THEORY OF COMPUTING, Volume 5 (2009), pp. 173–189
www.theoryofcomputing.org

All Pairs Bottleneck Paths and Max-Min Matrix Products in Truly Subcubic Time*†

Virginia Vassilevska Ryan Williams Raphael Yuster

Received: July 25, 2008; published: September 13, 2009.

Abstract: In the *all pairs bottleneck paths* (APBP) problem, one is given a directed graph with real weights on its edges. Viewing the weights as capacities, one is asked to determine, for all pairs (s,t) of vertices, the maximum amount of flow that can be routed along a *single* path from s to t. The APBP problem was first studied in operations research, shortly after the introduction of maximum flows and all pairs shortest paths.

We present the first truly subcubic algorithm for APBP in general dense graphs. In particular, we give a procedure for computing the $(\max, \min)$-product of two arbitrary matrices over $\mathbb{R} \cup \{\infty, -\infty\}$ in $O(n^{2+\omega/3}) \leq O(n^{2.792})$ time, where n is the number of vertices and ω is the exponent for matrix multiplication over rings. Max-min products can be used to compute the maximum bottleneck values for all pairs of vertices together with a "successor matrix" from which one can extract an explicit maximum bottleneck path for any pair of vertices in time linear in the length of the path.

ACM Classification: G.2.2, F.2.2

AMS Classification: 05C85, 68R10

Key words and phrases: bottleneck path, maximum capacity path, matrix multiplication, subcubic time

*This paper is based upon a preliminary version [29] appearing in the *Proceedings of the 39th Annual ACM Symposium on Theory of Computing (STOC)*, San Diego, California, 2007. The work was done while the first two authors were at Carnegie Mellon University.

†The first author was sponsored by the National Science Foundation under contracts no. CCR-0122581, no. CCR-0313148, and no. IIS-0121641. While at the Institute for Advanced Study, the first and second authors were supported by the National Science Foundation under grant no. CCF-0832797. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

 DOI: 10.4086/toc.2009.v005a009

1 Introduction

In recent years, researchers have found surprisingly strong connections between the complexity of fundamental graph problems and the complexity of matrix multiplication over a ring. Much of the prominent work in this area [22, 12, 24, 30] has developed fast algorithms for certain interesting cases of the all pairs shortest paths (APSP) problem in truly subcubic time, i. e., $O(n^{3-\delta})$ for some constant $\delta > 0$, where n is the number of vertices in the graph. Still, it remains to be seen if the general APSP problem can be solved in truly subcubic time. Several algorithms have been given for solving APSP in $n^{3-o(1)}$ time; the most recent development is by Chan [2] and runs in $O(n^3 \log\log^3 n/\log^2 n)$ time.

While we are still unable to give a bona fide subcubic algorithm for APSP, we do present such an algorithm for an intimately related problem, *all pairs bottleneck paths* (APBP). In this problem, one is given a directed graph with (arbitrary) real edge weights representing capacities, and the problem is to report, for all pairs (s,t) of vertices, the maximum amount of flow that can be routed from s to t along any single path. (This amount is given by the smallest weight edge on the path, a. k. a. the *bottleneck* edge.) Our algorithm for APBP runs in $O(n^{2+\omega/3}) \le O(n^{2.792})$ time, where ω is the exponent of matrix multiplication over a ring. We can also obtain explicit maximum bottleneck paths: after $\tilde{O}(n^{2+\omega/3})$ preprocessing, we can return an explicit simple maximum bottleneck path between any pair of vertices s,t in $O(\ell)$ time, where ℓ is number of edges in the returned path. That is, the algorithm can be used to efficiently *find* maximum bottleneck paths as well.

The APBP problem has been studied alongside APSP in several contexts. Pollack [21] introduced APBP (calling it the *maximum capacity route problem*), and showed how the cubic APSP algorithms of that time could be modified to solve it. Hu [13] proved that in *undirected* graphs, APBP can be solved in $O(n^2)$ time by simply taking the paths in a maximum spanning tree. Therefore the problem on undirected graphs can actually be solved in $O(n^2)$ time, which is optimal. The directed case of the problem has remained open until now, and recently appeared as an explicit goal in Shapira et al. [23]. Prior to our work, the fastest algorithm for general APBP used Fredman and Tarjan's implementation of Dijkstra's algorithm [9] on all nodes, in $O(mn + n^2 \log n)$ time, where m and n are the number of edges and nodes in the graph, respectively.

A problem related both to APSP and APBP is the *all pairs bottleneck shortest paths* problem (APBSP), first considered by [23]. Consider a scenario in which we want to get from location u to location v in as few hops as possible, and subject to this, we wish to maximize the flow that we can route from u to v. In other words, we want to compute for each pair of vertices, the shortest (unweighted) distance $d(u,v)$ and the maximum bottleneck weight $b(u,v)$ of a path of length $d(u,v)$ from u to v. Shapira et al. [23] gave a truly subcubic algorithm for APBSP in the *node-weighted* case. We show that the more general edge-weighted case can also be solved in subcubic time. Our solution runs in $\tilde{O}(n^{\frac{15+\omega}{6}}) \le O(n^{2.896})$ time.

Our method for APBP and APBSP is based on a new $O(n^{2+\omega/3})$ algorithm for computing the $(\max,\min)$-product of two $n \times n$ matrices with arbitrary entries from $\mathbb{R} \cup \{\infty, -\infty\}$.

Definition 1.1. The $(\max,\min)$-*product* of an $n \times \ell$ matrix A and an $\ell \times m$ matrix B is the $n \times m$ matrix $C = A \odot B$ such that

$$C[i,j] = \max_{k=1,\ldots,\ell} \min\{A[i,k], B[k,j]\} \quad \text{for all } i = 1,\ldots,n \text{ and } j = 1,\ldots,m.$$

This is the ordinary matrix product over the $(\max,\min)$ semiring with entries from $\mathbb{R}\cup\{\infty,-\infty\}$, and it is the natural generalization of the Boolean matrix product to totally ordered sets of arbitrary size. Besides its importance in flow problems, the $(\max,\min)$-product is also an important operation in fuzzy logic, where it is known as the *composition of relations* ([7], pg. 73). The ideas behind our $(\max,\min)$-product algorithm use ingredients from the dominance approaches of prior work; for more details, see Section 3.

Throughout this paper we use the standard addition-comparison computational model, along with random access to registers. In the algorithms of this paper, the only operations we actually use on real numbers are comparisons between them.

1.1 Related work

In addition to the work mentioned above, there are a few other interesting results on APBP that deserve mention. Karger et al. [16] show that any "path comparison" algorithm (that only accesses edge weights by comparing the weights of two different paths) requires $\Omega(n^3)$ time to compute both APSP and APBP. By way of fast matrix multiplication, our algorithm performs comparisons on rather unrelated pairs of edges, circumventing the above lower bound. Subramanian [26] proved that on random (Erdős-Rényi) graphs, both APBP and APSP can be solved in $O(n^2 \log n)$ time.

Very recently, Shapira et al. [23] have given algorithms for APBP in the special case where the *vertices* have weights, but not the edges. Their algorithms run in $O(n^{2.58})$ time and use fast rectangular matrix multiplication [4, 14]. Note that if $\omega = 2$, then their algorithms can be implemented to run in roughly $O(n^{2.5})$ time. Note that the vertex-weight case can be easily reduced to the edge-weight case, by setting the weight of an edge to be the minimum weight of its two endpoints. Their algorithm relies on the linearity of the number of weights. As the number of weights in the vertex-weight case is only n, but the number in the edge-weight case can be $\Omega(n^2)$, their techniques do not seem to apply to the latter case. The authors of [23] also stated the goal of finding a truly subcubic algorithm for $(\max,\min)$ matrix product as an open problem, which we resolve in this paper.

2 Preliminaries

For every graph (V,E) in this paper we let $n = |V|$ and $m = |E|$. We refer to the elements of V as nodes and vertices interchangeably. We refer to the elements of E as edges. Without loss of generality, the graphs in this paper are weakly connected, so that $m \geq n-1$.

For a positive integer k, we use $[k]$ to denote $\{1,\ldots,k\}$.

We use M^T to denote the transpose of a matrix M. As is common, we define $\omega \geq 2$ to be the infimum of all real numbers s such that matrix multiplication over a ring is in $O(n^s)$ arithmetic operations. The best known upper bound for ω is < 2.376, given by Coppersmith and Winograd [5].

We use a special matrix product in our algorithms, first defined by Matoušek [19].

Definition 2.1. Given two $n \times n$ matrices A and B over a totally ordered set, the *dominance product* is the $n \times n$ matrix $C = A \circledcirc B$ defined by

$$C[i,j] = \bigl|\{k \mid A[i,k] \leq B[k,j]\}\bigr|.$$

For technical reasons, we use the following definition of weight functions.

Definition 2.2. An *edge-weighted graph* $G = (V, E, w)$ consists of a directed graph (V, E) and a *weight function* $w : V \times V \to \mathbb{R} \cup \{-\infty, \infty\}$ with the following properties for all $u, v \in V$:

- $w(u, u) = \infty$,
- if $(u, v) \in E$ then $w(u, v) < \infty$,
- if $u \neq v$ and $(u, v) \notin E$ then $w(u, v) = -\infty$.

It is obvious that a standard weight function from E to $\mathbb{R}$ can be uniquely extended to a weight function in the above sense.

Definition 2.3. Given an edge-weighted graph $G = (V, E, w)$, a *bottleneck edge* of a path between vertices u and v is a smallest weight edge on that path. A *maximum bottleneck path* between u and v is a path whose bottleneck edge weight is equal to the maximum of the bottleneck edge weights of all paths from u to v.

3 The dominance approach

We begin by revisiting an approach used by Chan [3] and Vassilevska and Williams [28] to find improved algorithms for all pairs shortest paths and maximum node-weighted triangles, respectively. In this approach, one reduces a weighted graph problem to the *dominating pairs* problem from computational geometry, then uses a fast algorithm for that problem. The dominating pairs problem gives a set X of n points in k-dimensional space, and the task is to compute all pairs (x, y) where $x, y \in X$ and $x[i] \leq y[i]$ for all coordinates i.

Let M_X be the $n \times k$ matrix whose rows are the points of X. One way to determine dominating pairs is to compute the dominance product of M_X and M_X^T, as defined in Section 2. Then, $(M_X \circledleq M_X^T)[i, j] = k$ if and only if (i, j) is a dominating pair. The best known algorithm in terms of n for the dominance product of two $n \times n$ matrices is due to Matoušek [19].

Theorem 3.1 (Matoušek [19])**.** *The dominance product of two $n \times n$ matrices A and B with entries from a totally ordered set is computable in $O(n^{\frac{3+\omega}{2}})$ time.*

A nice advantage of the dominance approach is that sums of pairs of elements can be quickly compared to a global constant, which is useful in some weighted graph problems. For example, suppose we are given a constant K and an edge-weighted graph $G = (V, E, w)$ where $w : E \to \mathbb{R}$, and we want to compute for all pairs of vertices i, j whether there is a path of the form $i \to k \to j$ of total sum at least K. Then one can set up matrices A and B so that

$$A[i,k] := \begin{cases} K - w(i,k) & \text{if } (i,k) \in E \\ \infty & \text{otherwise,} \end{cases}$$
$$B[k,j] := \begin{cases} w(k,j) & \text{if } (k,j) \in E \\ -\infty & \text{otherwise.} \end{cases}$$

Then $(A \circledcirc B)[i,j] \neq 0$ if and only if there is a k for which $(i,k),(k,j) \in E$ and $K - w(i,k) \leq w(k,j)$, i. e., $w(i,k) + w(k,j) \geq K$. In this paper, we find a new application of the dominance approach, culminating in a genuinely subcubic algorithm for APBP.

In our applications that use a dominance product, we shall only want to perform comparisons with certain entries of the matrices. For example, suppose matrices A and B are over $\mathbb{R} \cup \{\infty\}$, such that A has mostly ∞ entries, while B has mostly finite entries. Then, in the computation of the dominance product $A \circledcirc B$, many of the comparisons $(A[i,k] \leq B[k,j])$ are false; it only makes sense to compare the *finite* entries of A with entries in B. To this end, we design a special algorithm for dominance product, in the case where one wishes to ignore large portions of the matrix A.

Theorem 3.2 (Sparse Dominance Product)**.** *Let A and B be $n \times n$ matrices with entries from a totally ordered set. Let $S \subseteq [n] \times [n]$ such that $|S| = m \geq n^{\omega-1}$. Let C be the matrix such that*

$$C[i,j] = \left|\{k \mid (i,k) \in S \text{ and } A[i,k] \leq B[k,j]\}\right|.$$

There is an algorithm `SD` *that, given A, B, and S, outputs C in $O(\sqrt{m} \cdot n^{\frac{1+\omega}{2}})$ time.*

Proof. Call the entries of A with coordinates in S the *relevant* entries of A. For every $j = 1, \ldots, n$, let L_j be the sorted list containing the relevant entries from A in column j, along with the entries from B in row j. Let g_j be the number of relevant entries of A in L_j, for all j. Clearly, $\sum_j g_j = m$. Pick a parameter r and partition each L_j into r consecutive buckets, such that every bucket contains at most $\lceil g_j/r \rceil$ relevant entries of A. Note that the bucket sizes are not necessarily uniform.

For every bucket number $b = 1, \ldots, r$, create Boolean matrices A_b and B_b:

$$A_b[i,j] := \begin{cases} 1 & \text{if } A[i,j] \text{ is in bucket } b \text{ of } L_j \\ 0 & \text{otherwise,} \end{cases}$$
$$B_b[j,k] := \begin{cases} 1 & \text{if } B[j,k] \text{ is in bucket } b' \text{ of } L_j \text{ and } b' > b \\ 0 & \text{otherwise.} \end{cases}$$

For each bucket number b, compute $C_b = A_b \times B_b$ (where $\times$ is matrix multiplication over the integers). This step takes $O(rn^{\omega})$ time and computes for every pair i,k and bucket number b, the number of j such that $A[i,j] \leq B[j,k]$, where $A[i,j]$ is in bucket b of L_j, and $B[j,k]$ is in a different bucket of L_j.

Initialize an $n \times n$ matrix D to be all zeroes. In every bucket b of L_j, there are at most $\lceil g_j/r \rceil$ relevant entries of A and some number t_{jb} of entries from B. Compare every A-entry with every B-entry in bucket b of L_j in $O(t_{jb} \cdot \lceil g_j/r \rceil)$ time; in particular, for each $A[i,j] \leq B[j,k]$ where $A[i,j]$ and $B[j,k]$ are in bucket b, increment $D[i,k]$. Over all j and b, this takes time on the order of

$$\sum_j \sum_b t_{jb} \cdot \lceil g_j/r \rceil \leq \sum_j (1 + g_j/r) \sum_b t_{jb} = \sum_j (1 + g_j/r)n = n^2 + \sum_j g_j n/r = n^2 + mn/r.$$

After all buckets of all lists are processed, $D[i,k]$ contains the number of j such that $A[i,j] \leq B[j,k]$, where $A[i,j]$, $B[j,k]$ are in the same bucket of L_j.

Finally, set $C = \sum_{b=1}^{r} C_b + D$. It is easy to verify from the above that the algorithm returns the desired C. The overall runtime of the above procedure is $O(n^2 + mn/r + rn^{\omega})$. Choosing $r = \sqrt{m} \cdot n^{\frac{1-\omega}{2}}$, the runtime is minimized to $O(\sqrt{m} \cdot n^{\frac{1+\omega}{2}})$. $\square$

We can give a slightly more general result using a lemma by Huang and Pan [14]. Fix $r \in (0,1]$. Define ω_r to be the infimum of all real numbers s such that multiplication of an $n \times n^r$ matrix and an $n^r \times n$ matrix over a ring can be done in $O(n^s)$ arithmetic operations. Observe that

$$\omega_r \leq 2(1-r) + r\omega\,,$$

since the product of an $n \times n^r$ and $n^r \times n$ matrix can be computed with $n^{2(1-r)}$ products on pairs of $n^r \times n^r$ matrices. Building on Coppersmith [4], Huang and Pan proved:

Lemma 3.3 (Huang and Pan [14])**.** *Let $\alpha = \sup\{0 \leq r \leq 1 \mid \omega_r = 2 + o(1)\} > 0.294$. Then for all $d \geq n^{\alpha}$, one can multiply an $n \times d$ with a $d \times n$ matrix in time*

$$O(d^{\frac{\omega-2}{1-\alpha}} \cdot n^{\frac{2-\omega\alpha}{1-\alpha}}) = O(d^{0.533} n^{1.844})\,,$$

where ω is the $n \times n$ matrix multiplication exponent.

Corollary 3.4. *There is an*

$$O\left(\min\left\{n^2 + (|S_A| \cdot |S_B|)^{\frac{\omega-2}{\omega-\alpha-1}} n^{\frac{2-\alpha\omega}{\omega-\alpha-1}}, n^{\omega} + \sqrt{|S_A| \cdot |S_B|} \cdot n^{\frac{\omega-1}{2}}\right\}\right)$$

algorithm for sparse dominance product, where S_A and S_B are subsets of $[n] \times [n]$, and the resulting matrix has $C[i,j] = |\{k \mid A[i,k] \leq B[k,j], (i,k) \in S_A, (k,j) \in S_B\}|$.

Proof. Suppose A has m_1 relevant entries and B has m_2. Sort each column k of A and row k of B together. For each k, let L_k be the sorted list for column/row k. Let D be a parameter to be chosen later. For every column/row k, let m_{1k} be the number of relevant entries of A in L_k, so that $\sum_k m_{1k} = m_1$.

Bucket each list L_k into consecutive buckets, so that each bucket (except for the last one) has m_1/D relevant elements of A. Compare elements within bucket b of list k in

$$\sum_b g_{bk} \lceil m_1/D \rceil$$

where g_{bk} is the number of relevant B-elements in bucket b of L_k. Overall, the runtime is $O(m_2 + \frac{m_1 m_2}{D})$.

To handle comparisons between buckets we do the following. Create matrices C and C' where C is $n \times O(D)$ and C' is $O(D) \times n$. The columns of C and rows of C' have indices (k,b) for bucket b of L_k, provided L_k has at least 2 buckets. We set $C[i,(k,b)]$ to be 1 if $A[i,k]$ is in bucket b of L_k. We set $C'[(k,b'),j]$ to be 1 if $B[k,j]$ is in some bucket $b > b'$ of L_k. Then clearly $C[i,(k,b)] \cdot C'[(k,b),j] = 1$ iff there is some $b' > b$ such that $B[k,j]$ is in bucket b' of L_k but $A[i,k]$ is in bucket $b < b'$ of L_k and when we sum these we always count different comparisons. The number of coordinates (k,b) is at most

$$\sum_{k\colon L_k \text{ has } \geq 2 \text{ buckets}} \left\lceil \frac{m_{1k} D}{m} \right\rceil \leq D \sum_k \frac{m_{1k}}{m_1} + \sum_{k\colon L_k \text{ has } \geq 2 \text{ buckets}} 1 \leq D + D = 2D\,.$$

We can compute the product of C and C' in $O(\lceil (\frac{D}{n}) \rceil n^{\omega})$ time. If $m_1 m_2 \geq n^{\omega+1}$, the best value for D is

$$m_1 m_2 / D = D n^{\omega-1} \implies D = \sqrt{m_1 m_2} / n^{\frac{\omega-1}{2}} \geq n\,.$$

The final runtime is then $O(\sqrt{m_1 m_2} n^{\frac{\omega-1}{2}})$.

If $m_1 m_2 < n^{\omega+1}$, then the product of C and C' can be computed in $O(n^{\omega})$ or $O(D^{\frac{\omega-2}{1-\alpha}} n^{\frac{2-\alpha\omega}{1-\alpha}})$ time by Lemma 3.3, where the current best value for α is < 0.294. In the first case, we set $D = n$ and the final runtime is $O(n^{\omega})$. In the second approach, the best value for D is

$$m_2 m_1 / D = D^{\frac{\omega-2}{1-\alpha}} n^{\frac{2-\alpha\omega}{1-\alpha}} \implies D = \frac{(m_1 m_2)^{\frac{1-\alpha}{\omega-\alpha-1}}}{n^{\frac{2-\alpha\omega}{\omega-\alpha-1}}}.$$

The final runtime becomes asymptotically

$$(m_1 m_2)^{\frac{\omega-2}{\omega-\alpha-1}} n^{\frac{2-\alpha\omega}{\omega-\alpha-1}} \le O\left((m_1 m_2)^{0.33} n^{1.21}\right).$$

□

4 All Pairs Bottleneck Paths

Armed with the sparse dominance product algorithm, we now turn to the all pairs bottleneck paths problem. We first show how to compute the $(\max, \min)$-product of matrices in truly subcubic time. Just as the $(\min, +)$-product (or distance product) can be used to find all pairs shortest paths [1], the $(\max, \min)$-product gives a way to solve APBP.

4.1 Max-Min Product

Recall that the $(\max, \min)$-product of two matrices A and B is defined to be the matrix C such that $C[i, j] = \max_k \ \min\{A[i,k], B[k, j]\}$. Clearly, the $(\max, \min)$-product of two matrices A and B can be modeled by an APBP computation on a three-layered graph, where the edge weights from the first to the second layer come from A and the edge weights from the second to the third layer come from B. Moreover, Corollary 4.2 states that APBP on an n-vertex graph can be solved in roughly the time it takes to compute a $(\max, \min)$-product of $n \times n$ matrices. This result follows from a more general result (Theorem 4.1) for *closed semirings* due to due to Fischer and Meyer [8], Furman [10, 11], and Munro [20].

A closed semiring is an algebraic structure weaker than a ring, so that $(R, \oplus, \odot, 0, 1)$ is a closed semiring if all of the following conditions hold:

(1) R is a set with $0, 1 \in R$ which is closed under the binary operations $\oplus$ and $\odot$;

(2) $\oplus$ is commutative, associative, idempotent, and 0 is an identity under $\oplus$;

(3) $\odot$ is associative, distributes over $\oplus$, and 1 is an identity under $\odot$;

(4) 0 is a multiplicative annihilator: $\forall x \in R: \ x \odot 0 = 0 \odot x = 0$;

(5) finally, there is a unary operation $*$ so that $a^* = 1 \oplus (a \odot a^*)$ for all $a \in R$.

The *transitive closure* of a square matrix A over a closed semiring R is always well-defined as the solution A^* to $A^* = I \oplus (A \odot A^*)$, where I is the identity matrix over R, and the $\oplus$ and $\odot$ operations on two matrices X, Y are given by $(X \oplus Y)[i, j] = X[i, j] \oplus Y[i, j]$, and $(X \odot Y)[i, j] = \bigoplus_k (X[i,k] \odot Y[k, j])$. Under these operations, the set of $n \times n$ matrices over a closed semiring R is also a semiring, with the identity and zero matrices playing the roles of 1 and 0.

Theorem 4.1 ([8, 10, 20], [1], pp. 204–206)**.** *If the product of two arbitrary $n \times n$ matrices over a closed semiring R can be computed in $M(n)$ time so that $M(2n) \geq 4M(n)$, then there exists a constant c such that the time $T(n)$ to compute the transitive closure of an arbitrary $n \times n$ matrix over R satisfies $T(n) \leq cM(n)$.*

Since $(\mathbb{R}, \min, \max, \infty, -\infty)$ is a closed semiring (it is also known as the *subtropical* semiring), we immediately obtain the following corollary.

Corollary 4.2. *Let $M(n)$ be such that $M(2n) \geq 4M(n)$. If the* $(\max, \min)$*-product of two arbitrary real $n \times n$ matrices is computable in $M(n)$ time, then the all pairs bottleneck paths problem on an n-vertex graph can be solved in $O(M(n))$ time.*

We note that the condition $M(2n) \geq 4M(n)$ is not really restrictive. Since we need to write the output, $M(n) \geq \Omega(n^2)$. If we assume that $M(n) = f(n)n^2$ for some nondecreasing function $f(n)$, then $M(2n) = f(2n)(2n)^2 = 4f(2n)n^2 \geq 4f(n)n^2 = 4M(n)$.

We now show how to compute the $(\max, \min)$-product in truly subcubic time, using the sparse dominance algorithm combined with another idea.

Theorem 4.3 (Max-Min Product)**.** *Given two $n \times n$ matrices A and B, the matrix C with*

$$C[i, j] = \max_k \ \min\{A[i,k], B[k, j]\}$$

can be computed in $O(n^{2+(\omega/3)})$ time. Moreover, for each pair of indices i, j, the algorithm returns an index k satisfying $\min\{A[i,k], B[k, j]\} = C[i, j]$.

Proof. We first compute for every pair i, j, the maximum $A[i,k]$ (over all k) such that $A[i,k] \leq B[k, j]$, storing the results in a matrix A'. Afterwards, we reverse the roles of A and B, computing for every pair i, j, the maximum $B[k, j]$ (over all k) such that $B[k, j] \leq A[i,k]$, storing the results in a matrix B'. Then we take

$$C[i, j] = \max\{A'[i, j], B'[i, j]\} .$$

Since the above two cases (of computing A' and B') are symmetric, it suffices to show how to compute A', where

$$A'[i, j] = \max_{k \,:\, A[i,k] \leq B[k, j]} A[i,k] .$$

To do this, we employ a strategy similar to one used to obtain *maximum witnesses* for matrix multiplication [17]. In particular, for each $i, j = 1, \ldots, n$, we "narrow down" the possible choices for an $A[i,k]$ such that $A[i,k] \leq B[k, j]$, to one of g possible entries. This is done by a careful application of $O(n/g)$ sparse dominance products, in $O(n^{2+(\omega/2)}/\sqrt{g})$ time. Then for each i, j, we directly check which of the

g possible entries are valid, if any. This takes $O(n^2 g)$ time. Choosing g optimally results in a subcubic time bound.

For every row i of matrix A, make a sorted list R_i of the entries in that row. Pick a parameter g. Partition the entries of each sorted list R_i into buckets, so that for every R_i there are $\lceil n/g \rceil$ buckets with at most g entries in each bucket. For every bucket value $b = 1, \ldots, \lceil n/g \rceil$, compute $C_b = \mathtt{SD}(A, B, S_b)$, where SD is the sparse dominance product from Theorem 3.2 and

$$S_b = \{(i,j) |\, A[i,j] \text{ is in bucket } b \text{ of } R_i\}\,.$$

Notice that for every bucket value b, we have $|S_b| \leq ng$. By Theorem 3.2, all matrices C_b can be computed in

$$O\left(\frac{n}{g} \cdot \sqrt{ng} \cdot n^{\frac{1+\omega}{2}}\right) = O\left(\frac{n^{2+(\omega/2)}}{\sqrt{g}}\right) \text{ time.}$$

Now for every pair i, j, we determine the *largest* bucket $b_{i,j}$ in R_i for which there exists a k such that $A[i,k] \leq B[k,j]$. (This is obtained by taking the largest $b_{i,j}$ such that $C_{b_{i,j}}[i,j] \neq 0$. Note we can easily compute $b_{i,j}$ during the computation of the C_b.) For every i, j, we then examine the entries in bucket $b_{i,j}$ of R_i to obtain the maximum $A[i,k]$ (and hence the corresponding k) such that $A[i,k] \leq B[k,j]$. Since there are at most g entries in a bucket, each pair i, j can be processed in $O(g)$ time. Therefore, this last step takes $O(n^2 g)$ time. To pick a value for g that minimizes the runtime, we set $n^2 g = n^{2+\omega/2}/\sqrt{g}$, obtaining $g = n^{\omega/3}$. The running time is hence $O(n^{2+(\omega/3)})$. □

Plugging in the best known value for ω by Coppersmith and Winograd [5], the runtime bound becomes $O(n^{2.792})$.

4.2 Computing explicit maximum bottleneck paths

By Corollary 4.2 we can obtain a matrix representing all pairs bottleneck path weights in an edge-weighted graph in $O(n^{2+(\omega/3)})$ time. To be able to compute actual paths, a bit more work is necessary. Since paths can have linear length in general, listing the optimal paths between all pairs of vertices might require cubic time. To tackle this hurdle, we proceed as is common in the shortest paths literature: instead of explicitly representing the optimal paths, we store an $n \times n$ matrix of *successor* nodes, so that given this matrix, a maximum bottleneck path P between any pair of vertices can be recovered in time linear in the length of P.

To build the successor matrix, we take an approach analogous to that used by Zwick [30] in solving the APSP problem. First, we compute APBP by repeatedly squaring the original adjacency matrix via $(\max, \min)$-product, instead of the approach in Aho et al. [1]. We also record, for every pair of nodes i, j, the last iteration $T[i,j]$ of the repeated squaring phase in which the bottleneck edge weight was changed, together with a witness vertex w_{ij} on a path from i to j, provided by the $(\max, \min)$-product computation in that iteration.

Given an iteration matrix T and a witness matrix w_{ij} (derived from a shortest path computation), Zwick [30] gives a procedure which computes a matrix of successors in $O(n^2)$ time, and another procedure that, given a matrix of successors and a pair of nodes, returns a simple shortest path between the nodes. Applying his procedures to our setting, we get simple maximum bottleneck paths. The major difference here is that our iteration values are obtained by repeated squaring, whereas Zwick's iteration values come from his random sampling algorithm for finding witnesses. We review Zwick's algorithm below.

ALGORITHM wit-to-suc(W,T):
$S \leftarrow 0$
for $\ell = 0$ **to** $\log n$ **do** $T_\ell = \{(i,j) \mid T[i,j] = \ell\}$
for every $(i,j) \in T_0$ **do** $S[i,j] = j$
for $\ell = 1$ **to** $\log n$ **do**
    for each $(i,j) \in T_\ell$ **do**
        $k = w_{ij}$
        while $S[i,j] = 0$ **do**
            $S[i,j] \leftarrow S[i,k]$, $i \leftarrow S[i,j]$
return S

Theorem 4.4. *The all pairs bottleneck paths problem can be solved in $O(n^{2+(\omega/3)})$ time. Furthermore, in $O(n^{2+(\omega/3)} \log n)$ time algorithm wit-to-suc computes a successor matrix from which for any i,j a simple maximum bottleneck path between i and j can be recovered in $O(\ell)$ time, where ℓ is the length of the returned path.*

Proof. Let w_{ij} and $T[i,j]$ for all vertex pairs i,j be provided by repeated squaring of the adjacency matrix using $(\max,\min)$-product.

Consider algorithm wit-to-suc. Let S be the matrix of successors that the algorithm computes. The algorithm processes vertex pairs (i,j) in increasing order of their iteration numbers $T[i,j]$. The idea is that if k is a witness for (i,j), then $T[i,k]$ is an earlier iteration of the squaring than $T[i,j]$, and hence $S[i,k]$ would be set before $S[i,j]$ is processed.

We claim by induction that after a value $S[i,j]$ is set, matrix S stores a simple maximum bottleneck path from i to j which can be recovered by following successors one by one. Our argument is similar to that of Zwick [30].

At iteration 0 of the algorithm, all pairs whose maximum bottleneck path is an edge are fixed. Suppose that at the iteration in which vertex pair (i,j) is processed, the claim holds for all vertex pairs (k,ℓ) that have been processed before (i,j) (and hence which have a nonzero $S[k,\ell]$ value). Now consider the iteration in which (i,j) is processed. Let $k = w_{ij}$. Since $S[i,k]$ is set, we can use its successor value to set $S[i,j]$ since we know that a maximum bottleneck path goes through k. We then take $S[i,j]$ and if its successor on the path to j has not been set, we set it to match $S[S[i,j],k]$. We continue processing consecutive successors similarly, until we encounter some i_0 for which $S[i_0,j]$ is set (i_0 exists as k is such a vertex). Since it is set, and the path from i to k is simple (by induction), $S[i_0,j]$ must have been set before (i,j) is processed. Hence by induction, the path from i_0 to j is simple and all successors for vertices on that path to j are set. But since no successors for vertices between i and i_0 were set, then the paths i to i_0 and i_0 to j are simple and nonoverlapping, and the overall path is simple and a maximum bottleneck path. Furthermore, now the successors of all vertices on the simple path are set in the S matrix.

The algorithm for determining successors from witnesses takes $O(n^2)$ time. Given a matrix of successors, obtaining the actual path from i to j is straightforward: find $S[i,j]$ and then recursively obtain the path from $S[i,j]$ to j. This clearly takes time linear in the length of the path. □

5 All Pairs Bottleneck Shortest Paths

We first recall the well-known *short-path–long-path method* [30, 12, 2]. This method is quite general and is used to tackle various all pairs path problems.

5.1 The short-path–long-path method

All pairs path problems can often be viewed as problems of computing the transitive closure of a given (adjacency) matrix over a given algebraic structure. The given algebraic structure consists of a set of elements R and two binary operations $\oplus : R \times R \to R$ and $\odot : R \times R \to R$ (where $\oplus$ is commutative) so that the $(\oplus, \odot)$ product of two $n \times n$ matrices A and B is well-defined as

$$(A \odot B)[i,j] = \bigoplus_{k \in [n]} A[i,k] \odot B[k,j]\,.$$

For instance, APSP is the problem of computing the transitive closure of a nonnegative real matrix with 0s on the diagonal, over the $(\min,+)$-semiring (also called *tropical*); APBP is the problem of computing the transitive closure of a real matrix with ∞s on the diagonal over the $(\max,\min)$-semiring (also called *subtropical*). The short-path–long-path method is particularly useful in cases when the $\odot$ operation in the underlying algebraic structure is not necessarily associative or commutative, or if it does not fully distribute over $\oplus$ (see e. g. [27]). In such a case, computing the transitive closure of an $n \times n$ matrix seems to require n iterations of the product (hence taking $\Omega(n^3)$ operations), whereas computing the transitive closure over a semiring, for instance, can be done in asymptotically the same time as computing the matrix product [8]. The method also applies when the all pairs path problem is not necessarily a transitive closure problem, but for which computing n $(\oplus, \odot)$ matrix products can be used to solve the problem (see e. g. [2]). This is the case for the all pairs bottleneck shortest paths problem.

In the method one first chooses a parameter $\ell < n$ and iteratively computes the underlying $(\oplus, \odot)$ matrix product on ℓ pairs of matrices, obtaining each consecutive pair from the previously computed products. This computation allows one to obtain best paths of length at most ℓ between all pairs of vertices, possibly collecting other information to be used later. A common instance of the product iteration phase is just computing the ℓ-th power of the adjacency matrix. This first phase of the method takes $O(M(n)\ell)$ time, where $M(n)$ is the time to compute the particular matrix product.

After this the following lemma is used to obtain in $O(n^2\ell)$ time a set of $O((n\log n)/\ell)$ vertices hitting all shortest paths between pairs of vertices at distance ℓ. The lemma follows directly from the analysis of the greedy algorithm for hitting set.

Lemma 5.1 ([18, 15, 25])**.** *Given a collection of N subsets of $\{1,\ldots,n\}$, each of size ℓ, one can find in $O(N\ell)$ time a set of at most $\frac{n(1+\ln N)}{\ell}$ elements of $\{1,\ldots,n\}$ hitting every one of the subsets.*

An easy way to achieve the above performance bounds by a randomized algorithm is to sample a set of $(cn\ln N)/\ell$ elements from $\{1,\ldots,n\}$ independently, uniformly at random for a constant $c > 1$. By a standard argument one can show that such a sample is a hitting set with probability at least $1-1/N^{c-1}$ (cf. [30]).

After obtaining the hitting set, one argues that for any pair of vertices a best path of length $\geq \ell$ (if one exists) must contain a node from the hitting set. One designs an algorithm for the single source version of the path problem, running in, say, $T(n)$ time. Then one runs this algorithm from all nodes in the hitting set in $O((nT(n)\log n)/\ell)$ time (possibly in both directions). Finally, one combines the results in $O((n^3 \log n)/\ell)$ time by considering every pair of nodes and every possible midpoint from the hitting set.

Suppose $T(n) = O(n^2)$ as is with most problems to which Dijkstra's algorithm applies. Then the overall running time is minimized when

$$M(n)\ell = \frac{n^3 \log n}{\ell}, \quad \ell = \sqrt{\frac{n^3}{M(n)} \log n},$$

and the runtime becomes

$$O\left(n^{1.5}\sqrt{M(n)\log n}\right).$$

5.2 An algorithm for APBSP

We first give an algorithm for the single source version of the bottleneck shortest paths problem (SSBSP).

Lemma 5.2. *SSBSP on a graph with m edges and n nodes can be solved in $O(m+n)$ time.*

Proof. The algorithm is an adaptation of breadth-first search. Let $G = (V,E,w)$ be the given edge-weighted graph and s the source node. We will maintain a set Q_i which at each stage i will contain nodes at (unweighted) distance i from s. The set needs to support insert, pop an element, go through the elements one by one. A linked list suffices to support all of these operations in $O(1)$ time.

Every node v in the graph has a bit $\mathsf{visited}(v)$ which is set if and only if an edge from an in-neighbor of v to v has been traversed. Node v has values $d(v)$ and $b(v)$ associated with it. Value $d(v)$ will be the (unweighted) shortest distance from s to v and $b(v)$ will be the maximum bottleneck edge on a shortest path from s to the v. Originally, $d(v) = \infty$ for $v \neq s$ and $d(s) = 0$, $b(v) = -\infty$ for all $v \neq s$ and $b(s) = \infty$, $\mathsf{visited}(v) = 0$ for all $v \neq s$, $\mathsf{visited}(s) = 1$.

We begin by inserting each out-neighbor v of s into Q_1. We set $d(v) = 1$, $b(v) = w(s,v)$ and $\mathsf{visited}(v) = 1$. We then process Q_1.

To process Q_i, repeat: pop a node v from Q_i; for all out-neighbors u of v:

- if u is not visited, insert u into Q_{i+1}, set $d(u) = i+1$, $b(u) = \max\{b(u), \min\{b(v), w(v,u)\}\}$, and $\mathsf{visited}(u) = 1$.

- else if u is visited and if $d(u) = i+1$, set $b(u) = \max\{b(u), \min\{b(v), w(v,u)\}\}$.

When Q_i is empty, process Q_{i+1} if Q_{i+1} is nonempty.

Correctness follows by induction: if the bottlenecks for nodes in Q_{i-1} are correct, then since every path of length i from s to u must be of the form P_{sv} followed by (v,u) where P_{sv} is a path of length

$i-1$ and $(v,u) \in E$, going through all nodes $v \in Q_{i-1}$ and setting $b(u) = \max\{b(u), \min\{b(v), w(v,u)\}\}$ computes the correct bottleneck weight.

The running time of the algorithm is $O(m+n)$ since we go through each edge at most once and a node is only accessed via an incoming edge or when popping it from a set Q_i. $\square$

Now we can apply the short-path–long-path method to prove:

Theorem 5.3. *APBSP on an n-node graph can be solved in* $O(n^{\frac{15+\omega}{6}}\sqrt{\log n}) \leq O(n^{2.896})$ *time.*

Proof. In our application of the method, the matrix product we use is the $(\max,\min)$-product, taking time $M(n) = O(n^{2+(\omega/3)})$ per iteration. Let $G = (V,E,w)$ be the given edge-weighted graph. Its adjacency matrix A is defined as follows for $j,k \in [n]$:

$$A[j,k] = \begin{cases} \infty & \text{if } j = k, \\ w(j,k) & \text{if } (j,k) \in E, \\ -\infty & \text{otherwise.} \end{cases}$$

For a parameter ℓ, we iterate the $(\max,\min)$-product on the adjacency matrix ℓ times as follows. At each iteration $i = 1,\ldots,\ell$ we have a matrix D^i containing the unweighted distance between any two nodes at distance at most i, and a matrix A^{i-1} which contains bottleneck values from iteration $i-1$. In iteration 1 we have A^0 and D^1 as follows: for $j,k \in [n]$,

$$A^0[j,k] = \begin{cases} \infty & \text{if } j = k \\ -\infty & \text{otherwise,} \end{cases} \quad \text{and} \quad D^1[i,j] = \begin{cases} 1 & \text{if } (i,j) \in E, \\ 0 & \text{if } i = j, \\ \infty & \text{otherwise.} \end{cases}$$

At each iteration i, we compute $C^i = A^{i-1} \odot A$ (where $\odot$ is the $(\max,\min)$-product). After computing C^i, we set for all $j,k \in [n]$

$$D^{i+1}[j,k] = \begin{cases} D^i[j,k] & \text{if } D^i[j,k] < \infty, \\ i & \text{if } C^i[j,k] > -\infty \text{ and } D^i[j,k] = \infty, \\ \infty & \text{otherwise.} \end{cases}$$

After D^{i+1} is computed, we create A^i by setting for all $j,k \in [n]$

$$A^i[j,k] = \begin{cases} C^i[j,k] & \text{if } D^{i+1}[j,k] = i, \\ -\infty & \text{otherwise.} \end{cases}$$

$A^i[j,k]$ is the maximum bottleneck edge weight on a path from j to k of length i if i is the shortest distance between j and k. Over all $i = 1,\ldots,\ell$, computing all A^i and D^{i+1} takes $O(\ell n^{2+(\omega/3)})$ time.

The short-path–long-path method requires that we find a hitting set S in $O(\ell n^2)$ time and do SSBSP from all nodes in S. By Lemma 5.2 this takes $O(n^3 \log n/\ell)$ time. We obtain all distances and bottlenecks by combining the results in $O(n^3 \log n/\ell)$ time as given by the method. We set

$$\ell = \sqrt{\frac{n^3}{M(n)} \log n} = n^{\frac{3-\omega}{6}} \sqrt{\log n},$$

and the runtime becomes

$$O\left(n^{1.5}\sqrt{M(n)\log n}\right) = O(n^{\frac{15+\omega}{6}}\sqrt{\log n}) = O(n^{2.896})\,.$$

□

6 Conclusion

We have provided the first truly subcubic algorithms for all pairs bottleneck paths and all pairs bottleneck shortest paths in general dense graphs, with no restrictions on edge weights or edge directions. Our approach combines several different ingredients from past work, along with a few new ideas, to reduce the problem of computing the $(\max,\min)$ matrix product to a small collection of 0-1 matrix products. Timothy Chan (personal communication) has observed that the running time of our algorithm can be slightly improved (from $n^{2.792}$ to $n^{2.781}$) by using fast rectangular matrix multiplication [4, 14]. More recently, Duan and Pettie [6] have extended our techniques to show that the $(\max,\min)$ matrix product can be computed in $O(n^{(3+\omega)/2}) = O(n^{2.688})$ time, the best known time for computing the dominance product. It is still an open problem whether the dominance or $(\max,\min)$-products can be computed in $O(n^{\omega})$ time.

The most pressing question from our work is if the ideas from our $(\max,\min)$ matrix product algorithm can be extended further to obtain a $O(n^{3-\delta})$ algorithm for the $(\min,+)$ matrix product (that is, the distance product). Note we already know that the dominance approach can be used to obtain the k *most significant bits* of the distance product in $O(2^k n^{(3+\omega)/2})$ time [28]. An affirmative answer would immediately imply a truly subcubic APSP algorithm for general graphs, resolving a longstanding and prominent open problem.

Acknowledgments. The authors would like to thank the anonymous referees for their valuable comments.

References

[1] A. V. AHO, J. E. HOPCROFT, AND J. ULLMAN: *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Boston, MA, 1974. 179, 180, 181

[2] T. M. CHAN: More algorithms for all-pairs shortest paths in weighted graphs. In *Proc. 39th STOC*, pp. 590–598. ACM Press, 2007. [STOC:1250790.1250877]. 174, 183

[3] T. M. CHAN: All-pairs shortest paths with real weights in $O(n^3/\log n)$ time. *Algorithmica*, 50(2):236–243, 2008. [Algorithmica:px2741688g4p4l18]. 176

[4] D. COPPERSMITH: Rectangular matrix multiplication revisited. *J. Complexity*, 13(1):42–49, 1997. [doi:10.1006/jcom.1997.0438]. 175, 178, 186

[5] D. COPPERSMITH AND S. WINOGRAD: Matrix multiplication via arithmetic progressions. *J. Symbolic Comput.*, 9(3):251–280, 1990. [doi:10.1016/S0747-7171(08)80013-2]. 175, 181

[6] R. DUAN AND S. PETTIE: Fast algorithms for $(\max,\min)$-matrix multiplication and bottleneck shortest paths. In *Proc. 19th SODA*, pp. 384–391. ACM Press, 2009. [SODA:1496770.1496813]. 186

[7] D. DUBOIS AND H. PRADE: *Fuzzy Sets and Systems: Theory and Applications*. Academic Press, 1980. 175

[8] M. J. FISCHER AND A. R. MEYER: Boolean matrix multiplication and transitive closure. In *Proc. 12th FOCS*, pp. 129–131. IEEE Comp. Soc. Press, 1971. 179, 180, 183

[9] M. L. FREDMAN AND R. E. TARJAN: Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987. [JACM:28869.28874]. 174

[10] M. E. FURMAN: Applications of a method of fast multiplication of matrices in the problem of finding the transitive closure of a graph. *Dokl. Akad. Nauk SSSR (in Russian)*, 194:524, 1970. 179, 180

[11] M. E. FURMAN: Applications of a method of fast multiplication of matrices in the problem of finding the transitive closure of a graph. *Soviet Math. Dokl. (in English)*, 11(5):1252, 1970. 179

[12] Z. GALIL AND O. MARGALIT: All pairs shortest paths for graphs with small integer length edges. *J. Comput. System Sci.*, 54:243–254, 1997. [JCSS:10.1006/jcss.1997.1385]. 174, 183

[13] T. C. HU: The maximum capacity route problem. *Oper. Res.*, 9(6):898–900, 1961. 174

[14] X. HUANG AND V. Y. PAN: Fast rectangular matrix multiplication and applications. *J. Complexity*, 14(2):257–299, 1998. [doi:10.1006/jcom.1998.0476]. 175, 178, 186

[15] D. S. JOHNSON: Approximation algorithms for combinatorial problems. *J. Comput. System Sci.*, 9:256–278, 1974. [JCSS:10.1016/S0022-0000(74)80044-9]. 183

[16] D. KARGER, D. KOLLER, AND S. PHILLIPS: Finding the hidden path: Time bounds for all-pairs shortest paths. *SIAM J. Comput.*, 22(6):1199–1217, 1993. [SICOMP:10.1137/0222071]. 175

[17] M. KOWALUK AND A. LINGAS: LCA queries in directed acyclic graphs. In *Proc. 32nd Int. Colloq. Autom Lang. Program. (ICALP)*, volume 3580, pp. 241–248. Springer-Verlag, 2005. [ICALP:252dpgap58fdhchf]. 180

[18] L. LOVÁSZ: On the ratio of optimal integral and fractional covers. *Discrete Math.*, 13:383–390, 1975. [doi:10.1016/0012-365X(75)90058-8]. 183

[19] J. MATOUŠEK: Computing dominances in E^n. *Inform. Process. Lett.*, 38(5):277–278, 1991. [IPL:10.1016/0020-0190(91)90071-O]. 175, 176

[20] J. I. MUNRO: Efficient determination of the transitive closure of a directed graph. *Inform. Process. Lett.*, 1(2):56–58, 1971. [IPL:10.1016/0020-0190(71)90006-8]. 179, 180

[21] M. POLLACK: The maximum capacity through a network. *Oper. Res.*, 8(5):733–736, 1960. 174

[22] R. SEIDEL: On the all-pairs-shortest-path problem in unweighted undirected graphs. *J. Comput. System Sci.*, 51:400–403, 1995. [JCSS:10.1006/jcss.1995.1078]. 174

[23] A. SHAPIRA, R. YUSTER, AND U. ZWICK: All-pairs bottleneck paths in vertex weighted graphs. In *Proc. 17th SODA*, pp. 978–985. ACM Press, 2007. [SODA:1283383.1283488]. 174, 175

[24] A. SHOSHAN AND U. ZWICK: All pairs shortest paths in undirected graphs with integer weights. In *Proc. 40th FOCS*, pp. 605–614. IEEE Comp. Soc. Press, 1999. [FOCS:10.1109/SFFCS.1999.814635]. 174

[25] S. K. STEIN: Two combinatorial covering theorems. *J. Combin. Theory Ser. A*, 16:391–397, 1974. [JCombThA:10.1016/0097-3165(74)90062-4]. 183

[26] C. R. SUBRAMANIAN: A generalization of Janson inequalities and its application to finding shortest paths. In *Proc. 10th SODA*, pp. 795–804. ACM Press, 1999. [SODA:314500.314917]. 175

[27] V. VASSILEVSKA: Nondecreasing paths in weighted graphs, or: How to optimally read a train schedule. In *Proc. 18th SODA*, pp. 465–472. ACM Press, 2008. [SODA:1347082.1347133]. 183

[28] V. VASSILEVSKA AND R. WILLIAMS: Finding a maximum weight triangle in $n^{3-\delta}$ time, with applications. In *Proc. 38th STOC*, pp. 225–231. ACM Press, 2006. [STOC:1132516.1132550]. 176, 186

[29] V. VASSILEVSKA, R. WILLIAMS, AND R. YUSTER: All-pairs bottleneck paths for general graphs in truly sub-cubic time. In *Proc. 39th STOC*, pp. 585–589. ACM Press, 2007. [STOC:1250790.1250876]. 173

[30] U. ZWICK: All pairs shortest paths using bridging sets and rectangular matrix multiplication. *J. ACM*, 49(3):289–317, 2002. [JACM:567112.567114]. 174, 181, 182, 183, 184

AUTHORS

Virginia Vassilevska
School of Mathematics
Institute for Advanced Study[1]
virgi@math.ias.edu
http://www.math.ias.edu/~virgi

[1] At the time of submission, this author was at Carnegie Mellon University.

Ryan Williams
School of Mathematics
Institute for Advanced Study[2]
ryanw@math.ias.edu
http://www.math.ias.edu/~ryanw

Raphael Yuster
Department of Mathematics
University of Haifa
raphy@math.haifa.ac.il
http://research.haifa.ac.il/~raphy/

ABOUT THE AUTHORS

VIRGINIA VASSILEVSKA obtained her Ph. D. in 2008 from the computer science department of Carnegie Mellon University under the supervision of Prof. Guy Blelloch. She is currently a postdoctoral fellow at the Institute for Advanced Study. Besides working on various theoretical problems, she enjoys some more practical exercises such as tennis.

RYAN WILLIAMS got his Ph. D. in computer science from Carnegie Mellon University in 2007. His advisor was Prof. Manuel Blum. Ryan is currently a postdoctoral fellow at the Institute for Advanced Study. In his spare time he enjoys having coffee and watching Auburn football.

RAPHAEL YUSTER is a professor at the mathematics department of the University of Haifa. His research interests include combinatorics, graph theory, algorithms, and probabilistic methods in combinatorics.

[2]At the time of submission, this author was at Carnegie Mellon University.