All Pairs Shortest Paths in weighted directed graphs – exact and almost exact algorithms

Uri Zwick *

Abstract

We present two new algorithms for solving the All Pairs Shortest Paths *(APSP) problem for weighted directed graphs. Both algorithms use fast matrix multiplication algorithms.*

The first algorithm solves the APSP problem for weighted directed graphs in which the edge weights are integers of small absolute value in $\tilde{O}(n^{2+\mu})$ time, where μ satisfies the equation $\omega(1,\mu,1) = 1 + 2\mu$ and $\omega(1,\mu,1)$ is the exponent of the multiplication of an $n \times n^{\mu}$ matrix by an $n^{\mu} \times n$ matrix. The currently best available bounds on $\omega(1,\mu,1)$, obtained by Coppersmith and Winograd, and by Huang and Pan, imply that $\mu < 0.575$. The running time of our algorithm is therefore $O(n^{2.575})$. Our algorithm improves on the $\tilde{O}(n^{(3+\omega)/2})$ time algorithm, where $\omega = \omega(1,1,1) < 2.376$ is the usual exponent of matrix multiplication, obtained by Alon, Galil and Margalit, whose running time is only known to be $O(n^{2.688})$.

The second algorithm solves the APSP problem almost *exactly for directed graphs with* arbitrary *non-negative real weights. The algorithm runs in $\tilde{O}((n^{\omega}/\epsilon)\cdot\log(W/\epsilon))$ time, where $\epsilon > 0$ is an error parameter and W is the largest edge weight in the graph, after the edge weights are scaled so that the smallest non-zero edge weight in the graph is 1. It returns estimates of all the distances in the graph with a stretch of at most $1+\epsilon$. Corresponding paths can also be found efficiently.*

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 with arbitrary real weights is $O(m^*n + n^2 \log n)$, where n is the number of vertices in the graph and m^* is the number of edges that participate in shortest paths (Dijkstra [9], Johnson [17], Fredman and Tarjan [12], Karger, Koller and Phillips [18], and McGeoch [20]).

The running time of the above algorithm may be as high as $\Omega(n^3)$. Can the APSP problem be solved in sub-cubic time? Fredman [11] showed that the APSP problem for weighted directed graphs can be solved *non-uniformly* in $O(n^{2.5})$ time. More precisely, for every n, there is a program that solves the APSP problem for graphs with n vertices using at most $O(n^{2.5})$ comparisons, additions and subtractions. But, a separate program has to be used for each input size. Furthermore, the size of the program that works on graphs with n vertices may be exponential in n. Fredman used this result to obtain a uniform algorithm that runs in $O(n^3((\log\log n)/\log n)^{1/3})$. Takaoka [23] slightly improved this bound to $O(n^3((\log\log n)/\log n)^{1/2})$. These running times are just barely sub-cubic.

The APSP problem is closely related to the problem of computing the min/plus product, or *distance product*, as we shall call it, of two matrices. If $A = (a_{ij})$ and $B = (b_{ij})$ are two $n \times n$ matrices, then their distance product $C = A \star B$ is an $n \times n$ matrix $C = (c_{ij})$ such that $c_{ij} = \min_{k=1}^{n} \{a_{ik} + b_{kj}\}$, for $1 \le i,j \le n$. A weighted graph $G = (V,E)$ on n vertices can be encoded as an $n \times n$ matrix $D = (d_{ij})$ in which d_{ij} is the weight of the edge (i,j), if there is such an edge in the graph, and $d_{ij} = +\infty$, otherwise. We also let $d_{ii} = 0$, for $1 \le i \le n$. It is easy to see that D^n, the n-th power of D with respect to distance products, is a matrix that contains the distances between all pairs of vertices in the graph (assuming there are no negative cycles). The matrix D^n can be computed using $\lceil \log_2 n \rceil$ distance products. It is, in fact, possible to show that the distance matrix D^n can be computed in essentially the same time required for just one distance product (see [1], Section 5.9).

Two $n \times n$ matrices over a *ring* can be multiplied using $O(n^{\omega})$ algebraic operations, where ω is the exponent of square matrix multiplication. The naive matrix multiplication algorithm shows that $\omega \le 3$. The best upper bound on ω is currently $\omega < 2.376$ (Coppersmith and Winograd [8]). The only lower bound available on ω is the naive lower

*Department of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel. E-mail address: zwick@math.tau.ac.il

bound $\omega \geq 2$. Unfortunately, the fast matrix multiplication algorithms cannot be used directly to compute distance products, as the set of integers, or the set of reals, is not a ring with respect to the operations min and plus.

Alon, Galil and Margalit [3] were the first to show that fast matrix multiplications algorithms can be used to obtain improved algorithms for the APSP problem for graphs with small integer weights. They obtained an algorithm whose running time is $\tilde{O}(n^{(3+\omega)/2})$ for solving the APSP problem for directed graphs with edge weights taken from the set $\{-1, 0, 1\}$. Galil and Margalit [14],[15], and independently Seidel [22], obtained $\tilde{O}(n^{\omega})$ time algorithms for solving the APSP problem for unweighted *undirected* graphs. Seidel's algorithm is much simpler. The algorithm of Galil and Margalit can be extended to handle small integer weights.

In this work we present an improved algorithm for solving the APSP problem for directed graphs with edge weights of small absolute value. The improved efficiency is gained by using *bridging sets* and by using *rectangular* matrix multiplications instead of square matrix multiplications, as used by Alon, Galil and Margalit [3]. It is possible to reduce a rectangular matrix multiplication into a number of square matrix multiplications. For example, the task of computing the product of an $n \times m$ matrix by an $m \times n$ matrix can be reduced to the task of computing $(n/m)^2$ products of $m \times m$ matrices. The running time of our algorithm, if we use this approach, is $\tilde{O}(n^{2+1/(4-\omega)})$, which is $\tilde{O}(n^{2.616})$, if we use the estimate $\omega < 2.376$. However, a more efficient implementation is obtained if we implement the rectangular matrix multiplications directly using the fastest rectangular matrix multiplication algorithms available. The running time of the algorithm is then $\tilde{O}(n^{2+\mu})$, where μ satisfies the equation $\omega(1, \mu, 1) = 1 + 2\mu$, where $\omega(1, \mu, 1)$ is the exponent of the multiplication of an $n \times n^{\mu}$ matrix by an $n^{\mu} \times n$ matrix. [1] The currently best available bounds on $\omega(1, \mu, 1)$, obtained by Coppersmith [7] and by Huang and Pan [16], imply that $\mu < 0.575$. The running time of out algorithm is therefore $O(n^{2.575})$, and possibly better.

If $\omega = 2$, as may turn out to be the case, then the running time of both our algorithm and the algorithm of Alon, Galil and Margalit, would be $\tilde{O}(n^{2.5})$. However, the running time of our algorithm may be $\tilde{O}(n^{2.5})$ even if $\omega > 2$. To show that the running time of our algorithm is $\tilde{O}(n^{2.5})$ it is enough to show that $\omega(1, \frac{1}{2}, 1) = 2$, i.e., that the product of an $n \times n^{1/2}$ matrix by an $n^{1/2} \times n$ matrix can be performed in $\tilde{O}(n^2)$ time. Coppersmith [7] showed that the product of an $n \times n^{0.294}$ matrix by an $n^{0.294} \times n$ matrix can be computed in $\tilde{O}(n^2)$ time.

The algorithm of Alon, Galil and Margalit [3] can also handle integer weights taken from the set $\{-M, \ldots, 0, \ldots, M\}$, i.e., integer weights of absolute value at most M. The running time of their algorithm is then $\tilde{O}(M^{(\omega-1)/2} n^{(3+\omega)/2})$, if $M \leq n^{(3-\omega)/(\omega+1)}$, and $\tilde{O}(M n^{(5\omega-3)/(\omega+1)})$, if $n^{(3-\omega)/(\omega+1)} \leq M$. Takaoka [24] obtained an algorithm whose running time is $\tilde{O}(M^{1/3} n^{(6+\omega)/3})$. The bound of Takaoka is better than the bound of Alon, Galil and Margalit for larger values of M. The running time of Takaoka's algorithm is sub-cubic for $M < n^{3-\omega}$.

Our algorithm can also handle small integer weights, i.e., weights taken from the set $\{-M, \ldots, 0, \ldots, M\}$. If rectangular matrix multiplications are reduced to square matrix multiplications, then the running time of the algorithm is $\tilde{O}(M^{1/(4-\omega)} n^{2+1/(4-\omega)})$. This running time is again sub-cubic for $M < n^{3-\omega}$ but, for every $1 \leq M < n^{3-\omega}$ the running time of our algorithm is faster than both the algorithms of Alon, Galil and Margalit and of Takaoka. The running time is further reduced if the rectangular matrix multiplications required by the algorithm are implemented using the best available algorithm. If $M = n^t$, where $t < 3 - \omega$, then the running time of the algorithm is $\tilde{O}(n^{2+\mu(t)})$, where $\mu = \mu(t)$ satisfies the equation $\omega(1, \mu, 1) = 1 + 2\mu - t$.

The new algorithm for solving the APSP problem for graphs with small integer weights is extremely simple and natural, despite the somewhat cumbersome bounds on its running time. We already noted that to compute all the distances in a weighed graph on n vertices represented by the matrix D it is enough to square the matrix D about $\log n$ times with respect to distance products. It turns out that if we are willing to repeat this process, say, $\log_{3/2} n$ times, then in the i-th iteration, instead of squaring the current matrix, it is enough to choose a set B_i of roughly $m_i = (2/3)^i n$ columns of the current matrix and multiply them by the corresponding m_i rows of the matrix. In fact, a randomly chosen set of about m_i columns would be a good choice with a very high probability! We have thus replaced the product of two $n \times n$ matrices in the i-th iteration by a product of an $n \times m_i$ matrix by an $m_i \times n$ matrix.

To convert distance products of matrices into normal algebraic products we use a technique suggested in [3] (see also Takaoka [24]), based on a previous idea of Yuval [25]. Suppose that $A = (a_{ij})$ and $B = (b_{ij})$ are two $n \times n$ matrices with elements taken from the set $\{-M, \ldots, 0, \ldots, M\}$. We convert A and B into two $n \times n$ matrices $A' = (a'_{ij})$ and $B' = (b'_{ij})$ where $a'_{ij} = (n+1)^{M-a_{ij}}$ and $b'_{ij} = (n+1)^{M-b_{ij}}$. It is not difficult to see that the distance product of A and B can be inferred from the algebraic product of A' and B' (see the next section). We pay, however, a high price for this solution. Each element of A' and B' is a huge number that about $M \log n$ bits, or about M words, are needed for its representation. An algebraic operation on elements of the matrices A' and B' *cannot* be viewed there-

[1] In general, $\omega(r, s, t)$ is the exponent of the multiplication of an $n^s \times n^r$ matrix by an $n^r \times n^t$ matrix.

fore as a single operation. Each such operation can be carried out, however, in $\tilde{O}(M \log n)$ time. We would have to take this factor into account in our complexity estimations.

Our work indicates that it may be possible to solve the APSP problem for directed graphs with small integer weights *uniformly* in $\tilde{O}(n^{2.5})$ time. Even if this were the case, there would still be a gap between the complexities of the directed and undirected versions of the APSP problem. As mentioned, the APSP for *undirected* graphs with small integer weights can be solved in $\tilde{O}(n^{\omega})$ time, as shown by Seidel [22] and by Galil and Margalit [14],[15].

We next show that the gap between the directed and the undirected versions of the APSP problem can be closed if we are willing to settle for *approximate* shortest paths. We say that a path between two vertices i and j is of stretch $1+\epsilon$ if its length is at most $1+\epsilon$ times the distance from i to j. It is fairly easy to see that paths of stretch $1+\epsilon$ between all pairs of vertices of an *unweighted* directed graph can be computed in $\tilde{O}(n^{\omega}/\epsilon)$ time. (This fact is mentioned in [14]). Stretch 2 paths, or at least stretch 2 distances, for example, may be obtained by computing the matrices A^{2^r}, for $1 \le r \le \lceil \log_2 n \rceil$, where A is the adjacency matrix of the graph, and Boolean products are used this time.

We extend this result and obtain an algorithm for finding stretch $1+\epsilon$ paths between all pairs of vertices of a directed graph with *arbitrary non-negative* real weights. The running time of the algorithm is $\tilde{O}((n^{\omega}/\epsilon) \cdot \log(W/\epsilon))$, where W is the largest edge weight in the graph after the edge weights are scaled so that the smallest non-zero edge weight is 1. Our algorithm uses a simple *adaptive scaling* technique. It is observed by Dor, Halperin and Zwick [10] that for any $c > 1$, computing stretch c distances between all pairs of vertices in an unweighted directed graph on n vertices is at least as hard as computing the Boolean product of two $n/3 \times n/3$ matrices. Our result is therefore very close to being optimal.

Algorithms for approximating the distances between all pairs of vertices in a weighted *undirected* graph were obtained by Cohen and Zwick [6]. They present an $\tilde{O}(n^2)$ algorithm for finding paths with stretch at most 3, an $\tilde{O}(n^{7/3})$ algorithm for finding paths of stretch $7/3$, and an $\tilde{O}(n^{3/2}m^{1/2})$ algorithm for finding paths of stretch 2. The algorithms of Cohen and Zwick [6] use ideas obtained by Aingworth, Chekuri, Indyk and Motwani [2] and by Dor, Halperin and Zwick [10] that design algorithms that approximate distances in unweighted undirected graphs with a small *additive* error. As can be seen from their running times, these algorithms are all purely combinatorial. They do not use fast matrix multiplication algorithms. It is also observed in [10] that for any $1 < c < 2$, computing stretch c distances between all pairs of vertices in an unweighted undirected graph on n vertices is again at least as hard as computing the Boolean product of two $n/3 \times n/3$ matrices. For $\epsilon < 1$, our algorithm is therefore close to optimal even for undirected graphs.

The rest of the paper is organized as follows. In the next section we present an algorithm that uses fast matrix multiplication to speed up the computation of distance products. In Section 3 we present a simple *randomized* algorithm for solving the APSP problem in directed graphs with small integer weights. In Section 4 we introduce the notion of *bridging sets* and explain how the randomized algorithm of the previous section can be converted into a deterministic algorithm. In Section 5 we present the new algorithm for obtaining an almost exact solution to the APSP problem for directed graphs with arbitrary non-negative real weights. Finally, we end in Section 6 with some concluding remarks and open problems.

2 Distance product of matrices

Let A be an $n \times m$ and B be an $m \times n$ matrix. The *distance product* of A and B, denoted $A \star B$, in an $n \times n$ matrix C such that

$$c_{ij} = \min_{k=1}^{m} \{a_{ik} + b_{kj}\} \text{ , for } 1 \le i, j \le n \text{ .}$$

The distance product of A and B can be computed naively in $O(n^2 m)$ time. Alon, Galil and Margalit [3] (see also Takaoka [24]) describe a way of using fast matrix multiplication, and fast integer multiplication, to compute distance products of matrices whose elements are taken from the set $\{-M, \ldots, 0, \ldots, M\} \cup \{+\infty\}$. The running time of their algorithm, when applied to rectangular matrices, is $\tilde{O}(M n^{\omega(1,r,1)})$, where $m = n^r$. Here $O(n^{\omega(1,r,1)})$ is the time required to compute an algebraic product of an $n \times n^r$ matrix by an $n^r \times n$ matrix when the entries in both matrices are small integers. We see, therefore, that the running time of this algorithm depends heavily on M. For large values of M the naive algorithm, whose running time is independent of M, is faster.

Algorithm **DIST-PROD**, whose description is given in Figure 1, uses the faster of these two methods to compute the distance product of an $n \times m$ matrix A and an $m \times n$ matrix B whose elements are integers. We let $m = n^r$. Elements in A and B that are of absolute value greater than M are treated as if they were $+\infty$. Algorithm **FAST-PROD**, called by **DIST-PROD**, computes the algebraic product of two integer matrices using the fastest rectangular matrix multiplication algorithm available, and using the Schönhage-Strassen [21] (see also [1]) algorithm for integer multiplication.

```
algorithm DIST-PROD(A, B, M)
if Mn^{ω(1,r,1)} ≤ n^{2+r}
then
```

$$a'_{ij} \leftarrow \begin{cases} (m+1)^{M-a_{ij}} & \text{if } |a_{ij}| \le M \\ 0 & \text{otherwise} \end{cases}$$

$$b'_{ij} \leftarrow \begin{cases} (m+1)^{M-b_{ij}} & \text{if } |b_{ij}| \le M \\ 0 & \text{otherwise} \end{cases}$$

$C' \leftarrow$ **FAST-PROD**(A', B')

$$c_{ij} \leftarrow \begin{cases} 2M - \lfloor \log_{(m+1)} c'_{ij} \rfloor & \text{if } c'_{ij} > 0 \\ +\infty & \text{otherwise} \end{cases}$$

`else`

$c_{ij} \leftarrow \min_{k=1}^{m} \{a_{ik} + b_{kj}\}$, $1 \le i, j \le n$.

```
endif
return C
```

Figure 1. Computing the distance product of two matrices.

Lemma 2.1 *Algorithm* **DIST-PROD** *computes the distance product of an $n \times n^r$ matrix by an $n^r \times n$ matrix whose finite entries are all of absolute value at most M in* $\tilde{O}(\min\{Mn^{\omega(1,r,1)}, n^{2+r}\})$ *time.*

Proof: If $n^{2+r} < Mn^{\omega(1,r,1)}$ then **DIST-PROD** computes the distance product of A and B using the naive algorithm that runs in $O(n^{2+r})$ time and we are done.

Assume, therefore, that $Mn^{\omega(1,r,1)} < n^{2+r}$. To see that the algorithm correctly computes the distance product of A and B in this case, note that for every $1 \le i, j \le n$ we have

$$c'_{ij} \;=\; \sum_{k=1}^{m} (m+1)^{2M-(a_{ik}+b_{kj})} \,,$$

where indices k for which $a_{ik} = +\infty$ or $b_{kj} = +\infty$ are excluded from the summation, and therefore

$$c_{ij} \;=\; \min_{k=1}^{m} \{a_{ik} + b_{kj}\} \;=\; 2M - \lfloor \log_{(m+1)} c'_{ij} \rfloor \,.$$

We next turn to the complexity analysis. If $Mn^{\omega(1,r,1)} \le n^{2+r}$ then **FAST-PROD** performs $\tilde{O}(n^{\omega(1,r,1)})$ arithmetical operations on $O(M \log n)$-bit integers. The Schönhage-Strassen integer multiplication algorithm multiplies two k-bit integers using $O(k \log k \log\log k)$ bit operations. Letting $k = O(M \log n)$, we get that the complexity of each arithmetic operation is $\tilde{O}(M \log n)$. Finally, the logarithms used in the computation of c_{ij} can be easily implemented using binary search. The complexity of the algorithm in this case is therefore $\tilde{O}(Mn^{\omega(1,r,1)})$, as required. □

There is, in fact, a slightly more efficient way of implementing **FAST-PROD**. Instead of computing the product of A' and B' using multiprecision integers, we can compute the product of A' and B' modulo about M different prime numbers with about $\log n$ bits each and then reconstruct the result using the Chinese remainder theorem.

What is known about $\omega(1, r, 1)$, the exponent of the multiplication of an $n \times n^r$ matrix by an $n^r \times n$ matrix? Note that $\omega = \omega(1,1,1)$ is the famous exponent of (square) matrix multiplication. The best bound on ω is currently $\omega < 2.376$ (Coppersmith and Winograd [8]). It is easy to see that a product of $n \times n^r$ matrix by an $n^r \times n$ matrix can be broken into $n^{2(1-r)}$ products of $n^r \times n^r$ matrices, and can therefore by computed in $O(n^{2+r(\omega-2)})$ time. It follows, therefore, that $\omega(1,r,1) \le 2 + r \cdot (\omega - 2)$. Better bounds are known. Coppersmith [7] showed that the product of $n \times n^{0.294}$ matrix by an $n^{0.294} \times n$ matrix can be computed using $\tilde{O}(n^2)$ arithmetical operations. Let $\alpha = \sup\{\, r : \omega(1,r,1) = 2 + o(1)\}$. It follows from Coppersmith's result that $\alpha \ge 0.294$. Note that if $\omega = 2 + o(1)$, then $\alpha = 1$. An improved bound for $\omega(1,r,1)$, for $\alpha \le r \le 1$ can be obtained by combining the bounds $\omega(1,1,1) < 2.376$ and $\omega(1,\alpha,1) = 2 + o(1)$. The following lemma is taken from Huang and Pan [16]:

Lemma 2.2 *Let* $\omega = \omega(1,1,1) < 2.376$ *and let* $\alpha = \sup\{\, r : \omega(1,r,1) = 2 + o(1)\} > 0.294$. *Then*

$$\omega(1,r,1) \le \begin{cases} 2 + o(1) & \textit{if } 0 \le r \le \alpha, \\ 2 + \frac{\omega-2}{1-\alpha}(r-\alpha) + o(1) & \textit{if } \alpha \le r \le 1. \end{cases}$$

We use these estimates in the next section.

Before ending this section, we introduce the notion of *witnesses* for distance products of matrices. An $n \times n$ matrix W is said to be a matrix of witnesses for the distance product $C = A \star B$ if for every $1 \le i, j \le n$ we have $c_{ij} = a_{i,w_{ij}} + b_{w_{ij},j}$. Witnesses for distance products will be used to reconstruct shortest paths.

Using ideas of Seidel [22], Galil and Margalit [13] and Alon and Naor [4], it is possible to extend algorithm **DIST-PROD** so that it would also return a matrix of witnesses. The running time of **DIST-PROD** would increase by only a polylogarithmic factor. The details will appear in the full version of this paper. In the sequel, we let $(C, W) \leftarrow$ **DIST-PROD**(A, B, M) denote an invocation of **DIST-PROD** that returns the product matrix C and a matrix of witnesses W.

3 A randomized algorithm

A simple randomized algorithm, **RAND-SHORT-PATH**, for finding distances, and a representation of shortest paths, `algorithm` **RAND-SHORT-PATH**(D)

$F \leftarrow D$; $W \leftarrow 0$
$M \leftarrow \max\{\ |d_{ij}| : d_{ij} \neq +\infty\}$
`for` $\ell \leftarrow 1$ `to` $\lceil \log_{3/2} n \rceil$ `do`
`begin`
  $s \leftarrow (3/2)^{\ell}$
  $B \leftarrow$ **RAND**$(\{1, 2, \ldots, n\}, (9 \ln n)/s)$
  $(F', W') \leftarrow$ **DIST-PROD**$(F[*, B], F[B, *], sM)$
  `for every` $1 \leq i, j \leq n$ `do`
  `if` $f'_{ij} < f_{ij}$ `then` $f_{ij} \leftarrow f'_{ij}$; $w_{ij} \leftarrow b_{w'_{ij}}$
`end`
`return` (F, W)

Figure 2. A randomized algorithm for finding shortest paths.

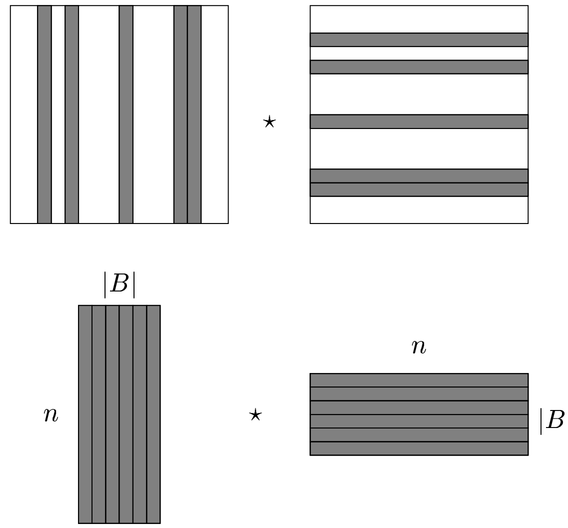

Figure 3. Replacing the square product $F \star F$ by the rectangular product $F[*, B] \star F[B, *]$.

between all pairs of vertices of a directed graph on n vertices in which all edge weights are taken from the set $\{-M, \ldots, 0, \ldots, M\}$ is given in Figure 2.

The input to **RAND-SHORT-PATH** is an $n \times n$ matrix D that contains the weights (lengths) of the edges of the input graph. We assume that the vertex set of the graph is $V = \{1, 2, \ldots, n\}$. The element d_{ij} is the weight of the directed edge from i to j in the graph, if there is such an edge, or $+\infty$, otherwise.

Algorithm **RAND-SHORT-PATH** starts by letting $F \leftarrow D$. The algorithm then performs $\lceil \log_{3/2} n \rceil$ iterations. In the ℓ-th iteration it lets $s \leftarrow (3/2)^{\ell}$. It then uses a function called **RAND** to produce a random subset B of $V = \{1, 2, \ldots, n\}$ obtained by selecting each element of V independently with probability $p = (9 \ln n)/s$. If $p \geq 1$, then **RAND** returns the set V. The algorithm then constructs the matrices $F[*, B]$ and $F[B, *]$. The matrix $F[*, B]$ is the matrix whose columns are the columns of F that correspond to the vertices of B. Similarly, $F[B, *]$ is the matrix whose rows are the rows of F that correspond to the vertices of B (see Figure 3). It then computes the distance product F' of the matrices $F[*, B]$ and $F[B, *]$ by calling **DIST-PROD**, putting a cap of sM on the absolute values of all the entries. The call also returns a matrix W' of witnesses. Finally, each element of F' is compared to the corresponding element of F. If the element of F' is smaller, then it is copied to F and the corresponding witness from W' is copied to W. (By $b_{w'_{ij}}$ we denote the w'_{ij}-th element of the set B.)

Let $\delta(i, j)$ denote the (weighted) distance from i to j in the graph. The following lemma is easily established:

Lemma 3.1 *At all times, for every $i, j \in V$:*

(i) $f_{ij} \geq \delta(i, j)$. (ii) If $f_{ij} \neq +\infty$, then either $f_{ij} = d_{ij}$ and $w_{ij} = 0$, or $w_{ij} \neq 0$ and $f_{ij} \geq f_{i,w_{ij}} + f_{w_{ij},j}$.

(iii) If $\delta(i, j) = \delta(i, k) + \delta(k, j)$ and if in the beginning of some iteration we have $f_{ik} = \delta(i, k)$, $f_{kj} = \delta(k, j)$, $|f_{ik}|, |f_{kj}| \leq sM$ and $k \in B$, then at the end of the iteration we have $f_{ij} = \delta(i, j)$.

Proof: Property (i) clearly holds when F is initialized to D. In each iteration, the algorithm chooses a set B and then lets

$$f'_{ij} \leftarrow \min\{\ f_{ik} + f_{kj} : k \in B\ ,\ |f_{ik}|, |f_{kj}| \leq sM\ \}$$
$$f_{ij} \leftarrow \min\{\ f_{ij}\ ,\ f'_{ij}\ \}$$

for every $i, j \in V$. For every k, we have $f_{ik} + f_{kj} \geq \delta(i, k) + \delta(k, j) \geq \delta(i, j)$, as follows from the induction hypothesis and the triangle inequality, and thus the new value of f_{ij} is again an upper bound on $\delta(i, j)$. Property (ii) also follows easily by induction.

If the conditions of property (iii) hold, then at the end of the iteration we have $f'_{ij} \leq f_{ik} + f_{kj} = \delta(i, k) + \delta(k, j) = \delta(i, j)$, and therefore also $f_{ij} \leq \delta(i, j)$. As $f_{ij} \geq \delta(i, j)$, by property (i), we get that $f_{ij} = \delta(i, j)$. $\square$

More interesting is the following lemma:

Lemma 3.2 *Let $s = (3/2)^{\ell}$, for some $1 \leq \ell \leq \lceil \log_{3/2} n \rceil$. With very high probability, if there is a shortest path from i to j in the graph that uses at most s edges then at the end of the ℓ-th iteration we have $f_{ij} = \delta(i, j)$.*

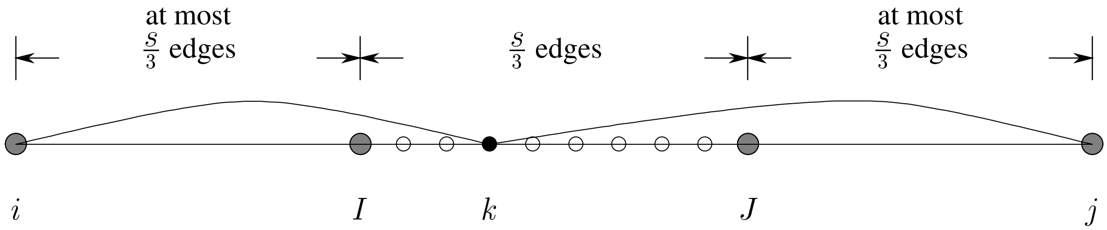

Figure 4. The vertices i and j are connected by a shortest path that uses at most s edges.

Proof: We prove the lemma by induction of ℓ. It is easy to see that the claim holds before the first iteration. We show now that if the claim holds for some $\ell - 1$, then it also holds for ℓ. Let i and j be two vertices connected by a shortest path that uses at most $s = (3/2)^\ell$ edges. Let p be such a shortest path from i to j. If the number of edges on p is at most $2s/3$, then after the $(\ell-1)$-st iteration we already have $f_{ij} = \delta(i,j)$ (with high probability). Suppose, therefore, that the number of edges on p is at least $2s/3$ and at most s.

To avoid technicalities, we 'pretend' at first that $s/3$ is an integer. We later indicate the changes needed to make the proof rigorous.

Let I and J be vertices on p such that I and J are separated, on p, by *exactly* $s/3$ edges, and such that i and I, and J and j are separated, on p, by *at most* $s/3$ edges. See Figure 4. Such vertices I and J can always be found as the path p is composed of at least $2s/3$ and at most s edges.

Let A be the set of vertices lying between I and J (inclusive) on p. Note that $|A| \geq s/3$. Let $k \in A$. As k lies on a shortest path from i to j, we have $\delta(i,j) = \delta(i,k) + \delta(k,j)$. As k lies between I and J, there are shortest paths from i to k, and from k to j that use at most $2s/3$ edges. By the induction hypothesis, we get that at the beginning of the ℓ-th iteration we have $f_{ik} = \delta(i,k)$ and $f_{kj} = \delta(k,j)$, with high probability. We also have $|f_{ik}|, |f_{kj}| \leq sM$. It follows, therefore, from Lemma 3.1(iii), that if there exist $k \in A \cap B$, where B is the set of vertices chosen at the ℓ-th iteration, then at the end of the ℓ-th iteration we have $f_{ij} = \delta(i,j)$, as required.

What is the probability that $A \cap B \neq \phi$? Let $p = (9 \ln n)/s$. If $p \geq 1$, then $B = V$ and clearly $A \cap B \neq \phi$. Suppose, therefore, that $p = (9 \ln n)/s < 1$. Each vertex then belongs to B independently with probability p. As $|A| \geq s/3$, the probability that $A \cap B = \phi$ is at most

$$\left(1 - \frac{9 \ln n}{s}\right)^{s/3} \leq \mathrm{e}^{-3 \ln n} = n^{-3} \, .$$

As there are less than n^2 pairs of vertices in the graph, the probability of failure during the entire operation of the algorithm is at most $n^2 \cdot n^{-3} = 1/n$. (We do not have to multiply the probability by the number of iterations, as it is enough to consider each pair of vertices just once.)

Unfortunately, $s/3$ is not an integer. To make the proof go through, we prove by induction a slight strengthening of the lemma. Define the sequence $s_0 = 1$ and $s_\ell = \lceil 3 s_{\ell-1}/2 \rceil$, for $\ell > 0$. Note that $s_\ell \geq (3/2)^\ell$. We show by induction on ℓ that, with high probability, for every $i, j \in V$, if there is a shortest path from i to j that uses at most s_ℓ edges, then at the end of the ℓ-th iteration $f_{ij} = \delta(i,j)$. The proof is almost the same as before. If p is a shortest path from i to j that uses at most s_ℓ edges, we consider vertices I and J on p such that I and J are separated by exactly $\lfloor s_\ell/2 \rfloor$ edges, and such that i and I, and J and j are separated by at most $\lceil s_\ell/2 \rceil$ edges. Repeating the above arguments we obtain a rigorous proof of the (strengthened) lemma. $\Box$

As each pair of vertices in a graph of n vertices is connected by a shortest path that uses at most n edges, assuming there are no negative cycles, we get that after the last iteration, F is, with high probability, the distance matrix of the graph.

It is also easy to see that the input graph contains a negative cycle if and only if $f_{ii} < 0$ for some $1 \leq i \leq n$. If there is a path from i to j that passes though a vertex contained in a negative cycle, we define the distance from i to j to be $-\infty$. Using a standard method, it is easy to identify all such pairs in $\tilde{O}(n^\omega)$ time. See [15] for the details.

If the input graph $G = (V,E)$ does not contain cycles of non-positive weight, then the matrix W can be easily used to obtain shortest paths. If there is a path from i to j, i.e., if $f_{ij} \neq +\infty$, then a shortest path from i to j can be obtained by concatenating a shortest path from i to w_{ij}, found recursively, and a shortest path from w_{ij} to j, also found recursively. If the graph does contain non-positive cycles, then the situation is slightly more complicated. For details, see the full version of this paper.

What is the complexity of **RAND-SHORT-PATH**? The time taken by the ℓ-th iteration is dominated by the time needed to compute the distance product of an $n \times m$ matrix by an $m \times n$ matrix, where $m = O((n \log n)/s)$, with entries of absolute value at most sM using **DIST-PROD**. If we assume that $s = n^{1-r}$ and $M = n^t$, then according to Lemma 2.1, this time is $\tilde{O}(\min\{n^{t+\omega(1,r,1)+(1-r)}, n^{2+r}\})$. Note that $\omega(1,r,1) + (1-r)$ is decreasing in r while $2+r$ is increasing in r. The running time of an iteration is maximized when $t + \omega(1,r,1) + (1-r) = 2 + r$, or equivalently, when $\omega(1, r, 1) = 1 + 2r - t$. As there are only $O(\log n)$ iterations, we get

Theorem 3.3 *Algorithm* **RAND-SHORT-PATH** *finds, with high probability, all distances in a graph on n vertices in which all the weights are integers with absolute values at most $M = n^t$, where $t \leq 3 - \omega$, in $\tilde{O}(n^{2+\mu(t)})$, where $\mu = \mu(t)$ satisfies $\omega(1, \mu, 1) = 1 + 2\mu - t$.*

Let us look more closely at the running time of the algorithm when $M = O(1)$. This is the case, for example, if all the weights in the graph belong to the set $\{-1, 0, 1\}$. The running time of the algorithm of Alon, Galil and Margalit in this case is $\tilde{O}(n^{(3+\omega)/2})$, which is about $O(n^{2.688})$. The running time of the new algorithm is $\tilde{O}(n^{2+\mu})$, where μ satisfies $\omega(1, \mu, 1) = 1 + 2\mu$. Using the naive bound $\omega(1, r, 1) \leq 2 + (\omega - 2)r$, we get that $\mu \leq \frac{1}{4-\omega} < 0.616$. Using the improved bound of Lemma 2.2, we get that $\mu \leq \frac{\alpha(\omega-1)-1}{\omega+2\alpha-4} < 0.575$.

Corollary 3.4
Algorithm **RAND-SHORT-PATH** *finds, with high probability, all distances in a graph on n vertices in which all the weights are taken from the set $\{-1, 0, 1\}$ in $O(n^{2.575})$ time.*

4 A deterministic algorithm

As before, we let $\delta(i, j)$ denote the (weighted) distance from i to j in the graph. We now let $\eta(i, j)$ denote the minimal number of edges on a shortest path from i to j. Note that if the graph is unweighted then $\delta(i, j) = \eta(i, j)$, for every $i, j \in V$.

Algorithm **RAND-SHORT-PATH** implicitly used the notion of *bridging sets* which we now formalize:

Definition 4.1 (Bridging sets) *Let $G = (V, E)$ be a weighted directed graph and let $s \geq 1$. A set of vertices B is said to be an* s-bridging set *if for every two vertices $i, j \in V$ such that $\eta(i, j) \geq s$, i.e., if all shortest paths from i to j use at least s edges, there exists $k \in B$, such that $\delta(i, j) = \delta(i, k) + \delta(k, j)$. The set B is said to be a* strong s-bridging set *if for every two vertices $i, j \in V$ such that $\eta(i, j) \geq s$, there exists $k \in B$, such that $\delta(i, j) = \delta(i, k) + \delta(k, j)$ and $\eta(i, j) = \eta(i, k) + \eta(k, j)$.*

The difference between bridging sets and strong bridging sets is depicted in Figure 5. All the paths shown there are shortest paths from i to j although they do not all use the same number of edges.

It is not difficult to see that if s is an integer then we can replace the condition $\eta(i, j) \geq s$ in the above definition by the condition $\eta(i, j) = s$.

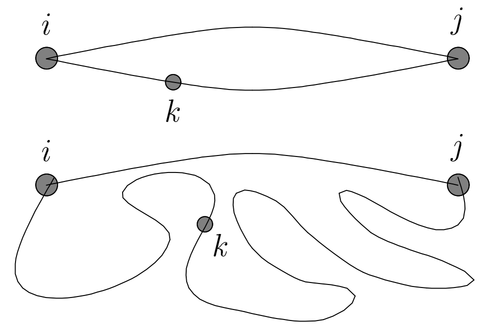

Figure 5. Bridging and strong bridging sets.

```
algorithm FIND-BRIDGE(W, s)
C ← φ
for every 1 ≤ i, j ≤ n do
    U ← SUB-PATH(W, i, j, s)
    if |U| ≥ s + 1 then C ← C ∪ {U}
end
B ← HITTING-SET(C)
return B
```

Figure 6. A deterministic algorithm for constructing an s-bridging set.

Reviewing the proof of Lemma 3.2, we see that algorithm **RAND-SHORT-PATH** remains correct as long as the set B used in the ℓ-th iteration is a *strong* $(s/3)$-bridging set. Implicit in the proof of Lemma 3.2 is the following result:

Lemma 4.2 *Let $G = (V, E)$ be a weighted directed graph on n vertices and let $s \geq 1$. If B is a random set obtained by running* **RAND**$(\{1, 2, \ldots, n\}, (3 \ln n)/s)$*, i.e., if each vertex of V is added to B independently with probability $(3 \ln n)/s$, then with high probability B is a* strong *s-bridging set.*

We next describe a deterministic algorithm, called **FIND-BRIDGE**, for finding s-bridging sets. A description of algorithm **FIND-BRIDGE** is given in Figure 6. It receives an $n \times n$ matrix of witnesses W using which it is possible to reconstruct shortest paths between all pairs of vertices $i, j \in V$ for which $\eta(i, j) \leq s$. We assume here, for simplicity, that the graph does not contain cycles of non-positive weight. We do *not* assume that the shortest paths encoded by W use a minimal number of edges.

Algorithm **FIND-BRIDGE** uses a procedure called **SUB-PATH** that receives two vertices $i, j \in V$ and an integer s and returns either an empty set, if no shortest path from i to j is encoded in W, or a set of at most $s+1$ vertices that constitute a shortest path from i to j that uses at most s edges, or a set of $s+1$ distinct vertices that lie on a shortest path from i to j. It is easy to see that **SUB-PATH** can be implemented to run in $O(s)$ time.

For every $i, j \in V$, let U_{ij} be the set obtained by calling **SUB-PATH**(W, i, j, s). If $\eta(i, j) = s$, then $|U_{ij}| \geq s+1$. Thus, if a set B *hits* all the sets U_{ij} such that $|U_{ij}| \geq s+1$, i.e., if $B \cap U_{ij} \neq \phi$ whenever $|U_{ij}| \geq s+1$, then B is s-bridging. Algorithm **FIND-BRIDGE** collects all the sets U_{ij} for which $|U_{ij}| \geq s+1$ into a collection of sets called $\mathcal{C}$. It then calls algorithm **HITTING-SET** to find a set that hits all the sets in this collection.

Algorithm **HITTING-SET** uses the greedy heuristic to find a set B that hits all the sets in the collection $\mathcal{C}$. As shown by Lovász [19] and Chvátal [5], the size of the hitting set returned by **HITTING-SET** is at most $(\ln \Delta) + 1$ times the size of the optimal *fractional* hitting set, where Δ is the maximal number of sets that a single element can hit. As each set in the collection $\mathcal{C}$ contains more than s elements, there is a fractional hitting set of size n/s. This fractional hitting set is obtained by giving each one of the n vertices of V a weight of $1/s$. As there are at most n^2 sets to hit, we get that $\Delta \leq n^2$. As a consequence we get that **FIND-BRIDGE** returns a bridging set of size at most $n(2\ln n+1)/s$. **HITTING-SET** can be easily implemented to run in time which is linear in the sum of the sizes of the sets in the collection. The running time of **FIND-BRIDGE** is therefore easily seen to be $O(n^2 s)$. We obtained, therefore, the following result:

Lemma 4.3 *If the matrix W can be used to reconstruct shortest paths between all pairs of vertices $i, j \in V$ for which $\eta(i, j) \leq s$, then algorithm* **FIND-BRIDGE** *finds an s-bridging set of size at most $n(2\ln n + 1)/s$. The running time of* **FIND-BRIDGE** *is $O(n^2 s)$.*

Unfortunately, the sets returned by **FIND-BRIDGE** are not necessarily strong bridging sets. But, if the input graph is *unweighted*, then an s-bridging set is also a strong s-bridging set. Thus, if we replace the call to **RAND** in **RAND-SHORT-PATH** by

```
if s ≤ n^{1/2} then
B ← FIND-BRIDGE(W, ⌊s/3⌋)
```

we obtain a deterministic algorithm for solving the APSP problem for *unweighted* directed graphs. We call this algorithm **SHORT-PATH**.

We compute new bridging sets only when $s \leq n^{1/2}$ as computing bridging sets for larger values of s may consume too much time. (Recall that the running time of **FIND-BRIDGE** is $O(n^2 s)$.) The algorithm remains correct as an s-bridging set is also an s'-bridging set for every $s' \geq s$. The use of a bridging set of size $\Theta(n^{1/2} \log n)$ in the iterations for which $s \geq n^{1/2}$ does not change the overall running time of the algorithm, as in all these iterations the required distance product can be computed using the naive algorithm in $\tilde{O}(n^{2.5})$ time. We thus get:

Theorem 4.4 *Algorithm* **SHORT-PATH** *solves the APSP problem for unweighted directed graphs* deterministically *in $\tilde{O}(n^{2+\mu})$ time, where $\mu < 0.575$ satisfies $\omega(1, \mu, 1) = 1 + 2\mu$.*

It is possible to modify **SHORT-PATH** so that it would also work for *weighted* directed graphs. We have no room to elaborate on this here. The details will appear in the full version of the paper.

5 Almost shortest paths

In this section we show that estimations with a relative error of at most ϵ of all the distances in a weighted directed graph on n vertices with *non-negative* integer weights bounded by M can be computed deterministically in $\tilde{O}((n^\omega/\epsilon) \cdot \log M)$ time. If the weights of the graphs are non-integral, we can scale them so that the minimal non-zero weight would be 1, multiply them by $1/\epsilon$, round them up and then run algorithm with the integral weights obtained. The running time of the algorithm would then be $\tilde{O}((n^\omega/\epsilon) \cdot \log(W/\epsilon))$, as claimed in the abstract and in the introduction.

For unweighted directed graphs, it is easy to obtain such estimates in $\tilde{O}(n^\omega/\epsilon)$ time. Let A be the adjacency matrix of the graph and let $\epsilon > 0$. By computing the Boolean matrices $A^{\lfloor (1+\epsilon)^\ell \rfloor}$ and $A^{\lceil (1+\epsilon)^\ell \rceil}$, for every $0 \leq \ell \leq \log_{1+\epsilon} n$, we can easily obtain estimates with a relative error of at most ϵ. The time required to compute all these matrices is $\tilde{O}(n^\omega/\epsilon)$. We next show that almost the same time bound can be obtained when the graph is weighted. The algorithm is again quite simple.

The main idea used to obtain almost shortest paths is *scaling*. A very simple scaling algorithm, called **SCALE**, is given in Figure 7. The algorithm receives an $n \times n$ matrix A containing *non-negative* elements. It returns an $n \times n$ matrix A'. The elements of A that lie in the interval $[0, M]$ are scaled, and rounded up, into the $R + 1$ different values $0, 1, \ldots, R$. We refer to R as the *resolution* of the scaling.

We next describe a simple algorithm for computing *approximate* distance products. The algorithm, called **APPROX-DIST-PROD**, is given in Figure 8. It receives two matrices A and B whose elements are non-negative integers. It uses *adaptive scaling* to compute a very accurate approximation of the distance product of A and B.

```
algorithm SCALE(A, M, R)
```

$$a'_{ij} \leftarrow \begin{cases} \lceil R a_{ij}/M \rceil & \text{if } 0 \le a_{ij} \le M \\ +\infty & \text{otherwise} \end{cases}$$

Return A'.

Figure 7. A simple scaling algorithm.

`algorithm` **APPROX-DIST-PROD**(A, B, M, R)
$C \leftarrow +\infty$
`for` $r \leftarrow \lfloor \log_2 R \rfloor$ `to` $\lceil \log_2 M \rceil$ `do`
`begin`
  $A' \leftarrow$ **SCALE**$(A, 2^r, R)$
  $B' \leftarrow$ **SCALE**$(B, 2^r, R)$
  $C' \leftarrow$ **DIST-PROD**(A', B', R)
  $C \leftarrow \min\{\ C\ ,\ (2^r/R) \cdot C'\ \}$
`end`
`return` C

Figure 8. Approximate distance products.

Lemma 5.1 *Let $\bar{C}$ be the distance product of the matrices obtained from the matrices A and B by replacing the elements that are larger than M by $+\infty$. Let M and R be powers of two. Let C be the matrix obtained by calling* **APPROX-DIST-PROD**(A, B, M, R). *Then, for every i, j we have $\bar{c}_{ij} \le c_{ij} \le (1 + \frac{4}{R})\bar{c}_{ij}$.*

Proof: The inequalities $\bar{c}_{ij} \le c_{ij}$ follow from the fact that elements are always rounded upwards by **SCALE**. We next show that $c_{ij} \le (1 + \frac{4}{R})\bar{c}_{ij}$. Let k be a witness for $\bar{c}_{ij}$, i.e., $\bar{c}_{ij} = a_{ik} + b_{kj}$. Assume, without loss of generality, that $a_{ik} \le b_{kj}$. Suppose that $2^{s-1} < b_{kj} \le 2^s$, where $1 \le s \le \log_2 M$ (the cases $b_{kj} = 0$ and $b_{kj} = 1$ are easily dealt with separately). If $s \le \log_2 R$, then in the first iteration of **APPROX-DIST-PROD**, when $r = \log_2 R$, we get $c_{ij} = \bar{c}_{ij}$. Assume, therefore, that $\log_2 R \le s \le \log_2 M$. In the iteration of **APPROX-DIST-PROD** in which $r = s$ we get that

$$\frac{2^r \cdot a'_{ik}}{R} \le a_{ik} + \frac{2^r}{R} \quad , \quad \frac{2^r \cdot b'_{kj}}{R} \le b_{kj} + \frac{2^r}{R} \ .$$

Thus, after the call to **DIST-PROD** we have

$$c_{ij} \le \frac{2^r \cdot a'_{ik}}{R} + \frac{2^r \cdot b'_{jk}}{R} \le a_{ik} + b_{kj} + \frac{2^{r+1}}{R} \le (1 + \frac{4}{R})\bar{c}_{ij} \ ,$$

as required. □

`algorithm` **APPROX-SHORT-PATH**(D, ϵ)
$F \leftarrow D$
$M \leftarrow \max\{\ d_{ij} : d_{ij} \ne +\infty\}$
$R \leftarrow 4\lceil \log_2 n \rceil / \ln(1+\epsilon)$
$R \leftarrow 2^{\lceil \log_2 R \rceil}$
`for` $\ell \leftarrow 1$ `to` $\lceil \log_2 n \rceil$ `do`
`begin`
  $F' \leftarrow$ **APPROX-DIST-PROD**(F, F, Mn, R)
  $F \leftarrow \min\{\ F\ ,\ F'\ \}$
`end`
`return` F

Figure 9. Approximate shortest paths.

If A and B are two $n \times n$ matrices, then the complexity of **APPROX-DIST-PROD** is $\tilde{O}(R \cdot n^{\omega} \cdot \log M)$. As we will usually have $R \ll M$, algorithm **APPROX-DIST-PROD** will usually be much faster than **DIST-PROD**, whose complexity is $\tilde{O}(M \cdot n^{\omega})$.

Algorithm **APPROX-SHORT-PATH**, given in Figure 9, receives as input an $n \times n$ matrix D representing the non-negative edge weights of a directed graph on n vertices, and an error bound ϵ. It computes estimates, with a stretch of at most $1 + \epsilon$, of all distances in the graph. Algorithm **APPROX-SHORT-PATH** starts by letting $F \leftarrow D$. It then simply squares F, using distance products, $\lceil \log_2 n \rceil$ times. Rather than compute these distance products exactly, it uses **APPROX-DIST-PROD** to obtain very accurate approximations of them.

Algorithm **APPROX-SHORT-PATH** uses a resolution R which is the smallest power of two greater than or equal to $4\lceil \log_2 n \rceil / \ln(1+\epsilon)$. Thus, $R = O((\log n)/\epsilon)$. Using Lemma 5.1, it is easy to show by induction that the stretch of the elements of F after the ℓ-th iteration is at most $(1 + \frac{4}{R})^{\ell}$. After $\lceil \log_2 n \rceil$ iterations, the stretch of the elements of F is at most

$$\left(1 + \frac{4}{R}\right)^{\lceil \log_2 n \rceil} \le \left(1 + \frac{\ln(1+\epsilon)}{\lceil \log_2 n \rceil}\right)^{\lceil \log_2 n \rceil} \le 1 + \epsilon \ .$$

As $R = O((\log n)/\epsilon)$, the complexity of each approximate distance product computed by **APPROX-SHORT-PATH** is $\tilde{O}((n^{\omega}/\epsilon) \cdot \log M)$. As only $\lceil \log_2 n \rceil$ such products are computed, this is also the complexity of the whole algorithm. We have thus established:

Theorem 5.2 *Algorithm* **APPROX-SHORT-PATH** *runs in $\tilde{O}((n^{\omega}/\epsilon) \cdot \log M)$ time and produces a matrix of estimated distances with a relative error of at most ϵ.*

As described, algorithm **APPROX-SHORT-PATH** finds approximate distances. It is easy to modify it so that it would also return a matrix W of witnesses using which approximate shortest paths could also be found.

6 Concluding remarks

The results of Seidel [22] and Galil and Margalit [14],[15] show that the complexity of the APSP problem for unweighted *undirected* graphs is $\tilde{O}(n^\omega)$. The exact complexity of the directed version of the problem is not known yet. In view of the results contained in this paper, there seem to be two plausible conjectures. The first is $\tilde{O}(n^{2.5})$. The second is $\tilde{O}(n^\omega)$. Galil and Margalit [14] conjecture that the problem for directed graphs is *harder* than the problem for undirected graphs. Proving, or disproving, this conjecture is a major open problem.

Another interesting open problem is finding the maximal value of M for which the APSP problem with integer weights of absolute value at most M can be solved in sub-cubic time. Our algorithm runs in sub-cubic time for $M < n^{3-\omega}$, as does the algorithm of Takaoka [24]. Can the APSP problem be solved in sub-cubic time, for example, when $M = n$?

We also presented an algorithm for obtaining almost exact solutions to the APSP problem for directed graphs with arbitrary non-negative real weights. For every desired accuracy ϵ, the algorithm performs only a polylogarithmic number of multiplications of matrices whose elements are small integers.

Acknowledgment

I would like to thank Victor Pan for sending me a copy of [16] and for answering several questions regarding it.

References

[1] A. Aho, J. Hopcroft, and J. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, 1974.

[2] D. Aingworth, C. Chekuri, P. Indyk, and R. Motwani. Fast estimation of diameter and shortest paths (without matrix multiplication). To appear in SIAM Journal on Computing. A preliminary version appeared in the 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. *Journal of Computer and System Sciences*, 54:255–262, 1997.

[4] N. Alon and M. Naor. Derandomization, witnesses for Boolean matrix multiplication and construction of perfect hash functions. *Algorithmica*, 16:434–449, 1996.

[5] V. Chvátal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4:233–235, 1979.

[6] E. Cohen and U. Zwick. All-pairs small-stretch paths. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms, New Orleans, Louisiana*, pages 93–102, 1997.

[7] D. Coppersmith. Rectangular matrix multiplication revisited. *Journal of Complexity*, 13:42–49, 1997.

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

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

[10] 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*, pages 452–461, 1996.

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

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

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

[14] Z. Galil and O. Margalit. All pairs shortest distances for graphs with small integer length edges. *Information and Computation*, 134:103–139, 1997.

[15] Z. Galil and O. Margalit. All pairs shortest paths for graphs with small integer length edges. *Journal of Computer and System Sciences*, 54:243–254, 1997.

[16] X. Huang and V. Pan. Fast rectangular matrix multiplications and applications. *Journal of Complexity*, 1998. To appear. Preliminary version appeared in the Proc. of the ACM Symposium on Parallel Algebraic and Symbolic Computation (PASCO'97), pp. 11-23.

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

[18] D. Karger, D. Koller, and S. Phillips. Finding the hidden path: time bounds for all-pairs shortest paths. *SIAM Journal on Computing*, 22:1199–1217, 1993.

[19] L. Lovász. On the ratio of optimal integral and fractional covers. *Discrete Mathematics*, 13:383–390, 1975.

[20] C. McGeoch. All-pairs shortest paths and the essential subgraph. *Algorithmica*, 13:426–461, 1995.

[21] A. Schönhage and V. Strassen. Schnelle multiplikation grosser zahlen. *Computing*, 7:281–292, 1971.

[22] R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of Computer and System Sciences*, 51:400–403, 1995.

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

[24] T. Takaoka. Subcubic cost algorithms for the all pairs shortest path problem. *Algorithmica*, 20:309–318, 1998.

[25] G. Yuval. An algorithm for finding all shortest paths using $N^{2.81}$ infinite-precision mutliplications. *Information Processing Letters*, 4:155–156, 1976.