Algorithmica (2006) 46: 181–192
DOI: 10.1007/s00453-005-1199-1

Algorithmica

A Slightly Improved Sub-Cubic Algorithm for the All Pairs Shortest Paths Problem with Real Edge Lengths[1]

Uri Zwick[2]

Abstract. We present an $O(n^3\sqrt{\log\log n}/\log n)$-time algorithm for the All Pairs Shortest Paths (APSP) problem for directed graphs with real edge lengths. This slightly improves previous algorithms for the problem obtained by Fredman, Dobosiewicz, Han, and Takaoka.

Key Words. Shortest paths, Min-plus products, Addition-comparison model, Decision trees, Bit-level parallelism.

1. Introduction. The input to the All Pairs Shortest Paths (APSP) problem is a directed graph $G = (V, E)$ with a length function $\ell\colon E \to \mathbb{R}$ defined on its edges. The goal is to find, for every pair of vertices $u, v \in V$, the distance from u to v in the graph, and possibly also a shortest path from u to v in the graph. (If there is a path from u to v in the graph that passes through a cycle of negative length, the distance from u to v is defined to be $-\infty$. If there is no path from u to v, the distance is defined to be $+\infty$.)

The APSP problem for directed graphs with real edge weights can be solved in $O(mn + n^2 \log n)$ time by running Dijkstra's [6] Single Source Shortest Path (SSSP) algorithm from each vertex, where $n = |V|$ and $m = |E|$ are the number of vertices and edges, respectively, in the graph. The quoted running time assumes the use of Fibonacci heaps [10], or an equivalent data structure. If some of the edge lengths are negative, then a preprocessing step described by Johnson [16] is necessary. A slightly improved running time of $O(mn + n^2 \log\log n)$ was recently obtained by Pettie [19], based on an approach initiated by Thorup [27], Hagerup [14], and Pettie and Ramachandran [20].

On dense graphs with $m = \Omega(n^2)$, the worst-case running times of the algorithms mentioned above is $\Theta(n^3)$. A running time of $O(n^3)$ is also obtained by the simple Floyd–Warshall algorithm (see [8] and [28]). Can the APSP problem be solved in sub-cubic, i.e., $o(n^3)$ time? An affirmative answer was provided by Fredman [9] who showed that the problem can be solved in $O(n^3(\log\log n/\log n)^{1/3})$. This time bound was subsequently improved to $O(n^3\sqrt{\log\log n/\log n})$ by Takaoka [25], to $O(n^3/\sqrt{\log n})$ by Dobosiewicz [7], to $O(n^3(\log\log n/\log n)^{5/7})$ by Han [15], and very recently to $O(n^3(\log\log n)^2/\log n)$, again by Takaoka [26]. We present here a further improved algorithm with a running time of $O(n^3\sqrt{\log\log n}/\log n)$.

[1] A preliminary version of this paper appeared in the *Proceedings of the* 15*th International Symposium on Algorithms and Computation* (*ISAAC* '04), Hong Kong, 2004, pp. 921–932.

[2] Department of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel. zwick@cs.tau.ac.il.

Received April 25, 2005. Communicated by H. Gabow.
Online publication April 28, 2006.

The complexity of the APSP problem is known to be the same as the complexity of the *min-plus* matrix multiplication problem (see Theorem 5.7 on page 204 of [1]). If $A = (a_{ij})$ and $B = (b_{ij})$ are two $n \times n$ matrices, we let $A * B$ be the $n \times n$ matrix whose (i, j)th element is $(A * B)_{ij} = \min_k \{a_{ik} + b_{kj}\}$. We refer to $A * B$ as the min-plus product of A and B. (It is trivial to see that the APSP problem can be solved by computing $\log n$ min-plus products. A more intricate argument, given in [1], shows that this extra $\log n$ can be avoided.)

The min-plus product can be naively computed using $O(n^3)$ additions and comparisons. Fredman [9] made the intriguing observation (see also Section 3) that the min-plus product of two $n \times n$ matrices can be inferred after performing only $O(n^{2.5})$ comparisons of sums of two matrix elements! The catch is that Fredman does not specify explicitly which comparisons should be made, nor how to infer the result from the outcome of these comparisons. In more exact terms, Fredman [9] shows that there is a *decision tree* for computing the min-plus product of two $n \times n$ real matrices whose depth is $O(n^{2.5})$. However, he does not construct such a decision tree explicitly.

Fredman [9] was able, however, to use his observation to obtain an explicit sub-cubic algorithm for the min-plus product, and hence for the APSP problem. This is done by explicitly constructing a decision tree of depth $O(m^{2.5})$ for the min-plus product of two $m \times m$ matrices, where $m = o(\log n)$. The size of this decision tree, which is exponential in m, is $o(n)$. As the product of two $n \times n$ matrices can be solved by computing $(n/m)^3$ products of $m \times m$ matrices, an $o(n^3)$ algorithm is obtained for the problem of multiplying two $n \times n$ matrices.

The main technique used by Fredman [9] to implement his algorithm is *table look-up*. Han [15] and Takaoka [25], [26] present more efficient algorithms based on similar ideas. Dobosiewicz [7] uses a somewhat different approach. The speed-up of his algorithm is obtained by using *bit-level parallelism*, i.e., the ability to operate simultaneously on $\log n$ bits contained in a single machine word. The exact computational model used by all these algorithms is discussed in the next section. We stress here that the *same* model is used in all cases, so our improved algorithm is *not* obtained by using a stronger machine model. Our algorithm uses both table look-ups and bit-level parallelism. It uses ideas appearing in the Boolean matrix multiplication algorithm of Arlazarov et al. [3]. It is also inspired by recent dynamic algorithms for the transitive closure in directed graphs (see, e.g., [17], [5], and [21]).

Much faster, and truly sub-cubic, algorithms are known for the standard matrix multiplication problem. Strassen [24] obtained an $O(n^{2.81})$-time algorithm. The best available bound is currently $O(n^{2.38})$, obtained by Coppersmith and Winograd [4]. It remains a major open problem whether these techniques could be used to obtain faster algorithms for the min-plus product of matrices with arbitrary *real* edge weights.

Fast algebraic matrix multiplication algorithms were used to obtain faster algorithms for the APSP problem with small integer edge lengths. Zwick [31], improving a result of Alon et al. [2], obtained an $O(n^{2.58})$ algorithm for the APSP problem for unweighted directed graphs. Even better algorithms are known for undirected graphs with small integer edge weights (see [22], [13], [12], and [23]). For more results on the APSP problem and its various variants, see [30].

The rest of this paper is organized as follows. In the next section we discuss the model of computation used. In Section 3 we review the ideas of Fredman [9]. In Section 4 we describe the algorithm of Dobosiewicz [7] on which our algorithm is based. In

Section 5 we present the Boolean matrix multiplication algorithm of Arlazarov et al. [3]. Finally, in Section 6 we present our algorithm which uses a combination of ideas from all previous sections. We end in Section 7 with some concluding remarks and open problems.

2. Model of Computation. We use the standard RAM model of computation (see, e.g., [1]). Each memory cell can either hold a *real* number or a w-bit integer. Usually, we assume that $w = \Theta(\log n)$, which is the standard realistic assumption, where n is the input size. All the bounds in the Abstract and Introduction make this assumption.

Reals numbers are treated in our model as an *abstract data type*. The only operations allowed on real numbers are additions and comparisons. (As explained below, subtractions can be simulated in our model.) No conversions between real numbers and integers are allowed. This model is sometimes referred to as the *addition-comparison* model (see, e.g., [30], [20], and [19]).

Although we are mainly interested in the case $w = \Theta(\log n)$, we explicitly describe the dependence of the running times of our algorithm on the word size w. This shows more clearly which logarithmic factors are the result of the word size, i.e., the effect of bit-level parallelism, and which are obtained using other techniques, e.g., table look-up. We always assume that $w \geq \log n$.

The operations allowed on integers are the standard arithmetical and logical operations. We can thus add two integers and compute their bitwise or. We also assume that we have an instruction that returns the index of one of the 1's in a non-zero word. We do not give an explicit list of the integer instructions assumed. The reason is that when $w = \Theta(\log n)$, which is the case we are most interested in, any conceivable instruction can be emulated, in constant time, using table look-up. In particular, even if there is no instruction for returning the index of, say, the leftmost 1 in a non-zero word, we can still find this index, in $O(1)$ time, using table look-up. The time needed for initializing the table will be negligible compared with the other operations performed by our algorithms.

As stated above, our model only allows additions and comparisons of real numbers. It is not difficult to see, however, that allowing subtractions does not change the strength of the model. We simply represent each intermediate result as the difference of two real numbers. When two difference $x_1 - y_1$ and $x_2 - y_2$ need to be compared, we do that by comparing $x_1 + y_2$ and $x_2 + y_1$. It is interesting to note that this simple observation also lies at the heart of Fredman's [9] technique.

Our realistic model of computation should be contrasted with the unrealistic, but nevertheless interesting, model used by Yuval [29]. He shows that the distance product of two matrices with *integer* elements can be computed by first converting the elements of the matrix into very high precision real numbers, performing a standard algebraic product, and then converting the elements back into integers. The conversion steps use the computation of exact exponentials and logarithms. More careful implementations of Yuval's algorithm, in realistic models of computation, combined with other techniques, form the basis of the algorithms of [13], [12], [23], and [31].

3. The Algorithm of Fredman. Let $A = (a_{ij})$ be an $n \times m$ matrix, and let $B = (b_{ij})$ be an $m \times n$ matrix. The distance product $C = A * B$ can be naively computed using

$O(mn^2)$ operations. Fredman [9] observed that the product can also be *deduced* after performing only $O(m^2n)$ operations.

THEOREM 3.1 [9]. *Let $A = (a_{ij})$ be an $n \times m$ matrix, and let $B = (b_{ij})$ be an $m \times n$ matrix with real elements. Then the distance product $C = A * B$ can be* deduced *from the information gathered by performing at most $O(m^2n)$ comparisons of differences of elements of the matrices A and B.*

PROOF. Let $a^i_{rs} = a_{ir} - a_{is}$ and $b^j_{rs} = b_{sj} - b_{si}$, for $i, j \in [n]$ and $s, r \in [m]$. These differences can be formed in $O(m^2n)$ time, and sorted using $O(m^2n \log(mn))$ comparisons. Fredman [9] actually shows that the differences can be sorted using only $O(m^2n)$ comparisons. (For a proof, see [9].)

Let $\bar{a}^i_{rs}$ be the index of a^i_{rs} in the sorted sequence, and let $\bar{b}^j_{rs}$ be the index of b^j_{rs} in the sequence, for $i, j \in [n]$ and $s, r \in [m]$. While sorting the sequence, we assume that ties are resolved in favor of the a^i_{rs} elements, i.e., if $a^i_{rs} = b^j_{r's'}$, then a^i_{rs} appears before $b^j_{r's'}$ in the sorted sequence and thus $\bar{a}^i_{rs} < \bar{b}^j_{r's'}$. With this convention we have $a^i_{rs} \leq b^j_{r's'}$ if and only if $\bar{a}^i_{rs} \leq \bar{b}^j_{r's'}$.

For every $i, j \in [n]$, we can now find an index $r = r_{ij} \in [m]$ for which $c_{ij} = a_{ir} + b_{rj}$ just by looking at the indices $\bar{a}^i_{rs}$ and $\bar{b}^i_{rs}$, without looking again at the elements of the matrices A and B. For every $i, j \in [n]$, we want to find an index r for which $a_{ir} + b_{rj} \leq a_{is} + b_{sj}$, for every $s \in [m]$. Note, however, that

$$\begin{aligned} a_{ir} + b_{rj} \leq a_{is} + b_{sj} \quad &\Leftrightarrow \quad a_{ir} - a_{is} \leq b_{sj} - b_{si} \quad \Leftrightarrow \quad a^i_{rs} \leq b^j_{rs} \\ &\Leftrightarrow \quad \bar{a}^i_{rs} \leq \bar{b}^j_{rs}. \end{aligned}$$

Thus, the outcome of every comparison needed to determine an appropriate index r is implied by the indices computed. □

The above "algorithm" does not explain *how* to use the indices $\bar{a}^i_{rs}$ and $\bar{b}^i_{rs}$ to determine the indices r_{ij}. It just says that these indices contain enough information to determine the result uniquely.

The fast "algorithm" for rectangular min-plus products of Theorem 3.1 can be used to obtain a fast "algorithm" for square min-plus products as follows:

THEOREM 3.2 [9]. *Let $A = (a_{ij})$ and $B = (b_{ij})$ be two $n \times n$ matrices with real elements. Then, the distance product $C = A * B$ can be* deduced *from the information gathered by performing at most $O(n^{2.5})$ comparisons of differences of elements of the matrices A and B.*

PROOF. Let $1 \leq m \leq n$ be a parameter to be chosen later. We split the matrix A into n/m matrices $A_1, A_2, \ldots, A_{n/m}$ of size $n \times m$, and the matrix B into n/m matrices $B_1, B_2, \ldots, B_{n/m}$ of size $m \times n$. Clearly, $A * B = \min_{i=1}^{k} A_i B_i$, where the min here is applied elementwise. Each distant product $A_i B_i$ can be determined, as described in the proof of Theorem 3.1, using $O(m^2n)$ comparisons. Computing the n/m products and

computing their elementwise minimum thus requires only

$$O\left(\frac{n}{m}\cdot(m^2n+n^2)\right)$$

comparisons. This expression is minimized for $m=\sqrt{n}$ and the resulting number of comparisons is then $O(n^{2.5})$. □

We note again that the "algorithm" given in the proof of Theorem 3.2 is not really an algorithm in the conventional sense of the word, as it does not specify how to infer the result of the distance product from the comparisons performed. More accurately, the theorem says that there is a *decision tree* for the min-plus product of two $n\times n$ matrices whose depth is $O(n^{2.5})$. Fredman [9] observes, however, that the decision tree whose existence is proved in Theorem 3.2 can be explicitly constructed for tiny values of n, and this can be used to lower slightly the cost of computing a min-plus product of two $n\times n$ matrices.

THEOREM 3.3 [9]. *Let $A=(a_{ij})$ and $B=(b_{ij})$ be two $n\times n$ matrices with real elements. Then, the distance product $C=A*B$ can computed in $O(n^3(\log\log n/\log n)^{1/3})$ time on a machine with $(\log n)$-bit words.*

Takaoka [25] simplified Fredman's explicit algorithm and reduced its running time to $O(n^3(\log\log n/\log n)^{1/2})$, again on a machine with $(\log n)$-bit words.

4. The Algorithm of Dobosiewicz. Dobosiewicz [7] discovered a slightly more efficient explicit implementation of Fredman's "algorithm" for rectangular min-plus products. Instead of using *table-lookup*, as done by Fredman [9] and Takaoka [25], the algorithm of Dobosiewicz simply uses *bit-level parallelism*.

THEOREM 4.1 [7]. *A distance product of an $n\times m$ matrix by an $m\times n$ matrix can be computed in $O(m^2n^2/w+n^2)$ time on a machine with w-bit words, where $w\le n/\log n$.*

PROOF. Dobosiewicz's algorithm is given in Figure 1. It receives an $n\times m$ matrix A, and an $m\times n$ matrix B. It returns two $n\times n$ matrices C and R. The matrix C contains the min-plus product $A*B$. The matrix R contains the minimal indices for the product, i.e., $c_{ij}=a_{i,r_{ij}}+b_{r_{ij},j}$, where $r_{ij}\in[m]$, for every $i,j\in[n]$.

The algorithm starts by setting $Z_i\leftarrow[n]$, for $i\in[n]$. The set Z_i contains all the indices j for which c_{ij} and r_{ij} were not determined yet. The algorithm maintains two other collections of sets, X_i, for $i\in[n]$, and Y_s, for $s\in[m]$. The cost of performing operations on these sets will be discussed later. We focus, first, on the correctness of the algorithm.

The main portion of the algorithm is a loop in which the variable r ranges over the values from 1 to m. In each iteration of the loop the algorithm identifies all pairs of indices (i,j), where $i,j\in[n]$, for which r is a minimal index, i.e., $(A*B)_{ij}=a_{ir}+b_{rj}$, and for which no other minimal index was found before, and sets the entries c_{ij} and

```
Algorithm (C_{n×n}, R_{n×n}) ← MULT(A_{n×m}, B_{m×n})
Let Z_i ← {1, 2, ..., n}, for i ∈ [n].
for r ← 1 to m
    Let a^i_{rs} ← a_{ir} − a_{is}, for i ∈ [n], s ∈ [m], s ≠ r.
    Let b^j_{rs} ← b_{sj} − a_{rj}, for j ∈ [n], s ∈ [m], s ≠ r.
    Form a sorted list L containing these 2(m − 1)n elements.
    Let X_i ← Z_i, for i ∈ [n].
    Let Y_s ← ∅, for s ∈ [m].
    for k ← 1 to 2(m − 1)n
        if L_k is a^i_{rs}, then X_i ← X_i − Y_s.
        if L_k is b^j_{rs}, then Y_s ← Y_s ∪ {j}.
    end
    for i ← 1 to n
        for every j ∈ X_i
            r_{ij} ← r.
            c_{ij} ← a_{ir} + b_{rj}.
        end
        Z_i ← Z_i − X_i.
    end
end
```

Fig. 1. The rectangular min-plus multiplication algorithm of Dobosiewicz.

r_{ij} accordingly. (Note that there may be several minimal indices for a pair (i, j). The algorithm will find the smallest one of them.)

As in the proof of Theorem 3.1, r is a minimal index for (i, j) if and only if $a_{ir}+b_{rj} \leq a_{is}+b_{sj}$, or, equivalently, $a_{ir}-a_{is} \leq b_{sj}-b_{rj}$, for every $s \in [m]$. The algorithm computes the differences $a^i_{rs} = a_{ir}-a_{is}$ and $b^j_{rs} = b_{sj}-b_{rj}$, for every $i, j \in [n], s \in [m]$, and forms a sorted list L containing them. (Again, as in the proof of Theorem 3.1, if $a^i_{rs} = b^j_{rs'}$, then the element a^i_{rs} is placed before $b^j_{rs'}$ in the list.) Then r is a minimal index for (i, j) if and only if a^i_{rs} appears before b^j_{rs} in the list, for every $s \in [m]$.

At the start of each iteration the algorithm sets $X_i \leftarrow Z_i$ for every $i \in [n]$. It then scans the elements of list L, one by one, while maintaining the following invariant: $j \in X_i$ if and only if $j \in Z_i$ and it is "still possible" that r is a minimal index for (i, j). To be more precise, $j \in X_i$ if and only if $j \in Z_i$ and for every $s \in [m]$, either a^i_{rs} appears before b^j_{rs} in L, or b^j_{rs} was not scanned yet. To help maintain this invariant, the algorithm also maintains, for every $s \in [m]$, a set Y_s that contains all the indices j for which b^j_{rs} was already encountered.

We see what actions should be taken to maintain the invariants when scanning the next element of L. If the scanned element is a^i_{rs}, then all the elements of Y_s should be removed from X_i. (Indeed, if $j \in Y_s$ then b^j_{rs} appears before a^i_{rs} in the list.) The algorithm thus appropriately performs $X_i \leftarrow X_i - Y_s$. If the scanned element is b^j_{rs}, we simply need to add j to Y_s, and the algorithm appropriately performs $Y_s \leftarrow Y_s \cup \{j\}$.

It is easy to see that when all the elements of L are scanned, $j \in X_i$ if and only if $j \in Z_i$ and r is a minimal index for (i, j). For each $j \in X_i$, the algorithm thus sets $r_{ij} \leftarrow r$ and $c_{ij} \leftarrow a_{ir} + b_{rj}$. The set X_i is then removed from Z_i, for $i \in [n]$. This completes the description and correctness proof of the algorithm.

We now discuss the complexity of the algorithm. Consider the cost of a single iteration of the algorithm with a given value of r. Computing the differences a^i_{rs} and b^j_{rs} takes $O(mn)$ time. Sorting them to form the list L takes $O(mn \log(mn))$ time. For each element of L we then perform two set operations on subsets of $[n]$. Each one of the sets X_i, Z_i, and Y_s can be represented as an n-bit vector which can be packed into n/w machine words. Each set operation can then be implemented in $O(n/w)$ time. The total cost of an iteration, excluding the cost of the last for loop, is thus $O(mn \log(mn) + mn^2/w)$, which is $O(mn^2/w)$ since we assume that $w \leq n/\log n$. Multiplying this by m, the number of iterations, we get a time bound of $O(m^2n^2/w)$.

To finish the proof it remains to bound the time spent in the double loop in which i ranges from 1 to n and j ranges over all the elements of X_i. To do so, note that for every $i \in [n]$, the m versions of the set X_i obtained in the m iterations are disjoint. This is so because X_i is initialized to Z_i at the beginning of each iteration, and the resulting set X_i is subtracted from Z_i at the end of each iteration. Thus, the total number of elements j extracted from the sets X_i in all iterations is exactly n^2. Extracting all the elements of a set X_i can be easily done in $O(|X_i| + n/w)$ time using a machine instruction that returns the index of one of the 1's in a non-zero machine word. (See Section 2.) The total time spent in the double loop is therefore $O(n^2 + mn/w)$. The total running time of the algorithm is therefore $O(m^2n^2/w + n^2)$, as required. □

THEOREM 4.2 [7]. *A distance product of two $n \times n$ matrices can be computed in $O(n^3/\sqrt{w})$ time on a machine with w-bit words, $w \leq n/\log n$.*

PROOF. As in the proof of Theorem 3.2, we break the product of two $n \times n$ matrices into n/m products of $n \times m$ by $m \times n$ matrices. The total time needed is then $O((n/m) \cdot (m^2n^2/w + n^2))$, which is minimized when we set $m = \sqrt{w}$. □

COROLLARY 4.3 [7]. *A distance product of two $n \times n$ matrices can be computed in $O(n^3/\sqrt{\log n})$ time on a machine with $(\log n)$-bit words.*

5. The Algorithm of Arlazarov et al. Arlazarov et al. [3] considered the different, though related, problem of computing the *Boolean*, i.e., the or-and product, of two Boolean matrices.

THEOREM 5.1 [3]. *The Boolean product of an $n \times \log n$ matrix by a $\log n \times n$ matrix can be computed in $O(n^2/w)$ time on a machine with w-bit words.*

PROOF. Let $A = (a_{ij})$ be an $n \times \log n$ matrix, and let $B = (b_{ij})$ be a $\log n \times n$ matrix. Let $C = AB$ be their $n \times n$ Boolean product. We let A_i, B_i, and C_i denote the ith row of A, B, or C, respectively. As we assume that $w \geq \log n$, each row A_i of A fits into a

```
Algorithm C_{n×n} ← BMULT(A_{n×log n}, B_{log n×n})

BT[0] ← 0
for i ← 0 to log n − 1
    for j ← 0 to 2^i − 1
        BT[2^i + j] ← B_i ∨ BT[j]
    end
end
for i ← 1 to n
    C_i ← BT[A_i]
end
```

Fig. 2. The Boolean matrix multiplication algorithm of Arlazarov et al.

single machine word. Each row of B and C, on the other hand, requires n/w machine words.

Each row A_i specifies a subset of the rows of B, of size at most $\log n$, that needs to be or'ed. Doing this naively would require $O((n \log n)/w)$ time for each row, and thus a total time of $O((n^2 \log n)/w)$.

We can save a $\log n$ factor as follows. For brevity, let $k = \log n$. For every k-bit word $x = x_1 \cdots x_k$, we let $BT[x] = \vee_{i=1}^{k} x_i B_i$, i.e., the or of the rows of B corresponding to the 1's in the word x. We can compute $BT[x]$, for every k-bit word x, in $O(2^k n/w) = O(n^2/w)$ time. Now $C_i = BT[A_i]$, for every $i \in [n]$. Thus, the rows C_i, for $i \in [n]$, can be looked up in the table BT, again in $O(n^2/w)$ time, as required. A complete description of the algorithm is given in Figure 2. □

THEOREM 5.2 [3]. *The Boolean product of two $n \times n$ matrices can be computed in $O(n^3/(w \log n))$ time on a machine with w-bit words.*

PROOF. As usual, we break the Boolean product of two $n \times n$ matrices into $n/\log n$ products of an $n \times \log n$ matrix by a $\log n \times n$ matrix. The total cost is then $O((n/\log n) \cdot (n^2/w)) = O(n^3/(w \log n))$. □

The Boolean product of two $n \times n$ matrices can be computed in $O(n^\omega)$ time, where $\omega < 2.376$ [4] is the algebraic matrix multiplication exponent (see, e.g., [11] and [18]). We do not know, however, how to utilize these fast algebraic or Boolean matrix multiplication algorithms to obtain faster algorithms for the min-plus product of matrices with real elements.

6. The New Algorithm. Using the idea used by Arlazarov et al. [3], we can obtain a slightly more efficient implementation of the algorithm of Dobosiewicz [7].

THEOREM 6.1. *A distance product of an $n \times m$ matrix by an $m \times n$ matrix can be computed in $O(m^2 n^2 \log(w \log n)/(w \log n))$ time on a machine with w-bit words.*

PROOF. The improved algorithm is obtained by providing a slightly more efficient implementation of the set operations used by the algorithm of Dobosiewicz [7].

Let t be a parameter to be chosen later. For simplicity, we assume that n/t is an integer. Each dynamic set Y_s, where $s \in [m]$, is maintained as follows. As long as Y_s contains at most t elements, we maintain a simple list $\mathcal{Y}_s$ containing the elements of Y_s. When the size of the list $\mathcal{Y}_s$ reaches t, we prepare a compressed representation Y_s^1 of $\mathcal{Y}_s$ using n/w words, and reset $\mathcal{Y}_s$ to the empty list. Elements added to the set Y_s are again added to the list $\mathcal{Y}_s$ until its length becomes t again. We then prepare a compressed representation of the set $Y_s^2 = \mathcal{Y}_s \cup Y_s^1$ and again empty $\mathcal{Y}_s$. Continuing in this way, we get $n/t - 1$ *snapshots* $Y_s^1, \ldots, Y_s^{n/t-1}$ of the set Y_s, and the list $\mathcal{Y}_s$ is always of size at most t. For every $s \in [m]$, we let v_s be the index of the last snapshot of Y_s created so far.

A set operation $X_i \leftarrow X_i - Y_s$ is now implemented as follows. We remove the elements in the list $\mathcal{Y}_s$, one by one, from X_i. This takes only $O(t)$ time. We also set $v_{is} \leftarrow v_s$ to signify that the elements of $Y_s^{v_s}$ should be removed from X_i, but we do not remove these elements as yet. These removals will be carried out at the final stage of the algorithm. Doing all these removals together will allow us to use a trick similar to the one used by Arlazarov et al. [3].

At the end of the sequence of update operations, we need to perform

$$X_i \leftarrow X_i - \bigcup_{s=1}^{m} Y_s^{v_{is}}, \qquad \text{for every} \quad i \in [n].$$

Note that for each $i \in [n]$ and $s \in [m]$ we have $0 \le v_{is} \le n/t - 1$. For brevity, let $b = n/t$. It would have been nice to be able to look up the value of $\bigcup_{s=1}^{m} Y_s^{v_{is}}$ in a table. Unfortunately, such a table will be too large as it would have to contain $b^m = (n/t)^m$ entries, which may be much larger than n. Let $k = \log n / \log(n/t)$. As $b^k = (n/t)^k = n$, we can afford to keep a table with $(n/t)^k$ entries. Thus, we can construct m/k tables, each of size n, such that the gth table will hold all the sets of the form $\bigcup_{s=gk+1}^{(g+1)k} Y_s^{v_{is}}$. Combining these m/k tables we get a two-dimensional table YT such that

$$YT\left[g, \sum_{s=gk+1}^{(g+1)k} v_s b^{s-(gk+1)}\right] = \bigcup_{s=gk+1}^{(g+1)k} Y_s^{v_s}, \qquad 0 \le g < \frac{m}{k}, \quad 0 \le v_s < b.$$

Each entry in the table is a subset of $[n]$ represented using n/w machine words. The time needed for constructing the table YT is proportional to its size in words, which is $O((m/k) \cdot n \cdot (n/w)) = O((mn^2 \log(n/t))/(w \log n))$. Each set of the form $\bigcup_{s=1}^{m} Y_s^{v_{is}}$ can now be formed in $O((m/k) \cdot (n/w))$ time by taking the union of m/k sets found in the table YT. The time needed for computing the n sets $\bigcup_{s=1}^{m} Y_s^{v_{is}}$, for $i \in [n]$, is therefore the same as the time needed for preparing the table YT. A full description of the proposed new way of implementing the set operations is given in Figure 3.

We now analyze the cost of executing all the set operations performed by the algorithm of Dobosiewicz [7] (see Figure 1) using the new implementation. We bound the cost of the $O(mn)$ set operations performed during one iteration of the algorithm. The total initialization cost is $O(mn/w)$. The total cost of handling instructions of the form $Y_s \leftarrow Y_s \cup \{i\}$ is $O(m \cdot (n + (n/t)(n/w)))$. The total cost of handling instructions of the form $X_i \leftarrow X_i - Y_s$ is $O(mn \cdot t)$. Adding the cost of the finalization stage, as discussed above,

Initialization:

for $s \leftarrow 1$ to m
  $v_s \leftarrow 0$.
  $\mathcal{Y}_s \leftarrow \emptyset$.
  $Y_s^0 \leftarrow \emptyset$.
end

$Y_s \leftarrow Y_s \cup \{i\}$:

if $|\mathcal{Y}_s| < t$ then
  $\mathcal{Y}_s \leftarrow \mathcal{Y}_s \cup \{i\}$
else
  $v_s \leftarrow v_s + 1$
  $Y_s^{v_s} \leftarrow Y_s^{v_s-1} \cup \mathcal{Y}_s$
  $\mathcal{Y}_s \leftarrow \{i\}$
end-if

$X_i \leftarrow X_i - Y_s$:

$v_{is} \leftarrow v_s$
for every $j \in \mathcal{Y}_s$
  $X_i \leftarrow X_i - \{j\}$
end

Finalization:

$b \leftarrow n/t$
$k \leftarrow \log n / \log b$

for $g \leftarrow 0$ to $m/k - 1$
  $YT[g, 0] \leftarrow \emptyset$
  for $s \leftarrow 1$ to k
    for $v \leftarrow 0$ to $b - 1$
      for $x \leftarrow 0$ to $b^{s-1} - 1$
        $YT[g, v \cdot b^s + x] \leftarrow Y_{gk+s}^v \cup YT[g, x]$
      end
    end
  end
end

for $i \leftarrow 1$ to n
  for $g \leftarrow 0$ to $m/k - 1$
    $ind \leftarrow 0$
    for $s \leftarrow gk + 1$ to $(g + 1)k$
      $ind \leftarrow b \cdot ind + v_{is}$
    end
    $X_i \leftarrow X_i - YT[ind]$
  end
end

Fig. 3. Speeding up the set operations used in the algorithm of Dobosiewicz.

we get that the total cost of an iteration is

$$O\left(mnt + \frac{mn^2 \log(n/t)}{w \log n}\right),$$

where we have neglected terms that are dominated by the two terms appearing above. To minimize the running time, we choose $t = n/(w \log n)$. The running time of an iteration is then $O((mn^2 \log(w \log n))/(w \log n))$. Multiplying this by the number of iterations we get the time bound claimed. □

THEOREM 6.2. *A distance product of two $n \times n$ matrices can be computed in $O(n^3/\sqrt{w \log n / \log(w \log n)})$ time on a machine with w-bit words.*

PROOF. Yet again, we break the Boolean product of two $n \times n$ matrices into n/m products of an $n \times m$ matrix by an $m \times n$ matrix. We compute each one of these rectangular products using the algorithm of Theorem 6.1. The total cost is then

$$O\left(\frac{n}{m} \cdot \left(\frac{m^2 n^2 \log(w \log n)}{w \log n} + n^2\right)\right).$$

Choosing $m = \sqrt{w \log n / \log(w \log n)}$, we get the time bound claimed. □

COROLLARY 6.3. *A distance product of two $n \times n$ matrices can be computed in $O(n^3\sqrt{\log\log n}/\log n)$ time on a machine with $(\log n)$-bit words.*

As an additional corollary, we get the main result of this paper.

COROLLARY 6.4. *The APSP problem for directed graphs with real edge weights can be solved in $O(n^3\sqrt{\log\log n}/\log n)$ time on a machine with $(\log n)$-bit words.*

7. Concluding Remarks. We have obtained a slightly improved sub-cubic algorithm for the APSP problem with real edge lengths. Unfortunately, we were not able to answer the following major open problem: Is there a genuinely sub-cubic algorithm for the APSP problem with real edge lengths, i.e., an algorithm that runs in $O(n^{3-\varepsilon})$ time, for some $\varepsilon > 0$?

The algorithm presented here is also the fastest known algorithm for the APSP problem with *integer* edge lengths taken, say, from the range $\{1, 2, \ldots, n\}$. Is there a genuinely sub-cubic algorithm for this version of the problem?

Note Added in Proof. An improved algorithm for the APSP problem with a running time of $O(n^3 \log n)$ was recently obtained by Timothy M. Chan.

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[2] 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.

[3] V.L. Arlazarov, E.C. Dinic, M.A. Kronrod, and I.A. Faradzev. On economical construction of the transitive closure of a directed graph. *Doklady Akademii Nauk SSSR*, 194:487–488, 1970. English translation in *Soviet Mathematics Doklady*, 11:1209–1210, 1970.

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

[5] C. Demetrescu and G.F. Italiano. Fully dynamic transitive closure: breaking through the $O(n^2)$ barrier. In *Proc. of* 41*st FOCS*, pages 381–389, 2000.

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

[7] W. Dobosiewicz. A more efficient algorithm for the min-plus multiplication. *International Journal of Computer Mathematics*, 32:49–60, 1990.

[8] R.W. Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

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

[10] 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.

[11] M.E. Furman. Application of a method of rapid multiplication of matrices to the problem of finding the transitive closure of a graph. *Doklady Akademii Nauk SSSR*, 194:524, 1970. English translation in *Soviet Mathematics Doklady*, 11:1252, 1970.

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

[13] 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.
[14] T. Hagerup. Improved shortest paths on the word RAM. In *Proc. of* 27*th ICALP*, pages 61–72, 2000.
[15] Y. Han. Improved algorithm for all pairs shortest paths. *Information Processing Letters*, 91(5):245–250, 2004.
[16] D.B. Johnson. Efficient algorithms for shortest paths in sparse graphs. *Journal of the ACM*, 24:1–13, 1977.
[17] V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *Proc. of* 40*th FOCS*, pages 81–91, 1999.
[18] I. Munro. Efficient determination of the transitive closure of a directed graph. *Information Processing Letters*, 1(2):56–58, 1971.
[19] S. Pettie. A new approach to all-pairs shortest paths on real-weighted graphs. *Theoretical Computer Science*, 312(1):47–74, 2004.
[20] S. Pettie and V. Ramachandran. Computing shortest paths with comparisons and additions. In *Proc. of* 13*th SODA*, pages 267–276, 2002.
[21] L. Roditty and U. Zwick. Improved dynamic reachability algorithms for directed graphs. In *Proc. of* 43*rd FOCS*, pages 679–688, 2002.
[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] A. Shoshan and U. Zwick. All pairs shortest paths in undirected graphs with integer weights. In *Proc. of* 40*th FOCS*, pages 605–614, 1999.
[24] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.
[25] T. Takaoka. A new upper bound on the complexity of the all pairs shortest path problem. *Information Processing Letters*, 43:195–199, 1992.
[26] T. Takaoka. A faster algorithm for the all-pairs shortest path problem and its application. In *Proc. of* 10*th COCOON*, pages 278–289, 2004.
[27] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM*, 46:362–394, 1999.
[28] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.
[29] G. Yuval. An algorithm for finding all shortest paths using $N^{2.81}$ infinite-precision multiplications. *Information Processing Letters*, 4:155–156, 1976.
[30] U. Zwick. Exact and approximate distances in graphs – a survey. In *Proc. of* 9*th ESA*, pages 33–48, 2001.
[31] U. Zwick. All-pairs shortest paths using bridging sets and rectangular matrix multiplication. *Journal of the ACM*, 49:289–317, 2002.