All-Pairs Shortest Paths in Unit-Disk Graphs in Slightly Subquadratic Time

Timothy M. Chan[1] and Dimitrios Skrepetos[2]

1 **Cheriton School of Computer Science, University of Waterloo, Canada**
tmchan@uwaterloo.ca
2 **Cheriton School of Computer Science, University of Waterloo, Canada**
dskrepet@uwaterloo.ca

Abstract

In this paper we study the all-pairs shortest paths problem in (unweighted) unit-disk graphs. The previous best solution for this problem required $O\left(n^2 \log n\right)$ time, by running the $O\left(n \log n\right)$-time single-source shortest path algorithm of Cabello and Jejčič (2015) from every source vertex, where n is the number of vertices. We not only manage to eliminate the logarithmic factor, but also obtain the first (slightly) subquadratic algorithm for the problem, running in $O\left(n^2 \sqrt{\frac{\log\log n}{\log n}}\right)$ time. Our algorithm computes an implicit representation of all the shortest paths, and, in the same amount of time, can also compute the diameter of the graph.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases unit-disk graphs, all-pairs shortest paths, computational geometry

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2016.24

1 Introduction

The all-pairs shortest paths (APSP) problem is one of the most known problems in the field of algorithms: given a graph $G = (V, E)$ with n vertices, where each edge has a real weight, we are asked to compute the shortest paths between all pairs of vertices. The classical algorithm of Floyd and Warshall solves the problem in $O\left(n^3\right)$ time. The first improvement for general weighted dense graphs came in 1976 when Fredman [16] obtained an algorithm of $O\left(n^3 \left(\frac{\log\log n}{\log n}\right)^{1/3}\right)$ time. This advancement sparked a number of algorithms that save polylogarithmic factors. For more results for APSP in graphs with arbitrary edge weights see [13, 32, 21, 34, 40, 35, 4, 22, 5, 23]. Recently, Williams [36] in a major breakthrough provided an algorithm that achieves superpolylogarithmic speedup, requiring $\frac{n^3}{2^{\Omega\left(\sqrt{\log n}\right)}}$ randomized time, later derandomized by Chan and Williams [10]. For the case of unweighted undirected graphs, Seidel [30] and Galil and Margalit [18] devised algorithms that solve the problem in matrix multiplication time. For results on directed and unweighted graphs and on graphs whose weights are small integers, see [17, 39, 18, 31, 33].

One very important class of graphs arising from computational geometry is *unit-disk graphs*. A unit-disk graph is the intersection graph of a set of unit disks, which is defined by creating a vertex for each unit disk and an edge between any two unit disks that intersect each other. Equivalently, given a set S of n points in the plane (the disk centers, after rescaling by a half), the unit-disk graph, $\mathrm{UD}(G)$, is defined by setting its vertex set to be S and creating an edge between any two points of S whose Euclidean distance is at most one. The edges are unweighted. The unit-disk graph with n vertices may contain $\Theta(n^2)$ edges, as every unit disk may intersect with every other unit disk. Since we aim for a subquadratic


© Timothy M. Chan and Dimitrios Skrepetos;
licensed under Creative Commons License CC-BY

solution to the all-pairs shortest path problem, we do not construct the set of the edges explicitly in our algorithms.

Unit-disk graphs are among the most studied families of graphs in geometry, with motivation from communication networks. Unit-disk graphs are related to planar graphs: by the circle packing theorem of Koebe–Andreev–Thurston any planar graph can be represented as a coin disk graph, although the disks may have different radii; on the other hand, planar graphs do not have large cliques unlike unit-disk graphs. Frederickson [15] gave an $O\left(n^2\right)$-time algorithm for solving the APSP problem in weighted planar graphs. Chan [6] improved the bound for unweighted directed planar graphs with an $O\left(n^2 \frac{\log\log n}{\log n}\right)$-time algorithm (and also considered general unweighted undirected sparse graphs). Wulff-Nilsen [37] independently developed another $O\left(n^2 \frac{\log\log n}{\log n}\right)$-time algorithm for computing the diameter of unweighted undirected planar graphs (and also announced similar results for the weighted case).

In this paper, we provide an algorithm for the APSP problem in unit-disk graphs that requires $O\left(n^2 \sqrt{\frac{\log\log n}{\log n}}\right)$ time. The previously fastest solution was to run from each vertex the single-source shortest path algorithm of Cabello and Jejčič [3], which required $O\left(n^2 \log n\right)$ total time. (See [29, 19] for other previous results on shortest paths in unit-disk graphs.) Therefore, we not only shave off the extra logarithmic factor of the previous result, but also provide the first (slightly) subquadratic solution to the problem. Our algorithm computes an implicit representation of the shortest paths: we encode the $O\left(n^2\right)$ shortest path distances and predecessors using bit-packing techniques (for use of bit-packing techniques in shortest path algorithms see [5]), so that we are still able to retrieve the shortest path distance of a pair of vertices in $O\left(1\right)$ time and the shortest path π itself in time linear in the number of vertices of π. In the same amount of time, we can also compute the diameter of the graph.

In recent years, obtaining polylogarithmic speedup for standard algorithmic problems have received considerable renewed attention. Such problems include 3SUM [20], Fréchet distance [1, 26], combinatorial Boolean matrix multiplication [8, 38], Klee's measure problem [7], CFL reachability [11], and many more. Our result can be seen as another contribution along this line of research.

The polylogarithmic improvement that we obtain for APSP in unit-disk graphs goes beyond standard word RAM tricks. First, in Section 2, we present a new algorithm for the *single-source* shortest path problem for unit-disk graphs, running in linear time after presorting the x- and y-coordinates of the input points. This improves over Cabello and Jejčič's single-source algorithm [3]. Their algorithm started with the Delaunay triangulation and performed repeated nearest neighbor queries, which inherently required $\Omega\left(n \log n\right)$ time even excluding preprocessing cost. Our new algorithm is instead based on a simple grid approach and exploits a linear-time Graham-scan-like procedure [28] for computing upper envelopes of presorted unit disks. (According to [3], Efrat has also observed an alternative, grid-based $O\left(n \log n\right)$-time algorithm, but his suggested solution seemed a bit more complicated and used a semi-dynamic data structure of Efrat et al. [25], which also inherently required $\Omega\left(n \log n\right)$ time even after presorting.)

Second, in Section 3, we extend the single-source algorithm to the case of *multiple* $\left(k = O\left(\sqrt{\frac{\log n}{\log\log n}}\right)\right)$ sources that lie in a *cluster*, i.e., in a common grid cell. In this case, we have to construct not just one but k upper envelopes, one for each source. This leads to a new kind of data structure problem: preprocess a set of unit disks (a "universe") so that given any subset of the universe, we can compute the upper envelope of the subset in slightly *sublinear* time. Our ideas can similarly be applied to the following (even more natural) problem, of independent interest: preprocess a point set (a universe) in 2D so that

given any subset of the universe, we can compute its convex hull, again in slightly sublinear time. Note that the input subset and the output can be encoded with linear number of bits, and thus slightly sublinear number of words. Solving problems for "preprocessed universes" is a relatively recent research direction (for example, see [9, 14, 12, 2]); our result with slightly sublinear time provides an unusual addition to this body of work.

Finally, in Section 4, to obtain our subquadratic-time APSP algorithm, we draw inspiration from previous algorithms for planar graphs [6, 37], which use planar separators to decompose the graph into regions of polylogarithmic size, and table-lookup techniques to handle each region. However, this approach does not directly apply to unit-disk graphs, because there could be large cliques and no small separators. On the other hand, when there are many large cliques, intuitively we should be able to exploit the multi-source algorithm from Section 3 to handle such clusters more efficiently. The challenge lies in how to carefully combine these two approaches. For unit-disk graphs, we end up avoiding planar separators and instead adopting a simpler *shifted grid* strategy [24]. This strategy is standard for geometric approximation algorithms, but we use the technique in a new and interesting way to design an *exact* algorithm (with an intricate balancing of parameters).

All of our algorithms operate in a standard unit-cost RAM model of computation, where each word can store either an input point or a $(\log n)$-bit integer. (We do *not* need to assume integer input coordinates from a bounded universe: the input coordinate values may be real numbers if we assume the availability of a few constant-degree algebraic predicates, as in most real-RAM algorithms in computational geometry. We are not cheating when using bit-packing and table-lookup tricks if they are done with $(\log n)$-bit words and without exotic word operations.)

2 Single-source shortest paths in linear time (after presorting)

In this section, we begin with the single-source problem: given a set S of n points, and a source point $s \in S$, we want to compute the shortest paths from s to all points in S in the unit-disk graph $\mathrm{UD}(G)$. We assume that the points in S have been presorted with respect to both x and y. The approach that we follow is a variant of the classical breadth-first search (BFS) algorithm.

The first step is to build a grid composed of square cells with side length $\frac{1}{\sqrt{2}}$. A cell c' is a *neighbor* of a cell c if the minimum distance between c and c' is at most 1. Clearly, the number of neighbors of a cell is constant. For each point p, as is the case in the classical BFS, we have a value for its shortest-path distance from s, $dist[p]$, and another for the predecessor, $pred[p]$, denoting the point before p in that path.

The algorithm iteratively performs the following step $n-1$ times. In each step ℓ we assume that we have already computed the distances and predecessors for all of the points that are at distance no more than $\ell - 1$ from s and that we have a list, called the *frontier*, containing all of the points being at distance exactly $\ell - 1$ from s. We find all points whose shortest-path distance from s is ℓ by finding all points at distance at most 1 from the points in the frontier, and we filter out the ones whose shortest-path distance from s has been found in earlier steps. For the remaining points, we update their distances and predecessors properly. Finally, we replace the frontier with the set of these points. It is easy to establish the correctness of the algorithm by induction. The concepts of the grid, cells, neighboring cells, and frontier points are depicted in Figure 1.

We say that a cell is *visited* at step ℓ if the algorithm performs an operation to any one of its points during that step. When the algorithm visits a cell, each point of that cell is used in operations at most twice. The following lemma is crucial for the rest of the section.

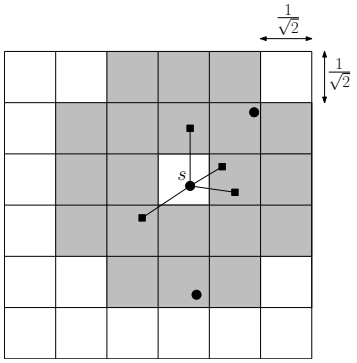

Figure 1 Here we see the grid created for six points depicted as black disks and squares. Each of the thirty-six squares corresponds to a cell. The neighbors of the cell of the third row and fourth column, containing the point s, are shaded gray. In the second step of the algorithm, where s is the source, we see the four points that are at distance one from s, depicted as squares, and the shortest path tree of s so far, depicted with line segments. Those four points compose the frontier for the second step.

▶ **Lemma 1.** *Each cell c is visited only a constant number of times.*

Proof. A cell c is visited either (i) when at least one of its points enters the frontier or (ii) when at least one of the points of a neighboring cell c' of c enters the frontier.

In case (i), we note that once at least one point of c enters the frontier in a step of the algorithm, then the rest of the points of c enter the frontier either in this step or in the next because any two points of c are at distance at most one. Also, a point can be in the frontier in exactly one step. Therefore, c has points in the frontier in at most two steps of the algorithm.

In case (ii), c is visited when one of its neighbors, c', has at least one frontier point. Since the number of the neighbors of c is constant and, by case one, any cell has frontier points at most twice, we conclude that a cell c is visited by its neighbors in a constant number of steps of the algorithm. ◀

To find points that are at distance at most 1 from the points in the frontier, we solve the following subproblem.

▶ **Subproblem 2.** *Given a set of n_r red points below a horizontal line h and another set of n_b blue points above h, both presorted by x, determine for each blue point whether there is a red point at distance at most one from it.*

▶ **Lemma 3.** *We can solve Subproblem 2 in $O(n_r + n_b)$ time.*

Proof. We consider the upper *envelope* of the unit disks centered at the red points; it is composed of the part of the boundary of the union of those unit disks that lies above h.

Notice that the parts of the boundaries of unit disks above h form a *pseudoline* arrangement. For each unit disk we can extend that part to a x-monotone curve, a pseudoline, from $x = -\infty$ to $x = \infty$ such that it intersects only once with the pseudoline of another unit disk. Moreover, we can make the order of the pseudolines at $x = -\infty$ coincide with the order of x-coordinates of the disk centers. An example of the pseudolines can be see in Figure 2. Then we can compute the upper envelope of these pseudolines by adapting the standard *Graham scan* algorithm [28] for computing upper envelopes of lines (i.e., convex hulls of points in the plane by duality). Graham scan takes linear, $O(n_r)$, time after presorting. Notice that we do not

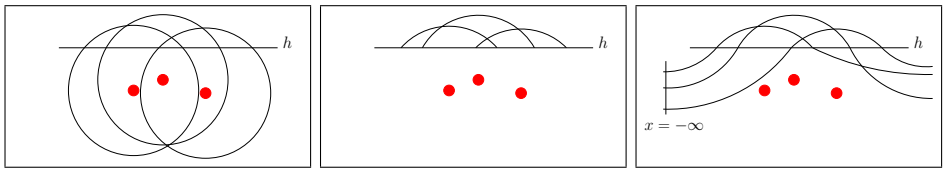

Figure 2 In the first subfigure we see the unit disks of three red points, depicted as disks, and a horizontal line h. In the second subfigure, we see the part of the boundaries of those unit disks lying above h. In the third subfigure, we see the pseudolines corresponding to each unit disk as defined in Lemma 3. Notice that the y-order of the pseudolines at $x = -\infty$ coincides with the x-order of the centers of their unit disks.

have to explicitly compute the pseudolines; the notion of the pseudolines is used only for intuition. The algorithm, assuming that the points are presorted, scans them from left to right, finds intersections between unit disks, and builds step-by-step the upper envelope.

Finally, to solve Subproblem 2, we want to determine for each blue point whether it is below the upper envelope of the red disks. This can done by performing a linear scan over the vertices of the upper envelope and the (presorted) blue points in $O(n_r + n_b)$ time. ◀

Putting everything together, we obtain the following theorem.

▶ **Theorem 4.** *Given a unit-disk graph, we can compute the shortest path tree from a given source in $O(n)$ time, if the input points have been pre-sorted by both x and y.*

Proof. In step ℓ of the BFS algorithm, we find the points at distance exactly ℓ from the source as follows. For each cell c having at least one point in the frontier and each neighbor c' of c, we use the subroutine from Lemma 3 on the input where the red points are the points of c that are in the frontier and the blue points are all the points of c'.

The cells c and c' are either horizontally or vertically separated; without loss of generality, we can assume the former by rotation. By Lemmas 1 and 3, the total time of the BFS is $O(n)$.

Note that to identify the predecessors during the BFS, we need to strengthen Subproblem 2 to report for each blue point a *witness* red point (if exists) that is at distance at most 1; the method in Lemma 1 can easily provide such witnesses. Initially assigning points to grid cells can be done in linear time (without the need for hashing, because of presortedness). ◀

Applying the above theorem n times immediately yields an $O(n^2)$-time algorithm for APSP for unit-disk graphs. In the subsequent sections, we aim for a slightly subquadratic algorithm.

3 Shortest paths for multiple sources in a cluster

In this section, we extend the single-source algorithm of the previous section to compute shortest paths from k source points $s_1, \ldots, s_k \in S$ to all vertices in S, where $k = O\left(\sqrt{\frac{\log n}{\log \log n}}\right)$ and $s_1, \ldots, s_k$ lie in a common grid cell. The approach is to run BFS from the multiple sources simultaneously but avoid a factor-k slowdown by using bit-packing tricks – this approach is inspired by the first APSP algorithm in [6].

First, we extend Lemma 1 to the case of k source points, lying close to each other. We say that a cell is *visited* at step ℓ if at the ℓ-th level of the BFS for at least one of the sources, the algorithm performs an operation to any one of its points.

▶ **Lemma 5.** *If the k source points are in the same cell, each cell is visited only a constant number of times.*

Proof. Let $dist_s[p]$ be the shortest-path distance between a source s and a point p. Observe that for any two sources s and s' in a common cell, $dist_s[p]$ and $dist_{s'}[p]$ can differ by at most one by the triangle inequality. Therefore, the first time that any point of a cell enters the frontier of any source, by the next step of the algorithm that point will enter the frontier of the rest of the sources. The rest of the proof is as in Lemma 1. ◀

If we apply the algorithm of Lemma 3 separately for the k sources, then the cost of this operation alone would be $O(nk)$, which we cannot afford. To overcome the issue, we introduce an extension of Subproblem 2.

▶ **Subproblem 6.** *Preprocess a universe R of n_r red points below a horizontal line h and another universe B of n_b blue points above h, so that given any subset $Q \subseteq R$ of the red universe, we can determine for each blue point whether there is a red point of Q at distance at most one from it, in* sublinear *total time.*

As we have seen in the proof of Lemma 3, the key to solving the subproblem lies in the construction of upper envelopes, so we concentrate on solving the following subproblem.

▶ **Subproblem 7.** *Preprocess a universe R of n_r red points below a horizontal line h, so that given any subset $Q \subseteq R$ of the red universe, we can compute the upper envelope of the unit disks centered at Q (specifically the part above h) in* sublinear *time.*

By sublinear we mean $O\left(\frac{n_r}{\text{polylog}\, n}\right)$ or $O\left(\frac{n_r+n_b}{\text{polylog}\, n}\right)$. The reason that sublinear time is feasible at all is that we can represent (i) the input subset Q as an n_r-bit vector, where the i-th bit denotes whether the i-th red point in x-order is in Q, and (ii) the output upper envelope is another n_r-bit vector, where the i-th bit denotes whether the disk defined by the i-th point participates in the upper envelope. We can pack either vector into $O\left(\frac{n_r}{\log n}\right)$ words.

▶ **Lemma 8.** *We can solve Subproblem 7 with $O(n_r \log n_r + n_r 2^g)$ preprocessing time and $O\left(n_r \frac{\log g}{g}\right)$ query time, for any given $g \leq \log n$.*

Proof. During the preprocessing phase, we divide the x-ordered sequence of red points into $O\left(\frac{n_r}{g}\right)$ chunks of at most g consecutive points each; the chunks lie in $O\left(\frac{n_r}{g}\right)$ disjoint slabs. The red points of each chunk compose $O(2^g)$ possible subsets, and we precompute the upper envelope for each such subset (specifically the part above h) in $O(g)$ time. The computation of all these envelopes, which we call *small upper envelopes*, needs $O\left(\frac{n_r}{g} 2^g g\right) = O(n_r 2^g)$ time. We store a lookup table containing all the small upper envelopes, each represented as an ordered array. In Figure 3, we see the chunks and the small upper envelope of each.

To answer a query for a given subset of the red points, we have to merge $O\left(\frac{n_r}{g}\right)$ small upper envelopes for the relevant subsets of the $O\left(\frac{n_r}{g}\right)$ chunks. Observe that the $O\left(\frac{n_r}{g}\right)$ small upper envelopes themselves can be viewed as a pseudoline arrangement (since the chunks are vertically separated). We can therefore apply Graham scan to compute the upper envelope of the $O\left(\frac{n_r}{g}\right)$ small upper envelopes, using a linear $O\left(\frac{n_r}{g}\right)$ number of *primitive operations*. We need two primitive operations: (i) finding the intersection point between two small upper envelopes, and (ii) determining whether a given point is above or below a small upper envelope. Both operations can be done in $O(\log g)$ time by binary searching (see [27]

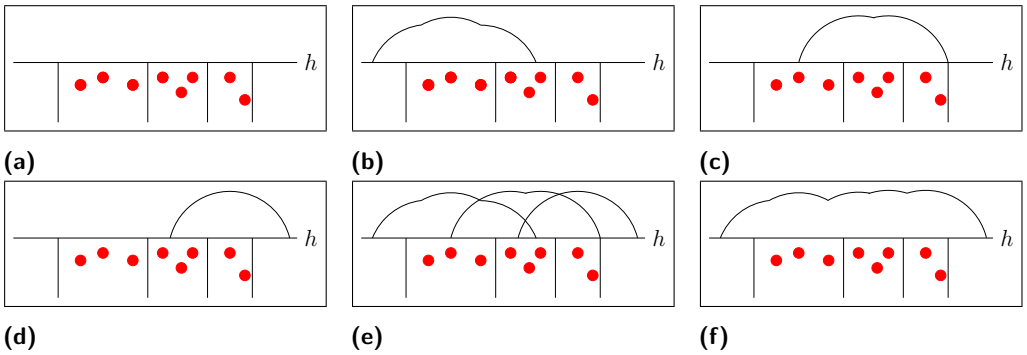

Figure 3 In the first subfigure we see the chunks of eight red points, drawn as disks, for $g = 3$, and a horizontal line h. Notice that the last chunk has only two points. In the next three subfigures, we see the small upper envelope associated with each chunk. The three small envelopes are seen together in the fifth subfigure. Finally, in the sixth subfigure we see the upper envelope of the small upper envelopes, which is the upper envelopes of all red points.

for (i)). Thus, Graham scan takes $O\left(\frac{n_r}{g}\log g\right)$ time. The output is a sequence of $O\left(\frac{n_r}{g}\right)$ pieces of small upper envelopes; we can convert each piece into the bit-vector format, in additional $O\left(\frac{n_r}{\log n}\right)$ total time. In Figure 3, we see the upper envelope. ◀

▶ **Lemma 9.** *We can solve Subproblem 6 with $O\left(n_r \log n_r + n_b \log n_b + n_r 2^g g + n_b g\right)$ preprocessing time and $O\left((n_r + n_b)\frac{\log g}{g}\right)$ query time, for any given $g \leq \log n$.*

Proof. We build on the method from Lemma 8. During the preprocessing phase, we also divide the x-ordered sequence of blue points into $O\left(\frac{n_b}{g}\right)$ chunks of at most g consecutive blue points each; the chunks lie in $O\left(\frac{n_b}{g}\right)$ disjoint slabs (regions each bounded by two vertical lines). We store the following extra structures:

1. For each small upper envelope e and each slab σ that contains at least one vertex of e, we precompute a g-bit vector where the i-th bit denotes whether the i-th blue point in σ is below e. We store all these vectors in a lookup table. There are $O\left(\frac{n_r}{g}2^g\right)$ small upper envelopes, each with $O(g)$ vertices; the total time is $O\left(n_r 2^g g\right)$.
2. For each slab σ, we precompute the arrangement of the $O(g)$ unit disks centered at the blue points in σ; the arrangement has $O\left(g^2\right)$ complexity and we can build a *point location* structure [28] in $O\left(g^2\right)$ time. For each face of the arrangement, we record a g-bit vector where the i-th bit denotes whether the face is inside the disk for the i-th blue point. There are $O\left(\frac{n_b}{g}\right)$ slabs; the total time is $O\left(n_b g\right)$.

To answer a query, we first construct the upper envelope E of the $O\left(\frac{n_r}{g}\right)$ small upper envelopes as described in the proof of Lemma 8. We then need to determine for each blue point whether it is below E.

We scan the slabs from left to right. Consider the next slab σ. Consider each small upper envelope e that contributes arcs to E inside σ. We compute a bit vector $\mathbf{v}_{\sigma,e}$ where the i-th bit denotes whether the i-th blue point in σ is below e, as follows:

1. If σ contains at least one vertex of e, then $\mathbf{v}_{\sigma,e}$ has already been precomputed in the lookup table.

2. If σ contains no vertices of e, then only one disk contributes to e inside σ; say the disk is defined by the red point q. We can determine $\mathbf{v}_{\sigma,e}$ by looking up the face containing q in the arrangement of the blue disks for σ, in $O(\log g)$ time by point location.

Finally, we take the bitwise-or of the bit vectors $\mathbf{v}_{\sigma,e}$ over all small upper envelopes e that contributes to E inside σ. The total number of bitwise-or operations and point location queries is $O\left(\frac{n_r+n_b}{g}\right)$, yielding total query time $O\left(\frac{n_r+n_b}{g}\log g\right)$. ◀

▶ **Theorem 10.** *Given a unit-disk graph, after $O\left(n\log n + n2^{O(k\log k)}\right)$-time preprocessing, we can compute an implicit representation of the shortest path trees for any $k \le \sqrt{\frac{\log n}{\log\log n}}$ source points lying in a unit-diameter square, in $O(n)$ time.*

Proof. For each point p, we maintain k-bit vectors *frontier*$[p]$ (resp. *found*$[p]$), where the i-th bit denotes whether p is in the frontier of the i-th source (resp. whether p has previously appeared in the frontier) in each step ℓ of the BFS algorithm.

In step ℓ of the BFS, we proceed as follows. For each cell c having at least one point in one of the k frontiers and for each neighbor c' of c, we use the subroutine from Lemma 9 on the input where the red universe contains all n_r points of c and the blue points are all n_b points of c'; for each of the k sources, we query with the red subset containing all points in its frontier in c. The total time for the k queries is $O\left(k(n_r+n_b)\frac{\log g}{g}\right) = O(n_r+n_b)$ by setting $g = k\log k$.

One technical issue concerns the generation of the bit-vector representation for these input red subsets (which are ordered by x or y depending on whether c and c' are horizontally or vertically separated). This can be done by taking each chunk of g red points $p_1,\ldots,p_g$, collecting the g k-bit vectors *frontier*$[p_1],\ldots,$*frontier*$[p_g]$, and performing a *transposition* to generate k g-bit vectors, where the j-th bit of the i-th vector is equal to the i-th bit of *frontier*$[p_j]$. The transposition involves simply shuffling bits between words and can be straightforwardly implemented in $O(1)$ time by table lookup if $kg \le \log n$. The outputs can similarly be converted by transposition to obtain vectors *out*$[p]$ for the blue points p in c', where the i-th bit denotes whether p is at distance at most one from some red point in c with respect to the i-th source.

We can update the k-bit vector *found*$[p]$ by taking the bitwise-or with *out*$[p]$. At the end of step ℓ, we can update the k-bit vector *frontier*$[p]$ by taking the bitwise-difference between the new and old *found*$[p]$ vectors. By Lemmas 5 and 9, the total time of the k simultaneous BFSs is $O(n)$.

Two remaining technical issues concern the encoding of the shortest path distances and of the predecessors. To address the former issue, for each point p, we store $dist_{s_1}[p]$ and work with the vector containing $dist_{s_i}[p] - dist_{s_1}[p]$ over all k sources s_i. Recall that these values are in $\{-1,0,1\}$ by the triangle inequality, and so the vector can be encoded in $O(k)$ bits; the total space over all n points is $O\left(n + \frac{nk}{\log n}\right) = O(n)$ words, and it is easy to update the distances of a point in constant time.

To address the latter issue, we need to strengthen Subproblem 6 so that given any blue point, we can report a witness red point (if exists) that is at distance at most one in constant time. Thus we need to perform a few modifications to the proof of Lemma 9. First in the lookup tables during the preprocessing phase, we record witnesses for the true bits for each g-bit vector, stored in a $(g\log g)$-bit *witness vector*. In the query algorithm, consider each slab σ. The upper envelope E inside σ consists of pieces of small upper envelopes. Divide σ into subslabs by drawing vertical lines at the endpoints of these pieces. If, for each subslab, we created a pointer from each blue point to the small upper envelope in the subslab, we

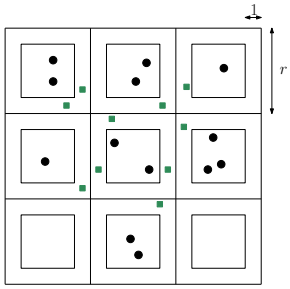

Figure 4 In this figure we see the shifted grid. We assume that the points have already been shifted. The nine big squares correspond to the supercells. The smaller square within each supercell corresponds to the region of the supercell that contains points at distance more than one from its boundary. The boundary points are drawn as green squares and the non-boundary points as black disks.

would need $O(kn)$ total time. To avoid that, for each subslab, we mark its rightmost blue point, which can be found by binary search in $O(\log g)$ time; the total time for that is $O\left(\frac{n_r+n_b}{g} k \log g\right) = O\left(n_r + n_b\right)$. Then we create a pointer from this marked point to the small upper envelope of its subslab. Finally, for each slab, we create a pointer from each blue point q to its successor among the marked blue points; these g pointers require $O\left(g \log g\right)$ bits, can be created in constant time, and can be stored in a $(g \log g)$-bit *pointer vector*, so the total time for this step is $O\left(\frac{n_b}{g} k\right) = O\left(\frac{n_b}{\log k}\right)$. Then, given any blue point q in σ, we can retrieve the pointer vector of σ, look up the marked successor of q, follow its pointer to the small upper envelope e in σ, and then look up q's witness with respect to $\mathbf{v}_{\sigma,e}$, all in constant time. ◀

4 All-pairs shortest paths in slightly subquadratic time

In this final section, we present our APSP algorithm for unit-disk graphs. Let r, a, b be parameters to be chosen later. We build a grid composed of square cells with side length r, where r is a parameter to be specified later $\left(\text{larger than } \frac{1}{\sqrt{2}}\right)$. Call these larger grid cells *supercells*. We say that a point $p \in S$ is a *boundary point* if it is at distance at most one from the boundary of some supercell. We begin with a standard *shifted* grid strategy by Hochbaum and Maass [24]. The supercell and the boundary points are depicted in Figure 4.

▶ **Lemma 11.** *There exists a translation of S such that the number of boundary points is $O\left(\frac{n}{r}\right)$. The translation can be found in linear time.*

Proof. Shift the points of S by a random vector from $\{1, \ldots, r\}^2$. The probability that a point $p \in S$ is a boundary point is at most $\frac{4}{r}$, so the expected number of boundary points is at most $\frac{4n}{r}$. (It is straightforward to derandomize in linear time.) ◀

After applying Lemma 11 to build the grid, our algorithm proceeds in four steps:

1. We first compute the shortest paths between the $O\left(\frac{n}{r}\right)$ boundary points and all points in S. For this step, we can run the single-source algorithm of Section 2 $O\left(\frac{n}{r}\right)$ times, requiring $O\left(\frac{n^2}{r}\right)$ total time.
2. Next, for each supercell γ_i that contains more than a points or more than b boundary points, we compute the shortest paths between all points in $S \cap \gamma_i$ and all points in S. This time, we use the multi-source algorithm of Section 3. The points in γ_i can be grouped

into $O\left(\frac{|S\cap\gamma_i|}{k}+r^2\right)$ clusters of size at most k, since the supercell can be decomposed into $O\left(r^2\right)$ cells of diameter one. The number of supercells with more than a points is $O\left(\frac{n}{a}\right)$ and the number of supercells with more than b boundary points is $O\left(\frac{n}{rb}\right)$. Hence, the total time for this step is $O\left(\sum_i\left(\frac{|S\cap\gamma_i|}{k}+r^2\right)\cdot n\right) = O\left(\left(\frac{n}{k}+\left(\frac{n}{a}+\frac{n}{rb}\right)r^2\right)\cdot n\right) = O\left(\frac{n^2}{k}+\frac{n^2r^2}{a}+\frac{n^2r}{b}\right)$.

3. For each supercell γ_i with at most a points and at most b boundary points, we compute the shortest paths between all pairs of points in $S\cap\gamma_i$. For this step, we can run a naive cubic-time APSP algorithm on $S\cap\gamma_i$, after adding an extra weighted edge between each boundary point u in γ_i and each point $p\in S\cap\gamma_i$, with weight $dist_u[p]$, which we have computed in step 1. (These extra edges take care of the possibility that the shortest paths may not stay inside γ_i.) The total time is $O\left(\sum_i|S\cap\gamma_i|^3\right) = O\left(\sum_i|S\cap\gamma_i|a^2\right) = O\left(na^2\right)$.
4. For each supercell γ_i with at most a points and at most b boundary points, we compute the shortest paths between all points $p\in S\cap\gamma_i$ and all points $q\in S-\gamma_i$. Each such path must pass through a boundary point in γ_i, i.e., we want to find $\min_u\left(dist_u[p]+dist_u[q]\right)$ where the minimum is over all boundary points u in γ_i.
   We describe a table lookup method inspired by the planar-graph APSP algorithm in [6]. For each connected component of the unit-disk graph of $S\cap\gamma_i$, pick a *representative* boundary point (if one exists) among the points of this component. For each point $p\in S\cap\gamma_i$, let $rep(p)$ be the representative boundary point that lies in p's connected component. For each point $q\in S-\gamma_i$, define $signature[q]$ to be the vector containing $signature_u[q] = dist_u[q]-dist_{rep(u)}[q]$ over all boundary points u in γ_i. Observe that these values are bounded by $O\left(r^2\right)$ by the triangle inequality (since the distance between u and $rep(u)$ is at most $O\left(r^2\right)$), and so the vector can be encoded in $O\left(b\log r\right)$ bits. All these values have already been computed in step 1, and so the signatures of all points over all supercells can be generated in $O\left(\frac{n^2}{r}\right)$ total time. (We can ignore empty signatures, i.e., supercells that do not have any boundary points.)
   For each $p\in S\cap\gamma_i$ and each $q\in S-\gamma_i$,
   $$\min_u\left(dist_u[p]+dist_u[q]\right) \;=\; dist_{rep(p)}[q] \;+\min_{u:rep(u)=rep(p)}\left(dist_u[p]+signature_u[q]\right).$$
   We can precompute the minimum in the right-hand side for each $p\in S\cap\gamma_i$ and each possible signature, and store all these minimums in a lookup table in $|S\cap\gamma_i|2^{O(b\log r)}$ time, for a total of $n2^{O(b\log r)}$ time.

The running time of the entire algorithm is

$$O\left(n\log n+n2^{O(k\log k)}+\frac{n^2}{r}+\frac{n^2}{k}+\frac{n^2r^2}{a}+\frac{n^2r}{b}+na^2+n2^{O(b\log r)}\right).$$

To balance the third and the fourth term, we set $k=r$. To balance the third and the sixth term, we set $r=\sqrt{b}$. To balance the third and the fifth term, we set $a=b^{3/2}$. Finally, we set $b=\frac{\varepsilon\log n}{\log\log n}$ for a sufficiently small constant ε, so that the second and the eighth terms get absorbed by the others. We then obtain the desired $O\left(n^2\sqrt{\frac{\log\log n}{\log n}}\right)$ time bound.

It is straightforward to modify the algorithm to retrieve any shortest-path distance in constant time, and retrieve any shortest path in time proportional to its size. It is also straightforward to modify the algorithm to compute the diameter (in step 4, for each $p\in S\cap\gamma_i$ and each possible signature, we need to find the point $q\in S-\gamma_i$ having this signature that maximizes $dist_{rep(p)}[q]$; all these maximums can be computed in $O\left(n\cdot\frac{n}{r}\right)$ total time by scanning the distance values from all boundary points). We conclude:

▶ **Theorem 12.** *Given a unit-disk graph we can compute an implicit representation of the shortest paths between all pairs, and the diameter of the graph, in* $O\left(n^2\sqrt{\frac{\log\log n}{\log n}}\right)$ *time in the RAM model.*

5 Conclusion

It is an intriguing open problem to compute the diameter of a unit-disk graph in truly subquadratic time, $O(n^{2-\varepsilon})$, for some positive constant ε. Another related problem is APSP in *weighted* unit-disk graphs, where the weight of an edge is defined as the Euclidean distance between the centers of the unit disks it connects.

Acknowledgement. We thank Anna Lubiw for posing the question of APSP in unit-disk graphs, and the participants of the algorithms problem session at Waterloo for discussion.

References

1 Pankaj K. Agarwal, Rinat Ben Avraham, Haim Kaplan, and Micha Sharir. Computing the discrete Fréchet distance in subquadratic time. *SIAM Journal on Computing*, 43:429–449, 2014.
2 Kevin Buchin and Wolfgang Mulzer. Delaunay triangulations in $O(\text{sort}(n))$ time and more. *Journal of the ACM (JACM)*, 58(2):6, 2011.
3 Sergio Cabello and Miha Jejčič. Shortest paths in intersection graphs of unit disks. *Computational Geometry*, 48:360–367, 2015.
4 Timothy M. Chan. All-pairs shortest paths with real weights in $O(n^3/\log n)$ time. *Algorithmica*, 50(2):236–243, 2008.
5 Timothy M. Chan. More algorithms for all-pairs shortest paths in weighted graphs. *SIAM Journal on Computing*, 39(5):2075–2089, 2010.
6 Timothy M. Chan. All-pairs shortest paths for unweighted undirected graphs in $o(mn)$ time. *ACM Transactions on Algorithms*, 8:1–17, 2012.
7 Timothy M. Chan. Klee's measure problem made easy. In *Proceedings of the 54th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 410–419, 2013.
8 Timothy M. Chan. Speeding up the Four Russians algorithm by about one more logarithmic factor. In *Proceedings of the 26th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 212–217, 2015. doi:10.1137/1.9781611973730.16.
9 Timothy M. Chan and Moshe Lewenstein. Clustered integer 3SUM via additive combinatorics. In *Proceedings of the 47th Annual ACM Symposium on Theory of Computing (STOC)*, pages 31–40, 2015.
10 Timothy M. Chan and Ryan Williams. Deterministic APSP, orthogonal vectors, and more: Quickly derandomizing Razborov–Smolensky. In *Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1246–1255, 2016.
11 Swarat Chaudhuri. Subcubic algorithms for recursive state machines. *ACM SIGPLAN Notices*, 43(1):159–169, 2008.
12 Bernard Chazelle and Wolfgang Mulzer. Computing hereditary convex structures. *Discrete & Computational Geometry*, 45(4):796–823, 2011.
13 Wlodzimierz Dobosiewicz. A more efficient algorithm for the min-plus multiplication. *International Journal of Computer Mathematics*, 32(1-2):49–60, 1990.
14 Esther Ezra and Wolfgang Mulzer. Convex hull of points lying on lines in $o(n \log n)$ time after preprocessing. *Computational Geometry*, 46(4):417–434, 2013.
15 Greg N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM Journal on Computing*, 16:1004–1022, 1987.

16 Michael L. Fredman. New bounds on the complexity of the shortest path problem. *SIAM Journal on Computing*, 5:83–89, 1976.

17 Noga Alon, Zvi Galil, and Oded Margalit. On the exponent of the all pairs shortest path problem. *Journal of Computer and System Sciences*, 54(2):255–262, 1997.

18 Zvi Galil and Oded Margalit. All pairs shortest distances for graphs with small integer length edges. *Information and Computation*, 134(2):103–139, 1997.

19 Jie Gao and Li Zhang. Well-separated pair decomposition for the unit-disk graph metric and its applications. *SIAM Journal on Computing*, 35(1):151–169, 2005.

20 Allan Grønlund and Seth Pettie. Threesomes, degenerates, and love triangles. In *Proceedings of the 55th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 621–630, 2014.

21 Yijie Han. Improved algorithm for all pairs shortest paths. *Information Processing Letters*, 91(5):245–250, 2004.

22 Yijie Han. An $O(n^3(\log\log n/\log n)^{5/4})$ time algorithm for all pairs shortest path. *Algorithmica*, 51(4):428–434, 2008.

23 Yijie Han and Tadao Takaoka. An $O(n^3 \log\log n/\log^2 n)$ time algorithm for all pairs shortest paths. In *Proceedings of the 13th Scandinavian Symposium on Algorithm Theory (SWAT)*, pages 131–141, 2012.

24 Dorit S. Hochbaum and Wolfgang Maass. Approximation schemes for covering and packing problems in image processing and VLSI. *Journal of the ACM*, 32(1):130–136, 1985.

25 Alon Efrat, Alon Itai, and Matthew J. Katz. Geometry helps in bottleneck matching and related problems. *Algorithmica*, 31(1):1–28, 2001.

26 Kevin Buchin, Maike Buchin, Wouter Meulemans, and Wolfgang Mulzer. Four Soviets walk the dog – with an application to Alt's conjecture. In *Proceedings of the 25th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1399–1413, 2014.

27 Mark H. Overmars and Jan van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23:166–204, 1981.

28 Franco P. Preparata and M. Ian Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, NY, 1985.

29 Liam Roditty and Michael Segal. On bounded leg shortest paths problems. *Algorithmica*, 59(4):583–600, 2011.

30 Raimund Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of Computer and System Sciences*, 51(3):400–403, 1995.

31 Avi Shoshan and Uri Zwick. All pairs shortest paths in undirected graphs with integer weights. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 605–614, 1999.

32 Tadao Takaoka. A new upper bound on the complexity of the all pairs shortest path problem. *Information Processing Letters*, 43(4):195–199, 1992. doi:10.1016/0020-0190(92)90200-F.

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

34 Tadao Takaoka. A faster algorithm for the all-pairs shortest path problem and its application. In *Proceedings of the 10th Annual International Conference on Computing and Combinatorics (COCOON)*, pages 278–289, 2004.

35 Tadao Takaoka. An $O(n^3 \log\log n/\log n)$ time algorithm for the all-pairs shortest path problem. *Information Processing Letters*, 96(5):155–161, 2005.

36 Ryan Williams. Faster all-pairs shortest paths via circuit complexity. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing (STOC)*, pages 664–673, 2014.

37 Christian Wulff-Nilsen. Constant time distance queries in planar unweighted graphs with subquadratic preprocessing time. *Computational Geometry*, 46(7):831–838, 2013.

38 Huacheng Yu. An improved combinatorial algorithm for Boolean matrix multiplication. In *Proceedings of the 42nd International Colloquium on Automata, Languages, and Programming (ICALP), Part I*, pages 1094–1105, 2015. doi:10.1007/978-3-662-47672-7_89.

39 Uri Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *Journal of the ACM*, 49(3):289–317, 2002.

40 Uri Zwick. A slightly improved sub-cubic algorithm for the all pairsshortest paths problem with real edge lengths. *Algorithmica*, 46(2):181–192, 2006. doi:10.1007/s00453-005-1199-1.