Fast Algorithms for (max, min)-Matrix Multiplication and Bottleneck Shortest Paths*

Ran Duan
University of Michigan

Seth Pettie
University of Michigan

Abstract

Given a directed graph with a capacity on each edge, the *all-pairs bottleneck paths* (APBP) problem is to determine, for all vertices s and t, the maximum flow that can be routed from s to t. For dense graphs this problem is equivalent to that of computing the $(\max, \min)$-transitive closure of a real-valued matrix. In this paper, we give a $(\max, \min)$-matrix multiplication algorithm running in time $O(n^{(3+\omega)/2}) \leq O(n^{2.688})$, where ω is the exponent of binary matrix multiplication. Our algorithm improves on a recent $O(n^{2+\omega/3}) \leq O(n^{2.792})$-time algorithm of Vassilevska, Williams, and Yuster. Although our algorithm is slower than the best APBP algorithm on *vertex* capacitated graphs, running in $O(n^{2.575})$ time, it is just as efficient as the best algorithm for computing the *dominance product*, a problem closely related to $(\max, \min)$-matrix multiplication.

Our techniques can be extended to give subcubic algorithms for related bottleneck problems. The *all-pairs bottleneck shortest paths* problem (APBSP) asks for the maximum flow that can be routed along a *shortest* path. We give an APBSP algorithm for edge-capacitated graphs running in $O(n^{(3+\omega)/2})$ time and a slightly faster $O(n^{2.657})$-time algorithm for vertex-capactitated graphs. The second algorithm significantly improves on an $O(n^{2.859})$-time APBSP algorithm of Shapira, Yuster, and Zwick. Our APBSP algorithms make use of new hybrid products we call the distance-max-min product and dominance-distance product.

1 Introduction

It is well known that many network optimization problems can be solved by reducing the problem to a series of fast matrix multiplications over a ring [5, 4, 11]. Among such problems are finding maximum weight matchings [13, 14, 15, 10], finding least common ancestors in DAGs [6], finding maximum weight subgraphs isomorphic to a fixed graph [22, 7], and computing shortest paths in weighted and unweighted graphs [2, 8, 9, 18, 16, 19, 24]. Generally speaking, algorithms based on matrix multiplication have a difficult time with *weighted* instances. They may not work at all for real-weighted graphs and, even for integer-weighted graphs, their running times often scale linearity with the maximum edge weight.

In recent years, however, researchers have successfully found "truly" subcubic time algorithms for some real-weighted optimization problems that improve on the best combinatorial algorithms for moderately dense graphs. Vassilevska and Williams [22] showed that it is possible to find a maximum weight triangle in a vertex-weighted graph in $O(n^{(3+\omega)/2})$ time; their algorithm has since been improved to $O(n^{\omega})$ [7] and generalized to finding larger induced subgraphs, not necessarily triangles. Here ω is the exponent of square matrix multiplication, which is known to be less than 2.376 [5].[1] Chan [3] has shown that various shortest path problems can be solved in sub-cubic time if the vertices are points in d-dimension space and the edge weights some well-behaved function of the vertex coordinates. As a special case, Chan showed that shortest paths in real *vertex*-weighted graphs can be solved in $O(n^{2.844})$ time.

Very recently Shapira et al. [17] and Vassilevska et al. [23] considered the *all pairs bottleneck paths* problem (APBSP, also known as the maximum capacity paths problem) in graphs with real *capacities* assigned to edges/vertices. It is shown that APBP can be computed in $O(n^{2+\mu}) = O(n^{2.575})$ time on vertex capacitated-graphs [17] and $O(n^{2+\omega/3}) = O(n^{2.792})$ time on edge capacitated graphs [23]. (Here $\mu \geq 1/2$ is a constant related to rectangular matrix multiplication.) Shapira et al. also considered a variation called all-pairs bottleneck *shortest* paths (APBSP), where one asks for the maximum capacity path among shortest paths. Their APBSP algorithm runs in $O(n^{(8+\mu)/3}) = O(n^{2.859})$ time. An unpublished algorithm of Vassilevska [20] computes APBSP on edge-capacitated graphs in $O(n^{(15+\omega)/6}) = O(n^{2.896})$ time.

*Email: {duanran, pettie}@umich.edu. This work was supported by NSF CAREER grant no. CCF-0746673.

[1] As in most other papers that use matrix multiplication, we abusively use the notation $O(n^x)$ to mean $O(n^{x+\epsilon})$ for all $\epsilon > 0$.

Our Results. In this paper we develop faster algorithms for $(\max, \min)$-product, APBP in edge-capacitated graphs, and all-pairs bottleneck shortest paths in both vertex and edge-capacitated graphs. We introduce a simple technique called *row balancing* (or *column balancing*) that decomposes a matrix into a sparse component and a dense component with uniform row (or column) density. Using this technique we exhibit an extremely simple algorithm for computing the *dominance product* on sufficiently sparse matrices in $O(n^\omega)$ time, as well as an algorithm for somewhat denser matrices that runs in time $O(\sqrt{mm'}n^{(\omega-1)/2})$. (This last bound was claimed earlier in [23]; it was based on a more complicated algorithm [21].) Using the sparse dominance product and row balancing we show how to compute the $(\max, \min)$-product (and, therefore, APBP) in $O(n^{(3+\omega)/2})$ time. This improves on the previous $O(n^{2+\omega/3})$ time algorithm [23]. As a stepping stone to our APBSP algorithms we develop fast algorithms for computing hybrid products that operate on *pairs* of matrices. In particular, we show how to compute the distance product ($(\min, +)$-product) of a pair of matrices under a dominance constraint on a second pair matrices, and, in a similar way, the $(\max, \min)$-product under a distance constraint. These hybrid products allow us to compute APBSP in $O(n^{(3+\omega)/2})$ time on edge-capacitated graphs and $O(n^{2.657})$ time on vertex-capacitated graphs, which are significant improvements over [20, 17], which run in $O(n^{(15+\omega)/6})$ and $O(n^{(8+\mu)/3})$ time, respectively.

Organization. In Section 3 we present our new algorithms for sparse dominance products and $(\max, \min)$-products, which leads directly to a faster APBP algorithm. In Section 4 we define new products called dominance-distance and distance-max-min, both of which operate on pairs of matrices. In Sections 4.3 and 4.4 we show how to compute APBSP in edge- and vertex-capacitated graphs using the distance-max-min product.

2 Definitions

In our paper, we assume w.l.o.g. that the capacities for edges or vertices are real numbers with the additional minimum and maximum elements $-\infty$ and ∞.

2.1 Row-Balancing and Column-Balancing

Most algorithms in this paper will use the concept of row-balancing (and column-balancing) for sparse matrices, in which we partition the dense rows into parts and reposition each part in a distinct row.

DEFINITION 2.1. *Let A be an $n \times p$ matrix with m finite elements. Depending on context, the other elements will either all be ∞ or all be $-\infty$. We assume the former below. The* row-balancing *of A, or $\mathbf{rb}(A)$, is a pair (A', A'') of $n \times p$ matrices, each with at most $k = \lceil m/n \rceil$ elements in each row. The row-balancing is obtained by the following procedure: First, sort all the finite elements in the ith row of A in increasing order, and divide this list into several parts $T_i^1, T_i^2, \ldots T_i^{a_i}$ such that all parts except the last one contain k elements and the last part ($T_i^{a_i}$) contains at most k elements. Let A' be the submatrix of A containing the last parts:*

$$A'[i,j] = \begin{cases} A[i,j] & \text{if } A[i,j] \in T^{a_i} \\ \infty & \text{otherwise} \end{cases}$$

Since the remaining parts have exact k elements, there can be at most $m/k \leq n$ of them. We assign each part to a distinct row in A'', i.e., we choose an arbitrary mapping $\rho : [n] \times [p/k] \to [n]$ such that $\rho(i,q) = i'$ if T_i^q is assigned row i'; it is undefined if T_i^q doesn't exist. Let A'' be defined as:

$$A''[i', j] = \begin{cases} A[i,j] & \text{if } \rho^{-1}(i') = (i,q) \text{ and } (i,j) \in T_i^q \\ \infty & \text{otherwise} \end{cases}$$

Thus, every finite $A[i,j]$ in A has a corresponding element in either A' or A'', which is also in the jth column. The column-balancing *of A, or $\mathbf{cb}(A)$, is similarly defined as (A'^T, A''^T), where $(A', A'') = \mathbf{rb}(A^T)$.*

2.2 Matrix Products

We use $\cdot$ to denote the standard $(+, \cdot)$-product on matrices and let $\circledcirc, \circledvee$, and $\star$ be the *dominance*, *max-min*, and *distance* products.

DEFINITION 2.2. **(Various Products)** *Let A and B be real-valued matrices. The products $\cdot, \circledcirc, \circledvee$, and $\star$ are defined as*

$$\begin{aligned} (A \cdot B)[i,j] &= \sum_k (A[i,k] \cdot B[k,j]) \\ (A \circledcirc B)[i,j] &= |\{k \mid A[i,k] \leq B[k,j]\}| \\ (A \circledvee B)[i,j] &= \max_k \min\{A[i,k], B[k,j]\} \\ (A \star B)[i,j] &= \min_k \{A[i,k] + B[k,j]\} \end{aligned}$$

In Section 4.2 we introduce hybrids of these called the dominance-distance and distance-max-min products.

3 Dominance and APBP

Matoušek [12] showed that the dominance product of two $n \times n$ matrices can be computed in $O(n^{(3+\omega)/2})$ time. However, in our algorithms we need the dominance product only for relatively sparse matrices. Theorem 3.1 shows that $A \circledcirc B$ can be computed in $O(n^\omega)$ time

when the number of finite elements is $O(n^{(\omega+1)/2})$. The algorithm behind this theorem is used directly in our APBP and APBSP algorithms. Using Theorem 3.1 as a subroutine we give a faster dominance product algorithm for somewhat denser matrices; however, these improvements have no implications for APBP or related problems. Theorem 3.2 was originally claimed by Vassilevska et al. [23]. Their algorithm, which does not appear in [23], is a bit more involved.

THEOREM 3.1. **(Sparse Dominance Product)** *Let A and B be two $n \times n$ matrices where the number of non-(∞) values in A is m_1 and the number of non-$(-\infty)$ values in B is m_2. Then $A \circledcirc B$ can be computed in time $O(m_1 m_2/n + n^\omega)$.*

Proof. Let $(A', A'') = \mathbf{cb}(A)$ be the column-balancing of A. We build two Boolean matrices $\hat{A}$ and $\hat{B}$ and compute $\hat{A} \cdot \hat{B}$ in $O(n^\omega)$ time.

$$\hat{A}[i,k] = 1 \quad \text{if } A''[i,k] \neq \infty$$
$$\hat{B}[k,j] = 1 \quad \text{if } B[k',j] \geq \max T_{k'}^{q'},\ (k',q') = \rho^{-1}(k)$$

One may verify that $\hat{A}[i,k] \cdot \hat{B}[k,j] = 1$ if and only if $B[k',j]$ is greater or equal to *all* the elements in the kth column of A'', which is the q'th part in k'th column of A, where $q' < a_{k'}$ is not the last part of column k'. What $(\hat{A} \cdot \hat{B})[i,j]$ does not count are dominances $A[i,k] \leq B[k,j]$, where either $A[i,k] \in T_k^q$ but $B[k,j]$ dominates some but not all elements in T_k^q, or $A[i,k] \in T_k^{a_k}$ (the last part of column k) and $B[k,j]$ does dominate all of $T_k^{a_k}$. We check these possibilities in $O(m_1 m_2/n)$ time. Each of the m_2 elements in B is compared against at most $\lceil m_1/n \rceil$ elements from A.

Using the procedure from Theorem 3.1 as a subroutine, we can compute $A \circledcirc B$ faster for denser matrices. The resulting algorithm is somewhat simpler than that of Vassilevska et al. [23].

THEOREM 3.2. **(Dense Dominance Product)** *Let A and B be two $n \times n$ matrices where m_1 is the number of non-(∞) elements in A and m_2 the number of non-$(-\infty)$ elements in B, where $m_1 m_2 \geq n^{1+\omega}$. Then $A \circledcirc B$ can be computed in time $O(\sqrt{m_1 m_2} n^{(\omega-1)/2})$.*

Proof. Let L be the sorted list of all the finite elements in A. We divide L into t parts $L_1, L_2, ..., L_t$, for a t to be determined, so each part has at most $\lceil m_1/t \rceil$ elements. Then we build Boolean matrices $\hat{A}_p, \hat{B}_p, A_p$, and B_p, for $1 \leq p \leq t$ as follows:

$$\hat{A}_p[i,k] = 1 \qquad \text{if } A[i,k] \in L_p$$
$$\hat{B}_p[k,j] = 1 \qquad \text{if } B[k,j] \geq \max L_p$$

$$A_p[i,k] = \begin{cases} A[i,k] & \text{if } A[i,k] \in L_p \\ \infty & \text{otherwise} \end{cases}$$
$$B_p[k,j] = \begin{cases} B[k,j] & \text{if } \min L_p \leq B[k,j] < \max L_p \\ -\infty & \text{otherwise} \end{cases}$$

Notice that every finite element of B is in at most one B_p. One may verify that

$$A \circledcirc B = \sum_{p=1}^{t} (\hat{A}_p \cdot \hat{B}_p + A_p \circledcirc B_p)$$

From Theorem 3.1, the computation of $A_p \circledcirc B_p$ takes time $O((m_1/t)|B_p|/n + n^\omega)$, where $|B_p|$ is the number of finite elements in B_p. Thus, the total time to compute $A \circledcirc B$ is $O(m_1 m_2/tn + tn^\omega)$. The theorem follows by setting $t = \sqrt{m_1 m_2}/n^{(1+\omega)/2}$.

3.1 Max-Min Product In this section we give an efficient algorithm for solving the max-min product of two matrices that uses the sparse dominance product as a key subroutine. One corollary is that all-pairs bottleneck capacities can be found in the same time bound [1]. By incurring an additional $\log n$ factor, we can find all-pairs bottleneck *paths* using existing techniques [24, 23]; see Appendix A for a review.

THEOREM 3.3. **(Max-Min Product)** *Given two real $n \times n$ matrices A and B, $A \otimes\!\!\!\!\bigcirc\ B$ can be computed in $O(n^{(3+\omega)/2}) \leq O(n^{2.688})$ time.*

Proof. It suffices to compute matrices C and C':

$$C[i,j] = \max_k \{A[i,k] \mid A[i,k] \leq B[k,j]\}$$
$$C'[i,j] = \max_k \{B[k,j] \mid A[i,k] \geq B[k,j]\}$$

since $(A \otimes\!\!\!\!\bigcirc\ B)[i,j] = \max\{C[i,j], C'[i,j]\}$. Below we compute C; the procedure for C' is obviously symmetric.

Let L be the sorted list (in increasing order) of all the elements in A and B. We evenly divide L into t parts $L_1, L_2, ..., L_t$, so each part has at most $\lceil 2n^2/t \rceil$ elements. Let A_r and B_r be the submatrices of A and B containing L_r:

$$A_r[i,j] = \begin{cases} A[i,j] & \text{if } A[i,j] \in L_r \\ \infty & \text{otherwise} \end{cases}$$
$$B_r[i,j] = \begin{cases} B[i,j] & \text{if } B[i,j] \in L_r \\ -\infty & \text{otherwise} \end{cases}$$

Let $(A'_r, A''_r) = \mathbf{rb}(A_r)$ be the row-balancing of A_r. After we compute $A_r \circledcirc B$, $A'_r \circledcirc B$, and $A''_r \circledcirc B$, for all r, we may determine $C[i,j]$ as follows:

1. Find the largest r such that $(A_r \circledcirc B)[i,j] > 0$. Thus, $C[i,j]$ must be in A_r.

2. Check whether $(A'_r \circledcirc B)[i,j] > 0$. If it is, since A'_r contains the largest part of each row in A_r, $C[i,j]$ must be in the ith row of A'_r. It follows that $C[i,j] = \max_k\{A'_r[i,k] \mid A'_r[i,k] \leq B[k,j]\}$.

3. If $(A'_r \circledcirc B)[i,j] = 0$, find the largest q such that $(A''_r \circledcirc B)[\rho(i,q),j] > 0$. It follows that $C[i,j] \in T_i^q$. We determine $C[i,j]$ be checking each element of T_i^q one by one.

Steps 1–3 take $O(n/t)$ time per element, for a total of $O(n^3/t)$ time. To compute $A_r \circledcirc B$ we begin by building two Boolean matrices $\hat{A}_r$ and $\hat{B}_r$ for all r such that:

$$\hat{A}_r[i,k] = 1 \qquad \text{if } A[i,k] \in L_r$$
$$\hat{B}_r[k,j] = 1 \qquad \text{if } B[k,j] \in L_{r+1} \cup \cdots \cup L_t$$

It is straightforward to see that $A_r \circledcirc B = A_r \circledcirc B_r + \hat{A}_r \cdot \hat{B}_r$: the inter-part comparisons are covered in $\hat{A}_r \cdot \hat{B}_r$ and the intra-part comparisons in $A_r \circledcirc B_r$. The products $A'_r \circledcirc B$ and $A''_r \circledcirc B$ can be computed in a similar fashion.

By Theorem 3.1 the time to compute $A_r \circledcirc B$, $A'_r \circledcirc B$, and $A''_r \circledcirc B$, for all r, is $t \cdot O(n^3/t^2 + n^\omega)$. In total the running time is $O(n^3/t + tn^\omega)$. The theorem follows by setting $t = n^{(3-\omega)/2}$.

Theorem 3.3 leads immediately to an algorithm computing all-pairs bottleneck *capacities* in $O(n^{(3+\omega)/2})$ time. We review in Appendix A an existing algorithm [24, 23] for finding explicit bottleneck *paths*.

COROLLARY 3.1. *APBP can be computed in $O(n^{(3+\omega)/2})$ time.*

4 Bottleneck Shortest Paths

In this section, we consider the All-Pairs Bottleneck Shortest Paths problem (APBSP) in both edge- and vertex-capacitated graphs. Let $D(u,v)$ be the unweighted distance from u to v and let $sc(u,v)$ be the maximum capacity path from u to v with length $D(u,v)$.

When the graph is *edge*-capacitated we give an APBSP algorithm running in $\tilde{O}(n^{(3+\omega)/2})$ time, matching the running time of our APBP algorithm. This is the first published subcubic APBSP algorithm for edge-capacitated graphs. It improves on an unpublished algorithm of Vassilevska [20], which runs in $O(n^{(15+\omega)/6}) = O(n^{2.896})$ time. For vertex-capacitated graphs our algorithm runs slightly faster, in $O(n^{2.657})$ time; this improves on a recent algorithm of Shapira et al. [17] running in $O(n^{(8+\mu)/3}) = O(n^{2.859})$ time.

In Section 4.1 we review some facts about rectangular matrix multiplication. In Section 4.2 we present fast algorithms for certain *hybrid* products based on dominance, distance, and max-min products. In Sections 4.3 and 4.4 we present our APBSP algorithms for edge- and vertex-capacitated graphs.

4.1 Rectangular Matrix Multiplication

In our algorithms we often use fast rectangular matrix multiplication algorithms [4, 11]. Let $\omega(r,s,t)$ to be the constant such that multiplying $n^r \times n^s$ and $n^s \times n^t$ matrices takes $O(n^{\omega(r,s,t)})$ time. We use the standard definitions of the constants α, β, and μ.

DEFINITION 4.1. *Let α be the maximum value satisfying $\omega(1,\alpha,1) = 2$ and let $\beta = \frac{\omega-2}{1-\alpha}$. Define μ to be the constant satisfying $\omega(1,\mu,1) = 1 + 2\mu$. If $\omega = 2$ then $\alpha = 1, \beta = 0$, and $\mu = 1/2$.*

Then following bounds on α, β, and μ can be found in [4, 11]:

LEMMA 4.1. *$\alpha > 0.294$, $\beta > 0.533$, and $\mu < 0.575$. For $s \geq \alpha$, $\omega(1,s,1) \leq 2 + \beta(s-\alpha)$.*

4.2 Hybrid Products

Our all-pairs bottleneck shortest path algorithms use products that are hybrids of dominance, distance, and max-min products.

DEFINITION 4.2. **(Dominance-Distance)** *Let $(A,\tilde{A})$ and $(B,\tilde{B})$ be pairs of real matrices. Their* dominance-distance *product is written $C = (A,\tilde{A}) \overset{\circledcirc}{\star} (B,\tilde{B})$, where*

$$C[i,j] = \min_{\substack{k: \\ \tilde{A}[i,k] \leq \tilde{B}[k,j]}} (A[i,k] + B[k,j])$$

In a similar fastion we define the distance-max-min product as a hybrid of distance and max-min.

DEFINITION 4.3. **(Distance-Max-Min)** *Let $(A,\tilde{A})$ and $(B,\tilde{B})$ be pairs of real matrices. Their* distance-max-min *product is defined as:*

$$\begin{aligned}(C,\tilde{C}) &= (A,\tilde{A}) \overset{\star}{\circledvee} (B,\tilde{B}) \\ &\textit{where} \\ C &= A \star B \\ \tilde{C}[i,j] &= \max_{\substack{k: \\ A[i,k]+B[k,j]=C[i,j]}} \min\{\tilde{A}[i,k], \tilde{B}[k,j]\}\end{aligned}$$

Our algorithms make use of Zwick's algorithm [24] for distance products in integer-weighted matrices.

THEOREM 4.1. **(Distance Product)** *[24] Let A and B be $n \times n^s$ and $n^s \times n$ matrices, respectively, whose elements are in $\{1,\ldots,M\}$. Then $A \star B$ can be computed in $O(\min\{n^{2+s}, Mn^{\omega(1,s,1)}\})$ time.*

THEOREM 4.2. **(Dominance-Distance Product)** *Let $A, \tilde{A}, B, \tilde{B}$ be matrices such that:*

$$A \in \{1, \ldots, M, \infty\}^{n \times n^s} \qquad \tilde{A} \in (\mathbb{R} \cup \{\infty\})^{n \times n^s}$$
$$B \in \{1, \ldots, M, \infty\}^{n^s \times n} \qquad \tilde{B} \in (\mathbb{R} \cup \{-\infty\})^{n^s \times n}$$

where M is an integer and $s \leq 1$. If the number of finite elements in $\tilde{A}$ and $\tilde{B}$ are m_1 and m_2, resp., then $(A, \tilde{A}) \overset{\otimes}{\star} (B, \tilde{B})$ can be computed in $O(m_1 m_2 / n^s + M n^{\omega(1,s,1)})$ time.

Proof. Let $(\tilde{A}', \tilde{A}'') = \mathbf{cb}(\tilde{A})$ be the column-balancing of $\tilde{A}$. We build two matrices $\hat{A}$ and $\hat{B}$, defined below. Here $(k', q') = \rho^{-1}(k)$.

$$\hat{A}[i,k] = \begin{cases} A[i,k'] & \text{if } \tilde{A}''[i,k] \neq \infty \\ \infty & \text{otherwise} \end{cases}$$

$$\hat{B}[k,j] = \begin{cases} B[k',j] & \text{if } \tilde{B}[k',j] \geq \max T_{k'}^{q'} \\ \infty & \text{otherwise} \end{cases}$$

In other words, $(\hat{A} \star \hat{B})[i,j]$ is the minimum $A[i,k'] + B[k',j]$ such that $\tilde{B}[k',j]$ dominates *all* of $T_{k'}^{q'}$, the part containing $A[i,k']$. Furthermore, $q' < a_{k'}$, i.e., $T_{k'}^{q'}$ is not the last part that appears in $\tilde{A}'$. What we must consider now are sums $A[i,k] + B[k,j]$ which could be smaller than $(\hat{A} \star \hat{B})[i,j]$. If $\tilde{A}[i,k] \in T_k^q$ then we must examine $\tilde{B}[k,j]$ if it dominates some, but not all, elements of T_k^q, or if $q = a_k$ and $\tilde{B}[k,j]$ dominates all of $T_k^{a_k}$. Each of the m_2 elements of $\tilde{B}$ participates in $\lceil m_1/n^s \rceil$ such sums, requiring $O(m_1 m_2 / n^s)$ time. The product $\hat{A} \star \hat{B}$ is computed in $O(M n^{\omega(1,s,1)})$ time.

Just as the max-min product may be applied directly to compute APBP, the distance-max-min product will be useful in computing APBSP on both edge- and vertex-capacitated graphs.

THEOREM 4.3. **(Distance-Max-Min Product)** *Let $A, \tilde{A}, B, \tilde{B}$ be matrices such that:*

$$A \in \{1, \ldots, M, \infty\}^{n \times n^s} \qquad \tilde{A} \in \mathbb{R}^{n \times n^s}$$
$$B \in \{1, \ldots, M, \infty\}^{n^s \times n} \qquad \tilde{B} \in \mathbb{R}^{n^s \times n}$$

where M is an integer and $s \leq 1$. Then $(A, \tilde{A}) \overset{\star}{\otimes} (B, \tilde{B})$ can be computed in $O(\min\{n^{2+s}, M^{1/2} \cdot n^{1+s/2+\omega(1,s,1)/2}\})$ time.

Proof. Recall that $(C, \tilde{C}) = (A, \tilde{A}) \overset{\star}{\otimes} (B, \tilde{B})$, where $C = A \star B$ and $\tilde{C}[i,j] = \max_k \min\{\tilde{A}[i,k], \tilde{B}[k,j]\}$ such that $A[i,k] + B[k,j] = C[i,j]$. (Note that we could always compute C and $\tilde{C}$ by the trivial algorithm in $O(n^{2+s})$ time.) We begin by computing C in $O(M n^{\omega(1,s,1)})$ time with Zwick's algorithm [24], then compute matrices $\tilde{C}_1, \tilde{C}_2$:

$$\tilde{C}_1[i,j] = \max_k \{\tilde{A}[i,k] \mid \tilde{A}[i,k] \leq \tilde{B}[k,j] \text{ and } A[i,k] + B[k,j] = C[i,j]\}$$
$$\tilde{C}_2[i,j] = \max_k \{\tilde{B}[k,j] \mid \tilde{A}[i,k] \geq \tilde{B}[k,j] \text{ and } A[i,k] + B[k,j] = C[i,j]\}$$

One can verify that $\tilde{C}[i,j] = \max\{\tilde{C}_1[i,j], \tilde{C}_2[i,j]\}$. Below we describe how to compute $\tilde{C}_1$; computing $\tilde{C}_2$ is symmetric.

Let L be the sorted list of all the elements in $\tilde{A}$ and $\tilde{B}$. We divide L into t parts, $L_1, L_2, ..., L_t$, so each part has $2n^{1+s}/t$ elements. Define the matrices A_r and B_r, for $1 \leq r \leq t$, as:

$$\tilde{A}_r[i,j] = \begin{cases} \tilde{A}[i,j] & \text{if } \tilde{A}[i,j] \in L_r \\ \infty & \text{otherwise} \end{cases}$$

$$\tilde{B}_r[i,j] = \begin{cases} \tilde{B}[i,j] & \text{if } \tilde{B}[i,j] \in L_r \\ -\infty & \text{otherwise} \end{cases}$$

Let $(\tilde{A}'_r, \tilde{A}''_r) = \mathbf{rb}(\tilde{A}_r)$ be the row-balancing of $\tilde{A}_r$. We compute the dominance-distance products G_r, G'_r, and G''_r, for $1 \leq r \leq t$, defined as:

$$G_r = (A, \tilde{A}_r) \overset{\otimes}{\star} (B, \tilde{B})$$
$$G'_r = (A, \tilde{A}'_r) \overset{\otimes}{\star} (B, \tilde{B})$$
$$G''_r = (A, \tilde{A}''_r) \overset{\otimes}{\star} (B, \tilde{B})$$

For every pair i, j we determine $\tilde{C}_1[i,j]$ as follows:

1. Find the largest r such that $G_r[i,j] = C[i,j]$, then $\tilde{C}_1[i,j]$ must be in $\tilde{A}_r$
2. Check whether $G'_r[i,j] = C[i,j]$. If it is, $\tilde{C}_1[i,j]$ must be in the ith row of $\tilde{A}'_r$. Check all the finite elements in that row one by one.
3. If $G'_r[i,j] \neq C[i,j]$, find the largest q such that $G''_r[\rho(i,q), j] = C[i,j]$. Thus, $\tilde{C}_1[i,j]$ must be in T_i^q, the qth part of the ith row of $\tilde{A}$. Check the elements in T_i^q one by one.

Steps 1–3 take $O(n^s/t)$ time per pair, that is, $O(n^{2+s}/t)$ time in total. What remains is to show that we can compute G_r, G'_r, G''_r in the stated bounds. To find G_r, we begin by constructing two matrices $\hat{A}_r$ and $\hat{B}_r$ such that:

$$\hat{A}_r[i,k] = \begin{cases} A[i,k] & \text{if } \tilde{A}[i,k] \in L_r \\ \infty & \text{otherwise} \end{cases}$$

$$\hat{B}_r[k,j] = \begin{cases} B[k,j] & \text{if } \tilde{B}[k,j] \in L_{r+1} \cup \cdots \cup L_t \\ \infty & \text{otherwise} \end{cases}$$

We compute $\hat{G}_r = \hat{A}_r \star \hat{B}_r$ using Zwick's algorithm [24] and $\tilde{G}_r = (A, \tilde{A}_r) \overset{\otimes}{\star} (B, \tilde{B}_r)$ using the algorithm from Theorem 4.2. One may verify that:

$$G_r[i,j] = \min\{\hat{G}_r[i,j], \tilde{G}_r[i,j]\}$$

If $G_r[i,j] = A[i,k] + B[k,j]$, the $\hat{G}_r$ matrix covers the case where $A[i,k]$ and $B[k,j]$ come from different parts and $\tilde{G}_r$ covers the case where they are both in part L_r. The matrices G'_r and G''_r are computed in a similar fashion.

In total, the time required to find G_r, G'_r, and G''_r, for $1 \leq r \leq t$, is $t \cdot O(Mn^{\omega(1,s,1)} + n^{2+s}/t^2)$, where the first term comes from [24] and the second from Theorem 4.2. (Recall that $\tilde{A}_r$ and $\tilde{B}_r$ have at most $2n^{1+s}/t$ finite elements.) We choose t to be $n^{1+s/2-\omega(1,s,1)/2}M^{-1/2}$, which makes the overall running time $O(M^{1/2} \cdot n^{1+s/2+\omega(1,s,1)/2})$.

4.3 APBSP with Edge Capacities For the APBSP problem, we use the "bridging sets" technique; see Zwick [24] and Shapira et al. [17]. A standard probabilistic argument shows that a small set of randomly selected vertices will cover a set of relatively long paths.

LEMMA 4.2. *[24] Let S be a set of paths between distinct pairs of vertices, each of length at least t, in a graph with n vertices. A set of $O(t^{-1}n\log n)$ vertices selected uniformly at random contains, with probability $1 - n^{-\Omega(1)}$, at least one vertex from each path in S. Such a set is called a t-bridging set. A t-bridging set can be found deterministically in $O(tn^2)$-time.*

THEOREM 4.4. *Given a real edge-capacitated graph on n vertices, APBSP can be computed in $O(n^{(3+\omega)/2}) = O(n^{2.688})$ time.*

Proof. We begin by computing unweighted distances in $O(n^{2+\mu}) = O(n^{2.575})$ time [24]. Let D and C be the distance and edge capacity matrices, respectively. For vertices at distance 1 or 2 it follows that:

$$sc(u,v) = \begin{cases} C[u,v] & \text{if } D[u,v] = 1 \\ (C \circledcirc C)[u,v] & \text{if } D[u,v] = 2 \end{cases}$$

In general, once $sc(u,v)$ is computed for u, v with $D[u,v] \leq t$, it can be computed for all u, v with $D[u,v] \leq 3t/2$ as follows. Let B be a bridging set for the set of bottleneck shortest paths with length $t/2$. We compute such a set if $t \leq \sqrt{n}$ and, if not, use the last bridging set when t was at most $\sqrt{n}$. Thus, $|B| = \tilde{O}(\max\{n/t, \sqrt{n}\})$. If $D[u,v]$ is between t and $3t/2$ there must be some vertex $b \in B$ that lies on the middle third of the bottleneck shortest path from u to v and, therefore, satisfies $D[u,b], D[b,v] \leq t$. In other words, $sc(u,v)$ can be derived from $sc(u,b)$ and $sc(b,v)$, both of which have already been computed. We have:

$$sc(u,v) = \max_{\substack{b \in B \\ D[u,b]+D[b,v]=D[u,v]}} \min\{sc(u,b), sc(b,v)\}$$

This is clearly an instance of the distance-max-min product of $n \times |B|$ and $|B| \times n$ matrices. If $B = \tilde{O}(\sqrt{n})$ we use the trivial $O(n^{2.5})$-time algorithm. Otherwise, let $n^s = |B| = \tilde{O}(n/t)$. By Theorem 4.3 this product can be computed in time:

$$\begin{aligned} & O(t^{\frac{1}{2}} \cdot n^{1+\frac{s+\omega(1,s,1)}{2}}) \\ &= O(n^{\frac{3+\omega(1,s,1)}{2}}) && t = n^{1-s} \\ &= O(n^{\frac{5+\beta(s-\alpha)}{2}}) && \omega(1,s,1) \leq 2 + \beta(s-\alpha) \\ &= O(n^{(3+\omega)/2}) && s \leq 1, \beta(1-\alpha) = \omega - 2 \end{aligned}$$

By Lemma 4.2 B can be computed in $O(tn^2) = O(n^{5/2}) = O(n^{(3+\omega)/2})$ time. The procedure above is obviously repeated just $\log_{3/2} n$ times, for a total running time of $O(n^{(3+\omega)/2})$.

4.4 APBSP with Vertex Capacities In this section, we consider the APBSP problem for vertex-capacitated graphs. There are two variants of the problem: closed-APBSP, where the endpoints of a path are taken into account, and open-APBSP, where they are not. However, Shapira et al. [17] showed that open-APBSP is reducible to closed-APBSP in $O(n^2)$ time. Thus we only consider the closed-APBSP problem in this paper. The Shapira et al. algorithm runs in time $O(n^{(8+\mu)/3}) \leq O(n^{2.859})$. Here we improve their result by the techniques introduced earlier.

Lemma 4.3 shows how bottleneck shortest paths can be found quickly for relatively close pairs of vertices. The proof borrows extensively from [17].

LEMMA 4.3. *Given a vertex-capacitated graph on n vertices, the bottleneck shortest paths can be computed for all pairs at distance at most n^t, in time $O(n^{(3+\omega+t-3\beta)/(2-\beta)})$.*

Proof. Number the vertices $V = \{v_1, v_2, ..., v_n\}$ in increasing order of capacity. We begin by computing the distance matrix D in $O(n^{2+\mu})$ time [24]. For each $s = 0, ..., n^t$, we compute two $n \times n$ Boolean matrices P_s and Q_s, where $P_s[i,j] = 1$ if and only if there is a path from v_i to v_j, of length at most s, in which v_j has minimum capacity, $Q_s[i,j] = 1$ if and only if there is a path from v_i to v_j, of length at most s, in which v_i has minimum capacity. From [17], the computation will take $O(n^{t+\omega})$ time, as follows. Let E be the adjacency

matrix of G and F be the Boolean matrix satisfying $F[i,j] = 1$ iff $i \geq j$. Then $P_0 = Q_0 = I$, $P_s = EP_{s-1} \wedge F$ and $Q_s = Q_{s-1}E \wedge F^T$.

We define two $n \times n$ matrices A and B:

$$A[i,j] = \begin{cases} D[i,j] & \text{if } P_{D[i,j]}[i,j] = 1 \\ \infty & \text{otherwise} \end{cases}$$

$$B[i,j] = \begin{cases} D[i,j] & \text{if } Q_{D[i,j]}[i,j] = 1 \\ \infty & \text{otherwise} \end{cases}$$

Then we just need to compute the bottleneck capacity matrix C in which:

$$C[i,j] = \min\{k \mid A[i,k] + B[k,j] = D[i,j]\}$$

By the definition of A and B, $A[i,k] = D[i,k]$, $B[k,j] = D[k,j]$, and v_k has the minimum capacity in both paths. Thus $sc(v_i, v_j)$ is just the capacity of $v_{C[i,j]}$.

To compute C, as in [6], partition A into $n \times n^r$ sub-matrices A_p and B into $n^r \times n$ sub-matrices B_p where A_p covers columns $(p-1)n^r + 1$ through pn^r and B_p covers the rows $(p-1)n^r + 1$ through pn^r. Then, for every p, we compute the distance product $C_p = A_p \star B_p$, which will take $O(n^{1-r} \cdot n^{\omega(1,r,1)+t})$ time. For $r > \alpha$ we have $\omega(1,r,1) \leq 2 + \beta(r-\alpha) = \omega - (1-r)\beta$. Thus, the time for this phase is $O(n^{\omega+t+1+\beta(r-1)-r})$

For every i, j, find the smallest p such that $C_p[i,j] = D[i,j]$, i.e., $C[i,j]$ will be in the range $[(p-1)n^r+1, pn^r]$. We check all possibilities one by one. This will take $O(n^{2+r})$ time. To balance the two bounds we choose $r = (\omega + t - 1 - \beta)/(2-\beta)$, making the total running time $O(n^{\frac{\omega+3+t-3\beta}{2-\beta}})$.

THEOREM 4.5. *Given a vertex-capacitated graph with n vertices, APBSP can be computed in $O(n^{\frac{3+\omega}{2} - \frac{\beta^2(3-\omega)}{4+2\beta(2-\beta)}}) = O(n^{2.657})$ time.*

Proof. This algorithm has two phases. In the first phase we use Lemma 4.3 to compute the bottleneck shortest paths for vertices at distance is at most n^t, for some properly selected t. In the second phase, we convert the vertex-capacitated graph to an edge-capacitated graph by giving each edge the minimum capacity of its endpoints. The algorithm from Theorem 4.4 will compute bottleneck shortest paths for the remaining vertex pairs in $O(n^{(3+\omega-\beta t)/2})$ time. To balance the two phases we choose $t = \beta(3-\omega)/(2+\beta(2-\beta))$, making the total running time: $O(n^{\frac{3+\omega}{2} - \frac{\beta^2(3-\omega)}{4+2\beta(2-\beta)}}) = O(n^{2.657})$.

5 Conclusion and Open Problems

We have established new bounds on the complexity of the max-min product and various *hybrid* products that take into account extra constraints, e.g., distance product with a dominance constraint, or max-min product with a distance constraint. As corollaries we established new bounds on the complexity of various all-pairs bottleneck problems in arbitrary directed graphs. The efficiency of our algorithms depended heavily on the simple *row balancing* and *column balancing* operations, which split a matrix into a sparse component and a dense component that has uniform density. This technique is quite basic and should be useful for designing subcubic algorithms for other graph optimization algorithms.

Our algorithms use sparse dominance products as a subroutine but only care to distinguish between two values: zero and non-zero. It remains open whether this simpler *binary* dominance product problem is provably equivalent to computing the general dominance product, or, for that matter, if either of these problems is equivalent to computing max-min products.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, Reading, MA, 1975.

[2] N. Alon, Z. Galil, and O. Margalit. On the exponent of the all pairs shortest path problem. *J. Comput. Syst. Sci.*, 54(2):255–262, 1997.

[3] T. Chan. More algorithms for all-pairs shortest paths in weighted graphs. In *Proc. 39th ACM Symposium on Theory of Computing (STOC)*, pages 590–598, 2007.

[4] D. Coppersmith. Rectangular matrix multiplication revisited. *J. Complex.*, 13(1):42–49, 1997.

[5] D. Coppersmith and T. Winograd. Matrix multiplication via arithmetic progressions. In *Proc. 19th ACM Symp. on the Theory of Computing (STOC)*, pages 1–6, 1987.

[6] A. Czumaj, M. Kowaluk, and A. Lingas. Faster algorithms for finding lowest common ancestors in directed acyclic graphs. *Theoretical Computer Science*, 380(1–2):37–46, 2007.

[7] A. Czumaj and A. Lingas. Finding a heaviest triangle is not harder than matrix multiplication. In *Proceedings 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 986–994, 2007.

[8] Z. Galil and O. Margalit. All pairs shortest distances for graphs with small integer length edges. *Information and Computation*, 134(2):103–139, 1997.

[9] Z. Galil and O. Margalit. All pairs shortest paths for graphs with small integer length edges. *J. Comput. Syst. Sci.*, 54(2):243–254, 1997.

[10] N. J. A. Harvey. Algebraic structures and algorithms for matching and matroid problems. In *Proceedings 47th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 531–542, 2006.

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

[12] J. Matoušek. Computing dominances in e^n. *Info. Proc. Lett.*, 38(5):277–278, 1991.

[13] M. Mucha and P. Sankowski. Maximum matchings via gaussian elimination. In *Proc. 45th Symp. on Foundations of Computer Science (FOCS)*, pages 248–255, 2004.

[14] M. Mucha and P. Sankowski. Maximum matchings in planar graphs via Gaussian elimination. *Algorithmica*, 45(1):3–20, 2006.

[15] P. Sankowski. Weighted bipartite matching in matrix multiplication time. In *Proceedings 33rd Int'l Symposium on Automata, Languages, and Programming (ICALP)*, pages 274–285, 2006.

[16] R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *J. Comput. Syst. Sci.*, 51(3):400–403, 1995.

[17] A. Shapira, R. Yuster, and U. Zwick. All-pairs bottleneck paths in vertex weighted graphs. In *SODA*, pages 978–985, 2007.

[18] A. Shoshan and U. Zwick. All pairs shortest paths in undirected graphs with integer weights. In *Proc. 40th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 605–614, 1999.

[19] T. Takaoka. Subcubic cost algorithms for the all pairs shortest path problem. *Algorithmica*, 20(3):309–318, 1998.

[20] V. Vassilevska. *Efficient Algorithms for Path Problems in Weighted Graphs*. PhD thesis, Carnegie Mellon University, August 2008.

[21] V. Vassilevska. Personal communication. 2008.

[22] V. Vassilevska and R. Williams. Finding a maximum weight triangle in $n^{3-\delta}$ time, with applications. In *STOC*, pages 225–231, 2006.

[23] V. Vassilevska, R. Williams, and R. Yuster. All-pairs bottleneck paths for general graphs in truly sub-cubic time. In *STOC*, pages 585–589, 2007.

[24] U. Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *J. ACM*, 49(3):289–317, 2002.

A Explicit Maximum Bottleneck Paths

The algorithm from Theorem 3.1 calculates the capacities of all bottleneck paths but does not return the paths as such. In this section we review some well known algorithms for actually generating the paths.

Let A_0 be the original capacity matrix of the graph (with ∞ along the diagonal) and let $A_q = A_{q-1} \otimes\!\!\!\otimes A_{q-1}$. Thus, $A_{\lceil \log n \rceil}[i,j]$ is the capacity of the bottleneck path between vertices i and j. Let W_q be the witness matrix for the qth iteration, i.e.:

$$W_q[i,j] = k \text{ s.t. } A_q[i,j] = \min\{A_{q-1}[i,k], A_{q-1}[k,j]\}$$

It is very simple to have our algorithms return the witness matrix. Let $I[i,j] = \min\{q \mid A_q[i,j] = A_{\lceil \log n \rceil}[i,j]\}$ be the iteration that establishes the bottleneck capacity between i and j. If the bottleneck path from i to j is composed of l edges we can return the path in $O(l)$ time as follows. If $I[i,j] = 0$ return the edge (i,j); otherwise, concatenate the paths from i to $W_{I[i,j]}[i,j]$ and from $W_{I[i,j]}[i,j]$ to j. The procedure above gives each edge in *amortized* constant time. Zwick [24] and Vassilevska et al. [23] gave simple procedures for finding the successor matrix S, given W, I, which allows us to generate the bottleneck path in $O(1)$ worst case time per edge. Let $S[i,j] = k$ if (i,k) is the first edge on the path from i to j.

It is straightforward to show that the **witness-to-successor** algorithm is correct and runs in $O(n^2)$ time; see [24, 23].

witness-to-successor(W, I)
  $S \leftarrow 0$
  For q from 0 to $\log n$
    $I_q \leftarrow \{(i,j) \mid I[i,j] = q\}$
  For every $(i,j) \in I_0$
    $S[i,j] \leftarrow j$
  For q from 1 to $\log n$
    For each $(i,j) \in I_q$
      $k \leftarrow W_q[i,j]$
      While $S[i,j] = 0$
        $S[i,j] \leftarrow S[i,k]$
        $i \leftarrow S[i,k]$
  Return S

The procedure above can easily be adapted to work with our APBSP algorithms.