Electronic Colloquium on Computational Complexity, Report No. 60 (2000)

All Pairs Shortest Paths using Bridging Sets and Rectangular Matrix Multiplication *

Uri Zwick †

August 16, 2000

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. Currently, the best available bounds on $\omega(1,\mu,1)$, obtained by Coppersmith, 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 $\tilde{O}((n^{\omega}/\epsilon)\log(W/\epsilon))$ 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 $O(mn + n^2 \log n)$, where n and m, respectively, are the number of vertices and edges in the graph. This algorithm is composed of a preliminary step, due to Johnson [Joh77], in which cycles of negative weight are found and eliminated, and a nonnegative weight function that induces the same shortest paths is found. The algorithm then proceeds by running Dijkstra's single source shortest paths algorithm (Dijkstra [Dij59]), implemented using Fibonacci heaps (Fredman and Tarjan [FT87]), from each vertex of the graph. For a clear description of the whole algorithm see Cormen, Leiserson and Rivest [CLR90], Chapters 21, 25 and 26.

For directed graphs with nonnegative edge weights, the running time of the above algorithm can be reduced to $O(m^*n + n^2 \log n)$, where m^* is the number of edges that participate in shortest paths (Karger, Koller and

*A preliminary version of this paper appeared in [Zwi98].

†Department of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel. E-mail address: zwick@math.tau.ac.il.
Work supported in part by THE ISRAEL SCIENCE FOUNDATION founded by The Israel Academy of Sciences and Humanities.

ISSN 1433-8092

Phillips [KKP93], and McGeoch [McG95]). For undirected graphs with nonnegative integer edge weights, a running time of $O(mn)$ can be obtained by running a recent single source shortest paths algorithm of Thorup [Tho99],[Tho00] from each vertex of the graph.

The running time of all the above mentioned algorithms may be as high as $\Omega(n^3)$. Can the APSP problem be solved in sub-cubic time? Fredman [Fre76] 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 $O(n^3((\log\log n)/\log n)^{1/3})$ time. Takaoka [Tak92] 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 [AHU74], 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 [CW90]). The only lower bound available on ω is the naive lower bound $\omega \ge 2$. 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 [AGM97] 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 edge weights. They obtained an algorithm whose running time is $\tilde{O}(n^{(3+\omega)/2})$ [1] for solving the APSP problem for directed graphs with edge weights taken from the set $\{-1, 0, 1\}$. Galil and Margalit [GM97a],[GM97b] and Seidel [Sei95], 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 has the advantage that it can be extended to handle small integer weights. The running time of their algorithm, when used to solve the APSP problem for undirected graphs with edge weights taken from the set $\{0, 1, \ldots, M\}$, is $\tilde{O}(M^{(\omega+1)/2} n^\omega)$. An improved time bound of $\tilde{O}(Mn^\omega)$ for the same problem was recently obtained by Shoshan and Zwick [SZ99].

In this paper 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 [AGM97]. 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 is easily reduced to the task of computing $(n/m)^2$ products of two $m \times m$ matrices. The running time of our algorithm, if we use this approach, is $\tilde{O}(n^{2+1/(4-\omega)})$, which is $O(n^{2.616})$, if we use the

[1] Throughout the paper, $\tilde{O}(f(n))$ stands for $O(f(n) \log^c n)$, for some $c > 0$.

estimate $\omega < 2.376$. However, a more efficient implementation is obtained if we compute 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. [2] Currently, the best available bounds on $\omega(1,\mu,1)$, obtained by Coppersmith [Cop97] and by Huang and Pan [HP98], imply that $\mu < 0.575$. The running time of our 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 [Cop97] showed that the product of an $n \times n^{0.294}$ by an $n^{0.294} \times n$ matrix can be computed in $\tilde{O}(n^2)$ time.

The algorithm of Alon, Galil and Margalit [AGM97] 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 $M \le n^{(3-\omega)/(\omega+1)}$, and $\tilde{O}(Mn^{(5\omega-3)/(\omega+1)})$, if $n^{(3-\omega)/(\omega+1)} \le M$. Takaoka [Tak98] 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 $1 \le M < n^{3-\omega}$ 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 computed 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 $\log_2 n$ 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 [AGM97] (see also Takaoka [Tak98]), based on a previous idea of Yuval [Yuv76]. 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 of $\log n$ bits each, are needed for its representation. An algebraic operation on elements of the matrices A' and B' *cannot* be viewed therefore 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.

[2] 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.

Our results 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 [Sei95] and by Galil and Margalit [GM97a],[GM97b]. (See also Shoshan and Zwick [SZ99].)

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 [GM97a]). 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 [DHZ00] that for any $c \ge 1$, 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 [CZ97]. 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 [CZ97] use ideas obtained by Aingworth, Chekuri, Indyk and Motwani [ACIM99] and by Dor, Halperin and Zwick [DHZ00] who designed 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 [DHZ00] that for any $1 \le c < 2$, 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 introduce the notion of *witnesses* for distance products. Such witnesses are used to reconstruct shortest paths. In Section 4 we present a simple *randomized* algorithm for solving the APSP problem in directed graphs with small integer weights. In Section 6 we introduce the notion of *bridging sets* and explain how the randomized algorithm of the previous section can be converted into a deterministic algorithm, if the input graph is unweighted. A deterministic algorithm for weighted graphs is then given in Section 7. In Section 8 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 9 with some concluding remarks and open problems.

2 Distance product of matrices

We begin with a definition of distance products.

```
algorithm dist-prod(A, B, M)
if Mn^ω(1,r,1) ≤ n^(2+r)
then
    a'_ij ← { (m+1)^(M−a_ij)   if |a_ij| ≤ M
            { 0                otherwise
    b'_ij ← { (m+1)^(M−b_ij)   if |b_ij| ≤ M
            { 0                otherwise
    C' ← fast-prod(A', B')
    c_ij ← { 2M − ⌊log_(m+1) c'_ij⌋   if c'_ij > 0
           { +∞                       otherwise
else
    c_ij ← min_{k=1}^{m} {a_ik + b_kj} , 1 ≤ i, j ≤ n .
endif
return C
```

Figure 1: Computing the distance product of two matrices.

Definition 2.1 (Distance products) *Let A be an $n \times m$ matrix 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*

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

In this definition, and in the rest of the paper, we use the convention that matrices are denoted by upper case letters, and that the elements of a matrix are denoted by the corresponding lower case letter.

The distance product of A and B can be computed naively in $O(n^2 m)$ time. Alon, Galil and Margalit [AGM97] (see also Takaoka [Tak98]) 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 $\tilde{O}(Mn^{\omega(1,r,1)})$, where $m = n^r$. Here $O(n^{\omega(1,r,1)})$ is the number of algebraic operations required to compute the standard algebraic product of an $n \times n^r$ matrix by an $n^r \times n$ matrix. 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 $+\infty$. (This feature is used by the algorithms described in the subsequent sections.) 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 [SS71] (see also [AHU74]) algorithm for integer multiplication.

Lemma 2.2 *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 $Mn^{\omega(1,r,1)} \le n^{2+r}$. 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. This reduces the running time, however, by only a polylogarithmic factor.

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 [CW90]). It is easy to see that a product of an $n \times n^r$ 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 $\omega(1,r,1) \le 2 + r(\omega - 2)$. Better bounds are known, however. Coppersmith [Cop97] showed that the product of an $n \times n^{0.294}$ matrix by an $n^{0.294} \times n$ matrix can be computed using $\tilde{O}(n^2)$ arithmetical operations. Let $\alpha = \sup\{\, 0 \le r \le 1 : \omega(1,r,1) = 2 + o(1)\}$. 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 [HP98]:

Lemma 2.3 *Let $\omega = \omega(1,1,1) < 2.376$ and let $\alpha = \sup\{\, 0 \le r \le 1 : \omega(1,r,1) = 2 + o(1)\} > 0.294$. Then*

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

Note that the upper bound on $\omega(1,r,1)$ given in Lemma 2.3 is a piecewise linear function. (See Figure 5 in Section 4.) Another well known fact (see, e.g., Pan [Pan85] or Burgisser, Clausen and Shokrollahi [BCS97]) regarding matrix multiplication, used in later sections, is the fact that $\omega(r,s,t)$, the exponent of computing the product of an $n^r \times n^s$ matrix and an $n^s \times n^t$ matrix, does not change if the order of its arguments is change. In particular:

Lemma 2.4 $\omega(1,1,r) = \omega(1,r,1) = \omega(r,1,1)$.

In other words, the cost of computing the product of an $n \times n^r$ matrix by an $n^r \times n$ matrix, and the cost of computing the product of an $n \times n$ matrix by an $n \times n^r$ matrix are asymptotically the same.

3 Witnesses for distance products

Next, we introduce the notion of *witnesses* for distance products of matrices. Witnesses for distance products are used to reconstruct shortest paths.

Definition 3.1 (Witnesses) *Let A be an $n \times m$ matrix and B be an $m \times n$ matrix. 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 $1 \le w_{ij} \le m$ and $c_{ij} = a_{i,w_{ij}} + b_{w_{ij},j}$.*

Using ideas of Seidel [Sei95], Galil and Margalit [GM93] and Alon and Naor [AN96], it is easy 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 are sketched below.

There is a simple, but expensive, way of computing witnesses for the distance product $C = A \star B$, where A is an $n \times m$ matrix, and B is an $m \times n$ matrix. Let $A' = (a'_{ij})$ and $B' = (b'_{ij})$ be matrices such that $a'_{ij} = ma_{ij} + j - 1$ and $b'_{ji} = mb_{ji}$, for every $1 \le i \le n$ and $1 \le j \le m$. If we compute the distance product $C' = A' \star B'$, then $\lfloor C'/m \rfloor$ is the distance product of $A \star B$ and $(C' \bmod m) + 1$ is a corresponding matrix of witnesses. Furthermore, all the witnesses in this matrix are the *smallest possible* witnesses. The drawback of this approach is that the entries of A and B are multiplied by m and this may slow down the operation **dist-prod** by a factor of m, which may be a huge factor.

There is, however, a much more efficient way of finding witnesses. We show, at first, how to find witnesses for elements that have *unique* witnesses. For $1 \le k \le m$ and $1 \le \ell \le \lceil \log_2 m \rceil + 1$, we let $bit_\ell(k)$ be the ℓ-th bit in the binary representation of k. (For concretness, assume that $bit_1(k)$ is the least significat bit in the representation of k.) For $1 \le \ell \le \lceil \log_2 m \rceil + 1$, let $I_\ell = \{1 \le k \le m \mid bit_\ell(k) = 1\}$. We also need the following definition which is also used in subsequent sections:

Definition 3.2 (Sampling) *Let A be an $n \times m$ matrix, and let $I \subseteq \{1, 2, \ldots, m\}$. Then, $A[*, I]$ is defined to be the matrix composed of the columns of A whose indices belong to I. Similarly, if B is an $m \times n$ matrix, then $B[I, *]$ is defined to be the matrix composed of the rows of B whose indices belong to I.*

To find witnesses for all elements of $A = B \star C$ that have a unique witness, we compute the $O(\log m)$ distance products $C_\ell = A[*, I_\ell] \star B[I_\ell, *]$, for $1 \le \ell \le \lceil \log_2 m \rceil + 1$. Let $C_\ell = (c^{(\ell)}_{ij})$. It is easy to see that $c^{(\ell)}_{ij} = c_{ij}$, if and only if there is a witness for c_{ij} whose ℓ-th bit is 1. If c_{ij} has a unique witness w_{ij}, then these conditions can be used to identify the individual bits in the binary representation of w_{ij}, and hence w_{ij} itself. Note that we do not have to know in advance whether c_{ij} has a unique witness. We just reconstruct a candidate witness w_{ij} and then check whether $c_{ij} = a_{i,w_{ij}} + b_{w_{ij},j}$.

What do we do with elements that have more than one witness? We use sampling. For every $1 \le r \le \log m$, we choose $s = c \log n$ random subsets $R_{r1}, \ldots, R_{rs}$ of $\{1, 2, \ldots, m\}$ of size $m/2^r$. For every such random set R_{rt}, where $1 \le r \le \log m$ and $1 \le t \le s$, we try to find unique witnesses for the product $A[*, R_{rt}] \star B[R_{rt}, *]$. When such a witness is found, we check whether it is also a witness for the original distance product $A \star B$. A simple calculation, identical to a calculation that appears in Seidel [Sei95], shows that if the constant c is taken to be large enough, then with very high probability, we will find in this way witnesses for all positions.

The above discussion gives a randomized algorithm for computing a matrix of witnesses for the distance product $A \star B$. The randomized algorithm uses $O(\log^3 n)$ ordinary distance products of matrices of equal or smaller size. The resulting algorithm can be derandomized using the results of Alon and Naor [AN96]. We thus obtain:

```
algorithm rand-short-path(D)
F ← D ; W ← 0
M ← max{ |d_ij| : d_ij ≠ +∞}
for ℓ ← 1 to ⌈log_{3/2} n⌉ do
begin
  s ← (3/2)^ℓ
  B ← rand({1, 2, ..., n}, (9 ln n)/s)
  (F', W') ← dist-prod(F[*, B], F[B, *], sM)
  for every 1 ≤ i, j ≤ n do
  if f'_ij < f_ij then f_ij ← f'_ij ; w_ij ← b_{w'_ij} endif
end
return (F, W)
```

Figure 2: A randomized algorithm for finding shortest paths.

Lemma 3.3 *An extended version of 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, and a corresponding matrix of witnesses, in $\tilde{O}(\min\{Mn^{\omega(1,r,1)}, n^{2+r}\})$ time.*

In the sequel, we let $(C, W) \leftarrow$ **dist-prod**(A, B, M) denote an invocation of the extended version of **dist-prod** that returns the product matrix C and a matrix of witnesses W.

4 A randomized algorithm for finding shortest paths

A simple randomized algorithm, **rand-short-path**, for finding distances, and a representation of shortest paths, 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 (or 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 Definition 3.2 and 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 that participate in the product. 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, i.e., the smallest weight of a directed

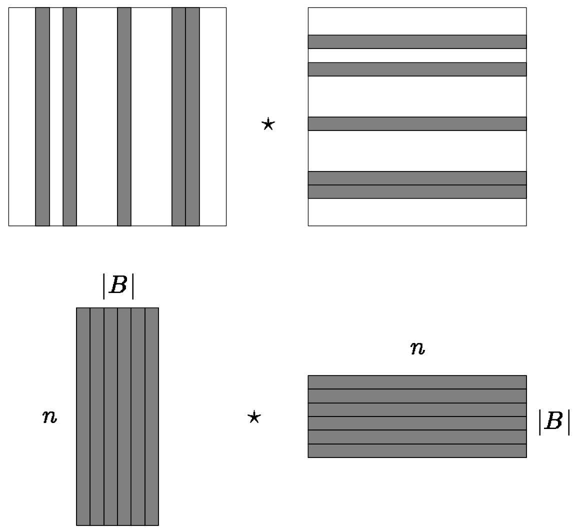

Figure 3: Replacing the square product $F \star F$ by the rectangular product $F[*, B] \star F[B, *]$.

path going from i to j. The weight of a path is the sum of the weights of its edges. The following lemma is easily established:

Lemma 4.1 *At any stage during the operation of* **rand-short-path**, *for every $i, j \in V$, we have:*

(i) $f_{ij} \geq \delta(i, j)$.

(ii) *If* $w_{ij} = 0$ *then* $f_{ij} = d_{ij}$. *Otherwise,* $1 \leq w_{ij} \leq n$ *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

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

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. Initially, $f_{ij} = d_{ij}$ and $w_{ij} = 0$, for every $i, j \in V$, so the condition is satisfied. Whenever f_{ij} is assigned a new value, we have $1 \leq w_{ij} \leq n$ and $f_{ij} = f_{i,w_{ij}} + f_{w_{ij},j}$. Until the next time f_{ij} is assigned a value we are thus guaranteed to have $f_{ij} \geq f_{i,w_{ij}} + f_{w_{ij},j}$, as the value of f_{ij} does not change, while the values of $f_{i,w_{ij}}$ and $f_{w_{ij},j}$ may only decrease.

Finally, if the conditions of property (iii) hold, then at the end of the iteration we have

$$f_{ij} \;\leq\; f'_{ij} \;\leq\; f_{ik} + f_{kj} \;=\; \delta(i, k) + \delta(k, j) \;=\; \delta(i, j)\,.$$

As $f_{ij} \geq \delta(i, j)$, by property (i), we get that $f_{ij} = \delta(i, j)$, as required. $\square$

More interesting is the following lemma:

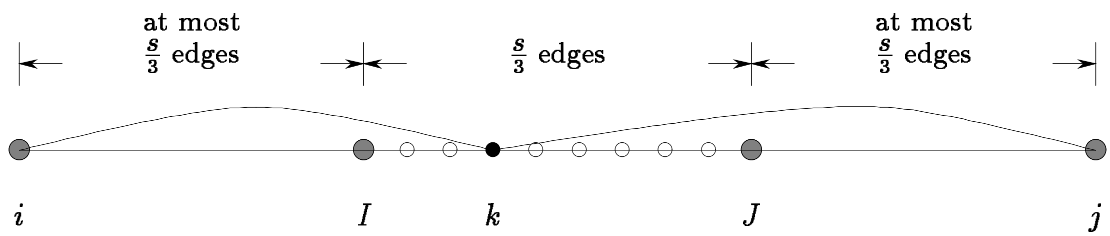

Figure 4: The correctness proof of **rand-short-path**.

Lemma 4.2 *Let $s = (3/2)^\ell$, for some $1 \le \ell \le \lceil \log_{3/2} n \rceil$. 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 $f_{ij} = \delta(i,j)$.*

Proof: We prove the lemma by induction of ℓ. It is easy to check that the claim holds for $\ell = 1$. We show next that if the claim holds for $\ell - 1$, 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, by the induction hypothesis, after the $(\ell - 1)$-st iteration we already have $f_{ij} = \delta(i,j)$ (with very 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 $|A| \ge s/3$. 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 very high probability. We also have $|f_{ik}|, |f_{kj}| \le sM$. It follows, therefore, from Lemma 4.1(iii), that if there exists $k \in A \cap B$, 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 $A \cap B \ne \phi$? Let $p = (9 \ln n)/s$. If $p \ge 1$, then $B = V$ and clearly $A \cap B \ne \phi$. Suppose, therefore, that $p = (9\ln n)/s < 1$. Each vertex then belongs to B independently with probability p. As $|A| \ge s/3$, the probability that $A \cap B = \phi$ is at most

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

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 each pair of vertices should only be considered at one of the iterations. If a pair $i, j \in V$ violates the condition of the lemma, then it also does so at the ℓ-th iteration, where ℓ is the smallest integer such that there is a shortest path from i to j that uses at most $s = (3/2)^\ell$ edges.)

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 $s_\ell = \lceil 3s_{\ell-1}/2 \rceil$, for $\ell > 0$. Note that $s_\ell \ge (3/2)^\ell$. 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 we have $f_{ij} = \delta(i,j)$.

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. $\square$

Combining Lemmas 4.1 and 4.2 with the fact that each pair of vertices in a graph of n vertices is connected by a shortest path that uses less than n edges, assuming there are no negative cycles in the graph, we get that after the last iteration, F is, with very high probability, the distance matrix of the graph. Furthermore, either $\delta(i,j) = d_{ij}$, or w_{ij} lies on a shortest path from i to j. This is stated formally in the following lemma:

Lemma 4.3 *If there are no negative weight cycles in the graph, then after the last iteration of* **rand-short-path**, *with very high probability, for every $i, j \in V$ we have*

(i) $f_{ij} = \delta(i,j)$.

(ii) If $w_{ij} = 0$ then $\delta(i,j) = d_{ij}$. Otherwise, $1 \le w_{ij} \le n$ and $\delta(i,j) = \delta(i, w_{ij}) + \delta(w_{ij}, j)$.

Proof: Condition (i) follows, as mentioned, from Lemma 4.2, the fact that in the last iteration $s \ge n$, and the fact that if $\delta(i,j) < +\infty$, and if there are no negative weight cycles in the graph, then there is a shortest path from i to j that uses at most $n-1$ edges.

Suppose now that $f_{ij} = \delta(i,j) < d_{ij}$. By Lemma 4.1$(ii)$ we get that after the last iteration we have $1 \le w_{ij} \le n$ and $f_{ij} \ge f_{i,w_{ij}} + f_{w_{ij},j}$, or equivalently, $\delta(i,j) \ge \delta(i, w_{ij}) + \delta(w_{ij}, j)$. But, by the triangle inequality we have $\delta(i,j) \le \delta(i, w_{ij}) + \delta(w_{ij}, j)$. Thus, $\delta(i,j) = \delta(i, w_{ij}) + \delta(w_{ij}, j)$, as required. $\square$

It is also easy to see that the input graph contains a negative cycle if and only if $f_{ii} < 0$ for some $1 \le i \le n$. 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 Galil and Margalit [GM97b] for the details.

The matrix W returned by **rand-short-path** contains a succinct representation of shortest paths between all pairs of vertices in the graph. Ways for reconstructing these shortest paths are described in the next section.

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 2.2, this time is $\tilde{O}(\min\{n^{t+\omega(1,r,1)+(1-r)}, n^{2+r}\})$. Graphs of the best available upper bounds on the functions $\omega(1,r,1)$ and $\omega(1,r,1) + (1-r)$ are given in Figure 5. (Also shown there is the function $2+r$.) 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 4.4 *Algorithm* **rand-short-path** *finds, with very high probability, all distances in the input graph, and a succinct representation of shortest paths between all pairs of vertices in the graph. If the input graph has n vertices, and the weights are all integers with absolute values at most $M = n^t$, where $t \le 3-\omega$, then its running time is $\tilde{O}(n^{2+\mu(t)})$, where $\mu = \mu(t)$ satisfies $\omega(1,\mu,1) = 1 + 2\mu - t$.*

If $M > n^{3-\omega}$ then fast matrix multiplication algorithms are never used by the algorithm and the running time is then $\tilde{O}(n^3)$.

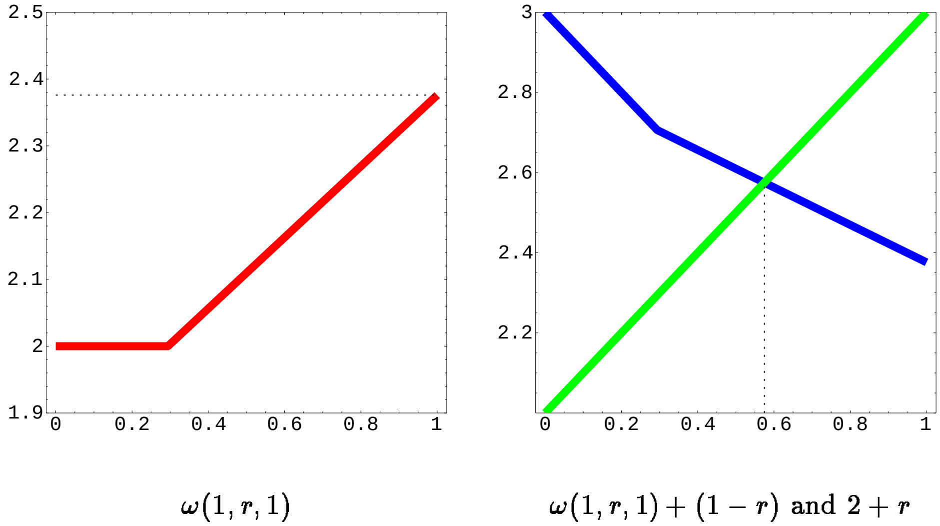

Figure 5: Best available bounds on the functions $\omega(1, r, 1)$ and $\omega(1, r, 1) + (1 - r)$, and the function $2 + r$.

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 2.3, we get that $\mu \leq \frac{\alpha(\omega-1)-1}{\omega+2\alpha-4} < 0.575$.

Corollary 4.5 *Algorithm* **rand-short-path** *finds, with very high probability, all distances, and a succinct representation of shortest paths between all pairs of vertices in the graph on n vertices in which all the weights are taken from the set $\{-1, 0, 1\}$ in $O(n^{2.575})$ time.*

5 Constructing shortest paths

A simple recursive algorithm, **path**, for constructing shortest paths is given in Figure 6. If there are no negative weight cycles in the graph, and if W is the matrix of witnesses returned by a successful run of **rand-short-path**, then **path**(W, i, j) returns a shortest path from i to j in the graph. If $w_{ij} = 0$, then the edge (i, j) is a shortest path from i to j. Otherwise, a shortest path from i to j is obtained by concatenating a shortest path from i to w_{ij}, found using a recursive call to **path**, and a shortest path from w_{ij} to j, found using a second recursive call to **path**. (The dot in next to last line in the description of **path** is used to denote concatenation.) If there is no directed path from i to j in the graph, then **path**(W, i, j) returns the "edge" (i, j) whose weight is $+\infty$.

Theorem 5.1 *If there are no negative weight cycles in the input graph, and if W is the matrix of witnesses returned by a successful run of* **rand-short-path**, *then* **path**(W, i, j) *returns a shortest path from i to j in the graph. The running time of* **path**(W, i, j) *is proportional to the number of edges in the path returned.*

Proof: For every $i, j \in V$, let t_{ij} be the number of the iteration of **rand-short-path** in which f_{ij} was set for the last time. If $f_{ij} = d_{ij}$, let $t_{ij} = 0$. We need the following claim:

Claim 5.2 *If $1 \leq w_{ij} \leq n$, then $t_{i,w_{ij}}, t_{w_{ij},j} < t_{ij}$.*

```
algorithm path(W, i, j)

if w_ij = 0 then
    return ⟨i, j⟩
else
    return path(W, i, w_ij) . path(W, w_ij, j)
endif
```

Figure 6: Constructing a shortest path using a matrix of witnesses.

Proof: Suppose that f_{ij} was set for the last time at the ℓ-th iteration. Let f^0_{rs} be the elements of the matrix F at the beginning of the ℓ-th iteration, and f^1_{rs} be these elements at the end of the ℓ-th iteration. By our assumption and by Lemma 4.3, we get that

$$\begin{aligned} f_{ij} &= f^1_{ij} = f^0_{i,w_{ij}} + f^0_{w_{ij},j} , \\ f_{ij} &= \delta(i,j) = \delta(i,w_{ij}) + \delta(w_{ij},j) . \end{aligned}$$

As $f^0_{i,w_{ij}} \geq \delta(i,w_{ij})$ and $f^0_{w_{ij},j} \geq \delta(w_{ij},j)$ (see Lemma 4.1(i)), we get that $f^0_{i,w_{ij}} = \delta(i,w_{ij})$ and $f^0_{w_{ij},j} = \delta(w_{ij},j)$. Thus, $f_{i,w_{ij}}$ and $f_{w_{ij},j}$ are already assigned their final values at the beginning of the ℓ-th iteration, and therefore $t_{i,w_{ij}}, t_{w_{ij},j} < \ell = t_{ij}$, as required. □

We now prove Theorem 5.1 by induction on t_{ij}. If $t_{ij} = 0$, then $w_{ij} = 0$, and **path**(W,i,j) returns the edge (i,j) which is indeed a shortest path from i to j. Suppose now that **path**(W,i,j) returns a shortest path from r to s for every r and s for which $t_{rs} < \ell$. Suppose that $t_{ij} = \ell$. By Claim 5.2, we get that $t_{i,w_{ij}}, t_{w_{ij},j} < \ell$. By the induction hypothesis, the recursive calls **path**(W,i,w_{ij}) and **path**(W,w_{ij},j) return shortest paths from i to w_{ij} and from w_{ij} to j. As $\delta(i,j) = \delta(i,w_{ij}) + \delta(w_{ij},j)$ (Lemma 4.3), the concatenation of these two shortest paths is indeed a shortest path from i to j, as required. □

There is, however, something unsatisfying with the behavior of **path**. While it is true that the call **path**(W,i,j) always returns a shortest path from i to j in the graph, the shortest path returned is not necessarily *simple*, i.e., it may visit certain vertices more than once. This may happen, of course, only if there are zero weight cycles in the graph. It is, of course, easy to convert a non-simple shortest path into a simple shortest path, by removing cycles, but the running time then is no longer proportional to the number of edges on the shortest path produced.

Another possible objection to the use of **path** is that it cannot produce shortest paths in *real time*. While it is true that a shortest path that uses ℓ edges can be found in $O(\ell)$ time, it may also take $\Omega(\ell)$ time just to find the second vertex on such a path.

To address these two issues, we show next that the matrix of witnesses W returned by **rand-short-path** can be easily converted into a matrix of *successors* (see, e.g., [CLR90], Chapter 25, were predecessors, instead of successors are considered). A matrix of successors can be easily used to construct trees of shortest paths.

Definition 5.3 (Successors) *A matrix S is a matrix of successors for a graph $G = (V,E)$ if for every $i,j \in V$, if there is a path from i to j in the graph, then the call* **s-path**(S,i,j)*, where* **s-path** *is the procedure given in Figure 7, returns a* simple *shortest path from i to j in the graph.*

```
algorithm s-path(S, i, j)

if s_ij = j then
    return ⟨i, j⟩
else
    return ⟨i, s_ij⟩ . s-path(S, s_ij, j)
endif
```

Figure 7: Constructing a shortest path using a matrix of successors.

```
algorithm wit-to-suc(W, T)

S ← 0

for ℓ ← 0 to max(T) do T_ℓ ← { (i, j) | t_ij = ℓ }

for every (i, j) ∈ T_0 do s_ij ← j

for ℓ ← 1 to max(T) do
for every (i, j) ∈ T_ℓ do
begin
    k ← w_ij
    while s_ij = 0 do s_ij ← s_ik ; j ← s_ij
end

return S
```

Figure 8: Constructing a matrix of successors.

Algorithm **wit-to-suc**, given in Figure 8, receives a matrix W of witnesses returned by **rand-short-path**, and a matrix T that gives the iteration number in which each element of W was set for the last time, as in the proof of Theorem 5.1. (It is very easy, of course, to modify **rand-short-path** so that it would also return this matrix.) It returns a matrix S of successors. Algorithm **wit-to-suc** works correctly even if there are zero weight cycles in the graph, but not if there are negative weight cycles in the graphs as then distances and shortest paths are not well defined.

Theorem 5.4 *If there are no negative weight cycles in the graph, if W is the matrix of witnesses returned by a successful run of* **rand-short-path**, *and if T is the corresponding matrix of iteration numbers, then algorithm* **wit-to-suc** *returns a matrix of successors. The running time of algorithm* **wit-to-suc** *is $O(n^2)$.*

Proof: Algorithm **wit-to-suc** begins by initializing all the elements of the $n \times n$ matrix S to 0. It then constructs, for each iteration number ℓ, the set T_ℓ of pairs (i,j) for which $t_{ij} = \ell$. It is easy to construct all these sets in $O(n^2)$ by bucket sorting. (In the description of **wit-to-suc**, $\max(T)$ denotes the maximal element in T. Note that $\max(T) = O(\log n)$.) Next, for every (i,j) such that $t_{ij} = 0$, it sets $s_{ij} \leftarrow j$. It then performs $\max(T)$ iterations, one of each iteration of **rand-short-path** in which values are changed.

We prove, by induction on the order in which the elements of the matrix S are assigned nonzero values, that if $s_{ij} \neq 0$, then **s-path**(S, i, j) returns a simple shortest path from i to j in the graph. This clearly holds after **wit-to-suc** sets $s_{ij} \leftarrow j$ for every $(i, j) \in T_0$, as the edge (i, j) is then a simple shortest path from i to j in the graph.

Suppose that **wit-to-suc** is now about to perform the `while` loop for a pair (i, j) for which $t_{ij} = \ell$. If $s_{ij} \neq 0$, then no new entries are assigned nonzero values. Suppose, therefore, that $s_{ij} = 0$. Let $k = w_{ij}$. By Claim 5.2, we get that $t_{ik} < \ell$ and , $t_{kj} < \ell$. Thus, s_{ik} and s_{kj} are already assigned nonzero values and by the induction hypothesis, the calls **s-path**(S, i, k) and **s-path**(S, k, j) return simple shortest paths in the graph from i to k, and from k to j. Let v be the first vertex on the path **s-path**(S, i, k) for which $s_{vj} \neq 0$. The vertex v is well defined as $s_{kj} \neq 0$. As $s_{vj} \neq 0$, we get, by the induction hypothesis, that **s-path**(S, v, j) traces a simple shortest path from v to j. The concatenation of the portion of **s-path**(S, i, k) from i to v, and of **s-path**(S, v, j) is clearly a shortest path from i to j. It is also simple as both portions are simple, and as for every u on the first portion, except v, we have $s_{uj} = 0$, while for every u on the second portion we have $s_{uj} \neq 0$. After the `while` loop corresponding to (i, j), **s-path**(S, i, j) returns this simple shortest path. Furthermore, if s_{uj} is changed by this `while` loop, then u lies on the first portion of this simple shortest path, and **s-path**(S, u, j) is the corresponding suffix of this simple shortest path, which is also a simple shortest path.

Finally, the complexity of the algorithm is $O(n^2)$ as each iteration of the `while` loop reduces the number of zero elements of S by one. □

6 A deterministic algorithm for unweighted graphs

In this section we describe a deterministic version of algorithm **rand-short-path** of Section 4. The version described here works only for *unweighted* directed graphs. A Slightly more complicated deterministic algorithm that works for weighted directed graphs is described in the next section. We start with the following useful definition:

Definition 6.1 ($\delta(i, j)$ and $\eta(i, j)$) *As before, let $\delta(i, j)$ denote the distance from i to j in the graph, i.e., the minimum weight of a path from i to j in the graph, where the weight of a path is the sum of the weights of its edges. Let $\eta(i, j)$ denote the minimum* number of edges *on a shortest path from i to j.*

If the graph is unweighted then $\delta(i, j) = \eta(i, j)$, for every $i, j \in V$. Note that $\eta(i, j)$ is not necessarily the distance from i to j in the unweighted version of the graph.

Algorithm **rand-short-path** implicitly used the notion of *bridging sets* which we now formalize:

Definition 6.2 (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 9. All the paths shown there, schematically, are shortest paths from i to j although they do not all use the same number of edges. If B is a strong s-bridging set, and if $\eta(i, j) = t$ and $t \geq s$, i.e., if the minimum number of edges on a

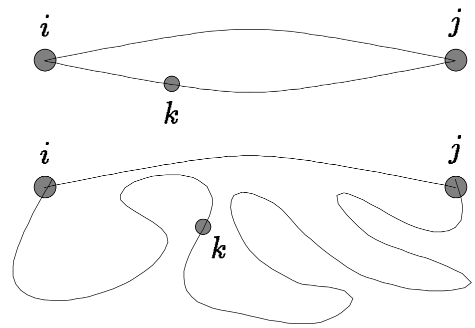

Figure 9: Bridging and strong bridging sets.

shortest path from i to j is t, and $t \geq s$, then there is a vertex $k \in B$ that lies on a shortest path from i to j that uses exactly t edges. The top drawing in Figure 9 illustrates the fact there there may be several shortest paths from i to j that use exactly t edges. A vertex k belonging to B is guaranteed to lie on at least one of them. If B is an s-bridging set, but not necessarily a strong s-bridging set, then a vertex k belonging to B is guaranteed to lie on a shortest path from i to j. But, this shortest path may use much more than t edges. This is illustrated in the bottom drawing of Figure 9.

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 definition of bridging, and strongly bridging, sets by the condition $\eta(i,j) = s$. Indeed, suppose the appropriate condition holds for every $u, v \in V$ such that $\eta(u,v) = s$. Suppose that $\eta(i,j) = t > s$. Consider a shortest path p from i to j that uses t edges. Let w be the s-th vertex on p, starting the count from 0. Then, clearly $\eta(i,w) = s$. Thus, a vertex $k \in B$ is guaranteed to lie on a shortest path from i to w. This vertex lies also on a shortest path from i to j, or on such a shortest path with a minimum number of edges, as required.

Reviewing the proof of Lemma 4.2, we see that algorithm **rand-short-path** produces correct results as long as the set B used in the ℓ-th iteration is a *strong* $(s/3)$-bridging set.

Lemma 6.3 *If in each iteration of* **rand-short-path** *the set B is a strong $(s/3)$-bridging set, then all distances returned by* **rand-short-path** *are correct.*

Proof: The proof is almost identical to the proof of Lemma 4.2. We show again, by induction on ℓ, that if $\eta(i,j) \leq (3/2)^\ell$, then after the ℓ-th iteration of the algorithm we have $f_{ij} = \delta(i,j)$. The basis of the induction is easily established. Suppose, therefore, that the claim holds for $\ell - 1$. We show that it also holds for ℓ. Let i and j be two vertices such that $2s/3 \leq \eta(i,j) \leq s$, where $s = (2/3)^\ell$. As in Lemma 4.2, let p be a shortest path from i to j that uses $\eta(i,j)$ edges, let I and J be two vertices on p such that I and J are separated, on p, by $s/3$ edges, and such that i and I, and J and j, are separated, on p, by at most $s/3$ edges (see Figure 4). As B, the set used in the ℓ-th iteration, is assumed to be a strong $(s/3)$-bridging set, and as $\eta(I,J) \geq s/3$, a vertex $k \in B$ is guaranteed to lie of a shortest path from I to J that uses $\eta(I,J)$ edges. This shortest path from I to J is not necessarily the portion of p going from I to J. Nonetheless, we still have $\delta(i,j) = \delta(i,k) + \delta(k,j)$ and $\eta(i,j) = \eta(i,k) + \eta(k,j)$. As $\eta(i,k) \leq \eta(i,J) \leq 2s/3$ and $\eta(k,j) \leq \eta(I,j) \leq 2s/3$ we get, by the induction hypothesis, that $f_{ik} = \delta(i,k)$ and $f_{kj} = \delta(k,j)$. After the distance product of the ℓ-th iteration we therefore have $f_{ij} = \delta(i,j)$, as required. □

In the proof of Lemma 6.3 we made heavy use of the assumption that B is a *strong* bridging set. If B were not a strong bridging set, we could not have deduced that $\eta(i,k), \eta(k,j) \leq 2s/3$ and the argument used in the proof would break down. Also implicit in the proof of Lemma 4.2 is the following result whose proof we do not repeat:

```
algorithm find-bridge(W, s)
C ← φ
for every 1 ≤ i, j ≤ n do
    U ← sub-path(W, i, j, s − 1) ∪ {i, j}
    if |U| ≥ s then C ← C ∪ {U} endif
end
B ← hitting-set(C)
return B
```

Figure 10: A deterministic algorithm for constructing an s-bridging set.

```
algorithm sub-path(W, i, j, s)
if w_ij = 0 or s = 0 then
    return φ
else
    U ← sub-path(W, i, w_ij, s − 1)
    return U ∪ {w_ij} ∪ sub-path(W, w_ij, j, s − |U| − 1)
endif
```

Figure 11: Finding up to s vertices on a shortest path from i to j.

Lemma 6.4 *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 very high probability B is a* strong *s-bridging set.*

We next describe a deterministic algorithm, called **find-bridge**, for finding s-bridging sets. (Unfortunately, the sets returned by **find-bridge** are not necessarily strong s-bridging sets.) A description of algorithm **find-bridge** is given in Figure 10. 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 $\eta(i, j) \leq s$. In other words, if $\eta(i, j) \leq s$, then **path**(W, i, j) produces a shortest path from i to j. We assume here, for simplicity, that the graph does not contain cycles of non-positive weight so the shortest path produced by **path**(W, i, j), when $\eta(i, j) \leq s$, is simple. We show later how to remove this simplifying assumption. We do *not* assume that the shortest path produced by **path**(W, i, j) uses a minimum number of edges, i.e., it may use more than $\eta(i, j)$ edges.

Algorithm **find-bridge** uses a procedure called **sub-path** that receives the matrix W, two vertices $i, j \in V$ and an integer s. The operation of **sub-path** is similar to the operation of **path**. It tries to construct a path from i to j using the witnesses contained in the matrix W. It counts, however, the number of intermediate vertices found so far on the path and stops the construction when s intermediate vertices are encountered. A simple recursive implementation of **sub-path** is given in Figure 11. The following lemma is easily verified.

Lemma 6.5 *If a call to* **path**(W, i, j) *constructs a simple path from i to j that passes through t intermediate* *vertices, then* **sub-path**(W,i,j,s) *returns the set of intermediate vertices on this path, if* $t \le s$, *or a subset of* s *intermediate vertices on this path, if* $t > s$. *The running time of* **sub-path**(W,i,j,s) *is* $O(s)$.

For every $i, j \in V$, let U_{ij} be the set obtained by adding the vertices i and j to the set obtained by calling **sub-path**(W,i,j,s). All the elements of U_{ij} are vertices on a shortest path from i to j. If $\eta(i,j) = s$, then by our assumption on W, **path**(W,i,j) returns a shortest path from i to j. This shortest path must use at least s edges and contain, therefore, at least $s-1$ intermediate vertices. It follows that $|U_{ij}| \ge s+1$. Thus, if a set B *hits* all the sets U_{ij} for which $|U_{ij}| \ge s$, i.e., if $B \cap U_{ij} \ne \phi$ whenever $|U_{ij}| \ge s$, then B is s-bridging. Algorithm **find-bridge** collects all the sets U_{ij} for which $|U_{ij}| \ge s$ 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 [Lov75] and Chvátal [Chv79], 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 at least 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 $\Delta \le n^2$. 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 6.6 *If the matrix W can be used to construct shortest paths between all pairs of vertices $i, j \in V$ for which $\eta(i,j) \le s$, 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⌋)
endif
```

we obtain a deterministic algorithm for solving the APSP problem for *unweighted* directed graphs. We call this algorithm **unwght-short-path**.

We compute new bridging sets only when $s \le n^{1/2}$ 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 $s' \ge s$. The use of a bridging set of size $\Theta(n^{1/2}\log n)$ in the iterations for which $s \ge n^{1/2}$ 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 6.7 *Algorithm* **unwght-short-path** *solves the APSP problem for unweighted directed graphs* deterministically *in $\tilde{O}(n^{2+\mu})$ time, where $\mu < 0.575$ satisfies $\omega(1,\mu,1) = 1+2\mu$.*

7 A deterministic algorithm for weighted graphs

In this section we present a deterministic version of algorithm **rand-short-path** for weighted directed graphs. The algorithm, called **short-path** is given in Figure 12. For simplicity, we assume at first that

```
algorithm short-path(D)
F ← D ; W ← 0
M ← max{ |d_ij| : d_ij ≠ +∞}
for ℓ ← 1 to ⌈log_2 n⌉ do
begin
    s ← 2^ℓ
    if s ≤ n^{1/2} then
        B ← find-bridge-upd(F, W, s/2)
    endif
    dist-prod-upd(F, W,  B, V, V,   sM)
    dist-prod-upd(F, W,  V, B, V, 2sM)
end
return (F, W)
```

Figure 12: A deterministic algorithm for finding shortest paths.

```
algorithm dist-prod-upd(F, W, A, B, C, L)
(F', W') ← dist-prod(F[A, B], F[B, C], L)
for every 1 ≤ i ≤ |A| and 1 ≤ j ≤ |C| do
if f'_ij < f_{a_i,c_j} then f_{a_i,c_j} ← f'_ij ; w_{a_i,c_j} ← b_{w'_ij} endif
```

Figure 13: A simple procedure for updating distances and witnesses.

the input graph does not contain negative weight cycles, nor zero weight cycles.

Algorithm **short-path** uses a simple procedure, called **dist-prod-upd**, that performs a distance product, by calling **dist-prod** of Section 2, and updates the distances and witnesses found so far. Algorithm **dist-prod-upd** is given in Figure 13. It receives the $n \times n$ matrices F and W that hold the distances and witnesses found so far. It also receives three subsets $A, B, C \subseteq V$, where $V = \{1, 2, \ldots, n\}$ is the set of all vertices. (In the calls made by **short-path**, two of the sets A, B and C would be V.) **dist-prod-upd** computes the distance product $F[A, B] \star F[B, C]$, putting a cap of L of the values of the entries of F that participate in the product. It then updates the matrices F and W accordingly. (By $F[A, B]$ we obviously mean the submatrix of F composed of the elements whose row index belongs to A, and whose column index belongs to B. Also, we let a_i denote the i-th elements of A.) Thus, the first call to **dist-prod-upd** in **short-path** computes the distance product $F[B, *] \star F$, while the second one computes the distance product $F[*, B] \star F[B, *]$, as in **rand-short-path**. By Lemma 2.4, we get that the cost of these two distance products is essentially the same.

Algorithm **short-path** constructs bridging sets by calling algorithm **find-bridge-upd** given in Figure 14. Algorithm **find-bridge-upd** is very similar to algorithm **find-bridge** of Section 6. The difference is that **find-bridge-upd** calls algorithm **sub-path-upd**, given in Figure 15, instead of algorithm **sub-path** called by **find-bridge**.

```
algorithm find-bridge-upd(F, W, s)
C ← φ
for every 1 ≤ i, j ≤ n do
    U ← sub-path-upd(F, W, i, j, i, j, s − 1) ∪ {i, j}
    if |U| ≥ s then C ← C ∪ {U} endif
end
B ← hitting-set(C)
return B
```

Figure 14: A deterministic algorithm for constructing an s-bridging set while updating some distances.

```
algorithm sub-path-upd(F, W, a, b, i, j, s)
if w_ij = 0 or s = 0 then
    return φ
else
    if f_ai + f_i,w_ij < f_a,w_ij then f_a,w_ij ← f_ai + f_i,w_ij ; w_a,w_ij ← i  endif
    if f_w_ij,j + f_j,b < f_w_ij,b then f_w_ij,b ← f_w_ij,j + f_j,b ; f_w_ij,b ← j  endif
    U ← sub-path-upd(F, W, a, b, i, w_ij, s − 1)
    return U ∪ {w_ij} ∪ sub-path-upd(F, W, a, b, w_ij, j, s − |U| − 1)
endif
```

Figure 15: Finding up to s vertices on a shortest path from i to j while updating distances.

A call to **sub-path**(W, i, j, s) returns a set of up to s intermediate vertices on a path from i to j. However, if $k \in$ **sub-path**(W, i, j, s), it is not guaranteed that $f_{ik}, f_{kj} < +\infty$, let alone $f_{ik} + f_{kj} \leq f_{ij}$. Algorithm **sub-path-upd** fixes this problem. The following lemma is easily verified.

Lemma 7.1 *If the matrices F and W satisfies the conditions $f_{ij} \geq \delta(i,j)$, for every $i,j \in V$, and $f_{ij} \geq f_{i,w_{ij}} + f_{w_{ij},j}$ whenever $1 \leq w_{ij} \leq n$, and if* **path**(W, i, j) *traces a path from i to j, then a call to* **sub-path-upd**(F, W, i, j, i, j, s) *returns a set of s intermediate vertices on this path, or the set of all intermediate vertices if there are less than s of them. If k is one of the vertices returned by the call, then after the call we have $f_{ik} + f_{kj} \leq f_{ij}$. The matrices F and W continue to satisfy the specified conditions. Furthermore, if before the call we have $f_{ij} = \delta(i,j)$, then after the call to we have $f_{ik} = \delta(i,k)$, $f_{kj} = \delta(k,j)$ and $\delta(i,j) = \delta(i,k) + \delta(k,j)$.*

Before proving the correctness of algorithm **short-path**, we prove a useful additional property of bridging sets.

Lemma 7.2 *Let B be an s-bridging set of a graph $G = (V, E)$ with no nonpositive weight cycles. Then, if $i, j \in V$ and $\eta(i,j) \geq s$, then there is a vertex $k \in B$ such that $\delta(i,j) = \delta(i,k) + \delta(k,j)$ and $\eta(i,k) \leq s$.*

Proof: By the definition of bridging sets, we get that there exists $k_1 \in B$ such that $\delta(i,j) = \delta(i,k_1) + \delta(k_1,j)$. If $\eta(i,k_1) \leq s$, we are done. Assume, therefore, that $\eta(i,k_1) > s$. Let k_1' be next to last vertex

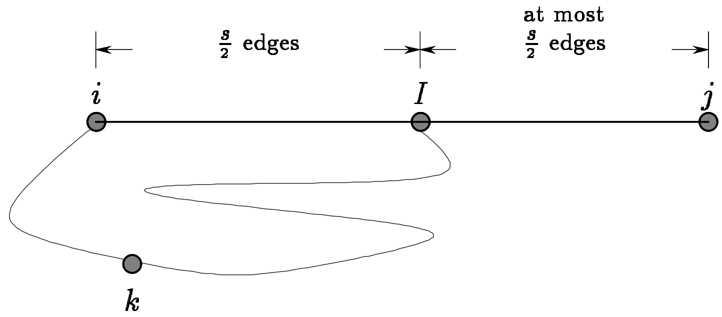

Figure 16: The correctness proof of **short-path**.

on a shortest path from i to k_1. Clearly, $k_1' \neq k_1$, $\delta(i,j) = \delta(i,k_1') + \delta(k_1',j)$ and $\eta(i,k_1') \geq s$. Thus, there exists $k_2 \in B$ such that $\delta(i,k_1') = \delta(i,k_2) + \delta(k_2,k_1')$, and therefore also $\delta(i,j) = \delta(i,k_2) + \delta(k_2,j)$. There is, therefore, a shortest path from i to j that passes through k_2, then through k_1', and then through k_1. As there are no nonpositive weight cycles in the graph, a shortest path must be simple and therefore $k_2 \neq k_1$. In general, suppose that we have found so far r distinct vertices $k_r, k_{r-1}, \ldots, k_1 \in B$ such that there is a shortest path from i to j that visits all these vertices. If $\eta(i,k_r) \leq s$, then we are done. Otherwise, we can find another vertex $k_{r+1} \in B$, distinct from all the previous vertices, such that there is a shortest path from i to j that passes though $k_{r+1}, k_r, k_{r-1}, \ldots, k_1$. As the graph is finite, this process must eventually end with a vertex from B satisfying our requirements. □

Theorem 7.3 *Algorithm* **short-path** *finds all distances, and a succinct representation of shortest paths between all pairs of vertices in a graph with no nonpositive weight cycles. If the input graph has n vertices and the edge weights are taken from the set $\{-M, \ldots, 0, \ldots, M\}$, where $M = n^t$ and $t \leq 3 - \omega$, then its running time is $\tilde{O}(n^{2+\mu(t)})$, where $\mu = \mu(t)$ satisfies $\omega(1,\mu,1) = 1 + 2\mu - t$.*

Proof: We prove, by induction, that after the ℓ-th iteration of **short-path** we have:

(i) $f_{ij} \geq \delta(i,j)$, for every $i,j \in V$.

(ii) If $w_{ij} = 0$ then $f_{ij} = d_{ij}$. Otherwise, $1 \leq w_{ij} \leq n$ and $f_{ij} \geq f_{i,w_{ij}} + f_{w_{ij},j}$.

(iii) If $\eta(i,j) \leq 2^\ell$, then $f_{ij} = \delta(i,j)$.

The proofs of properties (i) and (ii) are analogous to the proofs of properties (i) and (ii) of Lemma 4.1. We concentrate, therefore, on the proof of property (iii). It is easy to check that property (iii) holds before the first iteration. We show now that if it holds at the end of the $(\ell-1)$-st iteration, then it also holds after the ℓ-th iteration.

Let $i,j \in V$ be such that $\eta(i,j) \leq 2^\ell$. If $\eta(i,j) \leq 2^{\ell-1}$, then the condition $f_{ij} = \delta(i,j)$ holds already after the $(\ell-1)$-st iteration. Assume, therefore, that $2^{\ell-1} < \eta(i,j) \leq 2^\ell$. Let p be a shortest path from i to j that uses $\eta(i,j)$ edges. Let I be the vertex on p for which $\eta(i,I) = 2^{\ell-1}$. (See Figure 16.) Note that $\eta(I,j) \leq 2^{\ell-1}$. By the induction hypothesis, after the $(\ell-1)$-st iteration we have $f_{iI} = \delta(i,I)$ and $f_{Ij} = \delta(I,j)$. As B is an $2^{\ell-1}$-bridging set, we get, by Lemma 7.2, that there exists $k \in B$ such that $\delta(i,I) = \delta(i,k) + \delta(k,I)$ and $\eta(i,k) \leq 2^{\ell-1}$. Furthermore, as $k \in$ **sub-path-upd**$(W,i,I,i,I,s/2) \cup \{i,I\}$, we get, by Lemma 7.1, that $f_{ik} = \delta(i,k)$ and $f_{kI} = \delta(k,I)$. (The fact that $f_{ik} = \delta(i,k)$ follows also from the induction hypothesis, as $\eta(i,k) \le 2^{\ell-1}$.) As $\eta(i,I), \eta(i,k) \le 2^{\ell-1}$, we get that $|\delta(i,I)|, |\delta(i,k)| \le 2^{\ell-1}M$. Thus $|\delta(k,I)| = |\delta(i,I) - \delta(i,k)| \le |\delta(i,I)| + |\delta(i,k)| \le 2^{\ell}M$. To sum up, we have

$$\begin{array}{ccccc} f_{ik} = \delta(i,k) & , & f_{kI} = \delta(k,I) & , & f_{Ij} = \delta(I,j) \\ |f_{ik}| \le 2^{\ell-1}M & , & |f_{kI}| \le 2^{\ell}M & , & |f_{Ij}| \le 2^{\ell-1}M \end{array}$$

As $k \in B$ and $I, j \in V$, after the first distance product of the ℓ-th iteration, we get that

$$f_{kj} \le f_{kI} + f_{Ij} = \delta(k,I) + \delta(I,j) = \delta(k,j) ,$$

and thus $f_{kj} = \delta(k,j)$ and $|f_{kj}| < 2^{\ell+1}M$. As $i, j \in V$ and $k \in B$, after the second distance product we get that

$$f_{ij} \le f_{ik} + f_{kj} = \delta(i,k) + \delta(k,j) = \delta(i,j) ,$$

and thus $f_{ij} = \delta(i,j)$, as required. □

Finally, we describe the changes that should be made to **short-path** if we want it to detect negative weight cycles, and continue to work in the presence of zero weight cycles. Detecting negative weight cycles is easy. We simply check, after each iteration, whether $f_{ii} < 0$, for some $i \in V$. Making **short-path** work in the presence of zero weight cycles requires more substantial changes.

Before describing the changes required, let us review the problems caused by zero weight cycles. First, as mentioned in Section 5, the shortest paths returned by **path**(W,i,j) are not necessarily simple. Thus, calls to **sub-path**(W,i,j,s) and **sub-path-upd**(W,i,j,i,j,s) may return multisets with less than s distinct elements. As a consequence, the bridging set returned by **find-bridge**(W,s) and by **find-bridge-upd**(F,W,s) are not necessarily of size $O(n \log n/s)$. Second, Lemma 7.2, that plays a crucial role in the correctness proof of algorithm **short-path** no longer holds in the presence of zero weight cycles.

To fix these problems we use an approach that is similar to the one used in Section 5. After each iteration of **short-path** we call algorithm **wit-to-suc** convert the matrix of witnesses W into a matrix S of successors. As the complexity of **wit-to-suc** is $O(n^2)$, the extra cost involved is negligible. Even though W does not describe yet shortest paths between all pairs of vertices of the graph, it is not difficult to verify that if for some $i, j \in V$ the matrix W describes a shortest path from i to j in the graph, then S would describe a *simple* shortest path from i to j. Using S instead of W, it is then easy to find, in $O(s)$ time, the *first* s intermediate vertices on a shortest path from i to j. The bridging set returned by **find-bridge-upd** would then satisfy the condition of Lemma 7.2 and the correctness of the algorithm would follow.

8 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 $0 \le \ell \le \log_{1+\epsilon} n$, 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.

```
algorithm scale(A, M, R)
a'_ij ← { ⌈R a_ij / M⌉   if 0 ≤ a_ij ≤ M
        { +∞              otherwise
Return A'.
```

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

Figure 17: A simple scaling algorithm.

```
algorithm approx-dist-prod(A, B, M, R)
C ← +∞
for r ← ⌊log_2 R⌋ to ⌈log_2 M⌉ do
begin
    A' ← scale(A, 2^r, R)
    B' ← scale(B, 2^r, R)
    C' ← dist-prod(A', B', R)
    C ← min{ C , (2^r/R) · C' }
end
return C
```

Figure 18: Approximate distance products.

The main idea used to obtain almost shortest paths is *scaling*. A very simple scaling algorithm, called **scale**, is given in Figure 17. 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 18. 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.

Lemma 8.1 *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

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

```
algorithm approx-short-path(D, ε)
F ← D
M ← max{ d_ij : d_ij ≠ +∞}
R ← 4⌈log₂ n⌉/ ln(1 + ε)
R ← 2^⌈log₂ R⌉
for ℓ ← 1 to ⌈log₂ n⌉ do
begin
    F' ← approx-dist-prod(F, F, Mn, R)
    F ← min{ F , F' }
end
return F
```

Figure 19: Approximate shortest paths.

as required. □

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 19, 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 8.1, 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

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

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 8.2 *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.

9 Concluding remarks

The results of Seidel [Sei95] and Galil and Margalit [GM97a],[GM97b] 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 [GM97a] 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 [Tak98]. Can the APSP problem be solved in sub-cubic time, for example, when $M = n$?

Finally, we note that the shortest paths returned by the algorithms presented in this paper do not necessarily use a minimum number of edges. Producing shortest paths that do use a minimum number of edges seems to be a slightly harder problem. For more details, see Zwick [Zwi99].

Acknowledgment

I would like to thank Victor Pan for sending me a preprint of [HP98] and for answering several questions regarding matrix multiplication.

References

[ACIM99] D. Aingworth, C. Chekuri, P. Indyk, and R. Motwani. Fast estimation of diameter and shortest paths (without matrix multiplication). *SIAM Journal on Computing*, 28:1167–1181, 1999.

[AGM97] 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.

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The design and analysis of computer algorithms.* Addison-Wesley, 1974.

[AN96] N. Alon and M. Naor. Derandomization, witnesses for Boolean matrix multiplication and construction of perfect hash functions. *Algorithmica*, 16:434–449, 1996.

[BCS97] P. Burgisser, M. Clausen, and M.A. Shokrollahi. *Algebraic complexity theory.* Springer-Verlag, 1997.

[Chv79] V. Chvátal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4:233–235, 1979.

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to algorithms.* The MIT Press, 1990.

[Cop97] D. Coppersmith. Rectangular matrix multiplication revisited. *Journal of Complexity*, 13:42–49, 1997.

[CW90] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9:251–280, 1990.

[CZ97] 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.

[DHZ00] D. Dor, S. Halperin, and U. Zwick. All pairs almost shortest paths. *SIAM Journal on Computing*, 29:1740–1759, 2000.

[Dij59] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[Fre76] M.L. Fredman. New bounds on the complexity of the shortest path problem. *SIAM Journal on Computing*, 5:49–60, 1976.

[FT87] M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34:596–615, 1987.

[GM93] Z. Galil and O. Margalit. Witnesses for boolean matrix multiplication. *Journal of Complexity*, 9:201–221, 1993.

[GM97a] Z. Galil and O. Margalit. All pairs shortest distances for graphs with small integer length edges. *Information and Computation*, 134:103–139, 1997.

[GM97b] 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.

[HP98] X. Huang and V.Y. Pan. Fast rectangular matrix multiplications and applications. *Journal of Complexity*, 14:257–299, 1998.

[Joh77] D.B. Johnson. Efficient algorithms for shortest paths in sparse graphs. *Journal of the ACM*, 24:1–13, 1977.

[KKP93] D.R. Karger, D. Koller, and S.J. Phillips. Finding the hidden path: time bounds for all-pairs shortest paths. *SIAM Journal on Computing*, 22:1199–1217, 1993.

[Lov75] L. Lovász. On the ratio of optimal integral and fractional covers. *Discrete Mathematics*, 13:383–390, 1975.

[McG95] C.C. McGeoch. All-pairs shortest paths and the essential subgraph. *Algorithmica*, 13:426–461, 1995.

[Pan85] V. Pan. *How to multiply matrices faster*. Lecture notes in computer science, volume 179. Springer-Verlag, 1985.

[Sei95] R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of Computer and System Sciences*, 51:400–403, 1995.

[SS71] A. Schönhage and V. Strassen. Schnelle multiplikation grosser zahlen. *Computing*, 7:281–292, 1971.

[SZ99] A. Shoshan and U. Zwick. All pairs shortest paths in undirected graphs with integer weights. In *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science, New York, New York*, pages 605–614, 1999.

[Tak92] T. Takaoka. A new upper bound on the complexity of the all pairs shortest path problem. *Information Processing Letters*, 43:195–199, 1992.

[Tak98] T. Takaoka. Subcubic cost algorithms for the all pairs shortest path problem. *Algorithmica*, 20:309–318, 1998.

[Tho99] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM*, 46:362–394, 1999.

[Tho00] M. Thorup. Floats, integers, and single source shortest paths. *Journal of Algorithms*, 35:189–201, 2000.

[Yuv76] G. Yuval. An algorithm for finding all shortest paths using $N^{2.81}$ infinite-precision multiplications. *Information Processing Letters*, 4:155–156, 1976.

[Zwi98] U. Zwick. All pairs shortest paths in weighted directed graphs – exact and almost exact algorithms. In *Proceedings of the 39th Annual IEEE Symposium on Foundations of Computer Science, Palo Alto, California*, pages 310–319, 1998.

[Zwi99] U. Zwick. All pairs lightest shortest paths. In *Proceedings of the 31th Annual ACM Symposium on Theory of Computing, Atlanta, Georgia*, pages 61–69, 1999.

ECCC ISSN 1433-8092
http://www.eccc.uni-trier.de/eccc
ftp://ftp.eccc.uni-trier.de/pub/eccc
ftpmail@ftp.eccc.uni-trier.de, subject 'help eccc'