Finding a Maximum Weight Triangle in $n^{3-\delta}$ Time, With Applications

Virginia Vassilevska Ryan Williams
Computer Science Department*
Carnegie Mellon University
Pittsburgh, PA
{virgi, ryanw}@cs.cmu.edu

ABSTRACT

We present the first truly sub-cubic algorithms for finding a maximum node-weighted triangle in directed and undirected graphs with arbitrary real weights. The first is an $O(B \cdot n^{\frac{3+\omega}{2}}) = O(B \cdot n^{2.688})$ deterministic algorithm, where n is the number of nodes, ω is the matrix multiplication exponent, and B is the number of bits of precision. The second is a strongly polynomial randomized algorithm that runs in $O(n^{\frac{3+\omega}{2}} \log n)$ expected worst-case time. To achieve this, we show how to efficiently sample a weighted triangle uniformly at random, out of just those triangles whose total weight falls in some prescribed interval (W_1, W_2) for arbitrary weights W_1 and W_2. Previous approaches to the problem resulted in time bounds with either an *exponential* dependence on B, or a runtime of the form $\Omega(n^3/(\log n)^c)$. The algorithms are easily extended to finding a maximum node-weighted induced subgraph on $3k$ nodes in $\tilde{O}(n^{\frac{(3+\omega)k}{2}}) = \tilde{O}(n^{2.688k})$ time.

We give applications to a variety of problems, including a stable matching problem between buyers and sellers in computational economics, and discuss the possibility of extending our approach to a truly sub-cubic algorithm for computing all-pairs shortest paths on directed graphs with arbitrary weights.

*Both authors were supported by the NSF ALADDIN Center, under Grant No. 0122581.

STOC'06, May 21–23, 2006, Seattle, Washington, USA.

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

clique, dominating pairs, independent set, matrix multiplication, sub-cubic algorithm, triangle

1. INTRODUCTION

We revisit the fundamental problem of efficiently finding a triangle in a graph. In particular, we focus on the case where the graph has arbitrary real weights on its nodes, and we wish to determine if there is a triangle of maximum weight. We give a deterministic algorithm of the form $O(n^{3-\delta} \cdot \log W)$, where $\delta > 0$ and W is the largest weight, and a randomized algorithm that is $O(n^{3-\delta})$ (strongly polynomial) in an addition-comparison model.

Since the 1970's, it has been known that the time complexity of finding a triangle in an unweighted graph is at most the time complexity of matrix multiplication, cf. Itai and Rodeh [8]. Therefore, triangle-finding is possible in $O(n^\omega)$ time, where $\omega < 2.376$. The algorithm naturally extends to an algorithm for finding a $3k$-clique, and more generally an induced H-subgraph of size $3k$, in $O(k^2 n^{\omega k})$.

On the other hand, the time complexity of finding an optimal triangle in a weighted graph has been wide open. In general, reductions to fast matrix multiplication tend to fail miserably in the case of real-weighted graph problems. The most prominent example of this is the famous all-pairs shortest paths (APSP) problem. Seidel [12] and Galil and Margalit [5] developed $\tilde{O}(n^\omega)$ algorithms for undirected unweighted graphs. However, for arbitrary edge weights, the best algorithm known is the recent $O(n^3/\log n)$ one, by Chan [2]. When the edge weights are integers in $[-M, M]$, the problem is solvable in $\tilde{O}(Mn^{\omega})$ by Shoshan and Zwick [13], and $\tilde{O}(M^{0.681}n^{2.575})$ by Zwick [17], respectively. Earlier, a series of papers in the 70's and 80's starting with Yuval [16] attempted to speed up APSP directly using fast matrix multiplication. Unfortunately, these works require a model that allows for infinite-precision operations in constant time.

Our techniques have several interesting applications. We cite two here:

1. We can solve a generalization of the ubiquitous 3-Sum problem in computational geometry: given three tables of numbers T_1, T_2, T_3, along with a target K, is there $x \in T_1$, $y \in T_2$, and $z \in T_3$ such that $x + y + z = K$? Imposing an edge relation that constrains what possible pairs of numbers can be summed together is precisely the problem of finding a node-weighted triangle of a prescribed weight, which we can handle.

2. We consider a general buyer-seller problem from computational economics. Suppose we have a set of k commodities, along with n buyers and n sellers. Seller i has a reserve price v_{ij} for commodity j. Buyer i has a value p_{ij} for commodity j corresponding to the maximum price that i is willing to pay for the commodity.

   In the simplest case considered, a buyer ℓ *prefers* seller i to seller j if the number of commodities k for which $v_{ik} \le p_{\ell k}$ exceeds the number of commodities k for which $v_{jk} \le p_{\ell k}$. In the event of a tie, the buyer will prefer the seller with the cheapest sum of prices. In other words, the *perception* of buyer ℓ is that the ith seller has more affordable goods than the jth seller does. The preferences of sellers are defined analogously.

   A natural question is: how quickly can one determine a stable matching of buyers and sellers, under the above definition of preferences? The trivial algorithm takes $\Omega(n^2 k)$ time. Applying ideas from our triangle-finding algorithm, we can compute a stable matching in $O(n^2 + n\sqrt{kM(n,k)})$, where $M(n,k)$ is the complexity of multiplying an $n \times k$ matrix with a $k \times n$ matrix. For example, for $k = n$, the bound is $O(n^{2.688})$, and for $k \le n^{.294}$, the bound is $O(n^{2.147})$.

We are currently working to extend our algorithm to a sub-cubic algorithm for finding a maximum *edge-weighted* triangle. (The node-weighted version is a special case of the edge-weighted version.) We believe that if such an extension is possible, it should help us make further headway into the problem of finding a truly sub-cubic algorithm for APSP with arbitrary weights. This possibility will be discussed later in the paper.

2. PRELIMINARIES

Definition 2.1 *Let $A, B \in (\mathbb{R} \cup \{+\infty, -\infty\})^{n \times n}$. The* (min, +)-product *of A and B, denoted as $A \star B$, is the matrix C such that*

$$C[i,j] = \min_k \{A[i,k] + B[k,j]\}.$$

$A \star B$ is also sometimes called the distance product.

2.1 Model of computation

We use the familiar *addition-comparison model*, where additions and comparisons of input weights are allowed as unit operations. This model allows for the possibility of *strongly polynomial* algorithms, whose time complexity is independent of the weights in the input.

3. DOMINATING PAIRS

Given a set of points $\{v_1, \ldots, v_n\}$ in $\mathbb{R}^d$, the well-known *dominating pairs problem* is to find all pairs of points (v_i, v_j) such that for all $k = 1, \ldots, d$, $v_i[k] \le v_j[k]$. The key insight to our method is a connection between the problem of finding triangles and the problem of computing dominating pairs. This connection was inspired by recent work of Chan [2], who demonstrated how a $O(c^d n^{1+\varepsilon} + n^2)$ algorithm for computing dominating pairs in d dimensions can be used to solve the arbitrary APSP problem in $O(n^3/\log n)$ time.

In particular, we use an elegant algorithm by Matousek for computing dominating pairs in n dimensions [10]. Matousek's algorithm does more than determine dominances — it actually computes a matrix D such that

$$D[i,j] = |\{k \mid v_i[k] \le v_j[k]\}|.$$

We will call D the *dominance matrix* in the following.

Theorem 3.1 (Matousek [10]) *For a set S of n points in $\mathbb{R}^n$, the dominance matrix for S can be computed in $O\left(n^{\frac{3+\omega}{2}}\right)$ time.*

We outline Matousek's approach in the following paragraphs. For each coordinate $j = 1, \ldots, n$, sort the n points by coordinate j. This takes $O(n^2 \log n)$ time. Define the *jth rank of point v_i*, denoted as $r_j(v_i)$, to be the position of v_i in the sorted list for coordinate j.

Let $s \in [\log n, n]$ be a parameter to be determined later. Define n/s pairs of Boolean matrices (A_1, B_1), …, $(A_{n/s}, B_{n/s})$ as follows:

$$A_k[i,j] = 1 \iff r_j(v_i) \in [ks, ks+s),$$

$$B_k[i,j] = 1 \iff r_j(v_i) \ge ks + s.$$

Now, multiply A_k with B_k^T, obtaining a matrix C_k. Then $C_k[i,j]$ equals the number of coordinates c such that $v_i[c] \le v_j[c]$, $r_c(v_i) \in [ks, ks+s)$, and $r_c(v_j) \ge ks+s$.

Therefore, letting

$$C = \sum_{k=1}^{n/s} C_k,$$

we have that $C[i,j]$ is the number of coordinates c such that $\lfloor r_c(v_i)/s \rfloor < \lfloor r_c(v_j)/s \rfloor$.

Suppose we compute a matrix E such that $E[i,j]$ is the number of c such that $v_i[c] \le v_j[c]$ and $\lfloor r_c(v_i)/s \rfloor = \lfloor r_c(v_j)/s \rfloor$. Then, defining $D := C + E$, we will have the desired matrix

$$D[i,j] = |\{k \mid v_i[k] \le v_j[k]\}|.$$

To compute E, we use the n sorted lists. For each pair $(i,j) \in [n] \times [n]$, we look up v_i's position p in the sorted list for coordinate j. By reading off the adjacent points less than v_i in this sorted list (*i.e.* the points at positions $p-1$, $p-2$, *etc.*), and stopping when we reach a point v_k such that $\lfloor r_j(v_k)/s \rfloor < \lfloor r_j(v_i)/s \rfloor$, we obtain the list $v_{i_1}, \ldots, v_{i_\ell}$ of $\ell \le s$ points such that $v_{i_x}[j] \le v_i[j]$ and $\lfloor r_j(v_i)/s \rfloor = \lfloor r_j(v_{i_x})/s \rfloor$. For each $x = 1, \ldots, \ell$, we add a 1 to $E[i_x, i]$. Assuming constant time lookups and constant time probes into a matrix, this entire process takes only $O(n^2 s)$ time.

The runtime of the above procedure is $O(n^2 s + \frac{n}{s} n^\omega)$, where $\omega < 2.376$ is the matrix multiplication exponent. Choosing $s = n^{\frac{\omega-1}{2}}$, the time bound becomes $O\left(n^{\frac{3+\omega}{2}}\right)$.

3.1 The low-dimensional case

Here we sketch a slight but useful extension of the previous algorithm that uses rectangular matrix multiplication to get an improved time bound, when the points are in $\mathbb{R}^d$ with $d << n$.

Theorem 3.2 *Given a set of n points in $\mathbb{R}^d$, the dominance matrix can be computed in $O(n^{1.706} \cdot d^2 + n^{2+\varepsilon})$ time for all $\varepsilon > 0$, or in $O(n \cdot d^2 + n^\omega)$ time.*

So for example, when $d < n^{0.688}$, we can compute the dominance matrix in $O(n^\omega)$. We will just sketch the improvement, for sake of space. The A_k and B_k matrices in the above become $n \times d$ and $d \times n$ matrices, respectively. We replace the sum of C_k's in the above by a product of block matrices:

$$C = \left[\begin{array}{cccc} A_1 & A_2 & \cdots & A_{n/s} \end{array}\right] \cdot \left[\begin{array}{c} B_1 \\ B_2 \\ \vdots \\ B_{n/s} \end{array}\right]$$

that is, we are multiplying an $n \times dn/s$ and a $dn/s \times n$ matrix.

Computing E in this situation can be verified to take $O(n \cdot d \cdot s)$ time. As before, our goal is to choose s optimally so that this runtime is minimized. Results of Huang and Pan [7] coupled with the optimal choice of s show that we can either compute the dominances matrix in $O(n^{1.706} \cdot d^2 + n^{2+\varepsilon})$ time for all $\varepsilon > 0$, or $O(n \cdot d^2 + n^\omega)$ time. (Details omitted in this version.)

4. MAXIMUM WEIGHT TRIANGLE

We begin with a simple result for the node and edge weighted version of the maximum triangle problem. We have not seen it in the literature, but it is most likely folklore.

Given a (directed) graph $G = (V, A)$ with weights $w_e : A \to R$, $w_v : V \to R$ for some arithmetic domain R (integers, reals, *etc.*), one wishes to find a triangle, that is $u, v, w \in V$ so that $(u,v), (v,w), (w,u) \in A$ and $w_e(u,v) + w_e(v,w) + w_e(w,u) + w_v(u) + w_v(v) + w_v(w)$ is maximized.

Proposition 1 (Folklore) *A maximum node and edge weighted triangle can be found in $O(D(n))$, where $D(n)$ is the time complexity of* $(\min,+)$*-product.*

Proof. It is easy to see that the general case is equivalent to the edge-weighted case. Given w_e and w_v, define $w'_v(u) = 0$ for each $u \in V$ and $w'_e(u,v) = w_e(u,v) + (w_v(u) + w_v(v))/2$.

Let A be the weighted adjacency matrix of the graph with w'. Compute $B = (-A) \star (-A)$. Now take $C = A - B$ and find an i, j such that $C[i,j]$ is maximum. It follows that that (i,j) is an edge on the maximum weighted triangle. Finding an explicit triangle can be done by testing all vertices k for adjacency with (i,j). The runtime is $O(D(n) + n^2) = O(D(n))$. $\square$

Note that the best known time bounds for $D(n)$ are

- $O(n^3/\log n)$ for arbitrary real weights, by Chan [2], and
- $O(M \cdot n^\omega)$ for integer weights in $[-M, M]$, by Yuval [16] and Zwick [17].

Thus, prior to our work, the only known truly sub-cubic triangle algorithms were also *pseudo-polynomial*.

The above can be easily generalized to find a maximum weighted induced H-subgraph.

Corollary 4.1 *For an n node graph G, a maximum node and edge weighted induced H-subgraph on k nodes can be found in $O(D(n^{\lceil k/3 \rceil}) + k^2 n^{2\lceil k/3 \rceil})$, where $D(n)$ is the time complexity of* $(\min,+)$*-product.*

Proof. The idea is to construct a new graph G' on $O(n^{\lceil k/3 \rceil})$ nodes that has a triangle of weight W iff G has an induced H-subgraph of weight W. Let $H = H_1 \cup H_2 \cup H_3$, where H_i are disjoint and have at most $\lceil k/3 \rceil$ nodes each. Assume each node in each H_i is labelled $1, \ldots, |H_i|$. For each $i = 1, 2, 3$, create a vertex in G' for every ordered $|H_i|$-tuple of nodes of G that is an induced copy of H_i, where the jth node in the tuple corresponds to the node in H_i with label j. The node weight of a vertex in G' equals the sum of node and edge weights of the corresponding induced subgraph of G.

In G', put an edge between two vertices representing copies of H_i and H_j for $i \neq j$ iff the tuples they represent are node-disjoint, and the union of those tuples is a copy of $H_i \cup H_j$, under the same mapping that mapped the nodes of the first tuple to H_i and the nodes of the second tuple to H_j (*i.e.*, if the intertuple edges match the edges of $H_i \cup H_j$). The weight of such an edge (u, v) equals the sum of all weights of edges in G connecting the tuples represented by u and v.

The proof follows by observing that there is a weight-preserving bijection between weighted triangles in G' and induced weighted H-subgraphs of G. $\square$

4.1 Node weights

Given the close relationship of the above problem to the distance product of matrices, it is perhaps surprising that in the case of node weights, we can find a maximum weight triangle in sub-cubic time. We first present a weakly polynomial deterministic algorithm, then a randomized strongly polynomial algorithm.

4.1.1 Deterministic algorithm

THEOREM 4.1 *On graphs with integer weights, a maximum node-weighted triangle can be found in*

$$O(n^{(\omega+3)/2} \cdot \log W) \text{ time},$$

where W is the maximum weight of a triangle. On graphs with real weights, a maximum node-weighted triangle can be found in $O(n^{(\omega+3)/2} \cdot B)$ time, where B is the maximum number of bits in a weight.

PROOF. The idea is to obtain a procedure that, given a parameter K, returns an edge (i, j) from a triangle of weight at least K. Then one can binary search to find the weight of the maximum triangle, and try all possible vertices k to get the triangle itself.

We first explain the binary search. Without loss of generality, we assume that all edge weights are at least 1. Let W be the maximum weight of a triangle. Start by checking if there is a triangle of weight at least $K = 1$ (if not, there are no triangles). Then try $K = 2^\ell$ for increasing ℓ, until there exists a triangle of weight 2^ℓ but no triangle of weight $2^{\ell+1}$. This ℓ will be found in $O(\log W)$ steps. After this, we search on the interval $[2^\ell, 2^{\ell+1})$ for the largest K such that there is a triangle of weight K. This takes $O(\log W)$ time for integer weights, and $O(B)$ time for real weights with B bits of precision.

We now show how to return an edge from a triangle of weight at least K, for some given K. Let $V = \{1, \ldots, n\}$ be the set of vertices. For every $i \in V$, we make a point $f_i = (e(1), \ldots, e(n))$, where

$$e(j) = \begin{cases} K - w(i) & \text{if there is an edge from } i \text{ to } j \\ \infty & \text{otherwise.} \end{cases}$$

(In implementation, we can of course substitute a sufficiently large value in place of ∞.) We also make a point $g_i = (e'(1), \ldots, e'(n))$, where

$$e'(j) = \begin{cases} w(i) + w(j) & \text{if there is an edge from } i \text{ to } j \\ -\infty & \text{otherwise.} \end{cases}$$

Compute the dominance matrix $D(K)$ on the union of the sets $\{f_i\}$ and $\{g_i\}$. For all edges (i, j) in the graph, check if there exists a k such that $f_i[k] \leq g_j[k]$. This can be done by examining the appropriate entry in $D(K)$. If such a k exists, then we know there is a vertex k such that

$$K - w(i) \leq w(j) + w(k) \implies K \leq w(i) + w(k) + w(j),$$

that is, there exists a triangle of weight at least K using edge (i, j).

Observe that the above works for both directed and undirected graphs. $\square$

4.1.2 Randomized strongly-polynomial algorithm

In the above, the binary search over all possible weights prevents our algorithm from being strongly polynomial. We would like to have an algorithm that, in a comparison-based model, has a runtime with *no* dependence on the bitlengths of weights. Here we present an randomized algorithm that achieves this.

THEOREM 4.2 *On graphs with real weights, a maximum node-weighted triangle can be found in*

$$O(n^{(\omega+3)/2} \cdot \log n) \text{ expected worst-case time.}$$

We would like to somehow binary search over the collection of triangles in the graph to find the maximum. As this collection is $O(n^3)$, we would then have a strongly polynomial bound. Ideally, one would like to pick the "median" triangle from a list of all triangles, sorted by weight. But as the number of triangles can be $\Omega(n^3)$, forming this list is hopeless. Instead, we shall show how dominance computations allow us to efficiently and uniformly sample a triangle at random, whose weight is from any prescribed interval (W_1, W_2). If we pick a triangle at random and measure its weight, there is a good chance that this weight is close to the median weight. In fact, a binary search that randomly samples for a pivot can be expected to terminate in $O(\log n)$ time.

Let $W_1, W_2 \in \mathbb{R} \cup \{-\infty, \infty\}$, $W_1 < W_2$, and G be a node-weighted graph.

DEFINITION 4.1 *$\mathcal{C}(W_1, W_2)$ is defined to be the collection of triangles in G whose total weight falls in the range $[W_1, W_2]$.*

Lemma 4.1 *One can sample a triangle uniformly at random from $\mathcal{C}(W_1, W_2)$, in $O(n^{(\omega+3)/2})$ time.*

Proof. From the proof of Theorem 4.1, one can compute a matrix $D(K)$ in $O(n^{(\omega+3)/2})$ time, such that $D(K)[i,j] \neq 0$ iff there is a node k such that (i,k) and (k,j) are edges, and $w(i) + w(j) + w(k) > K$. In fact, the i,j entry of $D(K)$ is the *number* of distinct nodes k with this property.

Similarly, one can compute matrices $E(K)$ and $L(K)$ such that

- $E(K)[i,j] = |\{k|\ (i,k),(k,j) \in E,\ w(i) + w(j) + w(k) \leq K\}|$
- $L(K)[i,j] = |\{k|\ (i,k),(k,j) \in E,\ w(i) + w(j) + w(k) < K\}|$.

(This can be done by flipping the signs on all coordinates in the sets of points $\{f_i\}$ and $\{g_i\}$ from Theorem 4.1, then computing dominances, disallowing equalities for L.)

Therefore, if we take $F = E(W_1) - L(W_2)$, then $F[i,j]$ is the number of nodes k where there is a path from i to k to j, and $w(i) + w(j) + w(k) \in [W_1, W_2]$.

Let f be the sum of all entries in F. For each $(i,j) \in E$, choose (i,j) with probability $F[i,j]/f$. By the above, this step uniformly samples an edge from a random triangle. Finally, we look at the set of nodes S that are neighbors to both i and j, and pick each node in S with probability $\frac{1}{|S|}$. This step uniformly samples a triangle with edge (i,j). The final triangle is therefore chosen uniformly at random. □

Observe that there is an interesting corollary to the above.

Corollary 4.2 *In any graph, one can sample a triangle uniformly at random in $O(n^\omega)$ time.*

Proof. (Sketch) Multiplying the adjacency matrix with itself counts the number of 2-paths from each node to another node. Therefore one can count the number of triangles and sample just as in the above. □

We are now prepared to give the strongly polynomial algorithm.

Proof of Theorem 4.2. Start by choosing a triangle t uniformly at random from all triangles. By the corollary, this is done in $O(n^\omega)$ time.

Measure the weight W of t. Determine if there is a triangle with weight in the range (W, ∞), in $O(n^{(\omega+3)/2})$ time. If not, return t. If so, randomly sample a triangle from (W, ∞), let W' be its weight, and repeat the search with W'.

It is routine to estimate the runtime of this procedure, but we include it for completeness. Let $T(n,k)$ be the expected runtime for an n node graph, where k is the number of triangles in the current weight range under inspection. In the worst case,

$$T(n,k) \leq \frac{1}{k}\sum_{i=1}^{k-1} T(n,k-i) + c \cdot n^{(\omega+3)/2}$$

for some constant $c \geq 1$. But this means

$$T(n,k-1) \leq \frac{1}{k-1}\sum_{i=1}^{k-2} T(n,k-i) + c \cdot n^{(\omega+3)/2},$$

so

$$\begin{aligned} T(n,k) &\leq \left(\frac{1}{k} + \frac{k-1}{k}\right) \cdot T(n,k-1) \\ &\quad + \left(1 - \frac{k-1}{k}\right) cn^{(\omega+3)/2} \\ &= T(n,k-1) + \frac{c}{k} n^{(\omega+3)/2}, \end{aligned}$$

which solves to $T(n,k) = O(n^{(\omega+3)/2} \log k)$. □

4.1.3 An algorithm for sparse graphs

On sparse graphs, we can give a maximum node-weighted triangle algorithm which has comparable runtime to that of the unweighted version of the problem.

Theorem 4.3 *Let m be the number of edges in a graph with real weights on its nodes. A triangle of maximum node weight can be found in $O(m^{2-\frac{4}{5+\omega}}) = O(m^{1.46})$ time.*

Note that the best known algorithm for the unweighted version of this problem runs in $O(m^{2\omega/(\omega+1)}) = m^{1.41}$ time, by Alon, Yuster, and Zwick [1].

Proof. We follow the high/low technique of Alon, Yuster, and Zwick. Let $\Delta > 0$ be a parameter to be determined later. Divide the set of nodes into high degree nodes (those with degree at least Δ) and low degree nodes (those with degree less than Δ). By a counting argument, there are at most $2m/\Delta$ high degree nodes. Either a triangle contains only high degree nodes, or it contains some low degree node.

To cover the first case, restrict the graph to only its high degree nodes, and find the max node-weighted triangle in $\tilde{O}((m/\Delta)^{\frac{\omega+3}{2}})$ time. To cover the second case, iterate over all edges. When the current edge has a low degree node, check all of its $O(\Delta)$ neighbors for a triangle. This takes time $O(m \cdot \Delta)$. Setting Δ optimally to minimize runtime, $\Delta = m^{1-4/(5+\omega)}$, and $O(m^{2-4/(5+\omega)}) = O(m^{1.458})$. □

4.1.4 Maximum node-weighted H-subgraph

The result for node-weighted triangle can easily be extended to find a maximum node-weighted clique, and in general a maximum node-weighted induced H-subgraph. The transformation from Corollary 4.1 immediately gives

Theorem 4.4 *On graphs with arbitrary integer weights, a maximum node-weighted induced H-subgraph on $3k$*

nodes can be found in

$$O(n^{(\omega+3)k/2} \cdot \log n) \text{ expected worst-case time.}$$

5. TOWARDS TRULY SUB-CUBIC APSP

Starting with Floyd's n^3 algorithm [6], finding an $n^{3-\delta}$ algorithm for the general APSP has been a landmark problem in algorithms for at least forty years. It has seen its ups and downs. Kerr [9] in 1970 showed that the distance product requires $\Omega(n^3)$ on a straight-line program using $+$ and min. The first $o(n^3)$ algorithm was given by Fredman in 1976, running in $O(n^3 \cdot (\log\log n/\log n)^{1/3})$ time. The time complexity of APSP on dense graphs has gone through several improvements over the years, culminating in the aforementioned algorithm of Chan [2].

Computing the distance product quickly has long been considered as the key to a truly sub-cubic APSP algorithm, since it is known that the time complexity of APSP is no worse than that of the distance product of two arbitrary $n \times n$ matrices. Practically all APSP algorithms with runtime of the form $O(n^\alpha)$ have, at their core, some form of distance product. Therefore, any improvement on the complexity of distance product is interesting.

Here we show that *the most significant bits of $A \star B$* can be computed in sub-cubic time, again with no exponential dependence on edge weights. In previous work, Zwick [17] shows how to compute *approximate* distance products. Given any $\varepsilon > 0$, his algorithm computes distances d_{ij} such that the difference of d_{ij} and the exact value of the distance product entry is at most $O(\varepsilon)$. The running time of his algorithm is $O(\frac{W}{\varepsilon} \cdot n^\omega \log W)$. Unfortunately, guaranteeing that the distances are within ε of the right values, does not necessarily give any of the bits of the distances. Our strategy is to use dominance matrix computations.

Proposition 2 *Let $A, B \in (\mathbb{Z} \cup \{+\infty, -\infty\})^{n\times n}$. The k most significant bits of all entries in $A\star B$ can be determined in $O(2^k \cdot n^{\frac{3+\omega}{2}} \log n)$ time, assuming a comparison-based model.*

Proof. For a matrix M, let $M[i,:]$ be the ith row, and $M[:,j]$ be the jth row. For a constant K, define the set of vectors

$$M^L(K) := \{(M[i,1]-K, \ldots, M[i,n]-K) \mid i = 1, \ldots, n\}.$$

Also, define

$$M^R(K) := \{(-M[1,i], \ldots, -M[n,i]) \mid i = 1, \ldots, n\}.$$

Now consider the set of vectors

$$S(K) = A^L(K) \cup B^R(K).$$

Suppose we compute the matrix $C(K)$ defined by

$$C(K)[i,j] := |\{k \mid v_i[k] \leq v_j[k] \mid v_i, v_j \in S(K)\}|.$$

Then for any i, j,

$$\min_k\{A[i,k] + B[k,j]\} \leq K \iff C(K)[i,j] \neq 0.$$

Let $W = \max_{ij}\{A[i,j] + \max_{ij} B[i,j]\}$. Then $C(\frac{W}{2})$ gives the most significant bit of each entry in $A \star B$. To obtain the second most significant bit, compute $C(\frac{W}{4})$ and $C(\frac{3W}{4})$. The second bit of $(A \star B)[i,j]$ is given by the expression:

$$(C(\tfrac{W}{2})[i,j] \wedge C(\tfrac{3W}{4})[i,j]) \vee (\neg C(\tfrac{W}{2})[i,j] \wedge C(\tfrac{W}{4})[i,j]).$$

In general, to recover k bits of $(A \star B)$, one computes $C(\cdot)$ for 2^k values of K. We omit the details. $\square$

6. BUYER-SELLER STABLE MATCHING

In our final application, we show how the "dominance-comparison" ideas can be used to improve the runtime for solving a matching problem arising in computational economics. In this problem, we have a set of buyers and a set of sellers. Each buyer has a set of items he wants to purchase, together with a maximum price for each item which he is willing to pay for that item. In turn, each seller has a set of items she wishes to sell, together with a reserve price for each item which she requires to be met in order for the sale to be completed. Formally:

Definition 6.1 *An (n,k)-*Buyer-Seller *instance consists of*

- *a set $C = \{1, \ldots, k\}$ of* commodities [1]
- *an n-tuple of* buyers *$B = (b_1, \ldots, b_n)$ where $b_i = (B_i, p_i)$, s.t. $B_i \subseteq C$ are the commodities desired by buyer i, and $p_i : B_i \to \mathbb{R}^+$ is the maximum price function for buyer i*
- *an n-tuple of* sellers *$S = (s_1, \ldots, s_n)$ where $s_i = (S_i, v_i)$, s.t. $S_i \subseteq C$ are the commodities owned by seller i, and $v_i : S_i \to \mathbb{R}^+$ is the reserve price function for seller i*

A sale transaction for an item ℓ between a seller who owns ℓ and a buyer who wants ℓ can take place if the price the buyer is willing to pay is at least the reserve price the seller has for the item. Let us imagine that each buyer wants to do business with only one seller, and each seller wants to target a single buyer. Then the transaction between a buyer and a seller consists of all the items for which the buyer's maximum price meets the seller's reserve price.

Definition 6.2 *For a buyer (B_i, p_i) and a seller (S_j, v_j) the* transaction set *C_{ij} is defined as $C_{ij} = \{\ell \mid \ell \in B_i \cap S_j,\ p_i(\ell) \geq v_j(\ell)\}$. Denote by $\mathbf{C}$ the* transaction matrix *with entries $|C_{ij}|$.*

[1] We will use the words "commodities" and "items" interchangably.

The price *of C_{ij} is defined as $P_{ij} = \sum_{\ell \in C_{ij}} p_i(\ell)$, and the* reserve *of C_{ij} is defined as $R_{ij} = \sum_{\ell \in C_{ij}} v_j(\ell)$. Denote by P and R respectively the transaction* price *and* reserve *matrices with entries P_{ij} and R_{ij}.*

Further, we assume that every buyer i has a preference relation on the sellers j which depends entirely on P_{ij}, R_{ij} and $|C_{ij}|$. Conversely, every seller has a preference relation on the buyers determined by the same three values. More formally,

- buyer i has a (computable) preference function $f_i : \mathbb{R}^+ \times \mathbb{R}^+ \times \mathbb{Z}^+ \to \mathbb{Z}$ such that i prefers seller j to seller j' iff $f_i(P_{ij}, R_{ij}, |C_{ij}|) \geq f_i(P_{ij'}, R_{ij'}, |C_{ij'}|)$.
- Similarly, seller j has a (computable) preference function $g_j : \mathbb{R}^+ \times \mathbb{R}^+ \times \mathbb{Z}^+ \to \mathbb{Z}$ such that j prefers buyer i to buyer i' iff $g_j(P_{ij}, R_{ij}, |C_{ij}|) \geq g_j(P_{i'j}, R_{i'j}, |C_{i'j}|)$.

Ideally, each buyer wants to talk to his most preferred seller, and each seller wants to sell to her most preferred buyer. Unfortunately, this is not always possible for all buyers, even when the prices and reserves are all equal, and all preference functions equal $|C_{ij}|$. This is evidenced by the following example: Buyer 1 wants to buy item 2, buyer 2 wants to buy items 1 and 2, seller 1 has item 1, seller 2 has items 1 and 2. Here buyer 1 will not be able to get any items.

In a realistic setting, we want to find a buyer-seller matching so that there is no pair (b_i, s_j) for which b_i is not paired with s_j, such that both b_i and s_j would benefit from breaking their matches and pairing among each other. This is the *stable matching* problem, for which optimal algorithms are known when the preferences are known (*e.g.*, Gale-Shapley [4] can be implemented to run in $O(n^2)$). However, for large k, the major bottleneck in our setting is that of computing the preference functions of the buyers and sellers.

The obvious approach to compute P_{ij}, R_{ij} and $|C_{ij}|$ is to explicitly find the sets C_{ij}. This gives an $O(kn^2)$ algorithm to compute P_{ij}, R_{ij} and $|C_{ij}|$ for all pairs (i,j). Note that the sizes of P_{ij}, R_{ij} and $|C_{ij}|$ do affect the running time, hence our results are per arithmetic operation on numbers of this size.

Let f be a (computable) function. Let $T_f(b)$ be a time bound sufficient for computing $f(p,r,c)$, over all b-bit p, r and c. Define

$$T = \max_{i=1,\dots,n} \{T_{f_i}, T_{g_i}\}.$$

Then in time $O(kn^2 + Tn^2 + n^2 \log n)$, one can easily obtain for every buyer (seller) a list of the sellers (buyers) sorted by the buyer's (seller's) preference function.

Exploiting fast dominance computation, we can do better than the above trivial algorithm.

THEOREM 6.1 *The matrices P, R and $\mathbf{C}$ for an (n,k)-Buyer-Seller instance can be found in $O(n\sqrt{kM(n,k)})$ time, where $M(n,k)$ is the time required to multiply an $n \times k$ by a $k \times n$ matrix.*

PROOF. Using the dominance technique, we can compute matrix $\mathbf{C}$ as follows. For each buyer i we create a k-dimensional vector $\beta_i = (\beta_{i1}, \dots, \beta_{ik})$ so that $\beta_{ij} = p_i(j)$ if $j \in C_i$, and $\beta_{ij} = -\infty$ if $j \notin C_i$. For each seller i we create a k-dimensional vector $\sigma_i = (\sigma_{i1}, \dots, \sigma_{ik})$ so that $\sigma_{ij} = v_i(j)$ if $j \in S_i$, and $\sigma_{ij} = \infty$ if $j \notin S_i$. Computing the dominance matrix for these points computes exactly the number of items ℓ which buyer i wants to buy, seller j wants to sell, and $p_i(\ell) \geq v_j(\ell)$.

By a modification of Matousek's algorithm for computing dominances, we can also compute the matrices P and R.

We demonstrate how to find R. Recall that the dominance algorithm does a matrix multiplication $A_k \cdot B_k^T$ with entries $A_k[i,j] = 1$ iff $r_j(b_i) \in [ks, ks+s)$, and $B_k[i,j] = 1$ iff $r_j(s_i) \geq ks + s$ (using the notation from Theorem 3.1). Let B_k be the same, but redefine A_k to be

$$A_k[i,j] = \begin{cases} v_i(j) & \text{if } r_j(b_i) \in [ks, ks+s) \\ 0 & \text{otherwise} \end{cases}.$$

Similar modifications are made to the computation of the matrix E. Instead of adding 1 to the matrix entry $E[i_x, i]$ in the step for coordinate j, we add the corresponding reserve price $v_{i_x}(j)$. Determining P can be done analogously. $\square$

COROLLARY 6.1 *A buyer-seller stable matching can be determined in $O(n\sqrt{kM(n,k)} + n^2 \log n + n^2 T)$, where T is the maximum time to compute the preference functions of the buyers/sellers, given the buyer price and seller reserve sums for a buyer-seller pair.*

For instance, if $k = n$ and $T = O(\sqrt{n})$, the runtime of finding a buyer-seller stable matching is $O(n^{\frac{3+\omega}{2}}) = O(n^{2.688})$.

7. CONCLUSION

We have presented the first "truly" sub-cubic algorithm for finding a node-weighted triangle, along with several interesting applications that exploit dominance matrices in similar ways. We conjecture that our ideas can be extended to a sub-cubic algorithm for the edge-weighted case, and perhaps even to computing the distance product. As demonstrated above, we can estimate the maximum weight of a length-two path between two nodes, but one must be very conservative in the number of dominance matrix calls to have a chance of being sub-cubic.

Perhaps the most tangible open problem related to this work is to improve the runtime for computing a dominance matrix. Our intuition is that the following is true.

CONJECTURE: The dominance matrix for a set of n points in n dimensions is computable in $\tilde{O}(n^\omega)$ time.

We believe that one might prove the conjecture by invoking recursion on the two subproblems (the matrices C and E) of the current algorithm in an interesting way.

Note added in camera-ready: We have recently found a sub-cubic, deterministic strongly polynomial algorithm for max and min weight triangle. It runs in $\tilde{O}(n^{2.575})$ time, and uses *rectangular* matrix multiplication. However, this algorithm is more specialized, and does not extend to dominance computations, estimating the distance product, or computing a buyer-seller matching. Independently of us, Raphael Yuster [15] found an algorithm similar to our new one, as well as several extensions. The details will appear in later work [14].

8. ACKNOWLEDGEMENTS

We thank Raphael Yuster and the anonymous referees for their useful comments. In particular, one referee graciously suggested the possibility of a strongly polynomial algorithm for max-weight triangle.

9. REFERENCES

[1] N. Alon, R. Yuster, and U. Zwick. Color-Coding. *JACM* 42(4):844–856, 1995.

[2] T. M. Chan. All-Pairs Shortest Paths with Real Weights in $O(n^3/\log n)$ Time. In *Proc. WADS*, Springer-Verlag LNCS 3608, 318–324, 2005.

[3] D. Coppersmith and S. Winograd. Matrix Multiplication via Arithmetic Progressions. *JSC* 9(3):251–280, 1990.

[4] D. Gale and L. S. Shapley. College admissions and the stability of marriage. *Amer. Math. Monthly* 69:9–15, 1962.

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

[6] R. W. Floyd. Algorithm 97: Shortest path. *Comm. ACM* 5:345, 1962.

[7] X. Huang and V. Y. Pan. Fast rectangular matrix multiplication and applications. *Journal of Complexity*, 14(2):257–299, 1998.

[8] A. Itai and M. Rodeh. Finding a minimum circuit in a graph. *SIAM J. Computing*, 7(4): 413-423, 1978.

[9] L. R. Kerr. The effect of algebraic structure on the computational complexity of matrix multiplications. Ph.D. Thesis. Cornell University, Ithaca, NY, 1970.

[10] J. Matousek. Computing dominances in E^n. *Inf. Proc. Letters* 38(5): 277–278, 1991.

[11] J. Plehn and B. Voigt. Finding Minimally Weighted Subgraphs. In *Proc. 16th Int. Workshop on Graph-Theoretic Concepts in Comp. Sci.* (WG'90), Springer-Verlag, 1991.

[12] R. Seidel. On the all-pair-shortest-path problem. In *Proceedings of the 24th ACM Symposium on Theory of Computing* (STOC'92), 1992.

[13] A. Shoshan and U. Zwick. All Pairs Shortest Paths in Undirected Graphs with Integer Weights. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science* (FOCS'99), 1999.

[14] V. Vassilevska, R. Williams, and R. Yuster. Finding the smallest H-subgraph in real weighted graphs and related problems. Manuscript, in preparation.

[15] R. Yuster. Private communication.

[16] G. Yuval. An Algorithm for Finding All Shortest Paths Using $N^{2.81}$ Infinite-Precision Multiplications. *Inf. Proc. Letters* 4: 155–156, 1976.

[17] U. Zwick. All Pairs Shortest Paths using bridging sets and rectangular matrix multiplication. *JACM* 49(3):289–317, May 2002.