All-Pairs Small-Stretch Paths

Edith Cohen * *Uri Zwick* †

Abstract

Let $G = (V, E)$ be a *weighted* undirected graph. A path between $u, v \in V$ is said to be of stretch t if its length is at most t times the distance between u and v in the graph. We consider the problem of finding small-stretch paths between all pairs of vertices in the graph G.

It is easy to see that finding paths of stretch *less* than 2 between all pairs of vertices in an undirected graph with n vertices is at least as hard as the Boolean multiplication of two $n \times n$ matrices. We describe three algorithms for finding small-stretch paths between all pairs of vertices in a weighted graph with n vertices and m edges. The first algorithm, $\mathbf{STRETCH}_2$, runs in $\tilde{O}(n^{3/2}m^{1/2})$ time and finds stretch 2 paths. The second algorithm, $\mathbf{STRETCH}_{7/3}$, runs in $\tilde{O}(n^{7/3})$ time and finds stretch 7/3 paths. Finally, the third algorithm, $\mathbf{STRETCH}_3$, runs in $\tilde{O}(n^2)$ and finds stretch 3 paths.

Our algorithms are simpler, more efficient and more accurate than the previously best algorithms for finding small-stretch paths. Unlike all previous algorithms, our algorithms are *not* based on the construction of sparse spanners or sparse neighbourhood covers.

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 or undirected, graphs without negative cycles is $O(mn + n^2 \log n)$, where n is the number of vertices and m is the number of edges in the graph (Johnson [19], see also [13]). If the weights are non-negative, then a slightly sub-cubic algorithm whose running time is $O(n^3((\log\log n)/\log n)^{1/2})$, obtained by Takaoka [22] (slightly improving an algorithm of Fredman [16]), can also be used. In this work we consider approximate solutions for the version of the problem in which all the weights are *non-negative* and the graph is *undirected*.

Let $G = (V, E)$ be a weighted undirected graph. A path between $u, v \in V$ is said to be of stretch t if its length is at most $t\,\delta(u, v)$, where $\delta(u, v)$ is the distance between u and v in the graph. Similarly, we say that an estimate $\hat{\delta}(u, v)$ of the distance between u and v in G is of stretch t if $\delta(u, v) \le \hat{\delta}(u, v) \le t\,\delta(u, v)$.

The special case of the all-pairs shortest paths problem in which the input graph is *unweighted* is closely related to matrix multiplication. Recent works, by Alon, Galil and Margalit [3], Alon, Galil, Margalit and Naor [4], Galil and Margalit [17],[18] and Seidel [21] have shown that if integer 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 [12]). The results of Galil and Margalit [17] also apply to graphs with small integer weights and they have been recently improved by Takaoka [23]. There is no known way, however, of using matrix multiplication for speeding up APSP algorithms for graphs with arbitrary non-negative weights.

It is observed in Dor *et al.* [15] that distinguishing between distances 2 and 4 in unweighted undirected graphs is at least as hard as Boolean matrix multiplication. Finding estimated distances of stretch strictly less than 2 between any pairs of vertices, even in unweighted undirected graphs, is thus also at least as hard as Boolean matrix multiplication. It is also observed in [15] that distinguishing between distances 2 and $+\infty$ in unweighted *directed* graphs is again at least as hard as Boolean matrix multiplication. Thus finding all-pairs estimated distances of any *finite* stretch in directed graphs is again as hard as Boolean matrix multiplication. It is not surprising therefore that the approximation methods used here, and elsewhere, work only for undirected graphs.

Algorithms for finding small-stretch paths were obtained by Awerbuch, Berger, Cowen and Peleg [8], Cohen [10], and Dor, Halperin and Zwick [15]. Awerbuch

*AT&T Labs – Research, Murray Hill, NJ 07974, USA. E-mail address: edith@research.att.com.

†Department of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel, International Computer Science Institute, 1947 Center Street, Suite 600, Berkeley, CA 94704, and Department of Computer Science, University of California, Berkeley, CA 94720. E-mail address: zwick@cs.berkeley.edu.

et al. [8] presented an $\tilde{O}(mn^{64/t} + kn^{32/t})$ time algorithm for finding stretch t paths between k specified pairs of vertices. If paths between all pairs of vertices are required, the running time of the algorithm becomes $\tilde{O}(mn^{64/t} + n^{2+32/t})$. Cohen [10] improved this result and obtained an $\tilde{O}((m+k)n^{2/t})$ time algorithm for obtaining stretch $t+\epsilon$ paths between k specified pairs of vertices, where t is even and $\epsilon > 0$ is arbitrarily small. This becomes $\tilde{O}(n^{2+2/t})$ if all-pairs stretch $t+\epsilon$ paths are wanted. In particular, Cohen [10] obtains an $\tilde{O}(n^{5/2})$ time algorithm for finding all-pairs stretch $4+\epsilon$ paths. Dor *et al.* [15] describe an $\tilde{O}((m^{2/3}+n)n)$ time algorithm for finding all-pairs stretch 3 paths.

Cohen [11] describes efficient parallel algorithms for obtaining stretched distances and paths from a specified set of source vertices. The algorithms run in polylogarithmic time and perform $O(n^{\epsilon}(m+ns))$ work, where s is the number of source vertices and $\epsilon > 0$ is arbitrarily small. The estimated distances produced by these algorithms satisfy $\hat{\delta}(u,v) \le (1+\epsilon')\delta(u,v) + \omega_{\max}\cdot\text{polylog}(n)$, where $\epsilon' > 0$ is again arbitrarily small and $\omega_{\max}$ is the maximal edge weight.

We substantially improve the results of Awerbuch *et al.* [8], Cohen [10] and Dor *et al.* [15]. We obtain three algorithms that exhibit a tradeoff between running time and accuracy. The first algorithm, $\mathbf{STRETCH}_2$, runs in $\tilde{O}(n^{3/2}m^{1/2})$ time and finds all-pairs stretch 2 paths. The second algorithm, $\mathbf{STRETCH}_{7/3}$, runs in $\tilde{O}(n^{7/3})$ time and finds all-pairs stretch 7/3 paths. Finally, the third algorithm, $\mathbf{STRETCH}_3$, runs in $\tilde{O}(n^2)$ time and finds all-pairs stretch 3 paths. The running time of the algorithm $\mathbf{STRETCH}_3$ comes very close to the naive $\Omega(n^2)$ lower bound.

The algorithms of Awerbuch *et al.* [8] and Cohen [10] are essentially based on the construction of sparse *spanners*. Let $G=(V,E)$ be a weighted undirected graph. A subgraph $G'=(V,E')$ is a t-spanner of G if and only if for every $u,v \in G$ we have $\delta_{G'}(u,v) \le t\,\delta_G(u,v)$. One approach for finding stretch t paths in a graph $G=(V,E)$ is the following: find a sparse t-spanner $G'=(V,E')$ of G, and then find shortest, or almost shortest, paths in G'. As the graph G' usually contains less edges than the original graph G, finding paths, or small-stretch paths, in G' can be done more efficiently. Spanners were studied by many researchers, including Awerbuch [7], Peleg and Schäffer [20] Althöfer *et al.* [6], Chandra *et al.* [9], Awerbuch *et al.* [8] and Cohen [10].

It is easy to see that for any $t < 3$, the only t-spanner of a bipartite graph $G=(V,E)$ is the graph G itself. Spanners cannot be used therefore for finding paths of stretch less than 3. Our approximation algorithms use a direct approach for solving the problem. They are *not* based on the construction of sparse spanners or sparse neighbourhood covers.

Our algorithms are based on ideas used by Aingworth, Chekuri and Motwani [2] ((see also [1])) for obtaining an $\tilde{O}(n^{5/2})$ time algorithm for finding all-pairs *surplus* 2 paths in an *unweighted* undirected graph with n vertices. A path between two vertices $u,v \in V$ is said to be of surplus k if and only if its length is at most $\delta(u,v)+k$. The $\tilde{O}(n^{5/2})$ time algorithm of Aingworth *et al.* is improved by Dor, Halperin and Zwick [15] who present an $\tilde{O}(\min\{n^{3/2}m^{1/2}, n^{7/3}\})$ time algorithm, $\mathbf{APASP}_2$, for finding all-pairs surplus 2 paths in a graph with n vertices and m edges. Algorithm $\mathbf{APASP}_2$ is the first algorithm in a sequence of algorithms that exhibits a tradeoff between running time and accuracy. For every *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 finds surplus k paths in unweighted graphs. Algorithm $\mathbf{APASP}_4$, for example, runs in $\tilde{O}(\min\{n^{5/3}m^{1/3}, n^{11/5}\})$ time and finds surplus 4 paths. As mentioned, Dor *et al.* [15] also obtain an $\tilde{O}((m^{2/3}+n)n)$ time algorithm for finding all-pairs stretch 3 paths in *weighted* graphs.

The ideas used by Aingworth *et al.* [2] and Dor *et al.* [15] for obtaining *additive* error approximations of distances in *unweighted* graphs are adapted here for obtaining *multiplicative* error approximations of distances in *weighted* graphs. The algorithms presented here are simple modifications of the algorithms presented in [15]. Their analysis, however, is slightly more complicated due to the presence of weights.

2 Preliminaries

Our algorithms use two basic ingredients. The first ingredient is a simple partitioning algorithm described in Figure 1. A similar partitioning algorithm is used, implicitly, in [15]. The second ingredient is the classical single-source shortest paths algorithm of Dijkstra. We use a version of this algorithm described in Figure 2.

The input to the partitioning algorithm **partition** described in Figure 1 is an undirected graph $G=(V,E)$ with a weight function $\omega : E \to R^+$, and a decreasing sequence $s_1 > s_2 > \cdots > s_{k-1}$ of *degree thresholds*. We assume, for simplicity, that the edges in each adjacency list of the graph $G=(V,E)$ are sorted in increasing order of weight. If not, we can sort them in $O(m\log n)$ time. Though the input graph to **partition** is an undirected graph, this graph is considered within this algorithm to be *directed*, with each undirected edge represented by two anti-parallel edges. If $e=(u,v) \in E$ is a directed edge, we let $\text{ind}(e)$, the *index* of e, be the position of e in the sorted adjacency list of u.

Algorithm **partition** produces, among other things,

```
Algorithm partition(G, ω, ⟨s_1, s_2, ..., s_{k-1}⟩):
input:  (i) An undirected graph G = (V, E) with a weight function ω : E → R^+.
        (ii) A decreasing sequence s_1, s_2, ..., s_{k-1} of degree thresholds.

output: (i) A sequence E_1, E_2, ..., E_k, E^* of edge sets.
        (ii) A sequence D_1, D_2, ..., D_k of vertex sets.
For i ← 2 to k do E_i ← {e ∈ E | ind(e) ≤ s_{i-1}}
For i ← 1 to k-1 do (D_i, E_i^*) ← dominate((V, E_{i+1}), s_i)
E_1 ← E ; D_k ← V ; E^* ← ⋃_{1≤i<k} E_i^*
```

Figure 1: A partitioning algorithm.

a decreasing sequence of directed edge sets $E_1 \supseteq E_2 \supseteq \cdots \supseteq E_k$, where $E_1 = E$ and E_i, for $1 < i \le k$, includes the s_{i-1} lightest edges emanating from each vertex. If a vertex of G is of out-degree less than s_{i-1}, then E_i includes all the edges emanating from it. As each vertex contributes at most s_{i-1} edges to E_i, the number of edges in E_i, for $1 < i \le k$, is at most ns_{i-1}.

Let $U \subseteq V$ be a set of vertices. A set of vertices D is said to *dominate* the set U in the graph $G = (V, E)$ if and only if every vertex of U is connected by an outgoing edge to a vertex of D. We use the following observation, similar to an observation of Aingworth *et al.* [2] (see also [5], pp. 6-7). Aingworth *et al.* consider undirected graphs but the generalization to directed graphs is immediate.

Lemma 2.1. *Let $G = (V, E)$ be a directed graph with n vertices and m edges. Let $1 \le s \le n$. A set D of size $O((n \log n)/s)$ that dominates all the vertices of out-degree at least s in the graph can be found deterministically in $O(m + ns)$ time.*

Picking each vertex of V to be in D independently at random with probability $(c \log n)/s$, for some large enough $c > 0$, yields a desired dominating set with high probability. The deterministic algorithm of Aingworth *et al.* is, in a sense, a derandomization of this algorithm. It is obtained using the simple greedy approximation algorithm for the set covering problem. See [2] for details.

The partitioning algorithm uses an algorithm, called **dominate**(G, s), that receives a directed 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 out-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 out-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^* an edge for each dominated vertex. According to Lemma 2.1, algorithm **dominate** can be implemented to run in $O(m + ns)$ time.

Let $G = (V, E)$ be the input graph to algorithm **partition**. Let V_i, for $1 \le i < k$, be the set of vertices of G of out-degree at least s_i. The out-degree of each vertex of V_i in the graph (V, E_{i+1}) is *exactly* s_i (we assume here that s_i is integral). The call to **dominate**$((V, E_{i+1}), s_i)$ produces therefore a pair (D_i, E_i^*) such that D_i is of size $O((n \log n)/s_i)$ and it dominates V_i through the edges $E_i^* \subseteq E_{i+1}$.

Outside **partition**, the input graph $G = (V, E)$ and the output edge sets $E_1, E_2, \ldots, E_k$ and E^* are considered to be *undirected*. If $e = (u, v)$ is an undirected edge, we let $\mathrm{ind}_u(e)$ be the index of e in the sorted adjacency list of u and $\mathrm{ind}_v(e)$ be its index in the sorted adjacency list of v.

The assumption that the edges of G are sorted according to weight is not really necessary. We can use a linear time selection algorithm instead for selecting the s_i lightest edges emanating from each vertex. We have thus established:

Lemma 2.2. *Algorithm* **partition** *runs in $O(m + n\sum_{i=1}^{k-1} s_i)$ time, where n is the number of vertices and m is the number of edges in the input graph $G = (V, E)$, and $s_1, \ldots, s_{k-1}$ are the degree thresholds. For every $1 < i \le k$, the set $E_i \subseteq E$ is of size $O(ns_{i-1})$. The set $E^* \subseteq E$ is of size $O(kn)$. For every $1 \le i < k$, the set $D_i \subseteq V$ is of size $O((n \log n)/s_i)$ and if $u \in V$ is of degree at least s_i in G then there is an edge $e = (u, u') \in E_{i+1} \cap E^*$ such that $\mathrm{ind}_u(e) \le s_i$ and $u' \in D_i$.*

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

Lemma 2.3. (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*

Algorithm **dijkstra**$(G, \hat{\delta}, s)$:

input: (i) A graph $G = (V, E)$ and a source $s \in V$.
(ii) A matrix $\{\hat{\delta}(u,v)\}_{u,v}$ of upper bounds on distances.

output: An updated matrix $\{\hat{\delta}(u,v)\}_{u,v}$ of upper bounds on distances.

Run the classical Dijkstra algorithm, from s, on the graph $G = (V, E)$, where the weight of an edge $(u,v) \in E$ is taken to be $\hat{\delta}(u,v)$. For every $v \in V$, let $\delta_G(s,v)$ be the distance found between s and v.

For every $v \in V$, let $\hat{\delta}(s,v) \leftarrow \hat{\delta}(v,s) \leftarrow \min\{\, \hat{\delta}(s,v)\, ,\, \delta_G(s,v)\, \}$.

Figure 2: Dijkstra's single-source shortest paths algorithm.

of shortest paths, from s to all the vertices of V.

Dijkstra's algorithm appeared originally in [14], though the running time of the version described there is $O(n^2)$. For a more modern description of Dijkstra's algorithm see [13]. The running time of $O(m+n\log n)$ is attained by using *Fibonacci heaps*. As we shall usually ignore logarithmic factors in the sequel, we may as well use the simpler implementation of Dijkstra's algorithm that uses binary heaps and runs in $O((m+n)\log n)$ time.

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 small-stretch 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 all 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 $\hat{\delta}(u,v) = \omega(u,v)$, if $(u,v) \in E$, where $\omega(u,v)$ is the weight attached to the edge (u,v) in the input graph, and $\hat{\delta}(u,v) = +\infty$ otherwise. We find it convenient to use the version of Dijkstra's algorithm described in Figure 2. By **dijkstra**$((V,F), \hat{\delta}, s)$ we denote an invocation of Dijkstra's algorithm, from the vertex s, 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 at most $\omega(u,v)$. The weight of (u,v) may be less than $\omega(u,v)$, however, if a shorter path between u and v was found during a previous invocation of Dijkstra's algorithm. A call to **dijkstra**$((V,F), \hat{\delta}, s)$ updates the row and the column belonging to s in the matrix $\hat{\delta}$ with the distances found during this run.

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 $\delta(u,v) \le \hat{\delta}(u,v)$, where $\delta(u,v)$ is the distance between u and v in the input graph G. In the presentation that follows we describe algorithms that find all-pairs small-stretch *distances*. All our algorithms can be easily extended to produce small-stretch paths.

3 A stretch 2 algorithm

An $\tilde{O}(n^{3/2}m^{1/2})$ time algorithm for finding all-pairs stretch 2 distances is described in Figure 3. The algorithm, $\mathbf{STRETCH_2}$, is extremely simple, and is very similar to algorithm $\mathbf{apasp_2}$ of [15]. Algorithm $\mathbf{apasp_2}$ finds surplus 2 distances in unweighted graphs in $\tilde{O}(n^{3/2}m^{1/2})$ time and it is forms a 'half' of algorithm $\mathbf{APASP_2}$.

Algorithm $\mathbf{STRETCH_2}$ starts by a call to **partition**. This call produces a set E_2 containing the $(m/n)^{1/2}$ lightest edges touching each vertex in the in-

Algorithm $\mathbf{STRETCH}_2(G, \omega)$:

input: An undirected graph $G = (V, E)$ with a weight function $\omega : E \to R^+$.
output: A matrix $\{\hat{\delta}(u, v)\}_{u,v}$ of estimated distances.

$((E_1, E_2, E^*), (D_1, D_2)) \leftarrow \mathbf{partition}(G, (m/n)^{1/2})$

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

Repeat twice
For $i \leftarrow 1$ to 2 do
For every $u \in D_i$ run $\mathbf{dijkstra}(V, E_i \cup (\{u\} \times V), \hat{\delta}, u)$

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

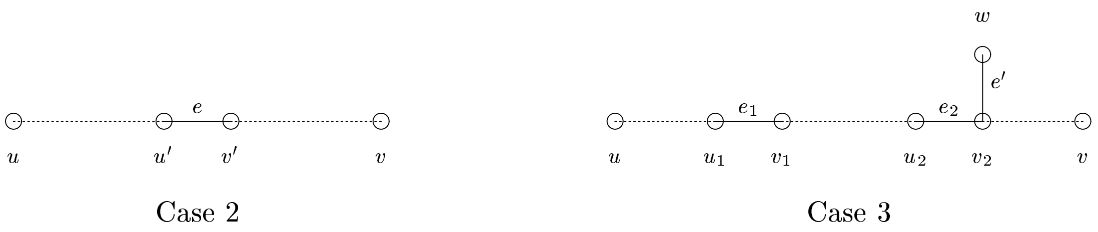

Figure 4: Cases 2 and 3 in the proof of Theorem 3.1.

put graph $G = (V, E)$, and a dominating set D_1 of size $\tilde{O}(n^{3/2}/m^{1/2})$ with the following property: if $u \in V$ is of degree at least $(m/n)^{1/2}$, then there is an edge $e = (u, u') \in E_2$ such that $\text{ind}_u(e) \le (m/n)^{1/2}$ and $u' \in D_1$. Recall that $E_1 = E$ and $D_2 = V$. The set E^* is not required by $\mathbf{STRETCH}_2$. Algorithm $\mathbf{STRETCH}_2$ then initializes the matrix $\{\hat{\delta}(u, v)\}_{u,v}$ of estimated distances with the edge weights.

The operation of $\mathbf{STRETCH}_2$ is then composed of two *iterations*. Each iteration is composed of two main steps. In the first step, Dijkstra's algorithm is run from each vertex of the dominating set D_1. In the second step, Dijkstra's algorithm is run from each vertex of $D_2 = V$. The edge set used when running **dijkstra** from a vertex $u \in D_1$ is $E_1 \cup (\{u\} \times V)$. As $E_1 = E$ this includes all the edges of the input graph. The edge set used when running **dijkstra** from a vertex $u \in D_2$, is $E_2 \cup (\{u\} \times V)$. This set is composed of the $(m/n)^{1/2}$ lightest edges touching each vertex, plus, edges connecting u with all the other vertices of the graph. In the first iteration, the weight of an edge $(u, v) \notin E$ is usually $+\infty$. In the second iteration, however, this weight is the upper bound on the distance between u and v found in the first iteration.

The operation of $\mathbf{STRETCH}_2$ is composed of relatively few invocations of Dijkstra's algorithm on the whole input graph, plus an invocation of Dijkstra's algorithm from each vertex on a much sparser graph. To be precise, Dijkstra's algorithm is invoked $\tilde{O}(n^{3/2}/m^{1/2})$ times on graphs with $O(m)$ edges, and $O(n)$ times on graphs with $O((nm)^{1/2})$ edges. The total runing time of all these invocations is therefore $\tilde{O}(n^{3/2}m^{1/2})$.

THEOREM 3.1. *Algorithm* $\mathbf{STRETCH}_2$ *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* $G = (V, E)$. *For every* $u, v \in V$, *we have* $\delta(u, v) \le \hat{\delta}(u, v) \le 2\,\delta(u, v)$.

Proof. The complexity analysis follows from the above discussion. We consider, therefore, the accuracy of the algorithm. For every $u, v \in V$ and $i = 1, 2$ let $\delta'_i(u, v)$ be value of $\hat{\delta}(u, v)$ after running **dijkstra** from all the vertices of D_i in the first iteration of the algorithm. Let $\delta''_i(u, v)$, for $i = 1, 2$, be the value of $\hat{\delta}(u, v)$ after running **dijkstra** from all the vertices of D_i in the second iteration of the algorithm. Note that $\delta''_2(u, v) \le \delta''_1(u, v) \le \delta'_2(u, v) \le \delta'_1(u, v)$ and that $\delta''_2(u, v)$ is the final value of $\hat{\delta}(u, v)$. Let $u, v \in V$ and consider a shortest path p between u and v. If x and y are vertices on p, we use $p_{x,y}$ to denote the portion of p that connects x and y. We consider three cases:

Case 1: All the edges on p belong to E_2.

The graph on which we run **dijkstra** from u contains all the edges of E_2 and, in particular, all the edges of p. We clearly get, in this case, that $\delta_2'(u,v) = \delta(u,v)$.

Case 2: Only one edge on p does not belong to E_2.

Let $e = (u',v')$ be the edge on p that does not belong to E_2 (see Figure 4). The edge e and all the edges on $p_{u,u'}$ belong to the graph $(V, E_2 \cup (\{v'\} \times V))$ on which we run **dijkstra** from v' in the second step of the first iteration. We get therefore that $\delta_2'(u,v') = \delta_2'(v',u) = \delta(u,v')$. The graph $(V, E_2 \cup (\{u\} \times V))$ on which we run **dijkstra** from u in the second step of the second iteration includes therefore an edge (u,v') of weight $\delta(u,v')$. This graph also contains all the edges of $p_{v',v}$. As a consequence, we get that $\delta_2''(u,v) = \delta(u,v)$.

Case 3: At least two edges on p do not belong to E_2.

Let $e_1 = (u_1,v_1)$ be the *first* edge on p not contained in E_2. Let $e_2 = (u_2,v_2)$ be the *last* edge on p not contained in E_2 (see Figure 4). We know that $e_1 \neq e_2$. Assume, without loss of generality, that $\omega(e_2) \le \omega(e_1)$ so, in particular, $\omega(e_2) \le \frac{1}{2}\,\delta(u,v)$ (otherwise, we switch the roles of u and v). As $e_2 \notin E_2$, we get that $\mathrm{ind}_{v_2}(e_2) > (m/n)^{1/2}$ and that the degree of v_2 is at least $(m/n)^{1/2}$. The vertex v_2 thus has a neighbour $w \in D_1$ such that $e' = (v_2,w) \in E_2$ and $\mathrm{ind}_{v_2}(e') \le (m/n)^{1/2}$. As $\mathrm{ind}_{v_2}(e') \le (m/n)^{1/2} < \mathrm{ind}_{v_2}(e_2)$, we get that $\omega(e') \le \omega(e_2)$.

As $w \in D_1$, the graph on which we run **dijkstra** from w in the first step of the first iteration includes all the edges of the input graph $G = (V,E)$. As a consequence we get that $\delta_1'(u,w) = \delta_1'(w,u) \le \delta(u,v_2) + \omega(e') \le \delta(u,v_2) + \omega(e_2)$. The graph on which we run **dijkstra** from u in the second step of the first iteration contains an edge (u,w) of weight $\delta_1'(u,w)$. This graph also contains the edge e', as $e' \in E_2$, and all the edges of $p_{v_2,v}$, as e_2 is the last edge on the path not belonging to E_2. As a consequence, we get that $\delta_2'(u,v) \le \delta_1'(u,w) + \omega(e') + \delta(v_2,v) \le \delta(u,v) + 2\omega(e_2) \le 2\,\delta(u,v)$.

In cases 1 and 2, the algorithm finds a shortest path between u and v. In case 3, the algorithm finds a stretch 2 path between u and v. This completes the proof of the theorem. □

If p is a path, we let $\omega(p)$ be the sum of the weights of the edges on the path. The argument used in the above proof can in fact be used to establish the following slightly stronger theorem.

THEOREM 3.2. *Let $\{\hat\delta(u,v)\}_{u,v}$ be the estimated distances produced by algorithm* **STRETCH$_2$**. *If p is a path between u and v in which the weight of the* second *heaviest edge is ω_2, then $\hat\delta(u,v) \le \omega(p) + 2\omega_2$.*

The path p here is not necessarily a shortest path between u and v. If p is a path of length one, then ω_2 is defined to be 0. It is easy to see that Theorem 3.2 implies Theorem 3.1.

Though the estimated distances produced by **STRETCH$_2$** may be, in the worst case, twice as large as the actual distances, Theorem 3.2 says that this can only happen when all the shortest paths between two vertices u and v are such that almost all their weight is concentrated on two edges of the path. Furthermore, the weights of these two heaviest edges should be of almost the same. In many practical situations, therefore, the stretch of most of the paths produced by **STRETCH$_2$** will be smaller than 2.

Algorithm **STRETCH$_2$** is similar to algorithm **apasp$_2$** of [15]. Algorithm **apasp$_2$** estimates distances in unweighted graphs with an *additive* error of at most 2. As can be seen from Theorem 3.2, the error of algorithm **STRETCH$_2$** is also additive. The additive error term is bounded this time in terms of weights of some of the edges that appear on a shortest path.

Algorithm **STRETCH$_2$** uses two iterations. Algorithm **apasp$_2$** uses only one. It is easy check that if u and v are connected by a path p such that the weight of the heaviest edge on p is ω_1, then after the first iteration we have $\hat\delta(u,v) \le \omega(p) + 2\omega_1$. While this is enough for getting surplus 2 for unweighted graphs, it may yield stretch 3 in the weighted case. The second iteration ensures that $\hat\delta(u,v) \le \omega(p) + 2\omega_2 \le \omega(p) + \omega_1 + \omega_2 \le 2\omega(p)$. While further iterations may improve some of the estimates produced by the algorithm, they seem to have no effect on its worst case behaviour.

4 A stretch 7/3 algorithm

An $\tilde O(n^{7/3})$ time algorithm for finding all-pairs stretch 7/3 distances is described in Figure 3. The algorithm, **STRETCH$_{7/3}$**, is similar to algorithm $\overline{\textbf{apasp}}_3$ of [15] which runs in $\tilde O(n^{7/3})$ time and finds surplus 2 paths in unweighted graphs. (Algorithms **apasp$_2$** and $\overline{\textbf{apasp}}_3$ together constitute algorithm **APASP$_2$**.)

Algorithm **STRETCH$_{7/3}$** is also similar to algorithm **STRETCH$_2$** presented in the previous section. There are two main differences. The first is that we now divide the edges into three classes, and not just two, as before. The second is that we effectively add the edge set $(D_1 \times V) \cup (D_2 \times V)$ to E_2 and the edge set $(D_1 \times V) \cup (D_2 \times D_2)$ to E_3.

Note that $|D_1| = \tilde O(n^{1/3})$, $|D_2| = \tilde O(n^{2/3})$, $|D_3| = |V| = n$ and that $|E_1| = O(n^2)$, $|E_2| = O(n^{5/3})$ and $|E_3| = O(n^{4/3})$. We get therefore that $|E_1'| = O(n^2)$, $|E_2'| = O(n^{5/3})$ and $|E_3'| = O(n^{4/3})$. We run Dijkstra's algorithm $\tilde O(n^{1/3})$ times on graphs with $O(n^2)$ edges, $\tilde O(n^{2/3})$ times on graphs with $O(n^{5/3})$ edges, and $O(n)$ times on graphs with $O(n^{4/3})$ edges. The total running time of the algorithm is therefore $\tilde O(n^{7/3})$.

`Algorithm` $\mathbf{STRETCH}_{7/3}(G,\omega)$:

`input:` An undirected graph $G=(V,E)$ with a weight function $\omega : E \rightarrow R^+$.
`output:` A matrix $\{\hat{\delta}(u,v)\}_{u,v}$ of estimated distances.

$((E_1,E_2,E_3,E^*),(D_1,D_2,D_3)) \leftarrow$ **partition**$(G,\langle n^{2/3},n^{1/3}\rangle)$

$E'_1 \leftarrow E_1$
$E'_2 \leftarrow E_2 \cup (D_1 \times V) \cup (D_2 \times V)$
$E'_3 \leftarrow E_3 \cup (D_1 \times V) \cup (D_2 \times D_2)$

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

Repeat twice
For $i \leftarrow 1$ to 3 do
For every $u \in D_i$ run **dijkstra**$(V, E'_i \cup (\{u\} \times V), \hat{\delta}, u)$

Figure 5: An $\tilde{O}(n^{7/3})$ time algorithm for computing stretch 7/3 distances.

THEOREM 4.1. *Algorithm* $\mathbf{STRETCH}_{7/3}$ *runs in* $\tilde{O}(n^{7/3})$ *time, where* n *is the number of vertices in the input graph* $G=(V,E)$. *For every* $u,v \in V$, *we have* $\delta(u,v) \le \hat{\delta}(u,v) \le \frac{7}{3}\,\delta(u,v)$.

Proof. For every $u,v \in V$ and $i=1,2,3$, let $\delta'_i(u,v)$ and $\delta''_i(u,v)$ be the values of $\hat{\delta}(u,v)$ after running **dijkstra** from all the vertices of D_i in the first and second iterations. Let $u,v \in V$ and consider a shortest path p between u and v. The proof consists of a case analysis similar to, but slightly more involved, than the one performed in the proof of Theorem 3.1. We consider five cases this time:

Case 1: There are at least two edges from $E_1 \setminus E_2$ on p.

This case is very similar to case 3 in the proof of Theorem 3.1. Let $e_1=(u_1,v_1)$ be the *first* edge from $E_1 \setminus E_2$ on p. Let $e_2=(u_2,v_2)$ be the *last* edge from $E_1 \setminus E_2$ on p. We know that $e_1 \ne e_2$ (see Figure 6). Assume, without loss of generality, that $\omega(e_2) \le \omega(e_1)$ so, in particular, $\omega(e_2) \le \frac{1}{2}\,\delta(u,v)$ (otherwise, we switch the roles of u and v). As $e_2=(u_2,v_2) \notin E_2$, we get that $\mathrm{ind}_{v_2}(e_2) > n^{2/3}$ and that the degree of v_2 is at least $n^{2/3}$. The vertex v_2 thus has a neighbour $w \in D_1$ such that $e'=(v_2,w) \in E_2$ and $\mathrm{ind}_{v_2}(e') \le n^{2/3}$. As $\mathrm{ind}_{v_2}(e') \le n^{2/3} < \mathrm{ind}_{v_2}(e_2)$, we get that $\omega(e') \le \omega(e_2)$. When we run **dijkstra** from $w \in D_1$ during the first iteration, we get that $\delta'_1(w,u) \le \delta(u,v_2)+\omega(e')$ and that $\delta'_1(w,v) \le \delta(v_2,v)+\omega(e')$. As $w \in D_1$, the graph on which we run **dijkstra** from $u \in D_3 = V$, in the third step of the first iteration, includes the edges (u,w) and (w,v). When we run **dijkstra** from u, we get, therefore, that $\delta'_3(u,v) \le \delta'_1(u,w)+\delta'_1(w,v) \le (\delta(u,v_2)+\omega(e'))+(\delta(v_2,v)+\omega(e')) \le \delta(u,v)+2\omega(e_2) \le 2\delta(u,v)$.

Case 2: There is exactly one edge from $E_1 \setminus E_2$ and at least one edge from $E_2 \setminus E_3$ on p.

Let $e=(u',v')$ be the only edge from $E_1 \setminus E_2$ on p. Assume, without loss of generality, that there is at least one edge from $E_2 \setminus E_3$ *after* e on p. Let $f_1=(u_1,v_1)$ be the first edge from $E_2 \setminus E_3$ after e on p. Let $f_2=(u_2,v_2)$ be the last edge from $E_2 \setminus E_3$ on p (see Figure 6). It may be the case that $f_1=f_2$.

As $e=(u',v') \notin E_2$, we get that $\mathrm{ind}_{v'}(e) > n^{2/3}$ and that the degree of v' is at least $n^{2/3}$. The vertex v' thus has a neighbour $w' \in D_1$ such that $e'=(v',w') \in E_2$ and $\mathrm{ind}_{v'}(e') \le n^{2/3}$. As $\mathrm{ind}_{v'}(e') \le n^{2/3} < \mathrm{ind}_{v'}(e)$, we get that $\omega(e') \le \omega(e)$. When we find distances from $w' \in D_1$ during the first iteration, we get that $\delta'_1(u,w') \le \delta(u,v')+\omega(e')$ and $\delta'_1(w',v) \le \delta(v',v)+\omega(e')$. When we find distances from $u \in D_3$, we get therefore that $\delta'_3(u,v) \le \delta(u,v)+2\omega(e')$.

As $f_1=(u_1,v_1) \notin E_3$, there is an edge $f'_1=(u_1,w_1) \in E_3$ such that $w_1 \in D_2$ and $\omega(f'_1) \le \omega(f_1)$. Similarly, there is an edge $f'_2=(u_2,w_2) \in E_3$ such that $w_2 \in D_2$ and $\omega(f'_2) \le \omega(f_2)$. When we find distances from $u' \in D_3 = V$ during the first iteration we get that $\delta'_3(u',w_1) \le \delta(u',u_1)+\omega(f'_1)$. When we find distances from $w_1 \in D_2$ during the second iteration, we get that $\delta''_2(w_1,u) \le \delta(u,u_1)+\omega(f'_1)$. When we find distances from $w_1 \in D_2$ during the first iteration, we get that $\delta'_2(w_1,w_2) \le \delta(u_1,v_2)+\omega(f'_1)+\omega(f'_2)$. Finally, when we find distances from $u \in D_3 = V$ during the second iteration we find that $\delta''_3(u,v) \le \delta''_2(u,w_1)+\delta''_2(w_1,w_2)+$

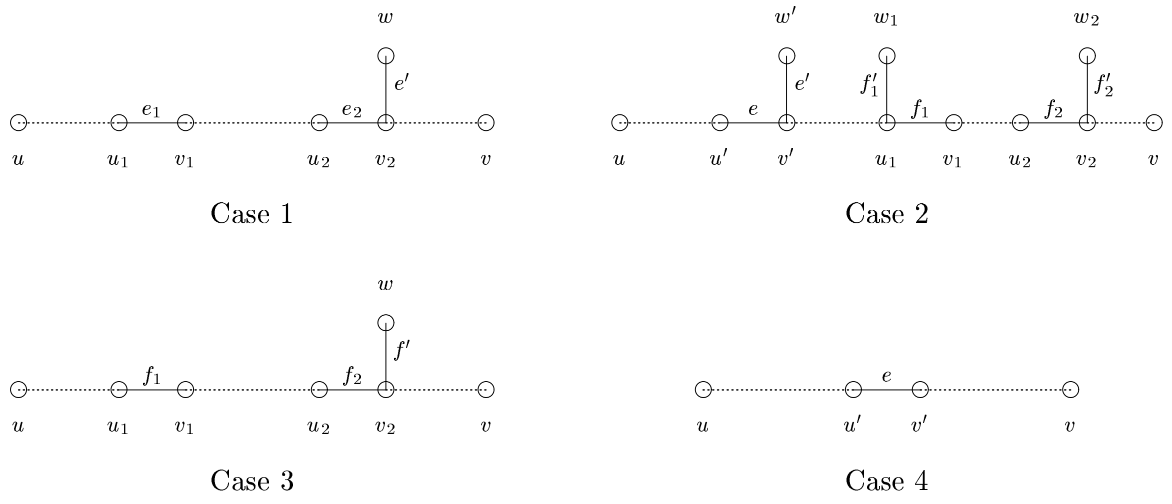

Figure 6: Cases 1 to 4 in the proof of Theorem 4.1.

$\omega(f_2') + \delta(v_2, v) \le \delta(u,v) + 2\omega(f_1') + 2\omega(f_2')$.

We get therefore that $\hat{\delta}(u,v) \le \min\{\,\delta(u,v) + 2\omega(e)\,,\, \delta(u,v) + 2\omega(f_1) + 2\omega(f_2)\,\}$. We now consider two subcases. If $f_1 \ne f_2$, then $\delta(u,v) \ge \omega(e) + \omega(f_1) + \omega(f_2)$ and it is easy to see that $\hat{\delta}(u,v) \le 2\,\delta(u,v)$. Assume therefore that $f_1 = f_2$. We get therefore that $\hat{\delta}(u,v) \le \min\{\,\delta(u,v) + 2\omega(e)\,,\, \delta(u,v) + 4\omega(f_1)\,\}$ while $\delta(u,v) \ge \omega(e) + \omega(f_1)$. It is easy to check that $\hat{\delta}(u,v) \le \frac{7}{3}\,\delta(u,v)$.

Case 3: There are no edges from $E_1 \setminus E_2$ on p and at least two edges from $E_2 \setminus E_3$ on p.

This case is similar to case 1. We omit the details.

Case 4: There is only one edge from $E_1 \setminus E_3$ on p.

This case is similar to case 2 of Theorem 3.1. We omit the details.

Case 5: All edges on p belong to E_3.

When we run **dijkstra** from $u \in D_3$ during the first iteration we find that $\delta_3'(u,v) = \delta(u,v)$.

This completes the proof of the theorem. Note that stretch 7/3 paths are obtained only in case 2. □

As in the previous section, the argument given proves the following slightly stronger result.

THEOREM 4.2. *Let $\{\hat{\delta}(u,v)\}_{u,v}$ be the estimated distances produced by algorithm* **STRETCH**$_{7/3}$. *If p is a path between u and v in which the weight of the heaviest edge is ω_1 and the weight of the second heaviest edge is ω_2, then $\hat{\delta}(u,v) \le \omega(p) + \min\{2\omega_1, 4\omega_2\}$.*

5 A stretch 3 algorithm

In this section we describe an $\tilde{O}(n^2)$ time algorithm for finding all-pairs stretch 3 distances. The algorithm, **STRETCH**$_3$, is described in Figure 7. It is a generalization of algorithm **APASP**$_\infty$ of Dor *et al.* [15] that runs in $\tilde{O}(n^2)$ time and finds stretch 3 distances and paths in unweighted graph. There are two main differences between algorithm **STRETCH**$_3$ and the previous two algorithms presented here. The first is that the edges are now divided into $\lfloor \log_2 n \rfloor$ classes. The second is that one iteration is now enough. We start by analyzing the running time of the algorithm.

LEMMA 5.1. *The running time of the algorithm* **STRETCH**$_3$ *is $\tilde{O}(n^2)$, where n is the number of vertices in the input graph $G = (V, E)$.*

Proof. For every $1 \le i \le k = \lfloor \log_2 n \rfloor$, algorithm **STRETCH**$_3$ invokes **dijkstra**'s algorithm $\tilde{O}(n/s_i)$ times on graphs with $O(ns_{i-1})$ edges (where $s_0 = n$). The total running time of the algorithm is therefore $\tilde{O}(n^2 \sum_{i=1}^{k} s_{i-1}/s_i) = \tilde{O}(n^2)$. □

For every $u, v \in V$ and for $1 \le i \le k$, we let $\delta_i(u,v)$ be the value of $\hat{\delta}(u,v)$ after running **dijkstra** from all the vertices of D_i. If p is a path, we let $\omega(p)$ be the sum of the weights of the edges on the path. The fact that the stretch of all the estimates found is at most 3 follows from the following lemma.

LEMMA 5.2. *Let p be a path connecting $u \in D_i$ and $v \in V$. Then, there is a set $\mathcal{E}_i(p)$ containing at most $i - 1$ edges of p not belonging to E_i such that $\delta_i(u,v) \le \omega(p) + 2\omega(\mathcal{E}_i(p))$.*

Proof. We prove the lemma by induction on i. For every $u \in D_1$ and $v \in V$ we have $\delta_1(u,v) = \delta(u,v)$ so the claim holds for $i = 1$. Suppose that the claim holds for every $1 \le j < i$. Let p be a path connecting $u \in D_i$ and $v \in V$. If all the edges on the path belong to E_i then $\delta_i(u,v) \le \omega(p)$ and we are done. Otherwise, let $e = (u', v')$ be the *last* edge on p that does not belong to E_i. Let v' be the endpoint of e closest to v. Let $1 \le j < i$ be such that $e \in E_j \setminus E_{j+1}$. As $e \notin E_{j+1}$, Algorithm $\mathbf{STRETCH}_3(G,\omega)$:

input: An undirected graph $G=(V,E)$ with a weight function $\omega: E \to R^+$.
output: A matrix $\{\hat{\delta}(u,v)\}_{u,v}$ of estimated distances.

Let $s_i = n2^{-i}$, for $1 \le i < k = \lfloor \log_2 n \rfloor$.
$((E_1,E_2,\ldots,E_k,E^*),(D_1,D_2,\ldots,D_k)) \leftarrow \mathbf{partition}(G,\langle s_1,s_2,\ldots,s_{k-1}\rangle)$

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

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

there is an edge $e' = (v',w) \in E_{j+1} \cap E^*$ such that $w \in D_j$ and $\omega(e') \le \omega(e)$.

Let p' be the path connecting u and w obtained by affixing e' to $p_{u,v'}$. By the induction hypothesis, applied to the path p', we get that

$$\begin{aligned} \delta_j(u,w) &= \delta_j(w,u) \le \omega(p') + 2\omega(\mathcal{E}_j(p')) \\ &\le \omega(p_{u,v'}) + \omega(e') + 2\omega(\mathcal{E}_j(p'))\ , \end{aligned}$$

where $\mathcal{E}_j(p')$ is a set of at most $j-1$ edges of p' not belonging to E_j. As $e' \in E_j$ we get that $e' \notin \mathcal{E}_j(p')$ and therefore all the edges of $\mathcal{E}_j(p')$ are from $p_{u,v'}$. As $e \in E_j$, we also get that $e \notin \mathcal{E}_j(p')$.

The graph on which we run **dijkstra** from $u \in D_i$ includes an edge (u,w) of weight at most $\delta_j(u,w)$. This graph also contains the edge $e' \in E^*$ and all the edges on the path $p_{v',v}$, as e was the last edge on p not contained in E_i. We get therefore that

$$\begin{aligned} \delta_i(u,v) &\le \delta_j(u,w) + \omega(e') + \omega(p_{v',v}) \\ &\le \omega(p_{u,v'}) + \omega(p_{v',v}) + 2\omega(e') + 2\omega(\mathcal{E}_j(p')) \\ &\le \omega(p) + 2\omega(e) + 2\omega(\mathcal{E}_j(p'))\ . \end{aligned}$$

As $\mathcal{E}_j(p')$ does not contain e, we can take $\mathcal{E}_i(p) = \mathcal{E}_j(p') \cup \{e\}$. The set $\mathcal{E}_i(p)$ contains at most $j \le i-1$ edges of p. All the edges of $\mathcal{E}_j(p')$ do not belong to E_i. The claim of the lemma follows. $\square$

Note that the claim of Lemma 5.2 holds for *any* path connecting u and v, not necessarily a shortest path. Combining the two lemmata, we get:

THEOREM 5.1. *Algorithm* $\mathbf{STRETCH}_3$ *runs in $\tilde{O}(n^2)$ time, where n is the number of vertices in the input graph $G=(V,E)$, and for every $u,v\in V$ we have $\delta(u,v) \le \hat{\delta}(u,v) \le 3\,\delta(u,v)$.*

Proof. The time bound follows from Lemma 5.1. Let p be a shortest path between u and v. As $\hat{\delta}(u,v) = \delta_k(u,v)$ and as $D_k = V$, we get, according to Lemma 5.2 that $\hat{\delta}(u,v) \le \omega(p) + 2\,\omega(\mathcal{E}_k(p))$. As $\mathcal{E}_k(p)$ is a subset of p we get that $\omega(\mathcal{E}_k(p)) \le \omega(p)$ and therefore $\hat{\delta}(u,v) \le 3\,\omega(p) \le 3\,\delta(u,v)$. This completes the proof of the theorem. $\square$

6 A family of stretch 3 algorithms

Let $\mathbf{STRETCH}_3^k$, for $k \ge 2$, be the algorithm obtained from $\mathbf{STRETCH}_3$ by replacing the threshold sequence by the sequence $\langle s_1,\ldots,s_{k-1}\rangle$, where $s_i = (m/n)^{1-i/k}$, for $1 \le i < k$. Algorithm $\mathbf{STRETCH}_3$ may be seen as the limit of this sequence of algorithms. Algorithm $\mathbf{STRETCH}_3^k$ is similar to algorithm $\mathbf{apasp}_k$ of Dor *et al.* [15]. Algorithm $\mathbf{STRETCH}_3^2$ is almost identical to algorithm $\mathbf{STRETCH}_2$ of Section 3. The only difference is that while two iterations are employed by $\mathbf{STRETCH}_2$, only one iteration is employed by $\mathbf{STRETCH}_3^2$. A second iteration does not improve the worst case behaviour of $\mathbf{STRETCH}_3^k$ for $k > 2$.

Using arguments similar to the ones used in the previous section we can prove the following theorem.

THEOREM 6.1. *Algorithm* $\mathbf{STRETCH}_3^k$ *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 $G=(V,E)$. If p is a path between u and v in which the weight of the i-th heaviest edge is ω_i, then $\hat{\delta}(u,v) \le \omega(p) + 2\sum_{i=1}^{k-1}\omega_i$. In particular, for every $u,v\in V$, we have $\delta(u,v) \le \hat{\delta}(u,v) \le 3\,\delta(u,v)$.*

The running time of the algorithms $\mathbf{STRETCH}_3^k$ decreases with k yet the stretch of the estimates produced by all these algorithms is at most 3. Consider, however, a pair of vertices u and v connected by a shortest path in which no single edge contributes more than, say, 1% to the total weight of the path. Algorithm $\mathbf{STRETCH}_2$, which runs in $\tilde{O}(n^{3/2}m^{1/2})$ time, is guaranteed to return a path between u and v of stretch at most 1.02, $\mathbf{STRETCH}_3^{11}$, which has a faster running time of $\tilde{O}(n^{21/11}m^{1/11}) \leq \tilde{O}(n^{23/11})$, is guaranteed to return a path of stretch at most 1.2. $\mathbf{STRETCH}_3$, which runs in $\tilde{O}(n^2)$ time, may, in certain cases, return a path of stretch 3.

7 Concluding remarks

We have presented very simple algorithms for finding all-pairs small-stretch paths in weighted undirected graphs. Algorithm $\mathbf{STRETCH}_2$ finds stretch 2 paths in $\tilde{O}(n^{3/2}m^{1/2})$ time. Stretch 2 is the smallest stretch achievable in a time which is below the time required for Boolean matrix multiplication. Algorithm $\mathbf{STRETCH}_3$ finds stretch 3 paths in $\tilde{O}(n^2)$ time. The running time of this algorithm is almost optimal. In many interesting cases, the stretch of the paths obtained by our algorithms are smaller than the worst case bounds given here.

Acknowledgement

We would like to thank Dorit Dor and Shay Halperin for some helpful discussions.

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.H. Spencer. *The probabilistic method.* Wiley, 1992.

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

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to algorithms.* The MIT Press, 1990.

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

[15] D. Dor, S. Halperin, and U. Zwick. All pairs almost shortest paths. In *Proceedings of the 37rd Annual IEEE Symposium on Foundations of Computer Science, Burlington, Vermont*, 1996. To appear.

[16] M.L. Fredman. New bounds on the complexity of the shortest path problem. *SIAM Journal on Computing*, 5:49–60, 1976.

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

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

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

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

[21] 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.

[22] T. Takaoka. A new upper bound on the complexity of the all pairs shortest path problem. *Information Processing Letters*, 43:195–199, 1992.

[23] T. Takaoka. Sub-cubic cost algorithms for the all pairs shortest path problem. In *Proceedings of the 21st International Workshop on Graph-Theoretic Concepts in Computer Science, Aachen, Germany*, Lecture Notes in Computer Science, Vol. 1017, pages 333–343. Springer-Verlag, 1995.