let the out-neighborhood $\mathcal{N}^{out}(v)$ be the set of all vertices w such that $(v, w) \in E$. Analogously, let $\mathcal{N}^{in}(v) = \{u \in V | (u, v) \in E\}$. For all $u, v \in V$, let $d(u, v)$ denote the distance from u to v.

For an array A, let $A[i, j]$ be the subarray of A from index i to index j, inclusive. If A is an array of lists, we define the size of $A[i, j]$ denoted $|A[i, j]|$ to mean the sum $\sum_{k=i}^{j} |A[k]|$ of the sizes of each list from $A[i]$ to $A[j]$.

We use $\log n$ to mean $\log_2 n$ and for convenience, we assume without loss of generality that n is a power of 2. For integers x, y we let $\lfloor x \rfloor_y$ denote the largest multiple of y that is at most x.

3 WARM-UP: A SIMPLE $O(n^{2+2/3}/\epsilon)$ TIME ALGORITHM

In this section we describe an algorithm for incremental SSSP on unweighted directed graphs with total update time $O(n^{2+2/3}/\epsilon)$. This algorithm illustrates the main ideas used in our $\tilde{O}(n^2 \log W/\epsilon)$ algorithm.

Theorem 3.1. *There is a deterministic algorithm that given an unweighted directed graph $G = (V, E)$ subject to edge insertions, a vertex $s \in V$, and $\epsilon > 0$, maintains for every vertex v an estimate $\hat{d}(v)$ such that after every update $d(s, v) \leq \hat{d}(v) \leq (1 + \epsilon)d(s, v)$, in total time $O(n^{2+2/3}/\epsilon)$.*

To obtain this result, we take inspiration from a simple property of *undirected* graphs: *Any two vertices at distance at least* 3 *have disjoint neighborhoods.* This observation is crucial in several spanner/hopset constructions as well as other graph algorithms (for example [13, 23, 34, 35, 42]), as well as partially dynamic SSSP on undirected graphs [18, 19]. In [18], Bernstein and Chechik exploit this property for partially dynamic undirected SSSP in the following way. The property implies that for any vertex v on a given shortest path from s to some t, the neighborhood of v is disjoint from almost all of the other vertices on this shortest path. Thus there cannot be too many high-degree vertices on any given shortest path, and therefore high-degree vertices are allowed to induce large additive error which can be exploited to increase the efficiency of the algorithm.

Whilst we would like to argue along the same lines, this property is unfortunately not given in *directed* graphs: there could be two vertices u and v at distance 3, and a third vertex z that only has in-coming edges from u and v. Clearly, u and v can now still be at distance 3 whilst their out-neighborhoods overlap. We overcome this issue by introducing *forward neighborhoods* $\mathcal{FN}(u)$ that only include vertices from the out-neighborhood $\mathcal{N}^{out}(u)$ that are estimated to be further away from the source vertex s than u. Now, suppose there are two vertices u and v both appear on some shortest path from s to some t and whose forward neighborhoods overlap. Let w be a vertex in $\mathcal{FN}(u) \cap \mathcal{FN}(v)$. Since w has a larger distance estimate than u and the edge (u, w) is in the graph, the distance estimates of u and w must be close, assuming that each distance estimate does not incur much error. Similarly, the distance estimates of v and w must be close. But then the distance estimates of u and v must also be close. Therefore, the forward neighborhood of each vertex on a long shortest path must only overlap with the forward neighborhoods of few other vertices on the path. In summary, our extension of the property to directed graphs is that if the distance estimates of u and v differ by a lot, then u and v have disjoint forward neighborhoods.

3.1 The Data Structure

In order to illustrate our approach, we present a data structure that only maintains approximate distances for vertices u that are at distance $d(s, u) > n^{2/3}$. This already improves the state of the art since we can maintain the exact distance $d(s, u)$ if $d(s, u) \leq n^{2/3}$ simply by using a classic ES-tree to depth $n^{2/3}$ which runs in time $O(mn^{2/3})$.

To understand the motivation behind our main idea, let us first consider a slightly modified version of the classic ES-trees that achieves the same running time: We maintain for each vertex $u \in V$ an array A_u with n elements where $A_u[i]$ is the set of all vertices $v \in \mathcal{N}^{out}(u)$ with $d(s, v) = i$. Then, when $d(s, u)$ decreases, the set of vertices in $\mathcal{N}^{out}(u)$ whose estimated distance from s decreases is exactly the set of vertices stored in $A_u[d^{NEW}(s, u) + 2, n]$ which we call the *forward neighborhood* $\mathcal{FN}(u)$ of u. (Recall that $A_u[i, j]$ is the subarray of A from index i to index j, inclusive.) That is since each such vertex v has at estimated distance more than $d(s, u) + 1$ thus relaxing the edge (u, v) is ensured to decrease v's distance. Thus, we only need to scan edges with tail u and head $v \in \mathcal{FN}(u)$, however, we also need to update A_u whenever an in-neighbor of u decreases its distance estimate.

For our data structure which we call a "lazy" ES-tree, we relax several constraints and use a lazy update rule. Instead of maintaining the exact value of $d(s, v)$ for all $v \in \mathcal{N}^{out}(u)$, we only maintain an *approximate* distance estimate $\hat{d}(v)$. Whilst we still maintain an array for each vertex $u \in V$, we now only update the position of v only after $\hat{d}(v)$ has decreased by at least $n^{1/3}$ or if (u, v) was scanned by u. To emphasize that this array is only updated occasionally, instead of using the notation A_u, we use the notation $\texttt{Cache}_u$. Again, we define $\texttt{Cache}_u[\hat{d}(u)+2, n]$ to be the *forward neighborhood* of u denoted $\mathcal{FN}(u) \subseteq N^{out}(u)$. Further, if $\mathcal{FN}(u)$ is small (say of size $O(n^{2/3})$), we say u is *light*. Otherwise, we say that u is *heavy*.

Now, we distinguish two scenarios for our update rule: if u is *light*, then we can afford to update the distance estimates of the vertices in $\mathcal{FN}(u)$ after every decrease of $\hat{d}(u)$. However, if u is *heavy*, then we only update the vertices in $\mathcal{FN}(u)$ after the distance estimate $\hat{d}(u)$ has been decreased by at least $n^{1/3}$ since the last scan of $\mathcal{FN}(u)$.

Additionally, for each edge (u, v), every time $\hat{d}(v)$ decreases by at least $n^{1/3}$, we update v's position in $\texttt{Cache}_u$.

Finally, we note that $|\mathcal{FN}(u)|$ changes over time and so we need to define the rules for when a vertex changes from light to heavy and vice versa more precisely. Initially, the graph is empty and we define every vertex to be light. Once the size of $\mathcal{FN}(u)$ is increased to $\gamma = 6n^{2/3}/\epsilon$, we set u to be heavy. On the other hand, when $|\mathcal{FN}(u)|$ decreases to $\gamma/2$, we set u to be light. Whenever u becomes light, we immediately scan all $v \in \mathcal{FN}(u)$ and decrease each $\hat{d}(v)$ accordingly. This completes the description of our algorithm.

3.2 Running Time Analysis

Let us now analyze the running time of the lazy ES-tree. For each vertex u, every time $\hat{d}(u)$ decreases by $n^{1/3}$, we might scan u's entire in- and out-neighborhoods. Since $\hat{d}(u)$ can only decrease at most n times, the total running time for this part of the algorithm is $O(nm/n^{1/3}) = O(mn^{2/3})$.

For every light vertex u, we scan $\mathcal{FN}(u)$ every time $\hat{d}(u)$ decreases. Since $\hat{d}(u)$ can only decrease at most n times and since u is light, the total running time for all vertices spent for this part of the algorithm is $O(\sum_{v \in V} n\gamma) = O(n^{2+2/3}/\epsilon)$.

Whenever a vertex u changes from heavy to light, we scan $\mathcal{FN}(u)$. If u only changes from heavy to light once per value of $\hat{d}(u)$, then the running time is $O(n^{2+2/3}/\epsilon)$ by the same argument as the previous paragraph. So, we only consider the times in which u toggles between being light and heavy whilst having the same value of $\hat{d}(u)$. Since the position of vertices in $\texttt{Cache}_u$ can only decrease, the only way for u to become heavy while keeping the same value of $\hat{d}(u)$ is if an edge is inserted. Since $\gamma/2$ edges must be inserted before u becomes heavy since it last became light, there were $\gamma/2$ edge insertions with tail u. Since each inserted edge is only added to a single $\mathcal{FN}(u)$ (namely to the forward neighborhood of its tail), we can amortize the cost of scanning the $\gamma/2$ vertices in $\mathcal{FN}(u)$ over the $\gamma/2$ insertions.

Combining everything, and since the classic ES-tree to depth $n^{2/3}$ takes at most $O(mn^{2/3})$ update time when run to depth $n^{2/3}$, we establish the desired running time.

3.3 Analysis of Correctness

Let us now argue that our distance estimates are maintained with multiplicative error $(1+\epsilon)$. The idea of the argument can be roughly summarized by the following points:

(1) the light vertices do not contribute any error,
(2) we can bound the error contributed by pairs of heavy vertices whose forward neighborhoods overlap, and
(3) the number of heavy vertices on any shortest path with pairwise disjoint forward neighborhoods is small.

We point out that while the main idea of allowing large error in heavy parts of the graphs is similar to [18], we rely on an entirely new method to prove that this incurs only small total error. We start our proof by proving the following useful invariant.

Invariant 3.2. *After every edge update, if $v \in \mathcal{FN}(u)$ then $|\hat{d}(v) - \hat{d}(u)| \le n^{1/3}$.*

Proof. First suppose that $\hat{d}(u) \le \hat{d}(v)$. Since $\hat{d}(u)$ and $\hat{d}(v)$ can only decrease, we wish to show that $\hat{d}(u)$ cannot decrease by too much without $\hat{d}(v)$ also decreasing. This is true simply because every time $\hat{d}(u)$ decreases by at least $n^{1/3}$, $\hat{d}(v)$ is set to at most $\hat{d}(u)+1$. Now suppose that $\hat{d}(u) > \hat{d}(v)$. Since $\hat{d}(u)$ and $\hat{d}(v)$ can only decrease, we wish to show that $\hat{d}(v)$ cannot decrease by too much while remaining in $\mathcal{FN}(u)$. This is true simply because every time $\hat{d}(v)$ decreases by at least $n^{1/3}$, we update v's position in $\texttt{Cache}_u$. If $\hat{d}(v) < \hat{d}(u)+2$ and v's position in $\texttt{Cache}_u$ is updated, then v leaves $\mathcal{FN}(u)$. □

Consider a shortest path $\pi_{s,t}$ for any $t \in V$, at any stage of the incremental graph G. Let $t_0 = s$. Then, for all i, let s_{i+1} be the first heavy vertex after t_i on $\pi_{s,t}$ and let t_{i+1} be the last vertex on $\pi_{s,t}$ whose forward neighborhood intersects with the forward neighborhood of s_{i+1} (possibly $t_{i+1} = s_{i+1}$). Thus, we get pairs $(s_1, t_1), (s_2, t_2), \dots, (s_k, t_k)$. Additionally, let $s_{k+1} = t$. Since the forward neighborhoods of all s_i's are disjoint and of size at least $\gamma/2$ (recall that s_i is heavy), we have that there are at most $k \le 2n/\gamma$ pairs (s_i, t_i).

For each i, let v_i be some vertex in $\mathcal{FN}(s_i) \cap \mathcal{FN}(t_i)$. Note that v_i exists by definition of t_i. By Invariant 3.2, $|\hat{d}(s_i) - \hat{d}(v_i)| \le n^{1/3}$ and $|\hat{d}(t_i) - \hat{d}(v_i)| \le n^{1/3}$. Thus, $\hat{d}(t_i) - \hat{d}(s_i) \le 2n^{1/3}$.

Let t'_i be the vertex on $\pi_{s,t}$ succeeding t_i (except $t'_0 = s$). If $t'_i \in \mathcal{FN}(t_i)$ then by Invariant 3.2, $\hat{d}(t'_i) - \hat{d}(t_i) \le n^{1/3}$. Otherwise, $t'_i \notin \mathcal{FN}(t_i)$ so $\hat{d}(t'_i) \le \hat{d}(t_i) + 1$. So regardless, we have $\hat{d}(t'_i) - \hat{d}(t_i) \le n^{1/3}$ and therefore, since $\hat{d}(t_i) - \hat{d}(s_i) \le 2n^{1/3}$, we have $\hat{d}(t'_i) - \hat{d}(s_i) \le 3n^{1/3}$.

We will show that if u is a light vertex and (u, v) is an edge, then $\hat{d}(v) \le \hat{d}(u) + 1$. Consider the last of the following events that occurred: a) edge (u, v) was inserted, b) $\hat{d}(u)$ was decremented, or c) $\hat{d}(u)$ became light. In case a), the algorithm decreases $\hat{d}(v)$ to be at most $\hat{d}(u)+1$. In cases b) and c), the algorithm updates the distance estimate of all vertices in $\mathcal{FN}(u)$, so if $\hat{d}(v) > \hat{d}(u)+1$ then $\hat{d}(v)$ is decreased to $\hat{d}(u)+1$. Thus we have shown that $\hat{d}(s_{i+1}) - \hat{d}(t'_i) = d(t'_i, s_{i+1})$.

Putting everything together, $\pi_{s,t}$ can be partitioned into (possibly empty) path segments $\pi_{s,t}[t'_i, s_{i+1}]$ and $\pi_{s,t}[s_{i+1}, t'_{i+1}]$. Observe that by definition for each path segment $\pi_{s,t}[t'_i, s_{i+1}]$, the vertices of all edge tails on that segment are light. Thus, by preceding arguments, we can now bound $\hat{d}(t)$ by

$$\hat{d}(t) \leq \sum_{i=0}^{k} \hat{d}(s_{i+1}) - \hat{d}(t_i') + \sum_{i=0}^{k-1} \hat{d}(t_{i+1}') - \hat{d}(s_{i+1})$$
$$< \sum_{i=0}^{k} d(t_i', s_{i+1}) + 3kn^{1/3} \leq d(s,t) + n^{2/3}\epsilon$$

The last inequality comes from our bound on k and the definition of γ. Thus, if $d(s,t) > n^{2/3}$ then $\hat{d}(t) \leq (1+\epsilon)d(s,t)$. Otherwise, $d(s,t) \leq n^{2/3}$ so the classic ES-tree up to depth $n^{2/3}$ finds the exact value of $d(s,t)$.

4 AN $\tilde{O}(n^2 \log W)$ UPDATE TIME ALGORITHM

In this section, we describe how to improve the construction above to derive an $\tilde{O}(n^2 \log W/\epsilon^{2.5})$ algorithm. We first prove the theorem below which gives a $\tilde{O}(n^2/\epsilon)$ bound for unweighted graphs and later we note that the the data structure can handle weighted graphs using standard edge rounding techniques.

Theorem 4.1 (Unweighted version of Theorem 1.1). *There is a deterministic algorithm that given an incremental unweighted directed graph $G = (V, E)$, a vertex $s \in V$, and $\epsilon > 0$, maintains for every vertex v an estimate $\hat{d}(v)$ such that after every update $d(s,v) \leq \hat{d}(v) \leq (1+\epsilon)d(s,v)$, and runs in total time $O(n^2 \log^5 n/\epsilon)$. Path queries can be answered in time linear in the number of edges on the path.*

4.1 Algorithm Overview

There are two main differences between our $\tilde{O}(n^2/\epsilon)$ time algorithm and our warm-up $O(mn^{2/3}/\epsilon)$ time algorithm from the previous section:

(1) Recall that the warm-up algorithm consisted of 1) a classic ES-tree of bounded depth to handle small distances, and 2) a "lazy" ES-tree (of depth n) to handle large distances. For our $\tilde{O}(n^2/\epsilon)$ time algorithm we will have $\log n$ ES-trees of varying degrees of laziness and to varying depths where each ES-tree is suited to handle a particular range of distances. In particular, for each i from 0 to $\log n - 1$, we have one lazy ES-tree that handles distances between 2^i and 2^{i+1}. The ES-trees that handle larger distances can tolerate more additive error, and are thus lazier.

(2) Recall that in the warm-up algorithm, each vertex v was of one of two types: light or heavy, depending the size of the forward neighborhood $\mathcal{FN}(v)$. For our $\tilde{O}(n^2/\epsilon)$ time algorithm, each vertex will be in one of $\Theta(\log n)$ heaviness levels. Roughly speaking, a vertex has heaviness i in the lazy ES-tree up to depth τ if $|\mathcal{FN}(u)| \approx 2^i \frac{n}{\tau}$.

Consider one of our $\log n$ lazy ES-trees. Let τ be its depth and let $\hat{d}_\tau(v)$ be its distance estimate for each vertex v. A central challenge caused by introducing $\log n$ heaviness levels for each lazy ES-tree is handling the event that a vertex changes heaviness level. We describe why unlike in the warm-up algorithm, handling changes in heaviness levels is not straightforward and requires careful treatment. In the warm-up algorithm, whenever a vertex u changes from heavy to light, we scan all $v \in \mathcal{FN}(u)$ and decrease each $\hat{d}(v)$ accordingly. Then, in the analysis of the warm-up algorithm, we argued that if u only changes from heavy to light once per value of $\hat{d}(u)$, we get the desired running time. Now that we have many heaviness levels and we are aiming for a running time of $\tilde{O}(n^2/\epsilon)$, we can no longer allow each vertex to change heaviness level every time we decrement $\hat{d}_\tau(u)$. In particular, suppose we are analyzing a lazy ES-tree up to depth D. Suppose for each vertex u, every time we decrement $\hat{d}_\tau(u)$, we change u's heaviness level and scan $\mathcal{FN}(u)$ as a result. Then since $|\mathcal{FN}(u)|$ could be $\Omega(n)$, the final running time would be $\Omega(n^2 D)$, which is too large. Thus, unlike in the warm-up algorithm, we require that the heaviness of each vertex does not change too often.

Without further modification of the algorithm, the heaviness level of a vertex u can change a number of times in succession. Suppose each index of $\mathtt{Cache}_u$ from index $\hat{d}_\tau(u) - \log n + 2$ to index $\hat{d}_\tau(u) + 1$ contains many vertices such that each of the next $\log n$ times we decrement $\hat{d}_\tau(u)$, $\mathcal{FN}(u)$ increases by enough that u increases heaviness level upon each decrement of $\hat{d}_\tau(u)$. We would like to forbid u from changing heaviness levels so frequently. To address this issue, we *change the definition* of the forward neighborhood $\mathcal{FN}(u)$.

In particular, if $\mathtt{Cache}_u$ contains many vertices in the set of indices that closely precede $\mathtt{Cache}_u[\hat{d}_\tau(u)]$, we *preemptively* add these vertices to $\mathcal{FN}(u)$. In the above example, instead of increasing the heaviness of u for every single decrement of $\hat{d}_\tau(u)$, we would preemptively increase the heaviness of u by a lot to avoid increasing its heaviness again in the near future. Roughly speaking, vertex u has heaviness $h(u)$ if $h(u)$ is the maximum value such that there are $\sim \frac{2^{h(u)} n}{\tau}$ vertices in $\mathtt{Cache}_u[\hat{d}_\tau(u) - 2^{h(u)}, \tau]$. (Note that this definition of heaviness is an oversimplification for the sake of clarity.)

Like in the warm-up algorithm, the heaviness level of a vertex u determines how often we scan $\mathcal{FN}(u)$. If a vertex u has heaviness $h(u)$, this means that we scan $\mathcal{FN}(u)$ whenever the value of $\hat{d}_\tau(u)$ becomes a multiple of $2^{h(u)}$.

In summary, when we decrement $\hat{d}_\tau(u)$, the algorithm does roughly the following:

- If the value of $\hat{d}_\tau(u)$ is a multiple of $2^{h(u)}$, scan all $v \in \mathcal{FN}(u)$ and decrement $\hat{d}_\tau(v)$ if necessary.
- If the value of $\hat{d}_\tau(u)$ is a multiple of $2^{h(u)}$, increase the heaviness of u if necessary.
- Regardless of the value of $\hat{d}_\tau(u)$, check if u has left the forward neighborhood of any other vertex w, and if so, decrease the heaviness of w if necessary.

4.2 The Data Structure

For each number τ between 1 and n such that τ is a power of 2, we maintain a "lazy ES-tree" data structure $\mathcal{E}_\tau$. The guarantee of the data structure $\mathcal{E}_\tau$ is that for each vertex $v \in V$ with $d(s,v) \in [\tau, 2\tau)$, the estimate $\hat{d}_\tau(v)$ maintained by $\mathcal{E}_\tau$ satisfies $d(s,v) \leq \hat{d}_\tau(v) \leq (1+\epsilon)d(s,v)$. Let $\tau_{max} = 2\tau(1+\epsilon)$. Since $\mathcal{E}_\tau$ does not need to provide a $(1+\epsilon)$-approximation for distances $d(s,v) > 2\tau$, the largest distance estimate maintained by $\mathcal{E}_\tau$ is at most τ_{max}. We use the distance estimate $\tau_{max}+1$ for all vertices that do not have distance estimate at most τ_{max}. For all $u \in V$, the final distance estimate $\hat{d}(u)$ is the minimum distance estimate $\hat{d}_\tau(u)$ over all data structures $\mathcal{E}_\tau$, treating each $\tau_{max}+1$ as ∞.

Definitions. We begin by making precise the definitions and notation from the algorithm overview section. For each data structure $\mathcal{E}_\tau$ and for each vertex $u \in V$ we define the following:

- $\hat{d}_\tau(u)$ is the distance estimate maintained by the data structure $\mathcal{E}_\tau$.
- $\texttt{Cache}_u$ is an array of τ_{max} lists of vertices whose purpose is to store (possibly outdated) information about $\hat{d}_\tau(v)$ for all $v \in \mathcal{N}_{out}(u)$. Every time we update the position of a vertex $v \in \mathcal{N}_{out}(u)$ in $\texttt{Cache}_u$, we move v to $\texttt{Cache}_u[\hat{d}_\tau(v)]$.
- $h(u)$ is the *heaviness* of u. Intuitively, if u has large heaviness, this means that u has a large *forward neighborhood* (defined later) and that we scan u's forward neighborhood infrequently.
- $\texttt{CacheInd}(u) = \lfloor \hat{d}_\tau(u) - 1 \rfloor_{2^{h(u)}}$. (Recall that $\lfloor x \rfloor_y$ is the largest multiple of y that is at most x.) The purpose of $\texttt{CacheInd}(u)$ is to define the forward neighborhood of u, which we do next.
- The *forward neighborhood* of u, denoted $\mathcal{FN}(u)$ is defined as the the set of vertices in $\texttt{Cache}_u[\texttt{CacheInd}(u), \tau_{max}]$. Note that $\mathcal{FN}(u)$ is defined differently from the warm-up algorithm due to reasons described in the algorithm overview section.
- $\texttt{Expire}_u$ is an array of τ_{max} lists of vertices whose purpose is to ensure that u leaves $\mathcal{FN}(v)$ once $\hat{d}_\tau(u)$ becomes less than $\texttt{CacheInd}(v)$. In particular, $v \in \texttt{Expire}_u[i]$ if $u \in \mathcal{FN}(v)$ and $\texttt{CacheInd}(v) = i$.
- We also define $\texttt{CacheInd}$ with a second parameter, which will be useful for calculating the heaviness of vertices. Let $\texttt{CacheInd}(v, 2^i) = \lfloor \hat{d}_\tau(v) - 1 \rfloor_{2^i}$. Note that $\texttt{CacheInd}(u, 2^{h(u)})$ is the same as $\texttt{CacheInd}(u)$.

Initialization. We assume without loss of generality that the initial graph is the empty graph. To initialize each $\mathcal{E}_\tau$, we initialize $\hat{d}_\tau(s)$ to 0, and for each $u \in V \setminus \{s\}$, we initialize $\hat{d}_\tau(u)$ to $\tau_{max}+1$. Additionally, for each $u \in V \setminus \{s\}$ we initialize the heaviness $h(u)$ to 0, and we initialize the arrays $\texttt{Cache}_u$ and $\texttt{Expire}_u$ by setting each of the $\tau_{max}+1$ fields in each array to an empty list.

The edge update algorithm. The pseudocode for the edge update algorithm is given in Algorithm 1. We also outline the algorithm in words.

The procedure InsertEdge(u, v) begins by updating $\texttt{Cache}_u$ and $\texttt{Expire}_v$ to reflect the new edge. Then, it calls IncrHeaviness(u) to check whether the heaviness of u needs to increase due to the newly inserted edge. Then, it initializes a set H storing edges.

Initially H contains only the edge (u, v). The purpose of H is to store edges (x, y) after the distance estimate $\hat{d}_\tau(x)$ has changed. We then extract one edge at a time and check whether the decrease in x's distance estimate also translates to a decrease of y's distance estimate by checking whether $\hat{d}_\tau(y) > \hat{d}_\tau(x) + 1$. If so, then $\hat{d}_\tau(y)$ can be decremented and we keep the edge in H. Otherwise, we learned that (x, y) cannot be used to decrease $\hat{d}_\tau(y)$ and we remove (x, y) from H. We point out that in our implementation a decrease of Δ is handled in the form of Δ decrements where the edge is extracted from H $\Delta+1$ times until it is removed from H.

The procedure Decrement(u, v) begins by decrementing $\hat{d}_\tau(v)$. Then, it checks whether $\hat{d}_\tau(v)$ is a multiple of $2^{h(v)}$. If so, it calls IncrHeaviness(v) to check whether the recent decrements of $\hat{d}_\tau(v)$ have caused $\mathcal{FN}(v)$ to increase by enough that the heaviness $h(v)$ has increased. Also, if $\hat{d}_\tau(v)$ is a multiple of $2^{h(v)}$, $\texttt{CacheInd}(v)$ and thus $\mathcal{FN}(v)$ have changed. Thus, we scan each vertex $w \in \mathcal{FN}(v)$ and update the position of w in $\texttt{Cache}_v$. Then, we insert for each such vertex $w \in \mathcal{FN}(v)$ the edge (v, w) into H which has the eventual effect of decreasing $\hat{d}_\tau(w)$ to value at most $\hat{d}_\tau(v)+1$. Since we perform these actions every $2^{h(u)}$ decrements of $\hat{d}_\tau(v)$, as we show later, we incur roughly $2^{h(u)}$ additive error on each out-going edge of v.

Additionally, the procedure Decrement(u, v) checks whether decrementing $\hat{d}_\tau(v)$ has caused v to expire from any of the forward neighborhoods that contain v. The vertices whose forward neighborhood v needs to leave are stored in $\texttt{Expire}_v[\hat{d}_\tau(v)+1]$. For each $w \in \texttt{Expire}_v[\hat{d}_\tau(v)+1]$, we update v's position in $\texttt{Cache}_w$ which causes v to leave $\mathcal{FN}(w)$. Then, we call DecrHeaviness(u) to check whether removing v from $\mathcal{FN}(w)$ has caused the heaviness of w to decrease.

The procedures IncrHeaviness(u) and DecrHeaviness(u) are similar. We first describe DecrHeaviness(u): On line 42, $h(u)$ is set to $\arg\max_{i \in \mathbb{N}}\{|\texttt{Cache}_u[\texttt{CacheInd}(u, 2^i), \tau_{max}]| \geq (2^i - 1)\frac{6n \log n}{\epsilon\tau}\}$. We note that $\texttt{Cache}_u$ may contain out-of-date information when DecrHeaviness(u) is called, however, we wish to update $h(u)$ based on up-to-date information. Thus, before line 42, we update $\texttt{Cache}_u$. However, we do not have time to update *every* index of $\texttt{Cache}_u$, so instead we only update the relevant indices. To do so, it suffices to first calculate the value i', which is the expression for $h(u)$ but using the out-of-date version of $\texttt{Cache}_u$, and then scan all $v \in \texttt{Cache}_u[\texttt{CacheInd}(u, 2^{i'}), \tau_{max}]$, updating the position of each such v in $\texttt{Cache}_u$.

Recall that a smaller value of $h(u)$ means that we scan $\mathcal{FN}(u)$ more often. Thus, after we decrease $h(u)$ in DecrHeaviness(u), the vertices $v \in \mathcal{FN}(u)$ might not have been scanned recently enough according to the new value of $h(u)$. Thus, to conclude the procedure DecrHeaviness(u), we scan each $v \in \mathcal{FN}(u)$ and add (u, v) to the set H so that Decrement(u, v) is called later.

The main difference between procedures IncrHeaviness(u) and DecrHeaviness(u) is that the constants in the expressions for calculating i' and $h(u)$ are different from each other, which ensures that u does not change heaviness levels too often. Additionally, the last step of DecrHeaviness(u) where we insert into H is not necessary for IncrHeaviness(u).

4.3 Analysis of Correctness

For each vertex t, the algorithm obtains the distance estimate $\hat{d}(t)$ by taking the minimum $\hat{d}_\tau(t)$ over all τ (excluding when $\hat{d}_\tau(t) = \tau_{max}+1$). The goal of this section, is to prove that

$$d(s,t) \leq \hat{d}(t) \leq (1+\epsilon)d(s,t)$$

for $d(s,t) \in [\tau, 2\tau]$. We prove this statement in two steps starting by giving a lower bound on $\hat{d}(t)$.

Algorithm 1: Algorithm for handling edge updates.

1 **Procedure** *InsertEdge*(u, v)
2   Add v to $\texttt{Cache}_u[\hat{d}_\tau(v)]$
3   **if** $\hat{d}_\tau(v) \geq \texttt{CacheInd}(u)$ **then**
4   Add u to $\texttt{Expire}_v[\texttt{CacheInd}(u)]$
5   IncrHeaviness(u)
6   **if** $\hat{d}_\tau(v) > \hat{d}_\tau(u) + 1$ **then**
7   Let H be a set storing edges (x, y)
8   H.Insert(u, v)
9   **while** $H \neq \emptyset$ **do**
10   Let tuple (x, y) be any tuple in H
11   **if** $\hat{d}_\tau(y) > \hat{d}_\tau(x) + 1$ **then**
12   Decrement(x, y)
13   **else**
14   H.Remove(x, y)
15 **Procedure** *Decrement*(u, v)
16     $\hat{d}_\tau(v) = \hat{d}_\tau(v) - 1$
17   **if** $\hat{d}_\tau(v)$ is a multiple of $2^{h(v)}$ **then**
18   IncrHeaviness(v)
19   **foreach** $w \in \mathcal{FN}(v)$ **do**
20   Move w to $\texttt{Cache}_v[\hat{d}_\tau(w)]$
21   Move v to $\texttt{Expire}_w[\texttt{CacheInd}(v)]$
22   H.Insert(v, w)
23   **foreach** $w \in \texttt{Expire}_v[\hat{d}_\tau(v) + 1]$ **do**
24   Move v to $\texttt{Cache}_w[\hat{d}_\tau(v)]$
25   Remove w from $\texttt{Expire}_v$
26   DecrHeaviness(w)
27 **Procedure** *IncrHeaviness*(u)
28     $i' \leftarrow \arg\max_{i \in \mathbb{N}}\{|\texttt{Cache}_u[\texttt{CacheInd}(u, 2^i), \tau_{max}]| \geq (2^i - 1)\frac{12n \log n}{\epsilon\tau}\}$
29   **if** $i' > h(u)$ **then**
30   **foreach** $v \in \texttt{Cache}_u[\texttt{CacheInd}(u, 2^{i'}), \tau_{max}]$ **do**
31   Move v to $\texttt{Cache}_u[\hat{d}_\tau(v)]$
32   Remove u from $\texttt{Expire}_v$
33         $h(u) \leftarrow \arg\max_{i \leq i'}\{|\texttt{Cache}_u[\texttt{CacheInd}(u, 2^i), \tau_{max}]| \geq (2^i - 1)\frac{6n \log n}{\epsilon\tau}\}$
34   **foreach** $v \in \mathcal{FN}(u)$ **do**
35   Add u to $\texttt{Expire}_v[\texttt{CacheInd}(u)]$
36 **Procedure** *DecrHeaviness*(u)
37     $i' \leftarrow \arg\max_{i \in \mathbb{N}}\{|\texttt{Cache}_u[\texttt{CacheInd}(u, 2^i), \tau_{max}]| \geq (2^i - 1)\frac{6n \log n}{\epsilon\tau}\}$
38   **if** $i' < h(u)$ **then**
39   **foreach** $v \in \mathcal{FN}(u)$ **do**
40   Move v to $\texttt{Cache}_u[\hat{d}_\tau(v)]$
41   Remove u from $\texttt{Expire}_v$
42         $h(u) \leftarrow \arg\max_{i \in \mathbb{N}}\{|\texttt{Cache}_u[\texttt{CacheInd}(u, 2^i), \tau_{max}]| \geq (2^i - 1)\frac{6n \log n}{\epsilon\tau}\}$
43   **foreach** $v \in \mathcal{FN}(u)$ **do**
44   Add u to $\texttt{Expire}_v[\texttt{CacheInd}(u)]$
45   H.Insert(u, v)

Lemma 4.2. *At all times, for all τ, for any $t \in V$, we have $d(s, t) \leq \hat{d}_\tau(t)$.*

Proof. It suffices to show that we only decrement $\hat{d}_\tau(v)$ if v has an in-coming edge from a vertex with distance estimate more than 1 below $\hat{d}_\tau(v)$. We only invoke the procedure Decrement(u, v) from line 12, and we invoke it under the condition that (u, v) is an edge and $\hat{d}_\tau(v) > \hat{d}_\tau(u) + 1$. Therefore after running Decrement(u, v) we still have $\hat{d}_\tau(v) \geq \hat{d}_\tau(u) + 1$. □

Let us next prove a small, but helpful lemma.

Lemma 4.3. *For all vertices $u, v \in V$, the index of $\texttt{Cache}_u$ containing v can only decrease over time.*

Proof. Whenever we insert v into to $\texttt{Cache}_u$ or move v to a new index in $\texttt{Cache}_u$, v is placed in $\texttt{Cache}_u[\hat{d}_\tau(v)]$. Since $\hat{d}_\tau(v)$ is monotonically decreasing over time, the lemma follows. □

Analogous to Invariant 3.2 from the warm-up algorithm, we can establish the invariant and the lemma below by careful case analysis which can be found in the full version.

Invariant 4.4. *For all $u, v \in V$, after processing each edge update, if $v \in \mathcal{FN}(u)$ then $|\hat{d}_\tau(v) - \hat{d}_\tau(u)| \leq 2^{h(u)}$.*

Lemma 4.5. *For all $u \in V$, $|\mathcal{FN}(u)| \geq (2^{h(u)} - 1)\frac{6n \log n}{\epsilon\tau}$ at all times except lines 24 to 26 and during DecrHeaviness(u).*

Given these preliminaries, we are now ready to prove the final lemma that establishes correctness of the algorithm.

Lemma 4.6. *After processing each edge update, for each $t \in V$ and each τ, $d(s, t) \leq \hat{d}_\tau(t)$ and if $d(s, t) \in [\tau, 2\tau)$ then $\hat{d}_\tau(t) \leq (1 + \epsilon)d(s, t)$.*

Proof. Our main argument is a generalization of the proof of correctness from the warm-up algorithm. Fix a heaviness level $h > 0$. Let $s = t_0$. Then, we define s_{i+1} be the first vertex with heaviness h after t_i on $\pi_{s,t}$ and let t_{i+1} be the last vertex on $\pi_{s,t}$ of heaviness h whose forward neighborhood intersects with the forward neighborhood of s_{i+1} (possibly $t_{i+1} = s_{i+1}$). Thus, we get pairs $(s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)$. Additionally, let $s_{k+1} = t$.

By definition, the forward neighborhoods of all s_i's are disjoint. By Lemma 4.5, for each s_i, $|\mathcal{FN}(s_i)| \geq (2^h - 1)\frac{6n \log n}{\epsilon\tau}$ and since all s_i's have disjoint forward neighborhoods, we have at most k pairs (s_i, t_i) with

$$k \leq \frac{n}{(2^h - 1)\frac{6n \log n}{\epsilon\tau}} \leq \frac{\epsilon\tau}{6(2^h - 1)\log n}.$$

For any i, let v_i be a vertex in $\mathcal{FN}(s_i) \cap \mathcal{FN}(t_i)$ (which exists by definition of t_i). By Invariant 4.4, we have $|\hat{d}_\tau(s_i) - \hat{d}_\tau(v_i)| \leq 2^h$ and $|\hat{d}_\tau(v_i) - \hat{d}_\tau(t_i)| \leq 2^h$. Thus, $\hat{d}_\tau(t_i) - \hat{d}_\tau(s_i) \leq 2^{h+1}$.

Let t'_i be the vertex on $\pi_{s,t}$ succeeding t_i (except $t'_0 = s$). If $t'_i \in \mathcal{FN}(t_i)$ then by Invariant 4.4, we have $\hat{d}_\tau(t'_i) - \hat{d}_\tau(t_i) \leq 2^h$ and otherwise, $t'_i \notin \mathcal{FN}(t_i)$ so $\hat{d}_\tau(t'_i) < \texttt{CacheInd}(t_i) < \hat{d}_\tau(t_i)$. So regardless, we have $\hat{d}_\tau(t'_i) - \hat{d}_\tau(t_i) \leq 2^h$. Combining this with the previous paragraph, we have $\hat{d}_\tau(t'_i) - \hat{d}_\tau(s_i) \leq 3 * 2^h$.

Now, let $h_{max} = \log n$ be the maximum heaviness level. We handle heaviness level h' (initially h_{max}) by finding the pairs (s_i, t_i)

for heaviness h' on the path π' (initially $\pi_{s,t}$). This partitions the path π' into segments $\pi'[t'_i, s_{i+1}]$ and $\pi'[s_{i+1}, t'_{i+1}]$. We observe that all arc tails in these path segments have heaviness less than h'. We contract the path segments $\pi'[s_{i+1}, t'_{i+1}]$ to obtain the new path π', decrement h' and recurse. We continue this scheme until h' is 0. By the previous analysis for each heaviness level h', summing over the distance estimate difference of vertex endpoints of each contracted segment we obtain at most $\frac{3(2^{h'})\epsilon\tau}{6(2^{h'}-1)\log n} \leq \frac{\epsilon\tau}{\log n}$ (since $h' > 0$) total error. Thus, each heaviness level larger than 0 contributes at most $\frac{\epsilon\tau}{\log n}$ additive error and overall they only induce additive error $\epsilon\tau$.

For $h' = 0$, we argue that the algorithm induces no error on edges on π' where each arc tail is of heaviness 0. We will show that if u is vertex of heaviness 0 and (u, v) is an edge, then $\hat{d}_\tau(v) \leq \hat{d}_\tau(u) + 1$. This is straightforward to see from the algorithm description, but we describe the argument in detail for completeness. Consider the last of the following events that occurred: a) edge (u, v) was inserted, b) $\hat{d}_\tau(u)$ was decremented, or c) the heaviness of $\hat{d}_\tau(u)$ became 0. Case a occurs in the InsertEdge(u, v) procedure where the algorithm decreases $\hat{d}_\tau(v)$ to be at most $\hat{d}_\tau(u) + 1$. Case b occurs in the Decrement(v) procedure. Here, the algorithm checks whether $\hat{d}_\tau(v)$ is a multiple of $2^{h(v)}$, which is true since $h(v) = 0$. Then the algorithm updates the distance estimate of all vertices in $\mathcal{FN}(u)$, so if $\hat{d}_\tau(v) > \hat{d}_\tau(u) + 1$ then $\hat{d}_\tau(v)$ is decreased to $\hat{d}_\tau(u) + 1$. Case c occurs in the DecrHeaviness(u) procedure where again the algorithm updates the distance estimate of all vertices in $\mathcal{FN}(u)$.

By definition, the path π' above is of length at most $d(s, t)$ and therefore we obtain an upper bound on $\hat{d}_\tau(t)$ of $d(s, t) + \epsilon\tau$. Then, when $d(s, t) \geq \tau$, the additive error of $\epsilon\tau$ is subsumed in the multiplicative $(1 + \epsilon)$-approximation, as required. □

4.4 Running Time Analysis

We will show that the total running time of each data structure $\mathcal{E}_\tau$ is $\tilde{O}(n^2/\epsilon)$. Since there are $O(\log n)$ values of τ, this implies that the total running time of the algorithm is $\tilde{O}(n^2/\epsilon)$. For the rest of this section we fix a value of τ. Let us start by proving a first invariant.

Invariant 4.7. *At all times, for all $u \in V$ and all integers i such that $h(u) < i \leq \log n$, $|\mathtt{Cache}_u[\mathtt{CacheInd}(u, 2^i), \tau_{max}]| \leq (2^i - 1)\frac{12n\log n}{\epsilon\tau}$.*

Proof. We note that the invariant is satisfied on initialization since $\mathtt{Cache}_u$ is initially empty. Let us now consider the events that could cause the invariant to be violated for some fixed i:

(1) $h(u)$ is decreased: We note that $h(u)$ is only decreased in line 42 of DecrHeaviness(u), where it is set to a value that satisfies the invariant. (In particular, $h(u)$ cannot decrease in IncrHeaviness(u) by Lemma 4.5.)
(2) A vertex v is added to $\mathtt{Cache}_u$: This scenario could only occur due to an insertion of an edge (u, v). However, after adding v to $\mathtt{Cache}_u$ (and u to $\mathtt{Expire}_v$), we directly invoke the procedure IncrHeaviness(u), which we analyze below.
(3) $\mathtt{CacheInd}(u, 2^i)$ is decreased: We note that $\mathtt{CacheInd}(u, 2^i)$ decreases only if $\hat{d}_\tau(u)$ decreases to a multiple of 2^i, in which case also call IncrHeaviness(u).

For the last two cases, it remains to prove that the procedure IncrHeaviness(u) indeed resolves a violation of the invariant. If we do not enter the **if** statement on line 29, then by the definition of i', the invariant is satisfied. If we do enter the **if** statement, then invariant is satisfied for all $i > i'$. By Lemma 4.3 the indices of vertices in $\mathtt{Cache}_u$ can only decrease and therefore during the course of IncrHeaviness(u), the size of $\mathtt{Cache}_u[\mathtt{CacheInd}(u, 2^{i'}), \tau_{max}]$ can only decrease. Thus, when IncrHeaviness(u) terminates, it is still the case that the invariant holds for all $i > i'$. On the other hand, if $i \leq i'$, then we set $h(u)$ on line 33 so that the invariant is satisfied.

□

We can now prove the most important lemma of this section bounding the time spent in the loops starting at lines 19, 30, 34, 39 and 43.

Lemma 4.8. *The total time spent in the loops starting in lines 19, 30, 34, 39 and 43 is $O(n^2 \log^4 n/\epsilon)$.*

Proof. We start our proof by pointing out that the time spent in the loop starting in line 34 is subsumed by the time spent by the loop in line 30 for the following reason. On line 33 the heaviness is chosen so that the forward neighborhood is over a more narrow range of indices that in loop on line 30. Furthermore, By Lemma 4.3 the indices of vertices in $\mathtt{Cache}_u$ can only decrease and therefore between lines 30 and 34, for all i the size of $\mathtt{Cache}_u[\mathtt{CacheInd}(u, 2^i), \tau_{max}]$ can only decrease.

Similarly, the running time spent in the loop starting in line 43 is subsumed by the running time of the loop starting in line 39. Thus, we only need to bound the running times of the loops starting in lines 19, 30, and 39. To bound their running times, we define the concept of i-scanning: we henceforth refer to the event of iterating through $\mathtt{Cache}_u[\mathtt{CacheInd}(u, 2^i), \tau_{max}]$ by *i-scanning* $\mathtt{Cache}_u$, for any $0 \leq i \leq \log n$, choosing the largest i applicable.

Lines 19, 30 and 39 all correspond to i-scanning $\mathtt{Cache}_u$: the loop on line 19 $h(u)$-scans $\mathtt{Cache}_u$, the loop on line 30 i'-scans $\mathtt{Cache}_u$ for i' chosen on line 28, and the loop at line 39 $h(u)$-scans $\mathtt{Cache}_u$. In the full version, we show how to bound the total number of i-scans to prove the claim below.

Claim 4.9. *For all $u \in V$ and all integers $0 \leq i \leq \log n$, the algorithm i-scans $\mathtt{Cache}_u$ at most $O(\tau \log^2 n/2^i)$ times over the course of the entire update sequence.*

Now, the running time of each of these i-scans can be bound by $O(2^i \frac{n\log n}{\epsilon\tau})$ by Invariant 4.7, so we obtain the claimed running time of

$$\sum_i O\left((\tau \log^2 n/2^i)\left(2^i \frac{n\log n}{\epsilon\tau}\right)\right) = O(n \log^4 n/\epsilon).$$

□

It is not hard to bound the total time spent in the loop on line 9 in the procedure InsertEdge(u, v) reusing claim 4.9. The proof is therefore deferred to the full version.

Lemma 4.10. *The total running time spent in the loop starting on line 9 excluding calls to Decrement(u, v) is bounded by $O(n^2 \log^4 n/\epsilon)$.*

We can now give a proof of the section's main theorem.

Proof of Theorem 4.1. We first prove that the total running time of a data structure $\mathcal{E}_\tau$ is $O(n^2 \log^5 n/\epsilon)$. We begin with the procedure InsertEdge(u, v). We note that this procedure takes constant time except for the **while** loop, if we ignore the calls to IncrHeaviness(u). Since there are at most n^2 edge insertions, the running time can be bounded by $O(n^2)$. Further, the total running time spend in the **while** loop starting in line 9 excluding calls to Decrement(u, v) is bounded by $O(n^2 \log^4 n/\epsilon)$ by lemma 4.10.

Let us bound the total time spent in procedure Decrement(u, v). We first observe that the loop on line 23 iterates through each vertex w in $\mathtt{Expire}_u[\hat{d}_\tau(u) + 1]$ removing each w from $\mathtt{Expire}_v$. Clearly, the number of iterations over the course of the entire algorithm can be bounded by the total number of times a vertex is inserted into $\mathtt{Expire}_v$ over all v. Since these insertions occur in the loops starting in lines 34 and 43, we have by lemma 4.8, that the time spend on the loop starting in line 23 is bound by $O(n^2 \log^4 n/\epsilon)$. Further, ignoring subcalls, each remaining operation in the procedure Decrement(u, v) takes constant time. We further observe that since each invocation of the procedure Decrement(u, v) decreases a distance estimate, the procedure is invoked at most $n\tau_{max} = O(n^2)$ times. Thus, we can bound the total time spent in procedure Decrement(u, v) by $O(n^2 \log^4 n/\epsilon)$.

For the procedures IncrHeaviness(u) and DecrHeaviness(u), we note that the calculations of i' and $h(u)$ on lines 28, 33, 37, and 42 can be implemented in $O(\log n)$ time using a binary tree over the elements of array $\mathtt{Cache}_u$ for each $u \in V$. We observe that both procedures receive at most $O(n^2 \log^4 n/\epsilon)$ invocations and since we already bounded the running times of the loops that call them. Thus, the total update time excluding loops can be bound by $O(n^2 \log^5 n/\epsilon)$. The loops take total time $O(n^2 \log^4 n/\epsilon)$ by Lemma 4.8. Using $\log n$ data structures, one for each distance threshold τ, we obtain the lemma. □

4.5 Weighted Graphs

It is not hard to extend the data structure to deal with weights $[1, W]$ by employing a standard edge-rounding technique [15, 16, 28, 49, 50, 58]. Due to lack of space, we defer a proof to the full version and only state the lemma.

Lemma 4.11. *There is a deterministic algorithm that given an incremental weighted directed graph $G = (V, E, w)$ with weights in $[1, W]$, a vertex $s \in V$, and $\epsilon > 0$, maintains for every vertex v an estimate $\hat{d}(v)$ such that after every update $d(s, v) \leq \hat{d}(v) \leq (1 + \epsilon)d(s, v)$ if $d(s, v) \in [\tau, 2\tau)$ for some $\tau \leq n$, and runs in total time $O(n^2 \log^6 n/\epsilon^{1.5})$. Path queries can be answered in time linear in the number of edges of the path.*

5 FINE-GRAINED LOWER BOUNDS FOR PARTIALLY DYNAMIC S-T SHORTEST PATHS

In this section we present several conditional lower bounds for the *s*-*t* Shortest Paths problem in the partially dynamic, i.e. incremental or decremental, setting. In the incremental setting, the assumption is that one starts with an empty graph and m edges are inserted one by one. In the decremental setting, one is given an initial m-edge graph, and then its edges are deleted one by one in some order until the empty graph is reached. We will assume that no preprocessing is done, and that all the work of the algorithm is done in the updates and queries, however, in some cases we will be able to allow arbitrary polynomial preprocessing time. Our lower bounds are based on several popular hypotheses. All hypotheses are for the Word-RAM model of computation with $O(\log n)$ bit words.

The first, the *BMM Hypothesis* is a hypothesis about "combinatorial" algorithms, simple algorithms that do not use the heavy machinery of fast matrix multiplication (as in [29, 30, 32, 36, 52, 54]). The hypothesis (see e.g. [3, 55]) states that in the Word-RAM model with $O(\log n)$ bit words, any combinatorial algorithm for computing the product of two $n \times n$ Boolean matrices requires $n^{3-o(1)}$ time. Due to the subcubic fine-grained equivalence of Boolean Matrix Multiplication (BMM) and Triangle detection [56], the hypothesis is equivalent to: any combinatorial algorithm for Triangle detection in n-vertex graphs requires $n^{3-o(1)}$ time in the Word-RAM model of computation with $O(\log n)$ bit words.

A generalization of the BMM hypothesis is the Combinatorial k-Clique Hypothesis for constant $k \geq 3$ that asserts that any combinatorial algorithm for k-Clique detection in n-vertex graphs requires $n^{k-o(1)}$ time in the Word-RAM model of computation with $O(\log n)$ bit words. When one removes the restriction to combinatorial algorithms, the k-Clique Hypothesis becomes that the current fastest k-Clique algorithms are essentially optimal. For k divisible by 3, the assertion is that $n^{\omega k/3-o(1)}$ time is necessary (see [1, 2, 24] for examples where this hypothesis is used).

For k not divisible by 3, the best known running times for k-Clique are not as clean. For instance, for 4-Clique the fastest known running time is $O(n^{3.252})$ using the fastest known rectangular matrix multiplication algorithm by Le Gall and Urrutia [47]. As long as $\omega > 2$, this algorithm would run in $O(n^{3+\delta})$ time for some $\delta > 0$, i.e. in truly supercubic time. Thus, the following quite weak 4-Clique Hypothesis would be quite plausible: There is a $\delta > 0$ so that $n^{3+\delta-o(1)}$ is needed to detect a 4-Clique in an n-node graph. Looking at the current best 4-Clique algorithms, of course, the 4-Clique Hypothesis is plausible even for $\delta = 0.252$.

Another way to circumvent the "combinatorial" nature of the BMM Hypothesis when using it for lower bounds on dynamic algorithms, is to instead use the Online Matrix Vector Multiplication (OMv) Hypothesis of Henzinger et al. [44]. The OMv Hypothesis is: Given an $n \times n$ Boolean matrix A, any algorithm that preprocesses A in poly(n) time needs total $n^{3-o(1)}$ time to answer n online queries that give a length n Boolean vector v and ask for the Boolean product Av. The OMv Hypothesis is known to imply the related so called OuMv Hypothesis: Given an $n \times n$ Boolean matrix A, any algorithm that preprocesses A in poly(n) time needs total $n^{3-o(1)}$ time to answer n online queries (u, v) where u and v are length n Boolean vectors by returning the Boolean product $u^T Av$ right after (u, v) is given.

We can generalize OuMv to define an analogous problem capturing 4-Clique. Define OMv3 to be the following problem: Given an $n \times n$ Boolean matrix A, preprocess it so that n queries of the following form can be answered online: the queries consist of three n length Boolean vectors u, v, w, and the answer of the query should be the Boolean value

$$\bigvee_{i,j,k} (u_i \wedge v_j \wedge w_k \wedge A[i,j] \wedge A[j,k] \wedge A[k,i]).$$

It is not hard to reduce 4-Clique to OMv3, even when the queries are given offline: we can assume that 4-Clique is given on a 4-partite graph with partitions V_1, V_2, V_3, V_4. Let A be the adjacency matrix of the subgraph induced by V_1, V_2, V_3, and for each $x \in V_4$, we can define the three Boolean vectors u^x, v^x, w^x, where $u^x[j] = 1$ only if $x \in V_1$ and (x, j) is an edge, $v^x[j] = 1$ only if $x \in V_2$ and (x, j) is an edge, and $w^x[j] = 1$ only if $x \in V_3$ and (x, j) is an edge. Then $(u_i^x \wedge v_j^x \wedge w_k^x \wedge A[i,j] \wedge A[j,k] \wedge A[k,i]) = 1$ only when $(i,j,k,x) \in V_1 \times V_2 \times V_3 \times V_4$ and $(i,x), (j,x), (k,x), (i,j), (j,k), (i,k)$ are all edges, i.e. whenever (i,j,k,x) is a 4-Clique.

Now, similarly to OuMv, since the queries to OMv3 are given in an online fashion, the problem seems harder than 4-Clique. The simple way to solve the problem, when given u, v, w seems to be to take the submatrices A^1, A^2, A^3 where A^1 restricts to the rows that u is one and columns that v is one, A^2 restricts to the rows that v is one and columns that w is one and A^3 restricts to the rows that w is one and columns that u is one, and then to compute the trace of $A^1 \cdot A^2 \cdot A^3$ in $O(n^\omega)$ time. In particular, there seems to be no way to use the fact that rectangular matrix multiplication can be done faster than by splitting into square blocks and using the fast square matrix multiplication algorithms.

We can thus make the following very plausible OMv3 Hypothesis, similar to the OuMv one, that any algorithm with polynomial preprocessing time needs $n^{\omega+1-o(1)}$ total time to solve OMv3.

The last hypothesis we will use concerns the k-Cycle problem (for constant k): given an m-edge graph, determine whether it contains a cycle on k vertices. All known algorithms for detecting k-cycles in directed graphs with m edges run at best in time $m^{2-c/k}$ for various small constants c [8, 31, 48, 57], even using powerful tools such as fast matrix multiplication. Ancona et al. [11] formulated a natural hypothesis completely consistent with the state of the art of cycle detection. This *k-Cycle Hypothesis* states that (in the Word-RAM model), for every constant $\epsilon > 0$, there exists a constant k, so that there is no $O(m^{2-\epsilon})$ time algorithm that can find a k-cycle in an m-edge graph.

5.1 Hardness from k-Cycle

The *k-Cycle Hypothesis* states that (in the Word-RAM model), for every constant $\epsilon > 0$, there exists a constant k, so that there is no $O(m^{2-\epsilon})$ time algorithm that can find a k-cycle in an m-edge graph.

We will reduce k-Cycle in m-edge, n-node graphs to incremental s-t SP in undirected or directed graphs, where one starts with an empty graph and inserts $O(m)$ edges, performing $O(n)$ queries. As $n = O(m)$ in connected graphs, the k-Cycle Hypothesis implies that as k grows, the amortized update/query time must be at least $m^{1-o(1)}$.

Theorem 5.1. *In the word-RAM model with $O(\log m)$ bit words, under the k-Cycle Hypothesis, there can be no constant $\epsilon > 0$ such that incremental s-t SP in directed or undirected m-edge graphs can be solved with $O(m^{2-\epsilon})$ preprocessing time and $O(m^{1-\epsilon})$ amortized update and query time.*

An analogous theorem holds in the decremental setting. We omit the details, but essentially one runs the reduction in reverse.

We note that with very minor modification, our reduction can be made to go from *minimum weight* k-cycle to incremental or decremental shortest s-t path in weighted graphs. Lincoln et al. [48] showed that under very believable assumptions (that min weight k-clique and also clique in hypergraphs require $n^{k-o(1)}$ time), min weight k-cycle requires $m^{2-1/k-o(1)}$ time, and hence Theorem 5.1 holds under even more standard assumptions for weighted graphs. For unweighted graphs, we do need the unweighted k-Cycle assumption. Even though this assumption has so far not been related to other standard hardness hypotheses, it is believable and completely consistent with the current state of algorithms.

We will now prove Theorem 5.1. The reduction is the natural extension of the reduction from Triangle detection to s-t SP in [3]. See Figure 1.

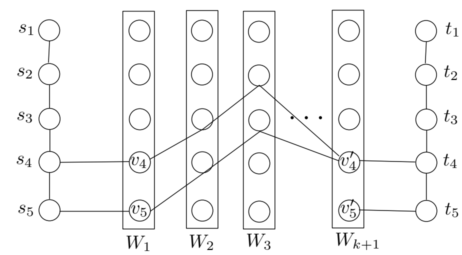

Figure 1: The edges between W_i and W_{i+1} are the edges of the original graph between V_i and V_{i+1}. The figure shows the state of the dynamic graph at stage 4 when one is searching for a k-cycle including v_4. A path from s_1 to t_1 that goes through edge (s_5, v_5) instead of (s_4, v_4) will have length at least $9+k$, whereas if there is a k-cycle through v_4, the shortest path will use (s_4, v_4) and (v_4', t_4) and will have length $8+k$.

First, suppose that incremental s-t SP can be solved with $O(m^{2-\epsilon})$ preprocessing time and $O(m^{1-\epsilon})$ update and query time for some constant $\epsilon > 0$. For that ϵ, let k be such that the k-Cycle Hypothesis asserts that there is no $O(m^{2-\epsilon})$ time algorithm for k-Cycle in m-edge graphs. We will obtain a contradiction via our reduction.

Let G be an m-edge, n-vertex graph in which we want to find a k-cycle. First we use color-coding [6, 7, 9] so that with polylogarithmic time overhead, we can assume that the vertices of G are partitioned into $V_1, V_2, \ldots, V_k$, so that if G contains a k-Cycle, one such cycle has its ith vertex in V_i, for each $i \in \{1, \ldots, k\}$.

Now, the vertices of our incremental graph will be as follows:

- For every $i \in \{1, \ldots, k\}$, there is a set of vertices W_i that contains for every $v \in V_i$ a vertex $v \in W_i$ representing it (slight abuse of notation here).
- Another copy of the vertices of V_1 in a set W_{k+1}. Call the copy of $v \in V_1$ in W_{k+1}, v'.
- A source vertex s_1, followed by vertices $s_1, \ldots, s_n$, all connected in a path.
- A sink vertex t_1, preceded by vertices $t_n, \ldots, t_2$, all connected in a path $t_n \to t_{n-1} \to \ldots \to t_1$.

Besides the path edges above, the remaining edges to be inserted before any queries are as follows: For every $i \in \{1, \ldots, k\}$, for every $u \in W_i, v \in W_{i+1}$, insert (u, v) as an edge if (u, v) was an edge of G.

Notice now that due to the color-coding, we can assume that to detect a k-Cycle in G we only need to check whether for some $v \in W_1$ and its copy $v' \in W_{k+1}$ there is a path of length k. Because of the layering, the distance between v and v' is k if there is a k-cycle in G going through v and it is $> k$ otherwise.

Now, the rest of the dynamic stages proceed as follows. Let the vertices of W_1 be $v_1, \ldots, v_n$, and their corresponding copies in W_{k+1} be $v'_1, \ldots, v'_n$. The stages go from 1 to n. In stage i, we insert an edge between s_{n+1-i} and $v_{n+1-i} \in W_1$ and an edge between v'_{n+1-i} and t_{n+1-i}. Then we query the distance between s_1 and t_1.

Now, at stage i, we have edges between s_{n+1-j} and v_{n+1-j} and between t_{n+1-j} and v'_{n+1-j} for all $j \in \{1, \ldots, i\}$.

The shortest path from s_1 to t_1 looks like this: go from s_1 to s_{n+1-j} (for some $j \in \{1, \ldots, i\}$) using the path of s-nodes, then take an edge to W_1, go through the layers $W_1 - W_{k+1}$ to a node of W_{k+1} and then to t_{n+1-r} (for some $r \in \{1, \ldots, i\}$) and then to t_1. Since going from a vertex in W_1 to a vertex in W_{k+1} gives distance at least k, the length of this path is at least $(n - j) + 2 + k + (n - r)$. If one of j or r is not equal to i (i.e. it is $< i$), the length of the path is $> 2(n - i) + k + 2$. If G contains a k-Cycle through v_{n+1-i}, however, there is a path from s_1 to t_1 going through v_{n+1-i}, v'_{n+1-i} and the edges of the k-Cycle, having total length exactly $2(n - i) + k + 2$.

Thus, in stage i, the distance between s_1 and t_1 is $2(n - i) + k + 2$ if there is a k-Cycle through v_{n+1-i}, and otherwise the distance is $> 2(n - i) + k + 2$. The total number of edge insertions is $O(m + n) = O(m)$ and the number of queries is $O(n) = O(m)$. Thus our supposedly efficient incremental algorithm would solve the k-Cycle problem in time $\tilde{O}(m \cdot m^{1-\epsilon}) = \tilde{O}(m^{2-\epsilon})$ time, a contradiction.

5.2 Hardness from OMv3 and 4-Clique

A weakness of the reduction from k-Cycle detection is that the result is only meaningful when the preprocessing time used by the algorithms is $O(m^{2-\epsilon})$ for some $\epsilon > 0$. For incremental algorithms, one could argue that since one starts with an empty graph, it is unclear how preprocessing could help at all. For decremental graphs however, one knows all the edges so preprocessing could help.

We present a reduction from OMv3 that (1) allows for arbitrary polynomial preprocessing, and (2) gives a higher conditional lower bound than the $m^{0.5-o(1)}$ amortized update/query lower bound that follows from OMv [44], as long as $\omega > 2$.

Moreover, even if instead of the OMv3 Hypothesis we only use the 4-Clique Hypothesis (still via the same reduction below as OMv3), we still obtain a higher than $m^{0.5-o(1)}$ update lower bound.

We will prove Theorem 5.2 below.

Theorem 5.2. *Suppose that incremental or decremental s-t Shortest Paths can be maintained with $P(m)$ preprocessing time and $u(m)$ amortized update and query time, then OMv3 can be solved with $P(O(n^2))$ preprocessing time and $n^2 \cdot u(O(n^2))$ total query time.*

If we assume the OMv3 Hypothesis, then we obtain that any incremental/decremental s-t Shortest paths algorithm with polynomial preprocessing time needs $m^{(\omega-1)/2-o(1)}$ amortized update or query time. For the current value of ω, the update lower bound is $\Omega(m^{0.686})$. If we assume the 4-Clique hypothesis we obtain that there exists a $\delta > 0$ such that any incremental/decremental s-t Shortest paths algorithm needs either $m^{(3+\delta)/2-o(1)}$ preprocessing time, or $m^{(1+\delta)/2-o(1)}$ amortized update or query time. For the current value of δ, this update lower bound is $\Omega(m^{0.626})$.

We now begin the proof of Theorem 5.2. We begin with a gadget that encodes the row/column indices $i \in [n]$ of A and dynamically encodes the n queries $(u^1, v^1, w^1), \ldots, (u^n, v^n, w^n)$ when they come.

We will describe the gadget $G(u)$ which will encode the u^is. The gadgets $G(v)$ and $G(w)$ are analogous. See Figure 2. $G(u)$ consists of $n(n+1)$ vertices: every $i \in [n]$ gets $n+1$ copies $(i, 0), (i, 1), \ldots, (i, n)$. The vertices $(i, 1), \ldots, (i, n)$ for each particular i are chained together in a path, so that for all $j \in \{1, \ldots, n-1\}$ there is an edge between (i, j) and $(i, j+1)$. This describes $G(u)$ (and also $G(v), G(w)$) before any queries come. On query u^ℓ, one inserts an edge from $(i, 0)$ to (i, ℓ) for each i for which $u^\ell[i] = 1$. (The insertions for $G(v)$ and $G(w)$ are analogous but with v^ℓ and w^ℓ instead of u^ℓ, respectively.)

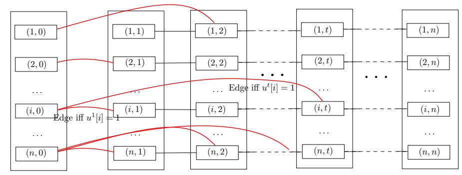

Figure 2: The gadget $G(u)$ encoding the positions in which the queries u^j are 1; in particular, if the current query is u^t, for each $j \leq t$ and each $i \in [n]$ there is a red edge from $(i, 0)$ to (i, j) whenever $u^j[i] = 1$, and the edges for u^t are inserted right after u^t is queried.

There are two copies of $G(u)$, $G(u)$ and $G'(u)$. We chain graphs $G(u), G(v), G(w), G'(u)$ together as follows. For every i, j such that $A[i, j] = 1$ we add edges from (i, n) of $G(u)$ to $(j, 0)$ of $G(v)$, from (i, n) of $G(v)$ to $(j, 0)$ of $G(w)$, and from (i, n) of $G(w)$ to $(j, 0)$ of $G'(u)$. See Figure 3.

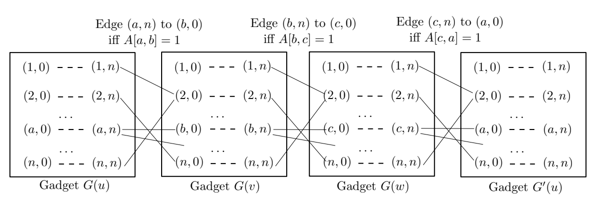

Figure 3: The "middle" gadget connecting the gadgets $G(u), G(v), G(w), G'(u)$.

Notice that so far we have $O(n^2)$ vertices and edges.

Claim 5.3. *Right after inserting the edges for u^ℓ, v^ℓ, w^ℓ into graphs $G(u), G(v), G(w), G'(u)$, the distance between $(i, 0)$ in $G(u)$ and (i, n) in $G'(u)$ is $3 + 4(n + 1 - \ell)$ if there are some j, k so that*

$$u^\ell[i] \wedge v^\ell[j] \wedge w^\ell[k] \wedge A[i, j] \wedge A[j, k] \wedge [k, i] = 1,$$

and the distance is $> 3 + 4(n + 1 - \ell)$ otherwise.

Proof. To get from $G(u)$ to $G'(u)$ one needs to use at least 3 edges (between $G(u)$ and $G(v)$, between $G(v)$ and $G(w)$ and between $G(w)$ and between $G'(u)$) and then also one needs to go from layer 0 ($(*, 0)$) to layer n ($(*, n)$) in each of the 4 gadgets. The shortest possible way to do this is to go through an edge from $(j, 0)$ to (j, ℓ) and then along the path from (j, ℓ) to (j, n), altogether having length $n + 1 - \ell$. Thus the shortest a path from $(i, 0)$ in $G(u)$ to (i, n) in $G'(u)$ is $3 + 4(n + 1 - \ell)$.

This minimal length is achievable if and only if (1) there are some j and k so that the edges $(i, 0)$ to (i, ℓ), $(j, 0)$ to (j, ℓ) and $(k, 0)$ to (k, ℓ) exist in $G(u), G'(u)$ and $G(v)$ and $G(w)$, respectively, and (2) also the edges (i, n) to $(j, 0)$ from $G(u)$ to $G(v)$, (j, n) to $(k, 0)$ from $G(v)$ to $G(w)$ and (k, n) to $(i, 0)$ from $G(w)$ to $G'(u)$ also exist. That is, iff $u^\ell[i] \wedge v^\ell[j] \wedge w^\ell[k] \wedge A[i, j] \wedge A[j, k] \wedge [k, i] = 1$. □

We will now complete the construction. Beyond the gadgets $G(u), G'(u), G(v), G(w)$ and the connections between them, we add two paths:

- The first consists of vertices $s_{\ell,i}$ for $\ell, i \in [n]$ and edges $(s_{\ell,i}, s_{\ell,i+1})$ when $i < n$ and $(s_{\ell,n}, s_{\ell+1,1})$ for $\ell < n$.
- The second similarly consists of vertices $t_{\ell,i}$ for $\ell, i \in [n]$ and edges $(t_{\ell,i}, t_{\ell,i+1})$ when $i < n$ and $(t_{\ell,n}, t_{\ell+1,1})$ for $\ell < n$.

The source and sink for the s-t shortest paths instance are $s_{n,n}$ and $t_{n,n}$. On query (u^j, v^j, w^j) to OMv3, we insert the already described edges into $G(u), G'(u), G(v), G(w)$ and then perform the following n insertions and queries: For each a from 1 to n, insert the edges $s_{j,a}$ to $(a, 0)$ in $G(u)$ and $t_{j,a}$ to (a, n) in $G'(u)$; then query the distance between $s_{n,n}$ and $t_{n,n}$. See Figure 4.

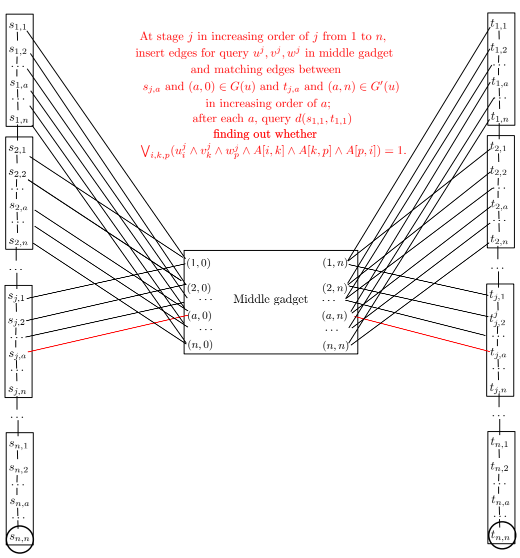

Figure 4: The full reduction connecting the middle gadget with the source and sink paths.

Claim 5.4. *After inserting the edges $s_{j,a}$ to $(a, 0)$ in $G(u)$ and $t_{j,a}$ to (a, n) in $G'(u)$, the distance from $s_{n,n}$ to $t_{n,n}$ is $2(n - j)n + 2(n - a + 1) + 3 + 4(n + 1 - j)$ if there are some b, c so that*

$$u^j[a] \wedge v^j[b] \wedge w^j[b] \wedge A[a, b] \wedge A[b, c] \wedge A[c, a] = 1,$$

and the distance is $> 2(n - j)n + 2(n - a + 1) + 3 + 4(n + 1 - j)$ otherwise.

Proof. The shortest path from $s_{n,n}$ to $t_{n,n}$ goes from $s_{n,n}$ up the s-path (on the left in Figure 4) to some node s^p_b, then along the edge $(s^p_b, (b, 0))$ to the first layer in the Middle gadget, then through the middle gadget, exiting it at some node (c, n) in the last layer, going to a node (r, c) on the t-path (on the right in Figure 4) down to $t_{n,n}$. The length of this path is the length of the subpath from $(b, 0)$ to (c, b) inside the middle gadget $+n(n-p)+(n-b)+n(n-r)+(n-c)+2$.

By Claim 5.3, the shortest possible distance between the first and last layers of the Middle gadget, after inserting the edges inside it for u^j, v^j, w^j is $3 + 4(n + 1 - j)$. Thus, for a particular choice of $p, r \le j$ and $b, c \in [n]$, the length of the above path is at least $3 + 4(n + 1 - j) + n(n - p) + (n - b) + n(n - r) + (n - c) + 2$.

If p and r are both $\le j - 1$, the length of the path is at least: $3 + 4(n + 1 - j) + 2n(n - j) + 2n + (n - b) + (n - c) + 2 > 3 + 4(n + 1 - j) + 2n(n - j) + 2(n - a) + 2$, since $b, c \le n$ and $a \ge 1$.

If $p \le j - 1$ and $r = j$ (the case $p = j$ and $r \le j - 1$ is similar), then since the only added edges from the jth part of the t path are between (c, n) and $t_{j,c}$ for $c \le a$ at this point, we also get that the length of the path is at least $3+4(n+1-j)+2n(n-j)+n+(n-a)+2 > 3 + 4(n + 1 - j) + 2n(n - j) + 2(n - a) + 2$.

Similarly, if $p, r = j$, and b or c is $< a$, the length of the path is at least $3+4(n+1-j)+2n(n-j)+2(n-a)+1+2 > 3+4(n+1-j)+2n(n-j)+2(n-a)+2$. Finally, by Claim 5.3, if $p = r = j, b = c = a$, then the length of the path is $3+4(n+1-j)+2n(n-j)+2(n-a)+2$ if there are some b, c such that $u^j[a] \wedge v^j[b] \wedge w^j[c] \wedge A[a, b] \wedge A[b, c] \wedge A[c, a] = 1$, and the length is larger otherwise. □

ACKNOWLEDGMENTS

The authors would like to thank all reviewers for useful suggestions to organize the article and for pointing out several small mistakes.

REFERENCES

[1] Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. 2015. If the Current Clique Algorithms are Optimal, So is Valiant's Parser. In *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*. 98–117.

[2] Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. 2018. If the Current Clique Algorithms Are Optimal, so Is Valiant's Parser. *SIAM J. Comput.* 47, 6 (2018), 2527–2555.

[3] Amir Abboud and Virginia Vassilevska Williams. 2014. Popular Conjectures Imply Strong Lower Bounds for Dynamic Problems. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*. 434–443.

[4] Josh Alman. 2019. Limits on the Universal Method for Matrix Multiplication. In *Proc. of CCC*. to appear.

[5] Josh Alman and Virginia Vassilevska Williams. 2018. Limits on All Known (and Some Unknown) Approaches to Matrix Multiplication. In *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018, Paris, France, October 7-9, 2018*. 580–591.

[6] Noga Alon, Raphael Yuster, and Uri Zwick. 1994. Color-coding: a new method for finding simple paths, cycles and other small subgraphs within large graphs. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing, 23-25 May 1994, Montréal, Québec, Canada*. 326–335.

[7] Noga Alon, Raphael Yuster, and Uri Zwick. 1995. Color-Coding. *J. ACM* 42, 4 (1995), 844–856.

[8] Noga Alon, Raphael Yuster, and Uri Zwick. 1997. Finding and Counting Given Length Cycles. *Algorithmica* 17, 3 (1997), 209–223.

[9] Noga Alon, Raphael Yuster, and Uri Zwick. 2016. Color Coding. In *Encyclopedia of Algorithms*. 335–338.

[10] Stephen Alstrup, Søren Dahlgaard, Arnold Filtser, Morten Stöckel, and Christian Wulff-Nilsen. 2019. Constructing Light Spanners Deterministically in Near-Linear

Time. In *27th Annual European Symposium on Algorithms (ESA 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[11] Bertie Ancona, Monika Henzinger, Liam Roditty, Virginia Vassilevska Williams, and Nicole Wein. 2019. Algorithms and Hardness for Diameter in Dynamic Graphs. In *Proceedings of ICALP*. to appear.

[12] Bertie Ancona, Monika Henzinger, Liam Roditty, Virginia Vassilevska Williams, and Nicole Wein. 2019. Algorithms and Hardness for Diameter in Dynamic Graphs. In *46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[13] Baruch Awerbuch. 1985. Complexity of network synchronization. *Journal of the ACM (JACM)* 32, 4 (1985), 804–823.

[14] Michael A Bender, Jeremy T Fineman, Seth Gilbert, and Robert E Tarjan. 2016. A new approach to incremental cycle detection and related problems. *ACM Transactions on Algorithms (TALG)* 12, 2 (2016), 14.

[15] Aaron Bernstein. 2009. Fully dynamic (2+ ϵ) approximate all-pairs shortest paths with fast query and close to linear update time. In *2009 50th Annual IEEE Symposium on Foundations of Computer Science*. IEEE, 693–702.

[16] Aaron Bernstein. 2016. Maintaining shortest paths under deletions in weighted directed graphs. *SIAM J. Comput.* 45, 2 (2016), 548–574.

[17] Aaron Bernstein and Shiri Chechi. 2018. Incremental topological sort and cycle detection in expected total time. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 21–34.

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

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

[20] Aaron Bernstein, Jacob Holm, and Eva Rotenberg. 2018. Online bipartite matching with amortized replacements. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 947–959.

[21] Aaron Bernstein, Maximilian Probst, and Christian Wulff-Nilsen. 2019. Decremental strongly-connected components and single-source reachability in near-linear time. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*. 365–376.

[22] Sayan Bhattacharya and Janardhan Kulkarni. 2020. An Improved Algorithm for Incremental Cycle Detection and Topological Ordering in Sparse Graphs. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2509–2521.

[23] Greg Bodwin and Virginia Vassilevska Williams. 2016. Better distance preservers and additive spanners. In *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 855–872.

[24] Karl Bringmann, Pawel Gawrychowski, Shay Mozes, and Oren Weimann. 2018. Tree Edit Distance Cannot be Computed in Strongly Subcubic Time (unless APSP can). In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*. 1190–1206.

[25] Keerti Choudhary and Omer Gold. 2018. Diameter Spanner, Eccentricity Spanner, and Approximating Extremal Graph Distances: Static, Dynamic, and Fault Tolerant. *CoRR* abs/1812.01602 (2018). arXiv:1812.01602 http://arxiv.org/abs/1812.01602

[26] Julia Chuzhoy, Yu Gao, Jason Li, Danupon Nanongkai, Richard Peng, and Thatchaphol Saranurak. 2019. A Deterministic Algorithm for Balanced Cut with Applications to Dynamic Connectivity, Flows, and Beyond. *arXiv preprint arXiv:1910.08025* (2019).

[27] 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 Theory of Computing (STOC 2019)*. ACM, New York, NY, USA, 389–400. https://doi.org/10.1145/3313276.3316320

[28] Edith Cohen. 1998. Fast algorithms for constructing t-spanners and paths with stretch t. *SIAM J. Comput.* 28, 1 (1998), 210–236.

[29] D. Coppersmith. 1997. Rectangular matrix multiplication revisited. *Journal of Complexity* 13 (1997), 42–49.

[30] D. Coppersmith and S. Winograd. 1990. Matrix multiplication via arithmetic progressions. *J. Symbolic Computation* 9, 3 (1990), 251–280.

[31] Mina Dalirrooyfard, Thuy Duong Vuong, and Virginia Vassilevska Williams. 2019. Graph pattern detection: Hardness for all induced patterns and faster non-induced cycles. In *Proceedings of STOC 2019*. to appear.

[32] A.M. Davie and A. J. Stothers. 2013. Improved bound for complexity of matrix multiplication. *Proceedings of the Royal Society of Edinburgh, Section: A Mathematics* 143 (4 2013), 351–369. Issue 02. https://doi.org/10.1017/S0308210511001648

[33] Friedrich Eisenbrand and Fabrizio Grandoni. 2004. On the complexity of fixed parameter clique and dominating set. *Theoretical Computer Science* 326, 1-3 (2004), 57–67.

[34] Michael Elkin and Ofer Neiman. 2016. Hopsets with constant hopbound, and applications to approximate shortest paths. In *Foundations of Computer Science (FOCS), 2016 IEEE 57th Annual Symposium on*. IEEE, 128–137.

[35] Michael Elkin and David Peleg. 2004. $(1+\epsilon,\beta)$-spanner constructions for general graphs. *SIAM J. Comput.* 33, 3 (2004), 608–631.

[36] François Le Gall. 2014. Powers of tensors and fast matrix multiplication. In *International Symposium on Symbolic and Algebraic Computation, ISSAC '14, Kobe, Japan, July 23-25, 2014*. 296–303.

[37] Maximilian Probst Gutenberg and Christian Wulff-Nilsen. 2020. Decremental sssp in weighted digraphs: Faster and against an adaptive adversary. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2542–2561.

[38] Maximilian Probst Gutenberg and Christian Wulff-Nilsen. 2020. Deterministic algorithms for decremental approximate shortest paths: Faster and simpler. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2522–2541.

[39] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. 2014. Decremental single-source shortest paths on undirected graphs in near-linear total update time. In *Foundations of Computer Science (FOCS), 2014 IEEE 55th Annual Symposium on*. IEEE, 146–155.

[40] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. 2014. Sublinear-time decremental algorithms for single-source reachability and shortest paths on directed graphs. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*. ACM, 674–683.

[41] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. 2015. Improved algorithms for decremental single-source reachability on directed graphs. In *International Colloquium on Automata, Languages, and Programming*. Springer, 725–736.

[42] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. 2016. Dynamic Approximate All-Pairs Shortest Paths: Breaking the O(mn) Barrier and Derandomization. *SIAM J. Comput.* 45, 3 (2016), 947–1006.

[43] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. 2017. Sublinear-Time Maintenance of Breadth-First Spanning Trees in Partially Dynamic Networks. *ACM Transactions on Algorithms (TALG)* 13, 4 (2017), 1–24.

[44] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. 2015. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*. ACM, 21–30.

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

[46] Valerie King. 1999. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*. IEEE, 81–89.

[47] Francois Le Gall and Florent Urrutia. 2018. Improved Rectangular Matrix Multiplication using Powers of the Coppersmith-Winograd Tensor. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*. 1029–1046.

[48] Andrea Lincoln, Virginia Vassilevska Williams, and R. Ryan Williams. 2018. Tight Hardness for Shortest Cycles and Paths in Sparse Graphs. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*. 1236–1252.

[49] Aleksander Madry. 2010. Faster approximation schemes for fractional multicommodity flow problems via dynamic graph algorithms. In *Proceedings of the forty-second ACM symposium on Theory of computing*. ACM, 121–130.

[50] Prabhakar Raghavan and Clark D Tompson. 1987. Randomized rounding: a technique for provably good algorithms and algorithmic proofs. *Combinatorica* 7, 4 (1987), 365–374.

[51] Liam Roditty and Uri Zwick. 2004. On dynamic shortest paths problems. In *European Symposium on Algorithms*. Springer, 580–591.

[52] A. Schönhage. 1981. Partial and Total Matrix Multiplication. *SIAM J. Comput.* 10, 3 (1981), 434–455.

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

[54] Virginia Vassilevska Williams. 2012. Multiplying matrices faster than coppersmith-winograd. In *Proceedings of the 44th Symposium on Theory of Computing Conference, STOC 2012, New York, NY, USA, May 19 - 22, 2012*. 887–898.

[55] Virginia Vassilevska Williams. 2018. On some fine-grained questions in algorithms and complexity. In *Proceedings of the International Congress of Mathematicians*. 3431–3475.

[56] Virginia Vassilevska Williams and Ryan Williams. 2018. Subcubic equivalences between path, matrix, and triangle problems. *Journal of the ACM (JACM)* 65, 5 (2018), 27.

[57] Raphael Yuster and Uri Zwick. 2004. Detecting short directed cycles using rectangular matrix multiplication and dynamic programming. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004, New Orleans, Louisiana, USA, January 11-14, 2004*. 254–260.

[58] Uri Zwick. 2002. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *Journal of the ACM (JACM)* 49, 3 (2002), 289–317.