All-Pairs Bottleneck Paths For General Graphs in Truly Sub-Cubic Time

Virginia Vassilevska*
Computer Science Dept.
Carnegie Mellon University
Pittsburgh PA
virgi@cs.cmu.edu

Ryan Williams†
Computer Science Dept.
Carnegie Mellon University
Pittsburgh PA
ryanw@cs.cmu.edu

Raphael Yuster
Department of Mathematics
University of Haifa
Haifa, Israel
raphy@research.haifa.ac.il

ABSTRACT

In the *all-pairs bottleneck paths* (APBP) problem (a.k.a. *all-pairs maximum capacity paths*), one is given a directed graph with real non-negative capacities on its edges and is asked to determine, for all pairs of vertices s and t, the capacity of a single path for which a maximum amount of flow can be routed 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 sub-cubic 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. Using this procedure, an explicit maximum bottleneck path for any pair of nodes can be extracted in time linear in the length of the path.

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems–*Computations on discrete structures; Geometrical problems and computations*

General Terms

Algorithms, Theory

Keywords

bottleneck path, maximum capacity path, matrix multiplication, sub-cubic time

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 [14, 7, 16, 21] has developed fast algorithms for certain interesting cases of the all-pairs shortest paths (APSP) problem in truly sub-cubic time, *i.e.* $O(n^{3-\delta})$ for some constant $\delta > 0$. Still, it remains to be seen if the general APSP problem can be solved in truly sub-cubic time. Several algorithms have been given for solving APSP in $n^{3-o(1)}$ time; the most recent development is by Chan [3] and runs in $O(n^3/\log^2 n)$ time.

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

The APBP problem has been studied alongside APSP in several contexts. Pollack [13] 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 [9] proved that in *undirected* graphs, APBP can be solved in $O(n^2)$ time by simply taking the paths in a maximum spanning tree. The directed case of the problem has remained open until now, and recently appeared as an explicit goal in [15]. Prior to our work, the best known algorithm for the general case of APBP had been $O(mn+n^2\log n)$ time (obtained by using Fredman and Tarjan's implementation of Dijkstra [6]).

Our method for APBP 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* $\ell \times m$ *matrix* B *is the unique matrix* C *such that*

$$C[i,j] = \max_{k=1,\ldots,\ell} \min\{A[i,k], B[k,j]\},$$

for all $i = 1, \ldots, n$ *and* $j = 1, \ldots, m$.

*Supported by a Computer Science Department PhD Scholarship.
†Supported by the NSF ALADDIN Center and a grant from Google, Inc.

The (max, min)-product is a 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* ([5], pp.73). The ideas behind our (max, min)-product algorithm use ingredients from the dominance approaches of prior work; for more details, see Section 4.

2. PRELIMINARIES

We use M^T to denote the transpose of a matrix M. As is typical, we define $\omega \geq 2$ to be the smallest real number such that matrix multiplication over a ring is in $O(n^{\omega+\varepsilon})$ arithmetic operations, for all $\varepsilon > 0$. The best known upper bound for ω is 2.376, given by Coppersmith and Winograd [4].

We use a special matrix product in our algorithms, first defined by Matousek [12].

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

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

To deal with technical issues with weight functions, we use the following definition.

Definition 2.2 *Given a graph $G = (V, E, w)$ with a weight function $w : E \to \mathbb{R}$, the* extension of w *is the unique function $\tilde{w} : (V \times V) \to \mathbb{R} \cup \{-\infty, \infty\}$ defined by:*

$$\tilde{w}(u, v) := \begin{cases} \infty & \text{if } u = v \\ -\infty & \text{if } (u, v) \notin E \\ w(u, v) & \text{otherwise.} \end{cases}$$

Throughout the paper, we shall assume that the weight function under consideration is an extension of some other weight function on edges.

Definition 2.3 *Given a graph $G = (V, E, w)$ with $w : E \to \mathbb{R}$, 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 maximum over the bottleneck edge weights of all paths from u to v.*

2.1 Model of computation

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.

3. RELATED WORK

In addition to the work mentioned in the introduction, there are a few other interesting results on APBP that deserve mention. Karger, Koller, and Phillips [10] 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 [17] proved that on random (Erdös-Renyi) graphs, both APBP and APSP can be solved in $O(n^2 \log n)$ time.

Very recently, Shapira, Yuster, and Zwick [15] have given algorithms for APBP in the special case where the *vertices* have capacities, but not the edges. Their algorithms use rectangular matrix multiplication, running in $O(n^{2.58})$ time. Note that the vertex-capacity case can be easily reduced to the edge-capacity case, by setting the capacity of an edge to be the minimum capacity of its two endpoints. Shapira, Yuster, and Zwick's algorithm relies on the linearity of the number of weights. As the number of capacities in the vertex-capacity case is only n, but the number in the edge-capacity case can be $\Omega(n^2)$, their techniques do not seem to apply to the latter case. The authors of [15] 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.

4. THE DOMINANCE APPROACH

We begin by revisiting an approach used by Chan [2] and Vassilevska and Williams [18] 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 of n points X 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 the Preliminaries. Then, $(M_X \circledcirc M_X^T)[i, j] = k$ 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 Matousek [12].

Theorem 4.1 (Matousek [12]) *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 a 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 weight 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 sub-cubic algorithm for APBP.

5. SPARSE DOMINANCE PRODUCT

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 A-matrix.

Theorem 5.1 (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 $|S| = m \geq n$. Let C be the matrix such that*

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

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^T in column 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 ring matrix multiplication). 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^T-entry in bucket b of column j in $O(t_{jb} \cdot \lceil g_j/r \rceil)$ time; in particular, for each $A[i,j] \leq B^T[k,j]$ where $A[i,j]$ and $B^T[k,j]$ are in bucket b, increment $D[i,k]$. Over all j and b, this takes time on the order of

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

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 $A[i,j]$, $B^T[k,j]$ 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 $O(\sqrt{m} \cdot n^{\frac{1+\omega}{2}})$. □

Remark 1 *If one replaces the $O(n^\omega)$ matrix multiplication algorithm in the above procedure with the fast sparse matrix multiplication of Yuster and Zwick [20], one can obtain an $O(\sqrt{|S_A| \cdot |S_B|} \cdot n^{\frac{\omega-1}{2}})$ algorithm for sparse dominance product, where S_A and S_B are subsets of $[n] \times [n]$, and the resulting matrix has $C[i,j] = |\{k \mid A[i,k] \leq B[k,j], (i,k) \in S_A, (k,j) \in S_B\}|$. However, this generalized algorithm did not turn out to be useful for our particular application.*

6. ALL-PAIRS BOTTLENECK PATHS

Armed with the sparse dominance product algorithm, we now turn to all-pairs bottleneck paths. We first show how to compute the $(\max, \min)$-product of matrices in truly sub-cubic 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 compute all-pairs bottleneck paths.

6.1 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 all-pairs bottleneck paths 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, APBP on an n vertex graph can be computed in roughly the time it takes to compute a $(\max, \min)$-product of $n \times n$ matrices.

Theorem 6.1 ([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 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, we immediately obtain the following corollary.

Corollary 6.1 *If the $(\max, \min)$-product of two arbitrary real $n \times n$ matrices is computable in $M(n)$ time, then all-pairs bottleneck paths of an n vertex graph is computable in $O(M(n))$ time.*

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

Theorem 6.2 (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 $O(n^{2+\frac{\omega}{3}})$ 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 [11]. 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 $O(n^{2+\frac{\omega}{2}}/\sqrt{g})$ 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 $\mathtt{SD}$ is the sparse dominance product from Theorem 5.1 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 5.1, all matrices C_b can be computed in

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

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 $n^2 g = \frac{n^{2+\omega/2}}{\sqrt{g}}$, obtaining $g = n^{\frac{\omega}{3}}$. The running time is hence $O(n^{2+\frac{\omega}{3}})$. □

Plugging in the best known value for ω by Coppersmith and Winograd [4], the runtime bound becomes $O(n^{2.79})$. If $\omega = 2$, then the above algorithm can run in $O(n^{2.67})$.

6.2 Computing Explicit Maximum Bottleneck Paths

By Corollary 6.1 we can obtain a matrix representing all-pairs bottleneck paths in an edge weighted directed graph in $O(n^{2+\frac{\omega}{3}})$ time. To compute the actual paths, a bit more work is necessary. We take an approach analogous to that used by Zwick [21] in computing all-pairs shortest paths. 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 vertices 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 [21] 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 vertices, returns a simple shortest path between the vertices. 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 6.3 *The all-pairs bottleneck paths problem can be computed in $O(n^{2+\frac{\omega}{3}})$ time. Furthermore, in $O(n^{2+\frac{\omega}{3}} \log n)$ 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 [21].

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. $\square$

7. CONCLUSION

We have provided the first truly sub-cubic algorithm for all-pairs bottleneck paths in general dense graphs, with no restrictions on edge weights or edge directions. Our algorithm 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 recently 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. We believe further improvements are probably possible along these lines.

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 [18]. An affirmative answer would immediately imply a truly sub-cubic APSP algorithm for general graphs, resolving a longstanding and prominent open problem. More modestly, it may be possible to use our methods to develop better maximum flow algorithms for the general case, as we are finding paths for which we can route a maximum amount of flow.

8. 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.

[2] T. M. Chan, All-pairs shortest paths with real weights in $O(n^3/\log n)$ time. In *Proc. Workshop on Algorithms and Data Struct.*, Springer-Verlag LNCS 3608:318–324, 2005.

[3] T. M. Chan, More algorithms for all-pairs shortest paths in weighted graphs. To appear in *Proc. of STOC*, 2007.

[4] D. Coppersmith, S. Winograd, Matrix multiplication via arithmetic progressions, *J. Symbolic Computation* 9(3):251–280, 1990.

[5] D. Dubois and H. Prade. Fuzzy Sets and Systems: Theory and Applications. *Academic Press*, 1980.

[6] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *JACM* 34(3):596–615, 1987.

[7] Z. Galil and O. Margalit. All-pairs shortest paths for graphs with small integer length edges. *JCSS* 54:243–254, 1997.

[8] Y. Han. An $O(n^3(\log\log n/\log n)^{5/4})$ Time Algorithm for All Pairs Shortest Paths. In *Proc. of ESA*, Springer-Verlag LNCS 4168:411–417, 2006.

[9] T. C. Hu. The Maximum Capacity Route Problem. *Operations Research* 9(6):898–900, 1961.

[10] D. Karger, D. Koller, and S. Phillips. Finding the Hidden Path: Time Bounds for All-Pairs Shortest Paths. *SIAM J. Computing* 22(6):1199–1217, 1993.

[11] M. Kowaluk and A. Lingas, LCA queries in directed acyclic graphs. In *Proc. of ICALP*, Springer-Verlag LNCS 3580:241–248, 2005.

[12] J. Matousek, Computing dominances in E^n. *Information Processing Letters* 38(5):277–278, 1991.

[13] M. Pollack. The Maximum Capacity Through a Network. *Operations Research* 8(5):733–736, 1960.

[14] R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *JCSS* 51:400–403, 1995.

[15] A. Shapira, R. Yuster and U. Zwick. All-Pairs Bottleneck Paths in Vertex Weighted Graphs. In *Proc. of SODA*, 978–985, 2007.

[16] A. Shoshan and U. Zwick. All Pairs Shortest Paths in Undirected Graphs with Integer Weights. *Proc. of FOCS*, 605–614, 1999.

[17] C. R. Subramanian. A generalization of Janson inequalities and its application to finding shortest paths. In *Proc. of SODA*, 795–804, 1999.

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

[19] V. Vassilevska, R. Williams, and R. Yuster. Finding the smallest H-subgraph in real weighted graphs and related problems. In *Proc. of ICALP*, Springer-Verlag LNCS 4051:262–273, 2006.

[20] R. Yuster and U. Zwick. Fast sparse matrix multiplication. *ACM Trans. on Algorithms* 1(1):2–13, 2005.

[21] Uri Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *JACM* 49(3):289–317, 2002.