More Algorithms for All-Pairs Shortest Paths in Weighted Graphs

Timothy M. Chan
School of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1, Canada
tmchan@uwaterloo.ca

ABSTRACT

In the first part of the paper, we reexamine the *all-pairs shortest paths* (APSP) problem and present a new algorithm with running time approaching $O(n^3/\log^2 n)$, which improves all known algorithms for general real-weighted dense graphs and is perhaps close to the best result possible without using fast matrix multiplication, modulo a few $\log\log n$ factors.

In the second part of the paper, we use fast matrix multiplication to obtain truly subcubic APSP algorithms for a large class of "geometrically weighted" graphs, where the weight of an edge is a function of the coordinates of its vertices. For example, for graphs embedded in Euclidean space of a constant dimension d, we obtain a time bound near $O(n^{3-(3-\omega)/(2d+4)})$, where $\omega < 2.376$; in two dimensions, this is $O(n^{2.922})$. Our framework greatly extends the previously considered case of small-integer-weighted graphs, and incidentally also yields the first truly subcubic result (near $O(n^{3-(3-\omega)/4}) = O(n^{2.844})$ time) for APSP in real-*vertex-weighted* graphs, as well as an improved result (near $O(n^{(3+\omega)/2}) = O(n^{2.688})$ time) for the all-pairs *lightest* shortest path problem for small-integer-weighted graphs.

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems

General Terms

Algorithms, Theory

Keywords

shortest paths, graph algorithms, computational geometry, matrix multiplication

1. INTRODUCTION

The all-pairs shortest paths (APSP) problem is unquestionably one of the most well known problems in algorithm design, frequently studied in textbooks; yet, the complexity of the problem has remained open to this day. For arbitrary dense (directed and undirected) real-weighted graphs with n vertices, the classical Floyd–Warshall algorithm [12] runs in $O(n^3)$ time. Fredman [16] was the first to realize the possibility of a subcubic algorithm, and since improvements have appeared in a number of papers; Table 1 summarizes the fascinating history. Notable among the more recent results are the $O(n^3/\log n)$ algorithm of this author [8], which is based on a simple *geometric* approach and does not require explicit table lookup or word tricks; and the $O(n^3 \log^{5/4} \log n/\log^{5/4} n)$ algorithm of Han [18], which amazingly breaks the $O(n^3/\log n)$ barrier by exploiting sophisticated word-packing tricks (implementable by table lookups), and is the best result known to date.

(We have ignored APSP algorithms for sparse graphs in the above discussion. Repeated applications of Dijkstra's single-source algorithm [12], combined with Johnson's preprocessing step if negative weights are permitted, imply an $O(n^2\log n + mn)$ time bound for any graph with m edges; the first term has been lowered to $O(n^2\log\log n)$ and $O(n^2\alpha(m,n))$ for directed and undirected graphs respectively, by Pettie [26] and Pettie and Ramanchandran [27].)

The first main result of this paper (Section 2) is an even faster algorithm for dense graphs with a running time of $O(n^3\log^3\log n/\log^2 n)$. As in the previous algorithms, the model of computation is the standard RAM with $\log n$ word size, with the standard instruction set. The new algorithm interestingly blends elements from the previous algorithms by the author [8] and by Han [18]: we use a different (yet remarkably simple) geometric approach and combine it with more word-packing tricks. The result not only improves a long line of earlier work, but also likely approaches the limit of what purely "combinatorial" algorithms can accomplish for the following reason: the Boolean matrix multiplication problem is a special case of APSP, and the fastest Boolean matrix multiplication algorithm known that does not rely on "algebraic" techniques (e.g., as used in Strassen's or Coppersmith and Winograd's algorithm) is still the classical "four-Russians" algorithm [6] from the 70s, with running time $O(n^3/\log^2 n)$. (The fastest known purely combinatorial algorithm for APSP for undirected unweighted graphs also takes $O(n^3/\log^2 n)$ time, by combining Feder and Motwani's graph compression technique [15] with one of the author's

time	ref.	year
$O(n^3)$	Dijkstra/Floyd–Warshall	1959/1962
$O(n^3 \log^{1/3} \log n / \log^{1/3} n)$	Fredman [16]	1976
$O(n^3 \log^{1/2} \log n / \log^{1/2} n)$	Takaoka [32]	1992
$O(n^3 / \log^{1/2} n)$	Dobosiewicz [14]	1990
$O(n^3 \log^{5/7} \log n / \log^{5/7} n)$	Han [17]	2004
$O(n^3 \log^2 \log n / \log n)$	Takaoka [33]	2004
$O(n^3 \log^{1/2} \log n / \log n)$	Zwick [38]	2004
$O(n^3 / \log n)$	Chan [8]	2005
$O(n^3 \log^{5/4} \log n / \log^{5/4} n)$	Han [18]	2006
$O(n^3 \log^3 \log n / \log^2 n)$	this paper	2007

Table 1: APSP algorithms for general dense real-weighted graphs.

methods for sparser graphs [9].) Thus, our result possibly marks the final chapter on the history of slightly subcubic APSP algorithms—that is, if one isn't too particular about $\log \log n$ factors.

The most tantalizing question in the area is whether in general the APSP problem could be solved in truly subcubic time ($O(n^{3-\delta})$ for some specific constant $\delta > 0$), by using fast matrix multiplication algorithms (like Strassen's or Coppersmith and Winograd's) as subroutines. Regrettably, this question will remain unanswered here. Nevertheless, in the second part of the paper (Section 3), we will describe new results along these lines for an important class of special cases.

At previous STOCs and FOCSs [4, 30, 31, 37], the case of graphs with small integer weights was addressed, where the weights lie in the range $\{1, \ldots, c\}$ for a constant c. In this case, Alon, Galil, and Margalit [4] gave an $\widetilde{O}(n^\omega)$ algorithm for undirected graphs and an $\widetilde{O}(n^{(3+\omega)/2}) = O(n^{2.688})$ algorithm for directed graphs. Here, $\omega < 2.376$ denotes the matrix multiplication exponent, and the $\widetilde{O}$ notation hides polylogarithmic and, in some cases, n^ε factors for an arbitrarily small constant $\varepsilon > 0$. The bound for directed graphs was improved to $O(n^{2.575})$ by Zwick [37], while the dependence of the constant factor on c for undirected graphs was improved by Shoshan and Zwick [31].

In this paper we investigate another class of (directed and undirected) weighted graphs that naturally arise in applications. In our framework, we assume that each vertex is associated with a constant number of parameters (coordinates), and that if an edge e exists from u to v, the weight of e is determined by evaluating a fixed algebraic function at the parameters at u and v (see Section 3 for a more precise formulation). Slightly more generally, we allow the weight of e to be determined by one of c possible functions, for any constant c (for simplicity, we ignore dependence of hidden constant factors on c, but they are of the form $c^{O(1)}$). Under these assumptions, we describe an algorithm with running time $\widetilde{O}(n^{3-(3-\omega)/(2\kappa+2)})$, where κ is a specific constant that depends on the particular family of functions (again see Section 3 for the precise definition of κ). The algorithm is obtained by using fast matrix multiplication, like previous algorithms, but with a somewhat different strategy, combined with geometric range searching techniques.

We also describe improvements in further special cases: If the ratio of the largest weight to the smallest weight is bounded by a constant, the running time can be reduced to $\widetilde{O}(n^{3-(3-\omega)/(\kappa+1)})$. Without this ratio assumption, we can also obtain an $\widetilde{O}(n^{3-(3-\omega)/(\kappa+1)})$ algorithm for the related problem of finding the shortest (i.e., minimum-weight) cycle in the graph.

To illustrate the generality of our framework, we mention a few consequences and relationships with known results:

- The prototypical instance of our framework is the Euclidean case, where points are embedded in two dimensions and the weight of an edge, if it exists, is taken to be the straight-line distance between its endpoints. (Edges may cross.) This is arguably one of the most natural settings for the APSP problem. In this case, $\kappa = 3$ and we obtain an $O(n^{2.922})$ algorithm. In a higher constant dimension d, the bound becomes $\widetilde{O}(n^{3-(3-\omega)/(2d+4)})$.

  Euclidean graphs and geometric graphs of course are of considerable interest and have been studied extensively in the literature through the years; for example, see [29] for an early paper about APSP in such graphs. For a more recent example, Narasimhan and Smid [25] investigated the problem of approximating the *stretch factor* of a Euclidean graph; our result immediately implies a subcubic *exact* algorithm for the stretch factor.

  It should be mentioned that the input graph need not be complete. Otherwise, Chan and Efrat [10] have already shown how to solve the single-source shortest path problem for geometric weight functions in subquadratic time, by using only range searching techniques without fast matrix multiplication; this automatically implies a (truly) subcubic algorithm for APSP. If the existence of an edge between two given points is also determined by some fixed algebraic expression evaluated at the points, then again the same previous techniques apply. Thus, the main feature behind the new algorithm is its ability to handle situations where the weights are geometrically determined but the graph (the edge set) itself is arbitrary.

- Our framework clearly extends the previously studied case of small integer weights, as we can simply take the c possible functions to be constants. Here, $\kappa = 1$ and we get a time bound of $\widetilde{O}(n^{(3+\omega)/2}) = O(n^{2.688})$, which is not as good as Zwick's algorithm [37] (but matches Alon *et al.*.'s original result [4]). However, unlike the previous algorithms for small integer weights, our framework applies more generally to the case where weights are taken from any fixed set of c real values—with the same running time $O(n^{2.688})$ if the maximum-to-minimum ratio is bounded, and $\widetilde{O}(n^{3-(3-\omega)/4}) = O(n^{2.844})$ if not. Even more generally, the same can be said for the case where the number of distinct weights out of each vertex is at most c.

- As another byproduct, we improve the main result of Zwick's STOC'99 paper [36] on the *all-pairs lightest shortest path* (APLSP) and the *all-pairs shortest lightest path* (APSLP) problem: for each pair (i, j), among all shortest paths from i to j, we seek one with the smallest length; or similarly, for each pair (i, j), among all paths from i to j of the shortest length, we seek one with the smallest weight. Zwick showed that both problems can be solved in $O(n^{2.747})$ time for directed graphs with weights from $\{1, \ldots, c\}$. In this case, we can immediately solve both problems by running our APSP algorithm, with the weight of each edge changed from x to $x+\delta$ for APLSP, or to $1 + \delta x$ for APSLP, for a sufficiently small fixed value $\delta = O(1/n)$; since the number of distinct edge weights and the maximum-to-minimum weight ratio are bounded by a constant, we get an $O(n^{2.688})$ algorithm, which is faster than Zwick's algorithm.

- Very recently, *vertex-weighted* graphs have been the subject of several papers (where the weight of a path/cycle is defined as the sum of the weights of its vertices). At last year's STOC, Vassilevska and Williams [34] showed that a problem related to APSP, namely, the minimum-weight triangle problem, can be solved in truly subcubic time, namely, $O(n^{2.688})$ time, for arbitrary real vertex weights. Subsequently, Vassilevska, Williams, and Yuster [35] improved this bound to $O(n^{2.575})$, and Czumaj and Lingas [13] further to $O(n^{2.376})$. Related problems have also been considered (like finding fixed subgraphs besides triangles), but we are not aware of any nontrivial results for cycles of length beyond 3, or for APSP itself. Our framework includes vertex-weighted graphs as special cases with $\kappa = 1$, and implies an $O(n^{2.844})$ algorithm for APSP and an $O(n^{2.688})$ algorithm for finding the shortest cycle of any fixed-constant length, or for finding the shortest cycle of unrestricted length, for arbitrary vertex-weighted graphs. This extends significantly, in a way, the original result of Vassilevska and Williams.

2. APSP FOR GENERAL GRAPHS IN NEAR $O(n^3/\log^2 n)$ TIME

It is well known [2] that the APSP problem for an arbitrary real-weighted graph with n vertices can be reduced to the problem of computing the *distance product* of two arbitrary real-valued, square $n \times n$ matrices (also known as the "min-plus matrix multiplication problem"): given two matrices $A = \{a_{ij}\}_{i,j}$ and $B = \{b_{ij}\}_{i,j}$, their distance product is defined as the matrix $C = \{c_{ij}\}_{i,j}$ with

$$c_{ij} := \text{the index } k \text{ that minimizes } a_{ik} + b_{kj}.$$

We denote this matrix by $A * B$. The reduction is done via a clever recursion and does not increase the asymptotic running time (if it exceeds $n^{2+\varepsilon}$).

In the author's previous paper [8], we observe that the problem of computing the distance product of a rectangular $n \times d$ matrix and a rectangular $d \times n$ matrix can be viewed as a geometric range searching problem and can be solved in $O(n^2)$ time for dimensions up to $d \approx \log n$ by using known techniques from computational geometry (specifically, a simple divide-and-conquer algorithm for computing *dominance* [28]). The distance product of two $n \times n$ matrices can then be solved in $O(n^2 \cdot n/d) = O(n^3/\log n)$ time by performing n/d such rectangular products and taking the element-wise minimum of the resulting matrices.

Here, we take a different geometric view of the problem of computing the distance product of $n \times d$ and $d \times n$ matrices (by thinking in terms of so-called *cuttings*), and propose a new solution for dimensions up to $d \approx \log n/\log\log n$. Although the value of d is marginally worse than the previous geometric approach, the rectangular product actually is computed in *sub*quadratic time! (The previous approach does not share this feature.) It is this feature that leads to the ultimate improvement for APSP.

How is it possible to compute the rectangular product in $o(n^2)$ time when the product itself has n^2 entries? Entries of this matrix are small integers from $\{1, \ldots, d\}$, so we can pack multiple entries in a single word. To be precise, let w denote the word size ($w = \Omega(\log n)$). We can store a list of n integers from $\{1, \ldots, d\}$ in *compressed* form in $O((n/w) \log d)$ words, with $\lfloor w/\log d \rfloor$ elements per word. Similarly, we can store a matrix $C = \{c_{ij}\}_{i,j\in\{1,\ldots,n\}}$ of integers from $\{1, \ldots, d\}$ as a compressed list $\langle c_{11}, \ldots, c_{1n}, \ldots, c_{n1}, \ldots, c_{nn}\rangle$, in $O((n^2/w) \log d)$ words, which is subquadratic for small d. We will describe an algorithm that requires time proportional to the number of words.

In the following, we will express running times in terms of both n and w and assume that certain nonstandard operations on words of size w can be performed in constant time. Although this assumption may not be "reasonable", it is without loss of generality if we set $w = \delta \log n$, for a constant $\delta > 0$. This is because we can implement table lookup on the standard RAM, i.e., we can precompute a table storing the outputs for all possible combinations of inputs in time $2^{O(w)}$, which is sublinear for a sufficiently small δ. To simplify presentation, we will ignore degenerate cases.

2.1 Geometry

We begin with a well-known geometric *cutting* lemma that is useful for solving range-searching-type problems. We state just one (very!) special case which is sufficient for our purposes, and include a proof to make the presentation self-contained.

Lemma 2.1. *Given a set H of n hyperplanes in $\mathbb{R}^d$ whose normals are parallel to c different directions, we can divide $\mathbb{R}^d$ into $O(c^d r^d)$ cells so that each cell intersects at most n/r hyperplanes.*

In addition, given a set P of n points, we can do the following in $O(cn \log n + c^d nr^{d-1})$ time: for each cell Δ that contains at least one point, we can compute the set P_Δ of all points of P inside Δ and a superset H_Δ of size $O(n/r)$ containing all hyperplanes of H intersecting Δ.

Proof. For each direction ξ, let H_ξ be the subset of hyperplanes orthogonal to ξ, sorted along ξ; put the $\lfloor |H_\xi|/r \rfloor$-th, $(\lfloor 2|H_\xi|/r \rfloor)$-th, ... hyperplanes of H_ξ in a subset R. Then $|R| \le cr$. Simply take the $O(|R|^d)$ cells of the arrangement of R. Every cell of the arrangement clearly intersects at most $\sum_\xi |H_\xi|/r = n/r$ hyperplanes.

For our purposes, it is not necessary to construct explicitly the arrangement of R, nor the polyhedral description of the cells. We can assign points of P to their corresponding cells directly by sorting the points and the hyperplanes along each direction ξ in time $O(cn\log n)$ (this can actually be reduced to $O(cn\log r)$ with more care). A list H_Δ can be easily generated in $O(n/r)$ time for each of the $O(c^d r^d)$ cells Δ. $\square$

Remark: By more powerful techniques from computational geometry [11, 24], the same result (ignoring dependences of the constant factors on d) actually holds for arbitrary hyperplanes with no restrictions on directions.

We now describe a new but surprisingly simple way of looking at the rectangular product problem geometrically, which leads to a slightly subquadratic algorithm:

Theorem 2.2. *Set $d = \delta\log n/\log w$ for a sufficiently small constant $\delta > 0$. Given an $n\times d$ real matrix A and a $d\times n$ real matrix B, we can compute the compressed matrix $C = A*B$ in $O((n^2/w)\log d)$ time.*

Proof. Form a set of n points $P = \{p_i\}_{i\in\{1,\ldots,n\}}$ from the matrix A, by letting

$$p_i \ := \ (a_{i1},\ldots,a_{id}).$$

Form a set of $O(d^2 n)$ hyperplanes $H = \{h_{jk\ell}\}_{j\in\{1,\ldots,n\},k,\ell\in\{1,\ldots,d\}}$ from the matrix B, by letting

$$h_{jk\ell} \ := \ \{(x_1,\ldots,x_d)\in\mathbb{R}^d \mid x_k + b_{kj} = x_\ell + b_{\ell j}\}.$$

Observe that for a connected region Δ and index j,

> if $h_{jk\ell}$ does not intersect Δ for all k and ℓ, then the index k that minimizes $x_k + b_{kj}$ is the same for all $(x_1,\ldots,x_d)\in\Delta$.

This observation suggests the following algorithm.

Apply Lemma 2.1 to H to obtain $d^{O(d)}r^d$ cells, as well as the lists P_Δ and H_Δ, in $O(d^{O(1)}n\log n + d^{O(d)}nr^{d-1})$ time (noting that the hyperplanes in H admit only $O(d^2)$ directions). For each cell Δ containing at least one point and for each $j = 1,\ldots,n$, arbitrarily choose a point $(x_1,\ldots,x_d)\in P_\Delta$ and set $c_{\Delta j}$ to the index $k\in\{1,\ldots,d\}$ that minimizes $x_k + b_{kj}$ (computable in $O(d)$ time each); the total time for this step is at most $d^{O(d)}nr^d$.

Fix a point $p_i\in P_\Delta$. Set the i-th row tentatively to $\langle c_{i1},\ldots,c_{in}\rangle := \langle c_{\Delta 1},\ldots,c_{\Delta n}\rangle$; this step takes time proportional to the length of the row, i.e., $O((n/w)\log d)$ time per point p_i. For each $h_{jk\ell}\in H_\Delta$, correct the value of c_{ij} by computing the actual index $k\in\{1,\ldots,d\}$ that minimizes $a_{ik}+b_{kj}$ (in $O(d)$ time) and resetting c_{ij} to this index; since $|H_\Delta|\le d^{O(1)}n/r$, the cost is $d^{O(1)}n/r$ per point p_i. At the end, all c_{ij} values would be correct.

The total time is

$$O(d^{O(d)}nr^d \ + \ d^{O(1)}n^2/r \ + \ (n^2/w)\log d).$$

Setting $r = n^{1/(d+1)}/d^c$ for a sufficiently large constant c yields a time bound of

$$O(d^{O(1)}n^{2-1/(d+1)} \ + \ (n^2/w)\log d).$$

The second term dominates for $d = \delta\log n/\log w$ for a sufficiently small δ. $\square$

Remarks: Compared to the geometric approach in [8], the new approach is more direct and does not need divide-and-conquer.

Although the geometric interpretation is important in the analysis (in regards to bounding the number of cells in the arrangement of R), the algorithm itself can actually be described entirely without any reference to geometry. In particular, the only primitive operations on the real numbers required are comparisons of differences/sums of pairs, as in all previous algorithms from Table 1.

2.2 Word Tricks

In order to use Theorem 2.2 effectively to compute the distance product of square matrices, we need to tackle one remaining issue: how to take the element-wise minimum of matrices in subquadratic time. To be precise, given two $n\times n$ matrices $X = \{x_{ij}\}_{i,j}$ and $Y = \{y_{ij}\}_{i,j}$ whose entries are indices, we want to compute the matrix $C = \{c_{ij}\}_{i,j}$ with

$$c_{ij} \ := \ \begin{cases} x_{ij} & \text{if } a_{ix_{ij}} + b_{x_{ij}j} \le a_{iy_{ij}} + b_{y_{ij}j} \\ y_{ij} & \text{otherwise.}\end{cases}$$

We denote this matrix by $X \wedge_{A,B} Y$.

This subproblem is encountered also in Han's paper [18], but his solution appears to be designed specifically for the matrices X and Y generated within his algorithm. The solution we describe below is more general, although it is marginally slower (costing another $\log\log n$ factor) and is inspired by the ideas outlined by Han.

We first state a few handy subroutines on manipulating compressed lists (some are well known, e.g., see [3] concerning (a)).

Lemma 2.3. *Given compressed lists $X = \langle x_1,\ldots,x_m\rangle$ and $Y = \langle y_1,\ldots,y_m\rangle$, where $x_i, y_i\in\{1,\ldots,u\}$,*

(a) we can sort X in $O(\lceil m/w\rceil\log u\log m)$ time;

(b) we can compute the list $\langle(x_1,y_1),\ldots,(x_m,y_m)\rangle$ in $O(\lceil m/w\rceil\log u)$ time;

(c) for a fixed mapping $f:\{1,\ldots,u\}\to\{1,\ldots,u\}$, we can compute the list $\langle f(x_1),\ldots,f(x_m)\rangle$ in $O(\lceil m/w\rceil\log u)$ time;

(d) for any given array $f[1],\ldots,f[u]\in\{1,\ldots,u\}$, we can compute the list $\langle f[x_1],\ldots,f[x_m]\rangle$ in $O(\lceil m/w\rceil\log(mu)\log m + u)$ time.

Proof.

(a) Let $\bar{w} = \lfloor w/\log u\rfloor$. We first show how to merge two sorted compressed lists X and Y by imitating the standard linear-time merging algorithm: In each iteration, we grab the next $\bar{w}$ elements of X, stored in a word z_1, and the next $\bar{w}$ elements of Y, stored in a word z_2. By a new word operation on z_1 and z_2, we can obtain a word z containing the $\bar{w}$ smallest elements in sorted order among the elements in z_1 and z_2. By another word operation on z_1 and z_2, we can also obtain the number j_1 (resp. j_2) of elements of z_1 (resp. z_2) that appear in z. After shifting over j_1 elements in X and j_2 elements in Y, we can continue to the next iteration. The running time is $O(\lceil m/\bar{w}\rceil)$.

Now, we show how to sort a compressed list X by imitating mergesort: just divide X into two sublists of $m/2$ elements, recursively sort each sublist, and merge. The running time is $O(\lceil m/\bar{w}\rceil \log m)$.

(b) Reset $\bar{w} = \lfloor w/(2\log u)\rfloor$. We simply apply the following word operation $m/\bar{w}$ times: given a word $\langle x_1, \ldots, x_{\bar{w}}\rangle$ and $\langle y_1, \ldots, y_{\bar{w}}\rangle$, output the word $\langle (x_1, y_1), \ldots, (x_{\bar{w}}, y_{\bar{w}})\rangle$.

(c) We simply apply the following word operation $m/\bar{w}$ times: given a word $\langle x_1, \ldots, x_{\bar{w}}\rangle$, output the word $\langle f(x_1), \ldots, f(x_{\bar{w}})\rangle$.

(d) Reset $\bar{w} = \lfloor w/\log(mu)\rfloor$. We first form the list of pairs $\{(i, x_i)\}_{i\in\{1,\ldots,m\}}$ by (b), and sort the list by (a) using x_i as the key.

We then split the sorted list into at most u sublists so that elements within the same sublists have the same x_i value. This can be done by repeated applications of the following word operation (and repeated shifting): given a word $\langle (i_1, x_{i_1}), \ldots, (i_{\bar{w}}, x_{i_{\bar{w}}})\rangle$, output the largest index j such that $x_{i_1} = \cdots = x_{i_j}$.

For each sublist $\langle (i_1, x), (i_2, x), \ldots\rangle$, change it to $\langle i_1, i_2, \ldots\rangle$ by (c), look up the value of $z = f[x]$, and change the sublist to $\langle (i_1, z), (i_2, z), \ldots\rangle$ by (b).

We then sort the union of the sublists by (a), this time, using i as the key, and finally map each element (i, z) to z by (c). $\square$

We now provide a slightly subquadratic algorithm for computing the element-wise minimum of two matrices:

Theorem 2.4. *Suppose $m \geq q^2 w$. Given an $m \times q$ matrix A and a $q \times m$ matrix B with real entries, and compressed $m \times m$ matrices X and Y with entries from $\{1, \ldots, q\}$, we can compute the compressed matrix $C = X \wedge_{A,B} Y$ in $O((m^2/w)\log^2 m)$ time.*

Proof. Sort the $O(mq^2)$ elements $a_{ik} - a_{i\ell}$ and $b_{j\ell} - b_{jk}$ over all $i, j \in \{1, \ldots, m\}$ and $k, \ell \in \{1, \ldots, q\}$; this step takes $O(mq^2 \log(mq))$ time. Let $\alpha[i, k, \ell]$ be the rank of $a_{ik} - a_{i\ell}$ and $\beta[j, k, \ell]$ be the rank of $b_{\ell j} - b_{kj}$ in the sorted list; these ranks are integers between 1 and $O(mq^2)$. Observe that

$$a_{ik} + b_{kj} \leq a_{i\ell} + b_{\ell j} \qquad \text{iff} \qquad \alpha[i, k, \ell] \leq \beta[j, k, \ell].$$

This observation suggests the following algorithm.

First generate the (compressed) $m \times m$ matrices $\{(i, x_{ij}, y_{ij})\}_{i,j}$ and $\{(j, x_{ij}, y_{ij})\}_{i,j}$ by Lemma 2.3(b) in $O((m^2/w)\log m)$ time. Next compute $\{\alpha_{ij} := \alpha[i, x_{ij}, y_{ij}]\}_{i,j}$ and $\{\beta_{ij} := \beta[j, x_{ij}, y_{ij}]\}_{i,j}$ by Lemma 2.3(d) in $O((m^2/w)\log^2 m + mq^2)$ time, since the number of distinct tuples $[i, x_{ij}, y_{ij}]$ is $O(mq^2)$; the first term dominates for $m \geq q^2 w$. Now, generate the matrix $\{(\alpha_{ij}, \beta_{ij}, x_{ij}, y_{ij})\}_{i,j}$ by Lemma 2.3(b) and map $(\alpha_{ij}, \beta_{ij}, x_{ij}, y_{ij})$ to x_{ij} if $\alpha_{ij} \leq \beta_{ij}$, and y_{ij} otherwise, by Lemma 2.3(c) in $O((m^2/w)\log m)$ time. $\square$

Corollary 2.5. *Suppose $n \geq q^2 w$. Given an $n \times q$ matrix A and a $q \times n$ matrix B with real entries, and compressed $n \times n$ matrices X and Y with entries from $\{1, \ldots, q\}$, we can compute the compressed matrix $C = X \wedge_{A,B} Y$ in $O((n^2/w)\log^2(qw))$ time.*

Proof. Set $m = q^2 w$. Divide X and Y into $O((n/m)^2)$ $m \times m$ submatrices, and apply Theorem 2.4 to every corresponding pair. The total time is $O((n/m)^2 \cdot (m^2/w)\log^2 m)$. $\square$

2.3 The APSP Algorithm

We can now put Theorem 2.2 and Corollary 2.5 together to get the final result.

Corollary 2.6. *Suppose $w = o(n^{1/3})$. Set $q = w\log n/\log^3 w$. Given an $n \times q$ matrix A and a $q \times n$ matrix B, we can compute $C = A * B$ in $O(n^2)$ time.*

Proof. Divide A into $n \times d$ matrices $A_1, \ldots, A_{q/d}$ and B into $d \times n$ matrices $B_1, \ldots, B_{q/d}$. Compute $C_i = A_i * B_i$ for all i by Theorem 2.2. Compute $C = C_1 \wedge_{A,B} \cdots \wedge_{A,B} C_{q/d}$ by Theorem 2.4. The total time is $O((q/d) \cdot (n^2/w)\log^2(qw))$, which is $O(n^2)$ for $q = O(dw/\log^2 w)$. (We can uncompress the resulting matrix with $O(n^2)$ additional time.) $\square$

Corollary 2.7. *Suppose $w = o(n^{1/3})$. Given $n \times n$ matrices A and B, we can compute $C = A * B$ in $O(n^3(\log^3 w)/(w\log n))$ time. Consequently, we can solve the APSP problem for arbitrary real-weighted directed graphs with n vertices within the same time bound.*

Proof. Divide A into $n \times q$ matrices $A_1, \ldots, A_{n/q}$ and B into $q \times n$ matrices $B_1, \ldots, B_{n/q}$. Compute $C_i = A_i * B_i$ for all i by Corollary 2.6 in $O(n^2 \cdot n/q)$ total time. Compute $C = C_1 \wedge_{A,B} \cdots \wedge_{A,B} C_{n/q}$ naively in $O(n^2 \cdot n/q)$ time. $\square$

3. APSP FOR GEOMETRIC GRAPHS IN TRULY SUBCUBIC TIME

In the previous section, we have used low-dimensional geometric techniques to solve a graph problem. In this section, we turn to geometric special cases of the graph problem.

Let $\mathcal{W}$ be a fixed set of c piecewise-algebraic functions over $\mathbb{R}^d \times \mathbb{R}^d$, each with a constant degree and a constant number of pieces, where the dimension d is now assumed to be a constant, and so is c. Consider a graph G, where the vertices are points $p_1, \ldots, p_n \in \mathbb{R}^d$, and the weight of the edge from p_i to p_j is chosen from the set $\{w(p_i, p_j) \mid w \in \mathcal{W}\} \cup \{\infty\}$. Such a graph G, and the corresponding weight matrix, are said to be *geometrically weighted*. We present a truly subcubic algorithm for APSP in geometrically weighted graphs.

As in several previous APSP algorithms for integer-weighted graphs (e.g., [4, 37]), the first idea is to divide shortest paths into two categories: those having lengths larger than a certain parameter ℓ, and those having lengths smaller than ℓ. Paths of large lengths are hit by a small subset of vertices and can therefore be found quickly. Previous algorithms like [4, 37] handle paths of small lengths by repeated-squaring-like strategies, but such strategies do not work here, because the square of a geometrically weighted matrix is no longer geometrically weighted.

We propose a different strategy: we show how to compute the distance product of a geometrically weighted matrix with an *arbitrary* matrix; paths of small lengths can then be found by computing such a product ℓ times. Although the resulting algorithm is not as good as previous algorithms in the case of small integer weights, it is more general; and in at least one scenario (namely, the APLSP problem), this strategy actually beats repeated squaring [36].

In the following, we abuse notation slightly and redefine $A * B$ to be the matrix $C = \{c_{ij}\}_{i,j}$ with $c_{ij} = \min_k(a_{ik} + b_{kj})$. Let $A \wedge B$ be the matrix $C = \{c_{ij}\}_{i,j}$ with $c_{ij} = \min\{a_{ij}, b_{ij}\}$. Let $\delta_G(p_i, p_j)$ denote the shortest path distance from i to j. For simplicity, we will only describe how to compute shortest-path distances; it is a straightforward matter to modify our algorithms to generate the predecessor matrix used for retrieving the shortest paths. In this section we will work with the real-RAM model, as is standard in the computational geometry literature; in particular, we will ignore thorny issues about sums of square roots in the case of Euclidean distances.

3.1 Geometry

We begin with another well-known geometric tool, a *partition theorem*, which plays a central role in range searching. Matoušek [23] established the original version; the version we need below follows from Agarwal and Matoušek's work on semialgebraic range searching [1].

Lemma 3.1. *There is a specific constant κ for which the following statement holds: Given any set P of n points in $\mathbb{R}^{d+1}$ and parameter $1 \le r \le n$, we can partition P into r subsets $P_1, \ldots, P_r$, each of size $O(n/r)$, and find r cells $\Delta_1 \supset P_1, \ldots, \Delta_r \supset P_r$, each of complexity $O(1)$, such that any surface of the form*

$$\{(x, z) \in \mathbb{R}^{d+1} \mid w(p, x) + z = c_0\} \quad (w \in \mathcal{W},\ p \in \mathbb{R}^d,\ c_0 \in \mathbb{R})$$

intersects at most $\widetilde{O}(r^{1-1/\kappa})$ cells. The subsets and cells can be constructed in $\widetilde{O}(n)$ time.

The best possible value of κ depends on the combinatorial complexity of a decomposition of an arrangement of surfaces (currently an open problem in general for dimensions greater than 4). Let d_{act} be the *actual dimension*, i.e., the number of variables that appear in the expression $w(p, x) + z$ for a fixed p and a fixed piece of a function $w \in \mathcal{W}$. Let d_{lin} be the *linearization dimension* [1] (roughly speaking, the number of variables needed to transform the expression $w(p, x) + z \le c$ into linear inequalities). It is known [1, 21] that the following value of κ works:

$$\kappa = \begin{cases} d_{\text{act}} & \text{if } d_{\text{act}} \le 4 \\ \min\{\lfloor (d_{\text{act}} + d_{\text{lin}})/2 \rfloor,\ 2d_{\text{act}} - 4\} & \text{otherwise.} \end{cases}$$

For example, suppose that $w(\cdot, \cdot)$ is the Euclidean distance function. Then $d_{\text{act}} = d + 1$, since $w(p, x) + z$ depends on $d+1$ variables (x, z); and $d_{\text{lin}} = d+2$, since $w(p, x) + z \le c$ can essentially be rewritten as $\|x - p\|^2 \le (c - z)^2$, i.e., $\|x\|^2 - z^2 - 2p \cdot x - 2cz \le c^2$, which is linear in the $d + 2$ variables $(x, z, \|x\|^2 - z^2)$. Thus, $\kappa = d + 1$.

For vertex-weighted graphs, each vertex is a point in $\mathbb{R}^1$ and $w(p, q) = p$, so $w(p, x) + z$ depends only on one variable z. Thus, $\kappa = d_{\text{act}} = 1$. For L_1 or L_∞ distances, $\kappa = 1$ similarly.

We now present a truly subcubic algorithm for computing the distance product of a geometrically weighted matrix and an arbitrary matrix, by combining fast matrix multiplication with the use of the partition theorem.

Theorem 3.2. *Given a geometrically weighted $n \times n$ matrix A and an arbitrary $n \times n$ matrix B, we can compute $C = A * B$ in $\widetilde{O}(n^{3-(3-\omega)/(\kappa+1)})$ time.*

Proof. For simplicity, assume that $\mathcal{W}$ consists of a single function $w(\cdot, \cdot)$. Let $p_1, \ldots, p_n \in \mathbb{R}^d$ be the points that define A. For each $j \in \{1, \ldots, n\}$, apply Lemma 3.1 to the point set $\{(p_k, b_{kj})\}_{k \in \{1,\ldots,n\}}$ to obtain r subsets $\{P_{\ell j}\}_\ell$ of size $O(n/r)$ and r cells $\{\Delta_{\ell j}\}_\ell$; the total time so far is $\widetilde{O}(n^2)$. Slightly abusing notation, we let $P_{\ell j}$ contain indices k rather than points (p_k, b_{kj}).

Let $S_i = \{k \mid a_{ik} = w(p_i, p_k)\}$. We want to compute $c_{ij} = \min_{k \in S_i}(w(p_i, p_k) + b_{kj})$ for each i and j.

For every $i, j \in \{1, \ldots, n\}, \ell \in \{1, \ldots, r\}$, first determine whether S_i intersects $P_{\ell j}$. This step reduces to multiplying an $n \times n$ Boolean matrix (whose rows correspond to bit vectors of the S_i's) and an $n \times nr$ Boolean matrix (whose columns correspond to bit vectors of the $P_{\ell j}$'s). Thus, this preprocessing can be done in $O(n^\omega r)$ time.

Now fix i and j. Let $\hat{c}_{ij}$ be the minimum of $\sup_{(x,z) \in \Delta_{\ell j}}(w(p_i, x) + z)$ over all ℓ such that $P_{\ell j}$ intersects S_i; this value is an upper bound on the actual value c_{ij} and can be computed in $O(r)$ time ($O(1)$ time per ℓ). Let γ_{ij} be the region $\{(x, z) \in \mathbb{R}^{d+1} \mid w(p_i, x) + z \le \hat{c}_{ij}\}$. Set c_{ij} to be the minimum of $w(p_i, p_k) + b_{kj}$ over all $k \in P_{\ell j} \cap S_i$ and over all ℓ with $\Delta_{\ell j}$ intersecting the boundary of γ_{ij}. Since the number of such ℓ's is $\widetilde{O}(r^{1-1/\kappa})$ and each $P_{\ell j}$ has $O(n/r)$ size, this step takes $\widetilde{O}(r^{1-1/\kappa} \cdot n/r) = \widetilde{O}(n/r^{1/\kappa})$ time.

To see why this correctly computes c_{ij}, note that since the actual value of c_{ij} is at most $\hat{c}_{ij}$, we can safely ignore all $k \in P_{\ell j}$ with $\Delta_{\ell j}$ strictly outside γ. On the other hand, we can ignore all $k \in P_{\ell j}$ with $\Delta_{\ell j}$ strictly inside γ, since by definition of $\hat{c}_{ij}$, such a subset $P_{\ell j}$ cannot intersect S_i.

The total running time is

$$\widetilde{O}(n^\omega r + n^3/r^{1/\kappa}).$$

Setting $r = n^{(3-\omega)/(1+1/\kappa)}$ yields the time bound. Note that if $\mathcal{W}$ contains c functions, we can run the above algorithm for each $w \in \mathcal{W}$ in turn and return the element-wise minimum of the results. □

Remarks: In the simple case of vertex-weighted graphs with $d = 1$, we of course do not need the partition theorem (we can divide a sorted list of n numbers in $\mathbb{R}^1$ into r sublists of size $O(n/r)$ in the obvious way). The resulting $\widetilde{O}(n^{(3+\omega)/2})$ algorithm bear superficial resemblance with some previous $\widetilde{O}(n^{(3+\omega)/2})$ algorithms, like Matoušek's dominance method [22].

The combination of geometric range searching techniques with fast matrix multiplication is rather interesting, although examples of such combinations have appeared before (e.g., see [7, 19, 20]).

Instead of the partition theorem, we can also prove Theorem 3.2 using a cutting lemma (like in the previous section). This alternative algorithm requires randomization but less machinery (since the original proof of the partition theorem has multiple steps, one of which use cuttings).

3.2 The APSP Algorithm

We need one more ingredient to deal with shortest paths of large lengths. The following lemma has been used in previous APSP algorithms (e.g., [4, 37]), and can be proved by random sampling, or deterministically by running the standard greedy algorithm for the hitting set problem (i.e., set cover in the dual).

Lemma 3.3. *Given a collection of N subsets of $\{1, \dots, n\}$ where each subset has size exactly ℓ, we can find a subset R of size $O((n/\ell)\log n)$ that hits all subsets in the collection, in $O(N\ell)$ time.*

We now derive an APSP algorithm from Theorem 3.2 and the preceding lemma:

Theorem 3.4. *We can solve the APSP problem for a geometrically weighted graph G with n vertices and m edges in $\widetilde{O}(n^{2-(3-\omega)/(2\kappa+2)}\sqrt{m}) = \widetilde{O}(n^{3-(3-\omega)/(2\kappa+2)})$ time.*

Proof. Let $A^{(1)} = A$ be the weight matrix of G. For each $s = 2, \dots, \ell$, compute $A^{(s)} = A * A^{(s-1)}$. This process requires $\ell - 1$ applications of Theorem 3.2 and produces the weight $a_{ij}^{(s)}$ of the shortest length-s path from i to j.

Next find a subset R that hits all $O(n^2)$ shortest length-ℓ paths found, by Lemma 3.3 in $O(n^2\ell)$ time. Let $B^{(0)} = B = \{b_{ij}\}_{i,j}$ where $b_{ij} := \delta_G(p_i, p_j)$ if $p_i \in R$, and ∞ otherwise. This matrix can be computed by $|R| = \widetilde{O}(n/\ell)$ applications of Dijkstra's single-source shortest path algorithm, each of which takes $O(n \log n + m)$ time.

For each $s = 1, \dots, \ell$, compute $B^{(s)} = A * B^{(s-1)}$. This requires another ℓ applications of Theorem 3.2.

Finally return $A^{(1)} \wedge \cdots \wedge A^{(\ell)} \wedge B^{(0)} \wedge \cdots \wedge B^{(\ell)}$. Correctness follows from the fact that a shortest path from p_i to p_j of length greater than ℓ can be decomposed into a shortest path from p_i to some $p_k \in R$ of length at most ℓ and a shortest path from p_k to p_j.

The total running time is

$$\widetilde{O}(\ell n^{3-(3-\omega)/(\kappa+1)} + mn/\ell).$$

Setting $\ell = \sqrt{m}/n^{1-(3-\omega)/(2\kappa+2)}$ yields the result. □

3.3 Further Special Cases

We show how to improve Theorem 3.4 in two special cases. The key new ingredient is a variant of Theorem 3.2 that is sensitive to the sparseness of one of the input matrices and the sparseness of the desired output:

Theorem 3.5. *Given a geometrically weighted $n \times n$ matrix A and an arbitrary $n \times n$ matrix B where B has $O(m)$ finite entries, we can compute any $O(m)$ specified entries of $C = A * B$ in $\widetilde{O}(n^\omega + mn^{1-(3-\omega)/(\kappa+1)})$ time.*

Proof. We modify the proof of Theorem 3.2. For each j, let $T_j = \{k \mid b_{kj} \neq \infty\}$. We apply Lemma 3.1 to the point set $\{(p_k, b_{kj})\}_{k \in T_j}$ but with a change of parameter: we still insist that each subset $P_{\ell j}$ has size $O(n/r)$, but the number of subsets is now reduced to $r_j := O(r|T_j|/n)$. Since the total number of subsets is $\sum_j r_j = O(rm/n)$, the preprocessing step—multiplying an $n \times n$ Boolean matrix and an $n \times rm/n$ matrix—can now be done in $O(\lceil rm/n^2 \rceil n^\omega) = O(n^\omega + rmn^{\omega-2})$ time. Like before, each entry c_{ij} can be computed in time $\widetilde{O}(r_j + r_j^{1-1/\kappa} \cdot n/r)$, which is at most $\widetilde{O}(r + n/r^{1/\kappa})$.

The total running time is

$$\widetilde{O}(n^\omega + rmn^{\omega-2} + mn/r^{1/\kappa}).$$

Setting $r = n^{(3-\omega)/(1+1/\kappa)}$ yields the result. □

The first special case is when the ratio of maximum to minimum weight is bounded by a constant c. We apply Theorem 3.5 in an interesting way to obtain this result:

Theorem 3.6. *We can solve the APSP problem for a geometrically weighted graph G with n vertices in $\widetilde{O}(n^{3-(3-\omega)/(\kappa+1)})$ time if all weights are between 1 and c.*

Proof. For each $s = 1, \dots, c\ell$, we will compute a matrix $A^{(s)} = \{a_{ij}^{(s)}\}_{i,j}$, where $a_{ij}^{(s)}$ is equal to $\delta_G(i,j)$ if the value is in the range $[s, s+1)$, and ∞ otherwise.

To do so, first let $A = \{a_{ij}\}_{i,j}$ be the weight matrix of G. Assuming that $A^{(1)}, \dots, A^{(s-1)}$ have been computed correctly, we generate $A^{(s)}$ as follows. Define $X^{(s)} = \{(i,j) \mid a_{ik}^{(s-b)} \neq \infty$ and $a_{kj} \in (b-1, b+1)$ for some k and for some $b \in \{1, \dots, c\}\}$. Computing $X^{(s)}$ reduces to c Boolean matrix multiplications and thus takes $O(n^\omega)$ time each. Define $Y^{(s)} = X^{(s)} - \{(i,j) \mid a_{ij}^{(t)} \neq \infty$ for some $t \in \{1, \dots, s-1\}\}$. By definition, this set $Y^{(s)}$ satisfies the following properties: (i) if $\delta_G(i,j) \in [s, s+1)$, then $(i,j) \in Y^{(s)}$; and (ii) if $(i,j) \in Y^{(s)}$, then $\delta_G(i,j) \in [s, s+2)$.

Now compute the product $A * (A^{(s-c)} \wedge \cdots \wedge A^{(s-1)})$ *restricted to* $Y^{(s)}$—i.e., for $(i,j) \notin Y^{(s)}$, the corresponding entry of the returned matrix need not be generated and is set to ∞. To obtain $A^{(s)}$, we take this resulting matrix further restricted to entries in the range $[s, s+1)$. By Theorem 3.2, the matrices $A^{(s)}$ for all $s = 1, \dots, c\ell$ can be computed in total time near

$$\begin{aligned} & \ell n^\omega + \sum_{s=1}^{c\ell} (|Y^{(s-c)} \cup \cdots \cup Y^{(s-1)}| + |Y^{(s)}|) n^{1-(3-\omega)/(\kappa+1)} \\ = \;& \widetilde{O}(\ell n^\omega + n^{3-(3-\omega)/(\kappa+1)}), \end{aligned}$$

since (ii) implies $\sum_s |Y^{(s)}| = O(n^2)$.

Next find a subset R that hits all shortest paths found of length exactly ℓ, by Lemma 3.3 in $O(n^2\ell)$ time. Let $B = \{b_{ij}\}_{i,j}$ and $B' = \{b'_{ij}\}_{i,j}$ where $b_{ij} := \delta_G(p_i, p_j)$ if $p_i \in R$ and ∞ otherwise, and $b'_{ij} := \delta_G(p_i, p_j)$ if $p_j \in R$ and ∞ otherwise. The two matrices can be computed by $|R| = \widetilde{O}(n/\ell)$ applications of Dijkstra's single-source/single-sink algorithm.

To finish, compute $B' * B$ naively in $O(n^2|R|)$ time and return $A^{(1)} \wedge \cdots \wedge A^{(c\ell)} \wedge (B' * B)$. Correctness follows from the fact that a shortest path from p_i to p_j of weight greater than $c\ell$ has length greater than ℓ and thus can be decomposed into a shortest path from p_i to some $p_k \in R$ and a shortest path from p_k to p_j.

The total running time is

$$\widetilde{O}(\ell n^\omega + n^{3-(3-\omega)/(\kappa+1)} + n^3/\ell).$$

Setting $\ell = n^{(3+\omega)/2}$ yields the result. □

Next, we consider the shortest cycle problem, which trivially reduces to APSP. We solve the cycle problem directly, using Theorem 3.5 and Lemma 3.3, to obtain a faster algorithm:

Theorem 3.7. *We can find the shortest (i.e., minimum-weight) cycle for a geometrically weighted graph G with n vertices with positive weights in $\widetilde{O}(n^{3-(3-\omega)/(\kappa+1)})$ time.*

Proof. Assume that ℓ is a power of 2. For each $t = 1, 2, 4, 8, \ldots, \ell$, we will compute a vertex subset $R^{(t)}$ of size $\widetilde{O}(n/t)$ that hits all shortest paths of length t, and for each $s = 1, \ldots, t$, we will compute the matrix $A^{(t,s)} = \{a_{ij}^{(t,s)}\}_{i,j}$, where $a_{ij}^{(t,s)}$ is the weight of the shortest length-s path from i to j if $i \in R^{(t/2)}$, and ∞ otherwise.

Given $R^{(t/2)}$, we can easily initialize $A^{(t,1)}$. We set $A^{(t,s)}$ to the product $A^{(t,s-1)} * A$ restricted to the $\widetilde{O}(n^2/t)$ entries in $\{(i,j) \mid i \in R^{(t/2)}\}$. By Theorem 3.5 (with the roles of A and B reversed), all the $A^{(t,s)}$ matrices can be generated in total time near

$$\sum_{t=1,2,4,\ldots,\ell} t \cdot [n^\omega + (n^2/t) \cdot n^{1-(3-\omega)/(\kappa+1)}]$$
$$= \widetilde{O}(\ell n^\omega + n^{3-(3-\omega)/(\kappa+1)}).$$

Having computed $A^{(t,t/2)}$, we set $R^{(t)}$ to be a subset of size $\widetilde{O}(n/t)$ that hits all $\widetilde{O}(n^2/t)$ shortest length-$(t/2)$ paths found that start at vertices in $R^{(t/2)}$, by Lemma 3.3 in $\widetilde{O}(n^2)$ time. Then indeed $R^{(t)}$ hits every shortest path π of length t, since the first $t/2$ vertices of π contain a vertex p_i of $R^{(t/2)}$ by induction, and the $t/2$ vertices of π after p_i contain a vertex of $R^{(t)}$.

Let $B = \{b_{ij}\}_{i,j}$ where $b_{ij} = \delta_G(i,j)$ if $i \in R^{(\ell)}$, and ∞ otherwise. This matrix can be computed by $\widetilde{O}(n/\ell)$ applications of Dijkstra's single-source algorithm. Let $C = \{c_{ij}\}_{i,j} = \bigwedge_{t=1,2,4,\ldots,\ell} \bigwedge_{s=1,\ldots,t} A^{(t,s)} \wedge B$. Return $\min_{i,j}(c_{ij} + a_{ji})$ as the weight of the shortest cycle (the cycle itself can be easily retrieved afterwards). Correctness follows since if the shortest cycle γ has length between $t/2$ and t, it must contain a vertex $p_i \in R^{(t/2)}$, and if γ has length greater than ℓ, it must contain a vertex $p_i \in R^{(\ell)}$. The weight of the path that starts at p_i and ends at p_i's predecessor p_j along γ is correctly computed as c_{ij}.

The total running time is

$$\widetilde{O}(\ell n^\omega + n^{3-(3-\omega)/(\kappa+1)} + n^3/\ell).$$

Setting ℓ near $n^{(3+\omega)/2}$ yields the result. □

Remark: We can also obtain the same $\widetilde{O}(n^{3-(3-\omega)/(\kappa+1)})$ time bound for the problem of finding the shortest simple cycle of a fixed length c_0, for any constant $c_0 \geq 3$. This follows more directly by applying Theorem 3.2 a constant number of times and using color-coding [5] to filter out nonsimple cycles.

4. REFERENCES

[1] P. K. Agarwal and J. Matoušek. On range searching with semialgebraic sets. *Discrete Comput. Geom.*, 11:393–418, 1994.

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

[3] S. Albers and T. Hagerup. Improved parallel integer sorting without concurrent writing. *Inf. Comput.*, 136:25–51, 1997.

[4] N. Alon, Z. Galil, and O. Margalit. On the exponent of the all pairs shortest path problem. *J. Comput. Sys. Sci.*, 54:255–262, 1997. Preliminary version at FOCS 1991.

[5] N. Alon, R. Yuster, and U. Zwick. Color-coding. *J. ACM*, 42:844–856, 1995.

[6] V. L. Arlazarov, E. C. Dinic, M. A. Kronrod, and I. A. Faradzev. On economical construction of the transitive closure of a directed graph. *Soviet Math. Dokl.*, 11:1209–1210, 1970.

[7] T. M. Chan. Dynamic subgraph connectivity with geometric applications. In *Proc. 34th ACM Sympos. Theory Comput.*, pages 7–13, 2002. *SIAM J. Comput.*, to appear.

[8] T. M. Chan. All-pairs shortest paths with real weights in $O(n^3/\log n)$ time. In *Proc. 9th Workshop Algorithms Data Struct.*, Lect. Notes Comput. Sci., vol. 3608, Springer-Verlag, pages 318–324, 2005. *Algorithmica*, to appear.

[9] T. M. Chan. All-pairs shortest paths for unweighted undirected graphs in $o(mn)$ time. In *Proc. 17th ACM-SIAM Sympos. Discrete Algorithms*, pages 514–523, 2006.

[10] T. M. Chan and Alon Efrat. Fly cheaply: on the minimum fuel consumption problem. *J. Algorithms*, 41(2):330–337, 2001.

[11] B. Chazelle. Cuttings. In *Handbook of Data Structures and Applications*, CRC Press, pages 25.1–25.10, 2005.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* McGraw-Hill, 2nd ed., 2001.

[13] A. Czumaj and A. Lingas. Finding a heaviest triangle is not harder than matrix multiplication. In *Proc. 18th ACM-SIAM Sympos. Discrete Algorithms*, 2007.

[14] W. Dobosiewicz. A more efficient algorithm for the min-plus multiplication. *Int. J. Computer Math.*, 32:49–60, 1990.

[15] T. Feder and R. Motwani. Clique partitions, graph compression and speeding-up algorithms. *J. Comput. Sys. Sci.*, 51:261–272, 1995. Preliminary version at STOC 1991.

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

[17] Y. Han. Improved algorithm for all pairs shortest paths. *Inform. Process. Lett.*, 91:245–250, 2004.

[18] Y. Han. An $O(n^3(\log\log n/\log n)^{5/4})$ time algorithm for all pairs shortest paths. In *Proc. 14th European Sympos. Algorithms*, Lect. Notes Comput. Sci., vol. 4168, Springer-Verlag, pages 411–417, 2006.

[19] H. Kaplan, N. Rubin, M. Sharir, and E. Verbin. Counting colors in boxes. In *Proc. 18th ACM-SIAM Sympos. Discrete Algorithms*, 2007.

[20] H. Kaplan, M. Sharir, and E. Verbin. Colored

intersection searching via sparse rectangular matrix multiplication. In *Proc. 22nd ACM Sympos. Comput. Geom.*, pages 52–60, 2006.

[21] V. Koltun. Almost tight upper bounds for vertical decompositions in four dimensions. *J. ACM*, 51:699–730, 2004.

[22] J. Matoušek. Computing dominances in E^n. *Inform. Process. Lett.*, 38:277–278, 1991.

[23] J. Matoušek. Efficient partition trees. *Discrete Comput. Geom.*, 8:315–334, 1992.

[24] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms.* Prentice-Hall, Englewood Cliffs, N.J., 1994.

[25] G. Narasimhan and M. Smid. Approximating the stretch factor of Euclidean graphs. *SIAM J. Comput.*, 39:978–989, 2000.

[26] S. Pettie. A new approach to all-pairs shortest paths on real-weighted graphs. *Theoret. Comput. Sci.*, 312:47–74, 2004.

[27] S. Pettie and V. Ramachandran. A shortest path algorithm for real-weighted undirected graphs. *SIAM J. Comput.*, 34:1398–1431, 2005.

[28] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, 1985.

[29] R. Sedgewick and J. S. Vitter. Shortest paths in Euclidean graphs. *Algorithmica*, 1:31–48, 1986.

[30] R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *J. Comput. Sys. Sci.*, 51:400–403, 1995. Preliminary version at STOC 1992.

[31] A. Shoshan and U. Zwick. All pairs shortest paths in undirected graphs with integer weights. In *Proc. 40th IEEE Sympos. Found. Comput. Sci.*, pages 605–614, 1999.

[32] T. Takaoka. A new upper bound on the complexity of the all pairs shortest path problem. *Inform. Process. Lett.*, 43:195–199, 1992.

[33] T. Takaoka. A faster algorithm for the all-pairs shortest path problem and its application. In *Proc. 10th Int. Conf. Comput. Comb.*, Lect. Notes Comput. Sci., vol. 3106, Springer-Verlag, pages 278–289, 2004.

[34] V. Vassilevska and R. Williams. Finding a maximum weight triangle in $n^{3-\delta}$ time, with applications. In *Proc. 38th ACM Sympos. Theory Comput.*, pages 225–231, 2006.

[35] V. Vassilevska, R. Williams, and R. Yuster. Finding the smallest H-subgraph in real weighted graphs and related problems. In *Proc. 33rd Int. Colloq. Automata, Languages, and Programming*, Lect. Notes Comput. Sci., vol. 4051, Springer-Verlag, pages 262–273, 2006.

[36] U. Zwick. All pairs lightest shortest paths. In *Proc. 31st ACM Sympos. Theory Comput.*, pages 61–69, 1999.

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

[38] U. Zwick. A slightly improved sub-cubic algorithm for the all pairs shortest paths problem with real edge lengths. In *Proc. 15th Int. Sympos. Algorithms and Computation*, Lect. Notes Comput. Sci., vol. 3341, Springer-Verlag, pages 921–932, 2004. *Algorithmica*, 46:181–192, 2006.