All Pairs Almost Shortest Paths

Dorit Dor * *Shay Halperin* * *Uri Zwick* *
Department of Computer Science
Tel Aviv University
Tel Aviv 69978, Israel

Abstract

Let $G = (V, E)$ be an unweighted undirected graph on n vertices. A simple argument shows that computing all distances in G with an additive one-sided error of at most 1 is as hard as Boolean matrix multiplication. Building on recent work of Aingworth, Chekuri and Motwani, we describe an $\tilde{O}(\min\{n^{3/2}m^{1/2}, n^{7/3}\})$ time algorithm $\mathbf{APASP}_2$ for computing all distances in G with an additive one-sided error of at most 2. The algorithm $\mathbf{APASP}_2$ is simple, easy to implement, and faster than the fastest known matrix multiplication algorithm. Furthermore, for every even $k > 2$, we describe an $\tilde{O}(\min\{n^{2-\frac{2}{k+2}}m^{\frac{2}{k+2}}, n^{2+\frac{2}{3k-2}}\})$ time algorithm $\mathbf{APASP}_k$ for computing all distances in G with an additive one-sided error of at most k.

We also give an $\tilde{O}(n^2)$ time algorithm $\mathbf{APASP}_\infty$ for producing stretch 3 estimated distances in an unweighted and undirected graph on n vertices. No constant stretch factor was previously achieved in $\tilde{O}(n^2)$ time.

*We say that a weighted graph $F = (V, E')$ k-*emulates *an unweighted graph $G = (V, E)$ if for every $u, v \in V$ we have $\delta_G(u, v) \leq \delta_F(u, v) \leq \delta_G(u, v) + k$. We show that every unweighted graph on n vertices has a 2-emulator with $\tilde{O}(n^{3/2})$ edges and a 4-emulator with $\tilde{O}(n^{4/3})$ edges. These results are asymptotically tight.*

Finally, we show that any weighted *undirected graph on n vertices has a 3-spanner with $\tilde{O}(n^{3/2})$ edges and that such a 3-spanner can be built in $\tilde{O}(mn^{1/2})$ time. We also describe an $\tilde{O}(n(m^{2/3} + n))$ time algorithm for estimating all distances in a* weighted *undirected graph on n vertices with a stretch factor of at most 3.*

1 Introduction

The all-pairs shortest paths (APSP) problem is one of the most fundamental algorithmic graph problems. The complexity of the fastest known algorithm for solving the problem for weighted directed graphs, without negative cycles, is $O(mn + n^2 \log n)$, where n and m are the number of vertices and edges in the graph (Johnson [22], see also [16]).

The special case of the all-pairs shortest paths problem in which the input graph is unweighted is closely related to matrix multiplication. It is fairly easy to see that solving the APSP problem exactly, even on unweighted graphs, is at least as hard as Boolean matrix multiplication. Recent works, by Alon, Galil and Margalit [3], Alon, Galil, Margalit and Naor [4], Galil and Margalit [20],[21] and Seidel [25] have shown that if matrix multiplication can be performed in $O(M(n))$ time, then the APSP problem for unweighted directed graphs can be solved in $\tilde{O}(\sqrt{n^3 M(n)})$ time and the APSP problem for unweighted undirected graphs can be solved in $\tilde{O}(M(n))$ time ($\tilde{O}(f)$ means $O(f \operatorname{polylog} n)$). The currently best upper bound on matrix multiplication is $M(n) = O(n^{2.376})$ (Coppersmith and Winograd [15]).

While the above results are extremely interesting from the theoretical point of view, they are of little use in practice as the fast matrix multiplication algorithms are better than the naive $O(n^3)$ time algorithm only for very large values of n. There is interest therefore in obtaining fast algorithms for the APSP problem that do *not* use fast matrix multiplication. The currently best *combinatorial* algorithm for the unweighted APSP problem is an $O(n^3 / \log n)$ time algorithm obtained by Feder and Motwani [18] (see also [10]). This offers only a marginal improvement over the naive $O(n^3)$ time algorithm.

As an algorithm for the APSP problem will yield an algorithm with a similar time bound for Boolean matrix multiplication, there is little hope of developing a combinatorial $O(n^{3-\epsilon})$ time algorithm for the APSP problem. The only hope for obtaining a practical algorithm whose running time is $O(n^{3-\epsilon})$ is by relaxing our requirements. We should be looking therefore at the problem of *approximating* distances and shortest paths.

Awerbuch *et al.* [9] and Cohen [12] considered the problem of finding stretch t all-pairs paths, where t is some fixed constant and a path is of stretch t if its length is at most t

* E-mail addresses: {ddorit,hshay,zwick}@math.tau.ac.il

times the distance between its endpoints. Cohen [12], improving the results of Awerbuch *et al.* [9], obtains, for example, an $\tilde{O}(n^{5/2})$ time algorithm for finding stretch $4+\epsilon$ paths and distances in weighted undirected graphs, for any $\epsilon > 0$ (all weights from now on are assumed to be positive). She also exhibits a tradeoff between the running time of the algorithm and the obtained stretch factor. For any even t, stretch $t+\epsilon$ paths between all pairs of vertices can be found in $\tilde{O}(n^{2+2/t})$ time. The works of Awerbuch *et al.* [9] and Cohen [12] are based on the construction of sparse spanners (Awerbuch [8], Peleg and Schäffer [24]). A t-spanner of a graph $G = (V, E)$ is a subgraph $G' = (V, E')$ of G such that for every $u, v \in V$ we have $\delta_{G'}(u, v) \leq t \cdot \delta_G(u, v)$, where $\delta_G(u, v)$ is the distance between the vertices u and v in the (possibly weighted) graph G.

A different approach all together was employed recently by Aingworth, Chekuri and Motwani [2] (see also [1]). They describe a simple and elegant $\tilde{O}(n^{5/2})$ time algorithm for finding all distances in unweighted and undirected graphs with an *additive* one-sided error of at most 2. They also make the very important observation that the small distances, and not the long distances, are the hardest to approximate. Based on the ideas of Aingworth *et al.* [2], Orlin (unpublished) obtained an $\tilde{O}(n^{7/3})$ time algorithm for finding all distances with an additive one-sided error of at most 4.

In this work we improve and extend the result of Aingworth *et al.* [2], and of Orlin, and obtain an $\tilde{O}(\min\{n^{3/2}m^{1/2}, n^{7/3}\})$ time algorithm, called $\mathbf{APASP}_2$, for finding all distances in unweighted and undirected graphs with an additive one-sided error of at most 2. The algorithm $\mathbf{APASP}_2$ is just the first in a sequence of algorithms $\mathbf{APASP}_k$, for even $k \geq 2$, that exhibits a trade-off between running time and accuracy. For any even $k > 2$, the algorithm $\mathbf{APASP}_k$ runs in $\tilde{O}(\min\{n^{2-\frac{2}{k+2}}m^{\frac{2}{k+2}}, n^{2+\frac{2}{3k-2}}\})$ time and has a one-sided error of at most k. The algorithm $\mathbf{APASP}_4$, for example, runs in $\tilde{O}(n^{5/3}m^{1/3}, n^{11/5})$ time. All algorithms described in this paper can be easily adapted to find almost shortest paths whose lengths are equal to the estimated distances.

In addition, we show that for any $k \geq 2$, the stretch of the estimates produced by the algorithm $\mathbf{APASP}_k$ is at most 3. As k increases, the running time of the algorithm $\mathbf{APASP}_k$ decreases. For $k = \Theta(\log n)$, the running time becomes $\tilde{O}(n^2)$. We let $\mathbf{APASP}_\infty$ be the algorithm $\mathbf{APASP}_k$ with $k = 2\lfloor \log n \rfloor$. The algorithm $\mathbf{APASP}_\infty$ produces stretch 3 distances in unweighted, undirected graphs in $\tilde{O}(n^2)$ time. As mentioned, no fixed stretch factor was previously achieved in $\tilde{O}(n^2)$ time.

We next introduce the notion of *emulators*. Emulators may be seen as the additive counterparts of spanners. We show that any graph on n vertices has a 2-emulator with $\tilde{O}(n^{3/2})$ edges, and a 4-emulator with $\tilde{O}(n^{4/3})$ edges. These can be constructed in $\tilde{O}(n^{5/2})$ and in $\tilde{O}(n^{7/3})$ time, respectively. We are not able to obtain sparser emulators. We are able, however, to construct 6-emulators of size $\tilde{O}(n^{4/3})$ in $\tilde{O}(n^2)$ time. The bounds on the number of edges in 2-emulators and 4-emulators are asymptotically tight. There are graphs on n vertices that cannot be 2-emulated by graphs with $n^{3/2-o(1)}$ edges, and there are graphs on n vertices that cannot be 4-emulated by graphs with $n^{4/3-o(1)}$ edges.

We are also able to obtain some results for *weighted* undirected graphs. We show that any weighted graph on n vertices has a 3-spanner with $\tilde{O}(n^{3/2})$ edges and that such a 3-spanner can be found in $\tilde{O}(mn^{1/2})$ time. Finally, we describe an $\tilde{O}(n(m^{2/3}+n))$ time algorithm for finding stretch 3 distances in a *weighted* undirected graph on n vertices. Extended and improved results for weighted graphs, including an $\tilde{O}(n^2)$ time algorithm for finding stretch 3 distances and an $\tilde{O}(n^{3/2}m^{1/2})$ time algorithm for finding stretch 2 distances appear in [14].

2 Preliminaries

The work of Aingworth *et al.* [2] is based on the following simple observation: there is a small set of vertices that dominates all the high degree vertices of a graph. A set of vertices D is said to *dominate* a set U if every vertex in U has a neighbour in D. This observation is also central to our work.

Lemma 2.1 ([2], see also [5], pp. 6-7) *Let $G = (V, E)$ be an undirected graph with n vertices and m edges. Let $1 \leq s \leq n$. A set D of size $O((n \log n)/s)$ that dominates all the vertices of degree at least s in the graph can be found deterministically in $O(m + ns)$ time.*

Note that as $s \leq n$, the running time of this deterministic algorithm is always at most $O(n^2)$. Picking each vertex of V independently at random with probability $(c \log n)/s$, for some large enough $c > 0$, will yield a desired dominating set of size $O((n \log n)/s)$ with high probability. A deterministic algorithm can be obtained using the simple greedy approximation algorithm for the set cover problem. See [2] for details.

In the subsequent sections, we use an algorithm, called $\mathbf{dominate}(G, s)$, that receives an undirected graph $G = (V, E)$ and a degree threshold s. The algorithm outputs a pair (D, E^*), where D is a set of size $O((n \log n)/s)$ that dominates the set of vertices of degree at least s in G. The set $E^* \subseteq E$ is a set of edges of G of size $O(n)$ such that for every vertex $u \in V$ with degree at least s, there is an edge $(u, u') \in E^*$ such that $u' \in D$. Once a dominating set D is obtained, the set E^* is easily obtained by adding to E^* a single edge for each vertex of degree at least s.

Another ingredient used by our algorithm is the classical Dijkstra's algorithm.

Lemma 2.2 (Dijkstra's algorithm) *Let $G = (V, E)$ be a weighted directed graph with n vertices and m edges. Let $s \in V$. Dijkstra's algorithm runs in $O(m + n \log n)$ time and finds distances, and a tree of shortest paths, from s to all the vertices of V. Furthermore, if the weights of the edges are integers in the range $\{1, 2, \ldots, n\}$ then the algorithm can be implemented to run in $O(m + n)$ time.*

Dijkstra's algorithm appeared originally in [17], though the running time of the version described there is $O(n^2)$. For a more modern description of Dijkstra's algorithm see [16]. The running time of $O(m + n \log n)$ is obtained by using *Fibonacci heaps* [19]. The observation that Dijkstra's algorithm can be implemented to run in $O(m+n)$ time if the weights are in the range $\{1, 2, \ldots, n\}$ is a simple exercise.

In all the algorithms described in this paper, except those of Section 7, we start with an *unweighted* undirected graph $G = (V, E)$. We then build many auxiliary *weighted* graphs and run Dijkstra's algorithm on each one of them. The weights of the edges in these auxiliary graphs will always be integers in the range $\{1, 2, \ldots, n\}$, so we can in fact use the simple $O(m + n)$ time implementation of Dijkstra's algorithm.

By running Dijkstra's algorithm from every vertex of a graph $G = (V, E)$, we get an $O(mn + n^2 \log n)$ time algorithm for solving the all pairs shortest paths problem (APSP). Our goal in this paper is to reduce the running time of APSP algorithms to as close to $\tilde{O}(n^2)$ as possible. To achieve this goal we are willing to settle for almost shortest paths instead of genuine shortest paths.

Our algorithms also involve many runs of Dijkstra's algorithm. Most of these runs, however, are performed on graphs with substantially less edges than the original input graph. A typical step in our algorithms is composed of choosing a vertex $u \in V$, choosing a set of edges F, and then running Dijkstra's algorithm, from u, on the graph $H = (V, F)$. The set of edges F is *not* necessarily a subset of the edge set E of the input graph. Furthermore, the set F *varies* from step to step. The weight of an edge $(u, v) \in F$ is taken to be the currently best upper bound on the distance between u and v in the input graph G. Bounds obtained in a run of Dijkstra's algorithm are used, therefore, in some of the subsequent runs.

In our algorithms, we use a *symmetric* $n \times n$ matrix, denoted $\{\hat{\delta}(u, v)\}_{u,v}$, to hold the currently best upper bounds on distances between all pairs of vertices in the input graph $G = (V, E)$. Initially $\hat{\delta}(u, v) = 1$, if $(u, v) \in E$, and $\hat{\delta}(u, v) = +\infty$ otherwise. By $\mathbf{dijkstra}((V, F), \hat{\delta}, u)$ we denote an invocation of Dijkstra's algorithm, from the vertex u, on the graph (V, F), where the weight of an edge $(u, v) \in F$ is taken to be $\hat{\delta}(u, v)$. The edges of F are considered to be *undirected*. As mentioned, an edge of F is not necessarily an edge of E. If $(u, v) \in F$ is an edge of the original graph then its weight is 1, otherwise, its weight is greater than 1. A call to $\mathbf{dijkstra}((V, F), \hat{\delta}, u)$ updates the row and the column belonging to u in the matrix $\hat{\delta}$ with the distances found during this run, if they are smaller than the previous estimates. Note that the matrix $\{\hat{\delta}(u, v)\}_{u,v}$ serves as both input and output of $\mathbf{dijkstra}$.

If the graph (V, F) is a subgraph of the input graph $G = (V, E)$, then a call to $\mathbf{dijkstra}((V, F), \hat{\delta}, u)$ amounts to running a BFS on (V, F) from u. When we want to stress this fact, we denote such a call by $\mathbf{bfs}((V, F), \hat{\delta}, u)$.

It should be clear from the above discussion that at any time during the run of our algorithms and for any $u, v \in V$ we have $\delta(u, v) \leq \hat{\delta}(u, v)$, where $\delta(u, v)$ is the distance between u and v in the input graph G.

3 Additive error 2

Aingworth *et al.* [2] obtained an $\tilde{O}(n^{5/2})$ algorithm for approximating all distances in an undirected and unweighted graph with a one-sided additive error of 2. We describe two faster algorithms that have the same accuracy. The first algorithm, $\mathbf{apasp}_2$, runs in $\tilde{O}(n^{3/2}m^{1/2})$ time. The second algorithm, $\overline{\mathbf{apasp}}_3$, runs in $\tilde{O}(n^{7/3})$. The first algorithm is faster if the input graph is sufficiently sparse, namely, if $m < n^{5/3}$. By combining these two algorithms we get the algorithm $\mathbf{APASP}_2$ mentioned in the abstract.

A description of the algorithm $\mathbf{apasp}_2$ is given in Figure 1. Throughout the paper, we let $\deg(v)$ denote the degree of a vertex v. The algorithm is extremely simple. It starts by partitioning the vertices of the graph G into two classes, V_1 and $V_2 = V \setminus V_1$. The class V_1 includes the high degree vertices, i.e., the vertices of degree at least $s_1 = (m/n)^{1/2}$. The class V_2 includes all the low degree vertices, i.e., the vertices of degree less than $s_1 = (m/n)^{1/2}$. A similar partition is used by Aingworth *et al.* [2] and also by Alon, Yuster and Zwick [6]. The edge set E_2 is composed of all the edges that touch a low degree vertex and therefore $|E_2| = O(n(m/n)^{1/2}) = O((nm)^{1/2})$. The algorithm proceeds by finding a set D_1 of size $\tilde{O}(n^{3/2}/m^{1/2})$ that dominates the vertices of V_1 and an edge set $E^* \subseteq E$ of size $O(n)$ such that for every $u \in V_1$ there exists $v \in D_1$ such that $(u, v) \in E^*$. Finally, the main part of the algorithm is composed of two steps. In the first, a BFS is pefromed on the graph G from every vertex $u \in D_1$. In the second and final step, Dijkstra's algorithm is run, from every $u \in V \setminus D_1$, on the graph $G_2(u) = (V, E_2 \cup E^* \cup (\{u\} \times D_1))$. It is important to note that Dijkstra's algorithm is not run on the input graph G, that may contain too many edges, and that a slightly different graph is used for each vertex $u \in V \setminus D_1$. The graph $G_2(u)$, on which Dijkstra's algorithm is run from u, includes all the edges that touch low degree vertices, edges that connect each high degree vertex with a vertex of the dominating set, and edges connecting u with all

Algorithm $\mathbf{apasp}_2$:

input: An unweighted undirected graph $G = (V, E)$.
output: A matrix $\{\hat{\delta}(u, v)\}_{u,v}$ of estimated distances.

$s_1 \leftarrow (m/n)^{1/2}$

$V_1 \leftarrow \{v \in V \mid \deg(v) \geq s_1\}$
$E_2 \leftarrow \{(u, v) \in E \mid \deg(u) < s_1 \text{ or } \deg(v) < s_1\}$

$(D_1, E^*) \leftarrow \mathbf{dominate}(G, s_1)$
For every $u, v \in V$ let $\hat{\delta}(u, v) \leftarrow \begin{cases} 1 & \text{if } (u, v) \in E, \\ +\infty & \text{otherwise.} \end{cases}$
For every $u \in D_1$ run $\mathbf{bfs}(G, \hat{\delta}, u)$
For every $u \in V \setminus D_1$ run $\mathbf{dijkstra}(\,(V, E_2 \cup E^* \cup (\{u\} \times D_1))\,, \hat{\delta}, u)$

Figure 1. An $\tilde{O}(n^{3/2}m^{1/2})$ time algorithm for computing surplus 2 distances

(a) (b)

Figure 2. (a) Case 1 in the proof of Theorem 3.1. (b) Case 1 in the proof of Theorem 3.2.

the vertices of the dominating set. The number of edges in $G_2(u)$ is therefore $O((nm)^{1/2})$. It is also important to note that the graph $G_2(u)$ is a *weighted* graph. The weight of the edges in $E_2 \cup E^*$ is 1, as in the unweighted graph. The weight of an edge $(u, v) \in \{u\} \times D_1$, however, is the distance between u and v in G. This distance was found by the BFS that started at $v \in D_1$.

Theorem 3.1 *The algorithm* $\mathbf{apasp}_2$ *runs in* $\tilde{O}(n^{3/2}m^{1/2})$ *time, where* n *is the number of vertices and* m *is the number of edges in the input graph* $G = (V, E)$*, and for every* $u, v \in V$ *we have* $\delta(u, v) \leq \hat{\delta}(u, v) \leq \delta(u, v) + 2$.

Proof: We start with the complexity analysis. Finding the dominating set D_1 requires, according to Lemma 2.1, only $O(n^2)$ time. As $|D_1| = \tilde{O}(n^{3/2}/m^{1/2})$, the total running time of all the BFS's is $\tilde{O}(n^{3/2}/m^{1/2} \cdot m) = \tilde{O}(n^{3/2}m^{1/2})$. As the number of edges in each graph on which Dijkstra's algorithm is applied is $\tilde{O}((nm)^{1/2})$, the total running time of all these calls is also $\tilde{O}(n \cdot (nm)^{1/2}) = \tilde{O}(n^{3/2}m^{1/2})$ and this is also the running time of the whole algorithm.

We now examine the accuracy of the algorithm. The weights attached to the weighted edges in the graphs $G_2(u)$ are distances in the graph G. This implies that the approximations produced by the algorithm cannot be too small. In other words, $\delta(u, v) \leq \hat{\delta}(u, v)$ for every $u, v \in V$. We now prove that the $\hat{\delta}(u, v) \leq \delta(u, v) + 2$ for every $u, v \in V$. Let u and v be two vertices in G. We consider the following two (non-exclusive but exhaustive) cases:

Case 1: There is a shortest path between u and v that contains a vertex from V_1.

Let w be the *last* vertex on the path that belongs to V_1 (see Figure 2(a)). All the edges on the path from w to v touch vertices in V_2 and therefore belong to E_2, and therefore to $G_2(u)$. Let $w' \in D_1$ be such that $(w, w') \in E^*$. The edge (w, w') also belongs to $G_2(u)$. As $w' \in D_1$, a weighted edge (u, w') was added to $G_2(u)$. The weight of this edge is $\delta(u, w')$, the distance between u and w' in G, found by the BFS from $w' \in D_1$. Note that $\delta(u, w') \leq \delta(u, w) + 1$. By running Dijkstra's algorithm from u, on $G_2(u)$, we find therefore that

$$\begin{aligned} \hat{\delta}(u, v) &\leq \delta(u, w') + \delta(w', w) + \delta(w, v) \\ &\leq (\delta(u, w) + 1) + 1 + \delta(w, v) = \delta(u, v) + 2\,. \end{aligned}$$

Case 2: There is a shortest path between u and v that does not contain any vertex from V_1.

Algorithm $\overline{\mathbf{apasp}}_3$:

input: An unweighted undirected graph $G = (V, E)$.
output: A matrix $\{\hat{\delta}(u, v)\}_{u,v}$ of estimated distances.

$s_1 \leftarrow n^{2/3}$; $s_2 \leftarrow n^{1/3}$

For $i \leftarrow 1$ to 2 let $V_i \leftarrow \{v \in V \mid \deg(v) \geq s_i\}$
For $i \leftarrow 2$ to 3 let $E_i \leftarrow \{(u, v) \in E \mid \deg(u) < s_{i-1} \text{ or } \deg(v) < s_{i-1}\}$

For $i \leftarrow 1$ to 2 let $(D_i, E_i^*) \leftarrow \mathbf{dominate}(G, s_i)$
$E^* \leftarrow E_1^* \cup E_2^*$
For every $u, v \in V$ let $\hat{\delta}(u, v) \leftarrow \begin{cases} 1 & \text{if } (u, v) \in E, \\ +\infty & \text{otherwise.} \end{cases}$

For every $u \in D_1$ run $\mathbf{bfs}(G, \hat{\delta}, u)$
For every $u \in D_2$ run $\mathbf{bfs}((V, E_2), \hat{\delta}, u)$
For every $u \in V$ run $\mathbf{dijkstra}(\,(V, E_3 \cup E^* \cup (D_1 \times V) \cup (D_2 \times D_2) \cup (\{u\} \times D_2))\,, \hat{\delta}, u)$

Figure 3. An $\tilde{O}(n^{7/3})$ time algorithm for computing surplus 2 distances

This shortest path is contained in (V, E_2) and therefore $\hat{\delta}(u, v) = \delta(u, v)$. □

Algorithm $\overline{\mathbf{apasp}}_3$, described in Figure 3, is similar to algorithm $\mathbf{apasp}_2$. Instead of dividing the vertices into two classes according to their degrees, we now divide them into three classes. Instead of using the 'threshold' $(m/n)^{1/2}$, we now use the two thresholds $n^{1/3}$ and $n^{2/3}$. Another important difference between $\overline{\mathbf{apasp}}_3$ and $\mathbf{apasp}_2$ is that the graphs on which Dijkstra's algorithm is run now contain the edges $(D_1 \times V) \cup (D_2 \times D_2)$. These edges are weighted. The weight of an edge $(u, v) \in D_1 \times V$ is the distance between u and v in G, found by the BFS from $u \in D_1$. The weight of an edge $(u, v) \in D_2 \times D_2$ is the distance between u and v in the graph (V, E_2). Note that this distance may be larger then the distance between u and v in G.

Theorem 3.2 *The algorithm $\overline{\mathbf{apasp}}_3$ runs in $\tilde{O}(n^{7/3})$ time, where n is the number of vertices in the input graph $G = (V, E)$, and for every $u, v \in V$ we have $\delta(u, v) \leq \hat{\delta}(u, v) \leq \delta(u, v) + 2$.*

Proof: We start again with the complexity analysis. Finding the two dominating sets D_1 and D_2 requires only $O(n^2)$ time. As $|D_1| = \tilde{O}(n^{1/3})$, $|D_2| = \tilde{O}(n^{2/3})$, $|E_2| = O(n^{5/3})$ and $|E_3| = O(n^{4/3})$, the BFS's from the vertices of D_1 take $\tilde{O}(n^{1/3} \cdot n^2) = \tilde{O}(n^{7/3})$ time and the BFS's from the vertices of D_2 take $\tilde{O}(n^{2/3} \cdot n^{5/3}) = \tilde{O}(n^{7/3})$ time. As $|D_1 \times V| = \tilde{O}(n^{4/3})$ and $|D_2 \times D_2| = \tilde{O}(n^{4/3})$, each graph on which Dijkstra's algorithm is run has only $\tilde{O}(n^{4/3})$ edges. The total time taken by all these runs is therefore $\tilde{O}(n \cdot n^{4/3}) = \tilde{O}(n^{7/3})$.

It is again clear that $\delta(u, v) \leq \hat{\delta}(u, v)$, for every $u, v \in V$. It remains to show therefore that $\hat{\delta}(u, v) \leq \delta(u, v) + 2$, for every $u, v \in V$. Let u and v be two vertices in G. We consider the following three cases:

Case 1: There is a shortest path between u and v that contains a vertex w from V_1.

Let $w' \in D_1$ such that $(w, w') \in E$ (see Figure 2(b)). The edges (u, w') and (w', v) belong to the graph on which Dijkstra's algorithm is run from u. The weights of these edges are $\delta(u, w')$ and $\delta(w', v)$, the distances found by the BFS on G from $w' \in D_1$. Note that $\delta(u, w') \leq \delta(u, w) + 1$ and $\delta(w', v) \leq 1 + \delta(w, v)$. By running Dijkstra's algorithm from u, we find therefore that

$$\hat{\delta}(u, v) \leq \delta(u, w') + \delta(w', v) \leq \delta(u, v) + 2 \,.$$

Case 2: There is a shortest path between u and v that contains vertices from V_2 but not from V_1.

This case is very similar to case 1 in the proof of Theorem 3.1. Let w be the *last* vertex on the path that belongs to V_2. All the edges on the path from w to v touch vertices in V_3 and therefore belong to the graph $G_3(u)$ on which Dijkstra's algorithm is run from u. Let $w' \in D_2$ be such that $(w, w') \in E^*$. The graph $G_3(u)$ contains weighted edges connecting u to all the vertices of D_2. It contains in particular a weighted edge (u, w'). The weight of this edge is the distance $\delta_2(u, w')$ between u and w' in the graph $G_2 = (V, E_2)$, found by the BFS from $w' \in D_2$ on G_2. As all the edges on the path from u to w, as well as (w, w') belong to E_2, we get that $\delta_2(u, w') \leq \delta(u, w) + 1$. By running

Dijkstra's algorithm from u we find therefore that

$$\hat{\delta}(u,v) \leq \delta_2(u,w') + \delta(w',w) + \delta(w,v) \leq \delta(u,v) + 2 \ .$$

Case 3: There is a shortest path between u and v that does not contain any vertex from V_2.

This shortest path is then contained in (V, E_3) and therefore $\hat{\delta}(u,v) = \delta(u,v)$. □

The above proof remains correct even if the edge set $D_2 \times D_2$ is not added to the graphs on which Dijkstra's algorithm is run. We have added it as it may improve the accuracy of the algorithm, in certain cases, and as it is used in the proof of Theorem 6.3. We can also replace the edge set $\{u\} \times D_1$, in algorithm $\mathbf{apasp}_2$, and the edge set $\{u\} \times D_1$, in algorithm $\overline{\mathbf{apasp}}_3$, by the larger edge set $\{u\} \times V$ without increasing the running times of the algorithms.

We can easily get a randomized version of algorithm $\overline{\mathbf{apasp}}_3$ which has the property that *all* reported distances greater than $n^{2/3}$ are, with high probability, correct. Similarly, we can get a randomized version of $\mathbf{apasp}_2$ for which all reported distances greater than $n^{3/2}/m^{1/2}$ are, with high probability, correct. We use the following simple observation (a similar idea is used by Ullman and Yannakakis [26]):

Theorem 3.3 *Let $G = (V, E)$ be a weighted directed graph with n vertices and m edges. Let $1 \leq s \leq n$. There is an $\tilde{O}(n^3/s)$ time randomized algorithm that finds, with high probability, the exact distance between any pair of vertices connected by a shortest path that uses at least s edges.*

Proof: Let D be a random set of vertices obtained by picking each vertex independently with probability $(c \log n)/s$, for some large enough $c > 0$. The expected size of D is $\tilde{O}(n/s)$. Run Dijkstra's algorithm from each vertex of D in both G, and the graph obtained from G by reversing the edges. The complexity of this step is $\tilde{O}(nm/s)$. For every $u \in D$ and $v \in V$, we now know $\delta(u,v)$ and $\delta(v,u)$ exactly. For every pair of vertices $u, v \in V$, let $\hat{\delta}(u,v) = \min_{w \in D}\{\delta(u,w) + \delta(w,v)\}$. The complexity of this step is $\tilde{O}(n^3/s)$. It is easy to see that $\hat{\delta}(u,v) = \delta(u,v)$ if and only if there is a shortest path between u and v that passes through a vertex of D. If there is a shortest path between u and v of length s, then with high probability, at least one of the vertices on the path will belong to D. As there are $O(n^2)$ pairs of vertices connected by shortest paths that use at most s edges, and as we can focus on one such path for each pair, we get that, with high probability, each one of these $O(n^2)$ paths will pass through a vertex of D, and the exact distances between all these pairs will be found. □

It follows that if the set D_1 in algorithm $\overline{\mathbf{apasp}}_3$ is chosen at random, by picking each element with probability $cn^{-2/3} \log n$, then, with high probability, if $\hat{\delta}(u,v) \geq n^{2/3}$ then $\hat{\delta}(u,v)$ is the exact distance between u and v. Long distances are therefore easier to compute.

4 Trading time and accuracy

Aingworth *et al.* [2] obtained their $\tilde{O}(n^{5/2})$ algorithm by splitting the vertices into two classes according to their degree. We obtain our $\tilde{O}(n^{7/3})$ time algorithm by dividing them into three classes. It is natural to try to divide the vertices into more classes. In Figure 4 we describe an algorithm $\mathbf{apasp}_k$ that divides the vertices into k classes and runs in $\tilde{O}(n^{2-1/k}m^{1/k})$ time. Algorithm $\mathbf{apasp}_2$ described in Figure 1 is a special case of this more general algorithm. We next show that algorithm $\mathbf{apasp}_k$ has an additive error of at most $2(k-1)$.

Theorem 4.1 *For every $2 \leq k = O(\log n)$, the algorithm $\mathbf{apasp}_k$ runs in $\tilde{O}(n^{2-1/k}m^{1/k})$ time, where n is the number of vertices and m is the number of edges in the input graph $G = (V, E)$, and for every $u, v \in V$ we have $\delta(u,v) \leq \hat{\delta}(u,v) \leq \delta(u,v) + 2(k-1)$.*

Proof: We start again with the complexity analysis. For every $1 \leq i \leq k$ and $u \in D_i$, let $E_i(u)$ be the edge set of the graph on which Dijkstra's algorithm is run from u. It is easy to check that $|D_i| = \tilde{O}(n^{2-i/k}/m^{1-i/k})$, and that $|E_i| = \tilde{O}(n \cdot (m/n)^{1-(i-1)/k}) = \tilde{O}(n^{(i-1)/k}m^{1-(i-1)/k})$, for every $1 \leq i \leq k$. The cost of running Dijkstra's algorithm from every $u \in D_i$, where $1 \leq i \leq k$, is therefore $\tilde{O}(n^{2-i/k}/m^{1-i/k} \cdot n^{(i-1)/k}m^{1-(i-1)/k}) = \tilde{O}(n^{2-1/k}m^{1/k})$. The total running time of the algorithm is therefore $\tilde{O}(k \cdot n^{2-1/k}m^{1/k}) = \tilde{O}(n^{2-1/k}m^{1/k})$, as $k = O(\log n)$.

For every $1 \leq i \leq k$ and $u, v \in V$, let $\delta_i(u,v)$ be the value of $\hat{\delta}(u,v)$ after running $\mathbf{dijkstra}$ from all the vertices of D_i. We now prove by induction on i, that if $u \in D_i$ and $v \in V$, then $\delta(u,v) \leq \delta_i(u,v) \leq \delta(u,v) + 2(i-1)$. Recall that $D_k = V$. For every $u, v \in V$, we get that $\hat{\delta}(u,v) = \delta_k(u,v) \leq \delta(u,v) + 2(k-1)$, as required.

Let $G_i(u) = (V, E_i(u))$ be the graph on which Dijkstra's algorithm is run from $u \in D_i$. For $i = 1$, the claim is clear, as for every $u \in D_1$ we have $G_1(u) = G$, and therefore $\delta_1(u,v) = \delta(u,v)$ for every $u \in D_1$ and $v \in V$. Suppose therefore that $i > 1$ and that the claim is true for $i-1$. Let $u \in D_i$ and let $v \in V$. Consider a shortest path p from u to v. If all the edges on p belong to E_i, then $\delta_i(u,v) = \delta(u,v)$ and we are done. Otherwise, there must be a vertex from V_{i-1} on the path p. The argument that follows is again similar to the argument used in case 1 in the proof of Theorem 3.1 and case 2 in the proof of Theorem 3.2. Let w be the last vertex from V_{i-1} on the path p. Let p' be the subpath of p that connects w and v. As all the vertices on p', except w, do not belong to V_{i-1}, all the edges of p' belong to E_i. Let $w' \in D_{i-1}$ be such that $(w, w') \in E^*$. The graph $G_i(u)$ contains the edge (w, w') and a weighted edge (u, w') whose weight is $\delta_{i-1}(u,w') = \delta_{i-1}(w',u)$. By the induction hypothesis,

Algorithm $\mathbf{apasp}_k$:

input: An unweighted undirected graph $G = (V, E)$.
output: A matrix $\{\hat{\delta}(u, v)\}_{u,v}$ of estimated distances.

For $i \leftarrow 1$ to $k-1$ let $s_i \leftarrow (m/n)^{1-\frac{i}{k}}$

For $i \leftarrow 1$ to $k-1$ let $V_i \leftarrow \{v \in V \mid \deg(v) \geq s_i\}$
For $i \leftarrow 2$ to k let $E_i \leftarrow \{(u, v) \in E \mid \deg(u) < s_{i-1} \text{ or } \deg(v) < s_{i-1}\}$

For $i \leftarrow 1$ to $k-1$ let $(D_i, E_i^*) \leftarrow \mathbf{dominate}(G, s_i)$
$E_1 \leftarrow E$; $D_k \leftarrow V$; $E^* \leftarrow \cup_{1 \leq i < k} E_i^*$

For every $u, v \in V$ let $\hat{\delta}(u, v) \leftarrow \begin{cases} 1 & \text{if } (u, v) \in E, \\ +\infty & \text{otherwise.} \end{cases}$

For $i \leftarrow 1$ to k do
For every $u \in D_i$ run $\mathbf{dijkstra}((V, E_i \cup E^* \cup (\{u\} \times V)), \hat{\delta}, u)$

Figure 4. An $\tilde{O}(n^{2-1/k}m^{1/k})$ time algorithm for computing surplus $2(k-1)$ distances

$\delta_{i-1}(w', u) \leq \delta(w', u) + 2(i-2) \leq \delta(u, w) + (2i-3)$. As a consequence, we get that

$$\begin{aligned} \delta_i(u, v) &\leq \delta_{i-1}(u, w') + \delta(w', w) + \delta(w, v) \\ &\leq (\delta(u, w) + (2i-3)) + 1 + \delta(w, v) \\ &\leq \delta(u, v) + 2(i-1) \ . \end{aligned}$$

This completes the proof of the theorem. □

As in the previous section, we can obtain a slightly better algorithm for denser graphs. Algorithm $\overline{\mathbf{apasp}}_k$ is described in Figure 5. There are two differences between $\mathbf{apasp}_k$ and $\overline{\mathbf{apasp}}_k$. The first is that the degree thresholds are now $s_i = n^{1-i/k}$, and not $s_i = (m/n)^{1-i/k}$. The second is that for any $1 \leq j_1, j_2 \leq k$ such that $i + j_1 + j_2 \leq 2k+1$, the edges of $D_{j_1} \times D_{j_2}$ are added to the graph on which Dijkstra's algorithm is run from every vertex $u \in D_i$.

Theorem 4.2 *For every $2 \leq k = O(\log n)$, the algorithm $\overline{\mathbf{apasp}}_k$ runs in $\tilde{O}(n^{2+1/k})$ time, where n is the number of vertices in the input graph $G = (V, E)$, and for every $u, v \in V$ we have $\delta(u, v) \leq \hat{\delta}(u, v) \leq \delta(u, v) + 2(\lfloor k/3 \rfloor + 1)$.*

The analysis of this algorithm is slightly more complicated than the analysis of the previous algorithms and it is omitted. To get an additive error of at most k, where $k > 2$ is even, we can either use the algorithm $\mathbf{apasp}_{\frac{k}{2}+1}$, whose running time is $\tilde{O}(n^{2-\frac{2}{k+2}}m^{\frac{2}{k+2}})$, or the algorithm $\overline{\mathbf{apasp}}_{(3k-2)/2}$, whose running time is $\tilde{O}(n^{2+2/(3k-2)})$. The combination of these two algorithms is the algorithm $\mathbf{APASP}_k$ mentioned in the abstract.

We can show that the multiplicative error of the estimates produced by the algorithm $\mathbf{apasp}_k$ is at most 3.

Theorem 4.3 *For every $2 \leq k = O(\log n)$, the algorithm $\mathbf{apasp}_k$ runs in $\tilde{O}(n^{2-1/k}m^{1/k})$ time, and for every $u, v \in V$ we have $\delta(u, v) \leq \hat{\delta}(u, v) \leq 3\delta(u, v) - 2$.*

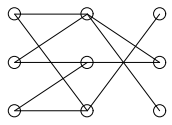
The proof of this theorem is again omitted. By taking $k = \Theta(\log n)$, we get an $\tilde{O}(n^2)$ time algorithm for finding stretch 3 approximate distances. An extension of this algorithm for weighted graph is presented in [14].

5 Boolean Matrix Multiplication

Let A and B be two Boolean $n \times n$ matrices. Construct a graph $G_{A,B} = (V, E)$ with

$$\begin{aligned} V &= \{u_1, \ldots, u_n\} \cup \{v_1, \ldots, v_n\} \cup \{w_1, \ldots, w_n\} \ , \\ E &= \{(u_i, v_k) \mid a_{ik} = 1\} \cup \{(v_k, w_j) \mid b_{kj} = 1\} \ . \end{aligned}$$

The graph corresponding to two 3×3 matrices is depicted in Figure 6. Let $C = A \times B$ (Boolean matrix multiplication). Clearly, $c_{ij} = 1$ if and only if $\delta_G(u_i, w_j) = 2$.



$$\begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} \times \begin{pmatrix} 0 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

Figure 6. Boolean matrix multiplication.

Algorithm $\overline{\mathbf{apasp}}_k$:

input: An unweighted undirected graph $G = (V, E)$.
output: A matrix $\{\hat{\delta}(u, v)\}_{u,v}$ of estimated distances.

For $i \leftarrow 1$ to $k-1$ let $s_i \leftarrow n^{1-\frac{i}{k}}$

For $i \leftarrow 1$ to $k-1$ let $V_i \leftarrow \{v \in V \mid \deg(v) \geq s_i\}$
For $i \leftarrow 2$ to k let $E_i \leftarrow \{(u, v) \in E \mid \deg(u) < s_{i-1} \text{ or } \deg(v) < s_{i-1}\}$

For $i \leftarrow 1$ to $k-1$ let $(D_i, E_i^*) \leftarrow \mathbf{dominate}(G, s_i)$
$E_1 \leftarrow E$; $D_k \leftarrow V$; $E^* \leftarrow \cup_{1 \leq i < k} E_i^*$

For every $u, v \in V$ let $\hat{\delta}(u, v) \leftarrow \begin{cases} 1 & \text{if } (u, v) \in E, \\ +\infty & \text{otherwise.} \end{cases}$

For $i \leftarrow 1$ to k do
For every $u \in D_i$ run $\mathbf{dijkstra}((V, E_i \cup E^* \cup (\{u\} \times V)) \cup (\cup_{i+j_1+j_2 \leq 2k+1} D_{j_1} \times D_{j_2})), \hat{\delta}, u)$

Figure 5. An $\tilde{O}(n^{2+1/k})$ time algorithm for computing surplus $O(k)$ distances

Furthermore, as the graph $G_{A,B}$ is bipartite, $c_{ij} = 1$ if and only if $\delta_G(u_i, w_j) \leq 3$. As an consequence we get

Theorem 5.1 *If all the distances in an undirected n vertex graph can be approximated with a one-sided additive error of at most one in $O(A(n))$ time, then Boolean matrix multiplication can also be performed in $O(A(n))$ time.*

By adding a disjoint path of length $k-2$ ending at each u_i, we get that distinguishing between distance k and $k+2$, for any *fixed* $k \geq 2$, is also as hard as Boolean matrix multiplication. Note that if $k \geq n^{2/3}$, then we can distinguish, with high probability, between distance k and $k+2$ in $\tilde{O}(n^{7/3})$ time.

Similarly, as any approximation algorithm that finds approximated distances of stretch strictly less than 2 can distinguish between distance 2 and distance 4, we get that getting approximate distances of stretch *less* than 2, between all pairs of vertices, is also as hard as Boolean matrix multiplication.

By turning the graph $G_{A,B}$ into a directed graph, where edges are directed to the right, we get that $c_{ij} = 1$ iff $\delta_G(u_i, w_j) < \infty$. Approximating distances in *directed* graphs, to within any *multiplicative* factor, is therefore as hard as Boolean matrix multiplication. It is not difficult to see that this remains so even if the directed graph is required to be strongly connected.

6 Distance Emulators

Closely related to the algorithms of Sections 3 and 4 is the notion of emulators.

Definition 6.1 (Emulators) *Let $G = (V, E)$ be an unweighted undirected graph. A weighted graph $F = (V, E')$ is said to be a* k-emulator *of G if and only if for every $u, v \in V$ we have $\delta_G(u, v) \leq \delta_F(u, v) \leq \delta_G(u, v) + k$.*

There is one significant difference, however, between emulators and the auxiliary graphs used in the algorithms of Sections 3 and 4. There, we constructed for each vertex u an auxiliary graph $G_k(u)$ that supplied good approximations to the distances from u to all the vertices of the graph. Here we want a single graph that will supply good approximations of *all* distances. Constructing a sparse k-emulator is therefore harder than computing surplus k distances.

The definition of k-emulators is related to the definition of *k-spanners* (Awerbuch [8], Peleg and Schäffer [24]). Let $G = (V, E)$ be a weighted undirected graph. A subgraph $G' = (V, E')$ of G is said to be a k-spanner of G if and only if for every $u, v \in V$ we have $\delta_{G'}(u, v) \leq k\delta_G(u, v)$. As G' is a subgraph of G, we always have $\delta_G(u, v) \leq \delta_{G'}(u, v)$. This definition differs from the definition of emulators in three respects. We require additive error, not multiplicative error. We do not insist on getting a subgraph of the original graph and we allow weighted edges. Althöfer *et al.* [7] also consider *Steiner spanners* in which vertices and edges may be added to the graph. Steiner spanners are more closely related to emulators. Liestman and Shermer [23] consider *additive spanners*. They are able, however, to obtain sparse additive spanners only for specific graphs such as pyramids, grids and hypercubes. Additive spanners, unlike emulators, must be subgraphs of the original graph. Emulators may be described as weighted additive Steiner spanners. The

definition of k-emulators is also related to the definition of *hop sets* (Cohen [13]).

Implicit in the work of Aingworth *et al.* [2] is an $\tilde{O}(n^{5/2})$ time algorithm for constructing 2-emulators with $\tilde{O}(n^{3/2})$ edges. We can get the following slightly stronger result.

Theorem 6.2 *Every unweighted undirected graph $G = (V, E)$ on n vertices can be 2-emulated by a* subgraph *$G' = (V, E')$ with $\tilde{O}(n^{3/2})$ edges. Such a subgraph can be constructed in $\tilde{O}(n^2)$ time.*

We omit the simple proof due to lack of space. A subgraph 2-emulator is also an additive 2-spanner and a multiplicative 3-spanner. In Section 7 (Theorem 7.3) we show that weighted graphs also have 3-spanners of size $\tilde{O}(n^{3/2})$. We present there a more efficient algorithm, whose running time is $\tilde{O}(mn^{1/2})$, for finding such 3-spanners. As there are bipartite graphs with $\Omega(n^{3/2})$ edges that do not contain cycles of length four [27], this result is tight, up to polylogarithmic factors. We can also show:

Theorem 6.3 *Every unweighted undirected graph $G = (V, E)$ on n vertices has a 4-emulator with $\tilde{O}(n^{4/3})$ edges. Such a graph can be constructed in $\tilde{O}(n^{7/3})$ time.*

Proof: It is not difficult to check that the graph $G_3 = (V, E_3 \cup E^* \cup (D_1 \times V) \cup (D_2 \times D_2))$ constructed by algorithm $\overline{\mathbf{apasp}}_3$ is a 4-emulator of $G = (V, E)$. □

It is tempting to claim that the k-emulators, for $k > 4$, can be similarly obtained by running $\overline{\mathbf{apasp}}_k$ with $k > 4$. Unfortunately, this is not true.

Let e_k be the infimum of all numbers for which each graph on n vertices has a k-emulator with $\tilde{O}(n^{1+e_k})$ edges. We have shown that $e_2 \leq 1/2$ and $e_4 \leq 1/3$. We conjecture that $e_k \to 0$ as $k \to \infty$. We are not able, however, to construct emulators with $o(n^{4/3})$ edges. We can, however, construct 6-emulators with $\tilde{O}(n^{4/3})$ edges in $\tilde{O}(n^2)$ time.

Theorem 6.4 *Let $G = (V, E)$ be an unweighted undirected graph of n vertices. A 6-emulator of G with $\tilde{O}(n^{4/3})$ edges can be constructed in $\tilde{O}(n^2)$ time.*

It is easy to see that k-emulators are Steiner $(k+1)$-spanners. Althöfer *et al.* [7] show that Steiner $(k+1)$-spanners of some graphs on n vertices must have $\tilde{\Omega}(n^{1+\frac{4}{3(k+3)}})$ edges. We can show that Steiner 3-spanners may require $\tilde{\Omega}(n^{3/2})$ edges and that Steiner 5-spanners may require $\tilde{\Omega}(n^{4/3})$ edges (where $\tilde{\Omega}(f) = \Omega(f \,/\, \text{polylog}\, n)$).

7 Stretched Paths and Distances

In this section we describe algorithms for finding stretched paths in *weighted* graphs. We use the following result which is part of the folklore.

Lemma 7.1 (Truncated Dijkstra) *Let $G = (V, E)$ be a weighted graph on n vertices. Suppose that the adjacency lists of the vertices of G are sorted according to weight. Let $v \in V$ be a vertex of G and let $1 \leq s \leq n$. Shortest paths from v to s vertices closest to v can be found in $\tilde{O}(s^2)$ time.*

The set of s vertices returned by the truncated Dijkstra algorithm running from v is not uniquely defined, as there may be many vertices at the same distance from v. All that we require is that if S is the set of vertices returned by the algorithm then for every $u \in S$ and $w \in V \setminus S$ we have $\delta(v, u) \leq \delta(v, w)$.

Theorem 7.2 *Let $G = (V, E)$ be a weighted undirected graph with n vertices and m edges. We can preprocess the graph in $\tilde{O}(m^{2/3}n)$ time so that given any two vertices $u, v \in V$, we can in $O(1)$ time output an estimated distance $\hat{\delta}(u, v)$ satisfying $\delta(u, v) \leq \hat{\delta}(u, v) \leq 3 \cdot \delta(u, v)$.*

Proof: Let s be a parameter to be chosen later. We run the truncated Dijkstra algorithm from every vertex $v \in V$ and find a set $N(v)$ of s vertices closest to v. The time required for finding these sets is $\tilde{O}(ns^2)$. Next, we find a set D of size $\tilde{O}(n/s)$ so that for every $v \in V$, there is $u \in D$ such that $u \in N(v)$. Such a set can be found in $O(ns)$ time. For every vertex $v \in V$, we keep a pointer to a vertex $u = P(v)$ such that $u \in D \cap N(v)$. We now run the full Dijkstra algorithm from all the vertices of D. The time required is $\tilde{O}(nm/s)$. We keep an $(n/s) \times n$ matrix with the distances from the vertices of D to all the other vertices of the graph. The time used so far is $\tilde{O}(ns^2 + nm/s)$. This is minimized by taking $s = m^{1/3}$. The total time is then $\tilde{O}(m^{2/3}n)$.

Given a pair of vertices u and v, we first check whether $v \in N(u)$. If so, we output the exact distance $\delta(u, v)$ computed during the truncated Dijkstra from u. Otherwise, we let $w = P(u) \in D \cap N(u)$ and we output the estimated distance $\hat{\delta}(u, v) = \delta(u, w) + \delta(w, v)$. The distance $\delta(u, w)$ was found during the truncated Dijkstra from u. The distance $\delta(w, v)$ was found during the full Dijkstra from w.

Clearly $\delta(u, v) \leq \hat{\delta}(u, v)$. If $v \in N(u)$, then $\hat{\delta}(u, v) = \delta(u, v)$. If $v \notin N(u)$, then $\delta(u, w) \leq \delta(u, v)$ and the estimate $\hat{\delta}(u, v)$ satisfies

$$\begin{aligned} \hat{\delta}(u, v) &= \delta(u, w) + \delta(w, v) \\ &\leq \delta(u, w) + (\delta(w, u) + \delta(u, v)) \\ &\leq 2\delta(u, w) + \delta(u, v) \;\leq\; 3\delta(u, v)\ , \end{aligned}$$

as required. □

Using essentially the same algorithm we can get:

Theorem 7.3 *Every weighted undirected graph $G = (V, E)$ on n vertices has a 3-spanner with $\tilde{O}(n^{3/2})$ edges. Such a 3-spanner can be constructed in $\tilde{O}(mn^{1/2})$ time.*

Proof: Let s be a parameter to be chosen later. Run the truncated Dijkstra algorithm from every vertex and find for every vertex $v \in V$ a set $N(v)$ of s vertices closest to v. Find a set D of size $\tilde{O}(n/s)$ such that for every $v \in V$, there is $u \in D \cap N(v)$. We then run a full Dijkstra from every vertex of D. The 3-spanner will be composed of the shortest paths trees found in all the truncated and full runs of Dijkstra's algorithm. The total number of edges will therefore be $\tilde{O}(ns + n^2/s)$. We choose $s = n^{1/2}$. The number of edges in the 3-spanner is then $\tilde{O}(n^{3/2})$ and the total running time is $\tilde{O}(ns^2 + nm/s) = \tilde{O}(n^2 + mn^{1/2})$. Note however, that if $mn^{1/2} \leq n^2$ then $m \leq n^{3/2}$ and the original graph is the required 3-spanner. □

If we are not interested in all distances, if we are willing to settle for a larger stretch factor, and if the number of edges in the graph is not too close to n^2, we can get more efficient algorithms. These will appear in the full version of the paper.

Acknowledgement

We would like to thank Howard Karloff for many helpful discussions and for comments on an earlier version of this extended abstract, and Edith Cohen and David Peleg for some clarifications regarding their papers.

References

[1] D. Aingworth, C. Chekuri, P. Indyk, and R. Motwani. Fast estimation of diameter and shortest paths (without matrix multiplication). Manuscript, 1996.

[2] D. Aingworth, C. Chekuri, and R. Motwani. Fast estimation of diameter and shortest paths (without matrix multiplication). In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms, Atlanta, Georgia*, pages 547–553, 1996.

[3] N. Alon, Z. Galil, and O. Margalit. On the exponent of the all pairs shortest path problem. In *Proceedings of the 32rd Annual IEEE Symposium on Foundations of Computer Science, San Juan, Puerto Rico*, pages 569–575, 1991.

[4] N. Alon, Z. Galil, O. Margalit, and M. Naor. Witnesses for boolean matrix multiplication and for shortest paths. In *Proceedings of the 33rd Annual IEEE Symposium on Foundations of Computer Science, Pittsburgh, Pennsylvania*, pages 417–426, 1992.

[5] N. Alon and J. Spencer. *The probabilistic method*. Wiley, 1992.

[6] N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. In *Proceedings of the 2nd European Symposium on Algorithms, Utrecht, The Netherlands*, Lecture Notes in Computer Science, Vol. 855, pages 354–364. Springer-Verlag, 1994. Journal version to appear in *Algorithmica*.

[7] I. Althöfer, G. Das, D. Dobkin, D. Joseph, and J. Soares. On sparse spanners of weighted graphs. *Discrete & Computational Geometry*, 9:81–100, 1993.

[8] B. Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32:804–823, 1985.

[9] B. Awerbuch, B. Berger, L. Cowen, and D. Peleg. Near-linear cost sequential and distributed constructions of sparse neighborhood covers. In *Proceedings of the 34rd Annual IEEE Symposium on Foundations of Computer Science, Palo Alto, California*, pages 638–647, 1993.

[10] J. Basch, S. Khanna, and R. Motwani. On diameter verification and boolean matrix multiplication. Technical Report STAN-CS-95-1544, Department of Computer Science, Stanford University, 1995.

[11] B. Chandra, G. Das, G. Narasimhan, and J. Soares. New sparseness results on graph spanners. In *Proceedings of the 8th Annual ACM Symposium on Computational Geometry, Berlin, Germany*, pages 192–201, 1992.

[12] E. Cohen. Fast algorithms for constructing t-spanners and paths with stretch t (extended abstract). In *Proceedings of the 34rd Annual IEEE Symposium on Foundations of Computer Science, Palo Alto, California*, pages 648–658, 1993.

[13] E. Cohen. Polylog-time and near-linear work approximation scheme for undirected shortest paths. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing, Montréal, Canada*, pages 16–26, 1994.

[14] E. Cohen and U. Zwick. All pairs small stretch paths. Submitted for publication, 1996.

[15] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9:251–280, 1990.

[16] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to algorithms*. The MIT Press, 1990.

[17] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[18] T. Feder and R. Motwani. Clique partitions, graph compression and speeding-up algorithms. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing, New Orleans, Louisiana*, pages 123–133, 1991.

[19] M. Fredman and R. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34:596–615, 1987.

[20] Z. Galil and O. Margalit. All pairs shortest distances for graphs with integer length edges. Submitted for publication, 1992.

[21] Z. Galil and O. Margalit. Witnesses for boolean matrix multiplication. *Journal of Complexity*, 9:201–221, 1993.

[22] D. Johnson. Efficient algorithms for shortest paths in sparse graphs. *Journal of the ACM*, 24:1–13, 1977.

[23] A. Liestman and T. Shermer. Additive graph spanners. *Networks*, 23:343–363, 1993.

[24] D. Peleg and A. Schäffer. Graph spanners. *Journal of Graph Theory*, 13:99–116, 1989.

[25] R. Seidel. On the all-pairs-shortest-path problem. In *Proceedings of the 24rd Annual ACM Symposium on Theory of Computing, Victoria, Canada*, pages 745–749, 1992.

[26] J. Ullman and M. Yannakakis. High-probability parallel transitive-closure algorithms. *SIAM Journal on Computing*, 20:100–125, 1991.

[27] R. Wenger. Extremal graphs with no C^4's, C^6's and C^{10}'s. *Journal of Combinatorial Theory, Series B*, 52:113–116, 1991.