

Two Optimization Problems for Unit Disks*

Lazar Milinković†

February 9, 2017

Abstract

We present an implementation of a recent algorithm to compute shortest-path trees in unit disk graphs in $O(n \log n)$ worst-case time, where n is the number of disks.

In the minimum-separation problem, we are given n unit disks and two points s and t , not contained in any of the disks, and we want to compute the minimum number of disks one needs to retain so that any curve connecting s to t intersects some of the retained disks. We present a new algorithm solving this problem in $O(n^2 \log^3 n)$ worst-case time and its implementation.

1 Introduction

In this paper we consider two geometric optimization problems in the plane where unit disks play a prominent role. For both problems we discuss efficient algorithms to solve them, provide an implementation of these algorithms, and present experimental results on the implementation.

The first problem we consider is computing a *shortest-path tree* in the (unweighted) intersection graph of unit disks. The input to the problem is a set \mathcal{D} of n disks of the same size, each disk represented by its center. The corresponding unit disk (intersection) graph has a vertex for each disk, and an edge connecting two disks D and D' of \mathcal{D} whenever D and D' intersect. An alternative, more convenient point of view, is to take as vertex set the set of centers of the disks, denoted by P , and connecting two points p and q of P whenever the Euclidean length $|pq|$ is at most the diameter of a disk. The graph is unweighted. Given a root $r \in P$, the task is to compute a shortest-path tree from r in this graph. See Figure 1.

The second problem we consider is the *minimum-separation problem*. The input is a set \mathcal{D} of n unit disks in the plane and two points s and t not covered by any disks of \mathcal{D} . We say that \mathcal{D} *separates* s and t if each curve in the plane from s to t intersects some disk of \mathcal{D} . The task is to find the minimum cardinality subset of \mathcal{D} that separates s and t . See the left of Figure 1 for an example of an instance. Formally, we want to solve

$$\begin{aligned} \min \quad & |\mathcal{D}'| \\ \text{s.t.} \quad & \mathcal{D}' \subseteq \mathcal{D} \text{ and } \mathcal{D}' \text{ separates } s \text{ and } t. \end{aligned}$$

Unit disks are the most standard model used for wireless sensor networks; see for example [8, 11, 22]. Often the model is referred as UDG. This model provides an appropriate trade off between simplicity and accuracy. Other models are more accurate, as for example discussed in [14, 16], but obtaining efficient algorithms for them is much more difficult.

*Supported by the Slovenian Research Agency, program P1-0297 and project L7-5459.

†FMF and FRI, University of Ljubljana, Slovenia.

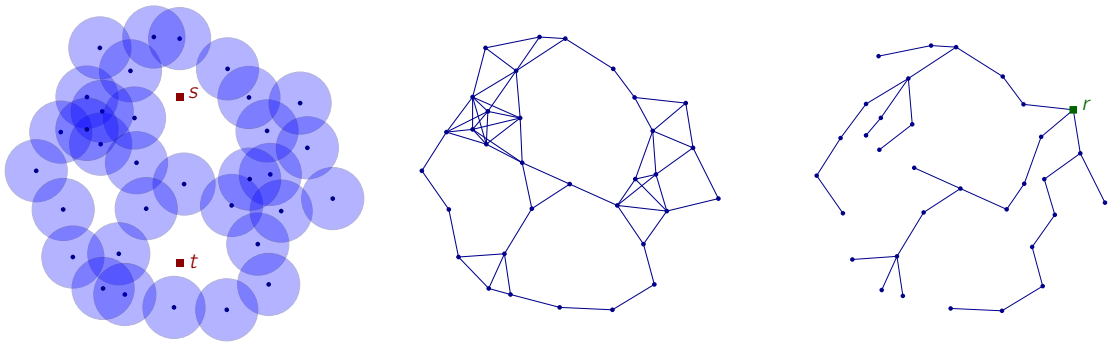


Figure 1: Left: unit disks and two additional points s and t . Middle: intersection graph of the disks. Right: a shortest-path tree in the graph.

While unit disks give a simple model, exploiting the geometric features of the model is often challenging. Shortest paths in unit disk graphs are essential for routing and are a basic subroutine for several other more complex tasks. A somehow unexpected application of shortest paths in unit-disk graphs to boundary recognition is given in [21]. The minimum-separation problem and variants thereof have been considered in [2, 9]. The problem is dual to the barrier-resilience problem considered in [1, 13, 15]. It is not obvious that the minimum-separation problem can be solved optimally in polynomial time, and the known algorithm for this uses as a subroutine shortest paths in unit disk graphs. Thus, both problems considered in this paper are related and it is worth to consider them together.

Our contribution We are aware of three algorithms to compute shortest-path trees in unit disk graphs in $O(n \log n)$ worst-case time: one by Cabello and Jejčič [3], one by Chan and Skrepetos [5], and one Efrat, Itai and Katz [7]. Here we report on an implementation of a modification of the algorithm in [3], and compare it against two obvious alternatives. The only complex ingredients in the algorithm is computing the Delaunay triangulation and static nearest-neighbour queries, but efficient libraries are available for this. The algorithm of [7] would be substantially harder to implement and it has worse constants hidden in the O -notation. The algorithm of [5] for single source shortest paths is implementable and we expect that it would work good in practice. However, this last algorithm has been published only very recently, when we had completed our research.

As mentioned before, it is not obvious that the minimum-separation problem can be solved in polynomial time. In particular, the conference version [10] of [9] gave 2-approximation algorithm for the problem. Cabello and Giannopoulos [2] provide an exact algorithm that takes $O(n^3)$ worst-case time and works for arbitrary shapes, not just disks. In this paper we improve this last algorithm to near-quadratic time for the special case of unit disks. The basic principle of the algorithm is the same, but several additional tools from Computational Geometry exploiting that we have unit disks have to be employed to reduce the worst-case running time. Furthermore, we implement a variant of the new, near-quadratic-time algorithm and report on the experiments.

Assumptions We will assume that *unit disk* means that it has radius $1/2$. Up to scaling the input data, this choice is irrelevant. However, it is convenient for the exposition because then the disks intersect whenever the distance between their centers is 1. The implementation and the experiments also make this assumption.

Henceforth P will be the set of centers of \mathcal{D} . All the computation will be concentrated on P . In particular, we assume that P is known. (For the shortest path problem, one could possibly consider weaker models based on adjacencies.)

We will work with the graph $G_{\leq 1}(P)$ with vertex set P and an edge between two points $p, q \in P$ whenever their Euclidean distance $|pq|$ is at most 1. In the notation we remove the dependency on P and on the distance. Thus we just use G instead of $G_{\leq 1}(P)$. For simplicity of the theoretical exposition we will sometimes assume that G is connected. It is trivial to adapt to the general case, for example treating each connected component separately. The implementation does not make this assumption.

Organization of the paper In Section 2 we discuss the theoretical algorithms for both problems and their guarantees. In Section 3 we discuss the implementations and the experimental results.

2 Description of algorithms

2.1 Shortest-path tree in unit-disk graphs

We describe here the algorithm of Cabello and Jeřčič [3] to compute a shortest path tree in G from a given root point $r \in P$. As it is usually done for shortest path algorithms, we use tables $dist[\cdot]$ and $\pi[\cdot]$ indexed by the points of P to record, for each point $p \in P$, the distance $d_G(s, p)$ and the ancestor of p in a shortest (s, p) -path.

The pseudocode of the algorithm, which we call UNWEIGHTEDSHORTESTPATH, is in Figure 2. We explain the intuition, taken almost verbatim from [3]. We start by computing the Delaunay triangulation $DT(P)$ of P . We then proceed in rounds for increasing values of i , where at round i we find the set W_i of points at distance exactly i in G from the source r . We start with $W_0 = \{r\}$. At round i , we use $DT(P)$ to grow a neighbourhood around the points of W_{i-1} that contains W_i . More precisely, we consider the points adjacent to W_{i-1} in $DT(P)$ as candidate points for W_i . For each candidate point that is found to lie in W_i , we also take its adjacent vertices in $DT(P)$ as new candidates to be included in W_i . For checking whether a candidate point p lies in W_i we use a data structure to find a nearest neighbour of p in W_{i-1} . If the distance from p to its nearest neighbour w in W_{i-1} is smaller than 1, then the shortest path tree is extended by connecting p to w .

Cabello and Jeřčič [3] show that the algorithm correctly computes the shortest-path tree from r . If for nearest neighbors we use a data structure that, for n points, has construction time $T_c(n)$ and query time $T_q(n)$, and the Delaunay triangulation is computed in $T_{DT}(n)$ time, then the algorithm takes $O(T_{DT}(n) + T_c(n) + nT_q(n))$ time. Standard tools in Computational Geometry imply that $T_{DT}(n) = O(n \log n)$, $T_c(n) = O(n \log n)$ and $T_q(n) = O(\log n)$. This leads to the following.

Theorem 1 (Cabello and Jeřčič [3]). *Let P be a set of n points in the plane and let r be a point from P . In time $O(n \log n)$ we can compute a shortest path tree from r in the unweighted graph $G_{\leq 1}(P)$.*

It is clear that, when computing the shortest path tree from several sources, we only need to compute the Delaunay triangulation once.

2.2 Minimum separation with unit-disk

Cabello and Giannopoulos [2] present an algorithm for the minimum separation problem that in the worst-case runs in cubic-time. The algorithm has one feature that is both an

```

UNWEIGHTEDSHORTESTPATH( $P, r$ )
1  build the Delaunay triangulation  $DT(P)$ 
2  for  $p \in P$ 
3       $dist[p] = \infty$ 
4       $\pi[p] = \text{NIL}$ 
5   $dist[r] = 0$ 
6   $W_0 = \{r\}$ 
7   $i = 1$ 
8  while  $W_{i-1} \neq \emptyset$ 
9      build data structure for nearest neighbour queries in  $W_{i-1}$ 
10      $Q = W_{i-1}$  // candidate points
11      $W_i = \emptyset$ 
12     while  $Q \neq \emptyset$ 
13          $q$  an arbitrary point of  $Q$ 
14         remove  $q$  from  $Q$ 
15         for  $qp$  edge in  $DT(P)$ 
16             if  $dist[p] = \infty$ 
17                  $w =$  nearest neighbour of  $p$  in  $W_{i-1}$ 
18                 if  $|pw| \leq 1$ 
19                      $dist[p] = i$ 
20                      $\pi[p] = w$ 
21                     add  $p$  to  $Q$ 
22                     add  $p$  to  $W_i$ 
23      $i = i + 1$ 
24 return  $dist[\cdot]$  and  $\pi[\cdot]$ 

```

Figure 2: Algorithm from [3] to compute a shortest path tree in the unweighted case.

106 advantage and a disadvantage: it works for any reasonable shapes, like segments or ellipses,
107 and not just unit disks. This means that it is very generic, which is good, but it cannot
108 exploit any properties of unit disks.

109 In this section we are going to describe an algorithm to solve the minimum separation
110 problem *for unit disks* in roughly quadratic time. The improvement is based on 3 ingredients.
111 The first ingredient is a reinterpretation of the algorithm of [2] for disks. In the original
112 algorithm, we had to select a point inside each shape. For disks there is a natural, obvious
113 choice, the center of the disk. This allows for a simpler description and interpretation of
114 the algorithm. We provide the description in Section 2.2.1

115 The second ingredient is the efficient algorithm for shortest-path trees for the graph
116 G . The third ingredient is a compact treatment of the edges of G using a few tools from
117 Computational Geometry, namely range trees, point-line duality, and nearest-neighbour
118 searches. This is explained in Section 2.2.2.

119 2.2.1 Generic algorithm specialized for unit disks

120 Let us first introduce some notation. Recall that s and t are the two points to separate.
121 Each walk W in the graph $G = G_{\leq 1}(P)$ defines a planar polygonal curve in the obvious

122 way: we connect the points of P with segments in the order given by W . We will relax the
 123 notation slightly and denote also by W the curve itself. For any spanning tree T of G and
 124 any edge $e \in E(G) \setminus E(T)$, let $\text{cycle}(T, e)$ be the unique cycle in $T + e$. Finally, for any
 125 walk in $G(P)$, let $\text{cr}_2(st, W)$ be the modulo 2 value of the number of crossings between
 126 the segment st and (the curve defined by) W . The following property is implicit in [2] and
 127 explicit in [4]:

128 Let T be any spanning tree of G . The set of unit disks with centers in P
 129 separate s and t if and only if there exists some edge $e \in E(G) \setminus E(T)$ such
 130 that $\text{cr}_2(st, \text{cycle}(T, e)) = 1$.

131 A consequence of this is that finding a minimum separation amounts to finding a
 132 shortest cycle in G that crosses the segment st an odd number of times. Moreover, one can
 133 show that we can restrict our search to a very concrete family cycles, as follows. Consider
 134 any optimal cycle W^* and let r^* be any vertex in W^* . Fix a shortest-path tree T_{r^*} from
 135 r^* in G . When there are many, the choice of T_{r^*} is irrelevant. Then, the set of cycles

$$136 \quad \{\text{cycle}(T_{r^*}, e) \mid e \in E(G) \setminus E(T_{r^*})\}$$

137 contains an optimal solution. This follows from the co-called 3-path condition. We include
 138 here the key property that implies this claim and spell out a self-contained proof. See [2]
 139 for very similar ideas.

140 **Lemma 2.** *Let W^* be a shortest cycle in G that crosses the segment st an odd number of*
 141 *times and let r^* be any vertex in W^* . Fix a shortest-path tree T_{r^*} from r^* in G . Then, the*
 142 *set of cycles $\{\text{cycle}(T_{r^*}, e) \mid e \in E(G) \setminus E(T_{r^*})\}$ contains a shortest cycle of G that crosses*
 143 *st an odd number of times.*

Proof. For any points p and q of P , let $T_{r^*}[p \rightarrow q]$ be the unique path contained in T_{r^*}
 from p to q . For every edge pq of G , let $\text{walk}(T_{r^*}, pq)$ be the closed walk that follows
 $T_{r^*}[r^* \rightarrow p]$, then the edge pq , and finally $T_{r^*}[q \rightarrow r^*]$. We then have the following relation
 modulo 2:

$$\begin{aligned} & \sum_{pq \in W^*} \text{cr}_2(st, \text{walk}(T_{r^*}, pq)) \\ &= \sum_{pq \in W^*} (\text{cr}_2(st, T_{r^*}[r^* \rightarrow p]) + \text{cr}_2(st, pq) + \text{cr}_2(st, T_{r^*}[q \rightarrow r^*])) \\ &= \sum_{pq \in W^*} \text{cr}_2(st, pq) \\ &= \text{cr}_2(st, W^*) \\ &= 1. \end{aligned}$$

144 In the second equality we have used that each path $T_{r^*}[r^* \rightarrow p]$ and its reverse $T_{r^*}[p \rightarrow r^*]$
 145 appears an even number of times in the sum, and thus cancel out modulo 2. Parity implies
 146 that, for some edge p_0q_0 of W^* , we have $\text{cr}_2(st, \text{walk}(T_{r^*}, p_0q_0)) = 1$. It must be that
 147 $p_0q_0 \notin E(T_{r^*})$ because for each edge pq of T_{r^*} it holds $\text{cr}_2(st, \text{walk}(T_{r^*}, pq)) = 0$.

148 Since $\text{cr}_2(st, \text{walk}(T_{r^*}, p_0q_0)) = \text{cr}_2(st, \text{cycle}(T_{r^*}, p_0q_0))$ because the path from r^* to
 149 the lowest common ancestor of p and q in T_{r^*} is counted twice on the left side of the
 150 equality, we have $\text{cr}_2(st, \text{cycle}(T_{r^*}, p_0q_0)) = 1$.

151 Since r^* is a vertex of W^* and p_0q_0 is an edge of W^* , the length of W^* is at least the
 152 length of $T_{r^*}[r^* \rightarrow p_0]$ plus 1, for the edge p_0q_0 , plus the length of $T_{r^*}[q_0 \rightarrow r^*]$. However,

153 this second part is exactly the length of $walk(T_{r^*}, p_0q_0)$, which is at least the length of
 154 $cycle(T_{r^*}, p_0q_0)$.

155 We have shown that, for some edge $p_0q_0 \in E(G) \setminus E(T_{r^*})$, the cycle $cycle(T_{r^*}, p_0q_0)$ is
 156 not longer than W^* and crosses st an odd number of times. The result follows. \square

157 Since we do not know a vertex r^* in the shortest cycle of G , we just try all possible roots
 158 as candidates. (This leads to the option of having a randomized algorithm, by selecting
 159 some roots at random, for the case where the optimal solution is large.) Thus, for each
 160 vertex r of G , we fix a shortest-path tree T_r from r in G , and then the size of the optimal
 161 solution is given by

$$162 \quad \min\{1 + d_G(r, p) + d_G(r, q) \mid r \in P, pq \in E(G) \setminus E(T_r), \text{cr}_2(st, cycle(T_r, pq)) = 1\}.$$

163 The values $\text{cr}_2(st, cycle(T_r, e))$ can be computed in constant amortized time per edge
 164 with some easy bookkeeping, as follows. Consider a fixed tree T_r . For each point $p \in P$ we
 165 store $N[p]$ as the parity of the number of crossings of the path in T_r from r to p . When
 166 p is not the root, the value $N[p]$ can be computed from the value of its parent $\pi[p]$ in T_r
 167 using that $N[p] = N[\pi[p]] + \text{cr}_2(st, p\pi[p])$. In the algorithm we have written it this way
 168 (lines 4–6), but one can also compute the values at the time of computing the shortest
 169 path tree T_r .

We then have for each shortest-path tree T_r

$$\begin{aligned} \forall pq \in E(G) \setminus E(T_r) : \quad & \text{cr}_2(st, cycle(T_r, pq)) = N[p] + N[q] + \text{cr}_2(st, pq) \pmod{2} \\ \forall pq \in E(T_r) : \quad & 0 = N[p] + N[q] + \text{cr}_2(st, pq) \pmod{2} \end{aligned}$$

170 because crossings that are counted twice cancel out modulo 2. In particular, the path in
 171 T_r from r to the lowest common ancestor of p and q is counted twice. This implies that we
 172 can just check for *all* edges pq of G whether the sum $N[p] + N[q] + \text{cr}_2(st, pq)$ is 0 modulo
 173 2. The final resulting algorithm, denoted as **GENERICMINIMUMSEPARATION**, is given in
 174 Figure 3.

```

GENERICMINIMUMSEPARATION( $P, s, t$ )
1   $best = \infty$  // length of the best separation so far
2  for  $r \in P$ 
3       $(dist[\ ], \pi[\ ]) =$  shortest path tree from  $r$  in  $G(P)$ 
      // Compute  $N[\ ]$ 
4       $N[r] = 0$ 
5      for  $p \in P \setminus \{r\}$  in non-decreasing values of  $dist[p]$ 
6           $N[p] = N[\pi[p]] + \text{cr}_2(st, p\pi[p]) \pmod{2}$ 
7      for  $pq \in E(G(P))$ 
8          if  $N[p] + N[q] + \text{cr}_2(pq, st) \pmod{2} = 1$ 
9               $best = \min\{best, dist[p] + dist[q] + 1\}$ 
10 return  $best$ 

```

Figure 3: Adaptation of the generic algorithm to compute the minimum separation for unit disks.

175 Let us look into the time complexity of the algorithm. For each point $r \in P$ we have to
 176 compute a shortest-path tree in G . This can be done in $O(n \log n)$ in our case, as discussed

177 in Section 2.1. Then, for each edge pq of G some constant amount of work is done. Thus
 178 for each point r we spend $O(n \log n + |E(G)|)$. This is cubic in the worst-case. We could
 179 get an improved running time if we can treat all the edges of G compactly. This is what
 180 we explain next.

181 2.2.2 Compact treatment of edges

182 From now on we will assume that s is the origin and t is the point $(0, \tau)$, with $\tau \geq 0$. Thus,
 183 the segment st is vertical and t is above s . The implementation just assumes that st is
 184 vertical with s below t . A simple rigid transformation can be applied to the input to get
 185 to this setting.

186 We will use the data structure in the following lemma. It is essentially a multi-level
 187 data structure consisting of a 2-dimensional range tree T with a data structure for nearest
 188 neighbour at each node of the secondary structure of T .

189 **Lemma 3.** *Let B be a set of n points with positive x -coordinates. We can preprocess
 190 B in $O(n \log^3 n)$ time such that, for any query point a with negative x -coordinate, we
 191 can decide in $O(\log^3 n)$ time whether the set $\{b \in B \mid ab \text{ intersects } \sigma \text{ and } |ab| \leq 1\}$ is
 192 empty. The same data structure can handle queries to know whether the set $\{b \in B \mid$
 193 $ab \text{ does not intersect } \sigma \text{ and } |ab| \leq 1\}$ is empty.*

194 *Proof.* We are going to use point-line duality and range trees. These are standard concepts
 195 in Computational Geometry; see for example [6, Chapters 5 and 8]. We assume that the
 196 reader is familiar with the topic. Figure 4 may be helpful in the following discussion.

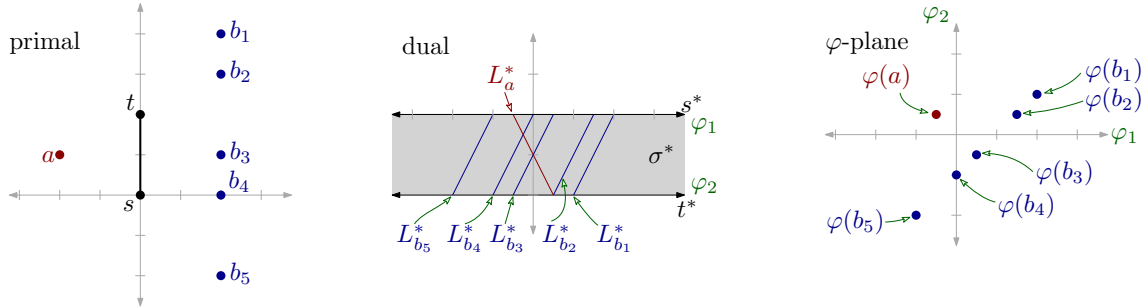


Figure 4: Transformation in the proof of Lemma 3.

197 We use the following precise point-line duality: the non-vertical line $\ell \equiv y = mx + c$ is
 198 mapped to the point $\ell^* = (m, -c)$ and vice-versa. Let \mathbb{L} be the set of non-vertical lines.
 199 Let σ be the line segment st . Let σ^* be the set of points dual to non-vertical lines that
 200 intersect σ . Thus

$$201 \quad \sigma^* = \{\ell^* \mid \ell \in \mathbb{L}, \ell \cap \sigma \neq \emptyset\}.$$

202 Since we assumed that $s = (0, 0)$ and $t = (0, \tau)$, in the dual space σ^* is the horizontal slab

$$203 \quad \sigma^* = \{(m, -c) \in \mathbb{R}^2 \mid 0 \leq c \leq \tau\}.$$

204 For every point $p \in \mathbb{R}^2$, outside the y -axis, let L_p^* be the set of points dual to the lines
 205 through p that intersect σ :

$$206 \quad L_p^* = \{\ell^* \mid \ell \in \mathbb{L}, p \in \ell, \text{ and } \sigma \cap \ell \neq \emptyset\}.$$

207 In the dual space, L_p^* is a segment with endpoints $(\varphi_1(p), 0)$ and $(\varphi_2(p), -\tau)$, for some
 208 values $\varphi_1(p)$ and $\varphi_2(p)$ that are easily computable. Namely, $\varphi_1(p)$ is the slope of the line

through p and $(0, 0)$ while $\varphi_2(p)$ is the slope of the line through p and $(0, \tau)$. The segment L_p^* is contained in the slab σ^* and has the endpoints on different boundaries of σ^* . Finally, define the mapping $\varphi(p) = (\varphi_1(p), \varphi_2(p))$. Thus, φ maps points in the plane with nonzero x -coordinate to points in the plane.

Let a be any point to the left of the y -axis and let b be a point to the right of the y -axis. The segment ab intersects σ if and only if L_a^* intersects L_b^* . Namely, an intersection of L_a^* and L_b^* is dual to the line through a and b . The segments L_a^* and L_b^* intersect if and only if the order of their endpoints on the boundaries of σ^* are reversed. Moreover, since a is to the left of the y -axis and b is to the right of the y -axis, if the segment ab intersects σ , then $\varphi_1(a)$, the slope of the line through a and $(0, 0)$, is smaller than $\varphi_1(b)$, the slope of the line through b and $(0, 0)$. Thus we have the following property:

$$ab \cap \sigma \neq \emptyset \iff \varphi_1(a) \leq \varphi_1(b) \text{ and } \varphi_2(a) \geq \varphi_2(b).$$

Given a point a to the left of the y axis, the set of points $b \in B$ with the property that ab intersects σ corresponds to the points b with $\varphi(b)$ in the bottom-right quadrant with apex $\varphi(a)$.

We can use a 2-dimensional range tree to store the point set $\varphi(B)$, where each point $b \in B$ is identified with its image $\varphi(b)$. Moreover, for each node v in the secondary level of the range tree, we store a data structure for nearest neighbours for the canonical set $P(v)$ of points that are stored below v in the secondary structure.

For any query $a \in A$, the points $b \in B$ such that ab intersects σ are obtained by querying the 2-dimensional range tree for the points of $\varphi(B)$ contained in the quadrant

$$\{(x, y) \mid \varphi_1(a) \leq x \text{ and } \varphi_2(a) \geq y\}.$$

This means that we get the set $\{b \in B \mid ab \text{ intersects } \sigma\}$ as the union of canonical subsets $P(v_1), \dots, P(v_k)$ for $k = O(\log^2 n)$ nodes in the secondary levels of the 2-dimensional range tree. For each such canonical subset $P(v_i)$, we query for the nearest neighbour of a . If for some v_i we get a nearest neighbour at distance at most 1 from a , then we know that $\{b \in B \mid ab \text{ intersects } \sigma \text{ and } |ab| \leq 1\}$ is non-empty. Otherwise the set is empty.

The construction time of the 2-dimensional range tree is $O(n \log n)$. Each point appears in $O(\log^2 n)$ canonical subsets $P(v)$. This means that $\sum_v |P(v)| = O(n \log^2 n)$, where the sum iterates over all nodes v in the secondary data structure. Since for each node v in the secondary level we build a data structure for nearest neighbours, which takes $O(|P(v)| \log |P(v)|)$, the total construction time is $O(n \log^3 n)$. For the query time, the standard 2-dimensional range tree takes $O(\log^2 n)$ time to find the $O(\log^2 n)$ nodes v_1, \dots, v_k such that

$$\bigcup_{i=1}^k P(v_i) = \{b \in B \mid ab \text{ intersects } \sigma\},$$

and then we need additional $O(\log n)$ time per node to query for a nearest neighbor.

Answering the queries for $\{b \in B \mid ab \text{ does not intersect } \sigma \text{ and } |ab| \leq 1\}$ is done similarly (and the same data structure works), we just have to query for 2 of the other quadrants. (The top-left quadrant of $\varphi(a)$ is always empty.) \square

Inside the data structure of Lemma 3 we are using a data structure for nearest neighbours with construction time $O(n \log n)$ and query time $O(\log n)$. If we would use another data structure for nearest neighbours with construction time $T_c(n)$ and query time $T_q(n)$, then the construction time in Lemma 3 becomes $O(T_c(n \log^2 n))$ and the query time is $O(T_q(n) \cdot \log^2 n)$.

From the theoretical perspective it would be more efficient to compute the union

$$\bigcup_{b \in B} \{(x, y) \in \mathbb{R}^2 \mid x < 0, |(x, y)b| \leq 1, (x, y) \text{ intersects } \sigma\}$$

and make point location there. Since the regions cannot have many crossings, good asymptotic bounds can be obtained. However, such approach seems to be only of theoretical interest and the improvement on the overall result is rather marginal.

Consider now a fixed root r . Assume that we have computed the shortest path tree T_r and the corresponding tables $\pi[\]$, $\text{dist}[\]$ and $N[\]$, as discussed in Section 2.2.1. We group the points by their distance from r :

$$W_i = \{p \in P \mid \text{dist}[p] = i\}, \quad i = 0, 1, \dots$$

A standard property of BFS trees, that also holds here, is that all the distances from the root for any two adjacent vertices differ by at most 1. That is, the neighbours of a point $p \in P$ in G are contained in $W_{\text{dist}[p]-1} \cup W_{\text{dist}[p]} \cup W_{\text{dist}[p]+1}$. We will exploit this property.

We make groups L_i^j and R_i^j (where L stands for left and R for right) defined by

$$\begin{aligned} L_i^j &= \{p \in P \mid \text{dist}[p] = i, p.x < 0, N[p] = j\}, \quad \text{where } j = 0, 1 \text{ and } i = 0, 1, \dots \\ R_i^j &= \{p \in P \mid \text{dist}[p] = i, p.x > 0, N[p] = j\}, \quad \text{where } j = 0, 1 \text{ and } i = 0, 1, \dots \end{aligned}$$

We are interested in edges pq of G such that $N[p] + N[q] + \text{cr}_2(st, pq) = 1 \pmod{2}$. Up to symmetry (exchanging p and q), this is equivalent to pairs of points (p, q) in one of the following two cases:

- for some $i \in \mathbb{N}$ and some $j \in \{0, 1\}$, we have $p \in L_i^j \cup R_i^j$, $q \in L_i^{1-j} \cup R_i^{1-j} \cup L_{i-1}^{1-j} \cup R_{i-1}^{1-j}$, $|pq| \leq 1$, and pq does not cross st ;
- for some $i \in \mathbb{N}$ and some $j \in \{0, 1\}$, we have $p \in L_i^j \cup R_i^j$, $q \in L_i^j \cup R_i^j \cup L_{i-1}^j \cup R_{i-1}^j$, $|pq| \leq 1$, and pq crosses st .

Each one of these cases can be solved efficiently. Up to symmetry, we have the following cases:

- If we want to search the candidates $(p, q) \in L_i^j \times L_{i'}^{1-j}$ (that cannot cross st since they are on the same side of the y -axis), we first preprocess $L_{i'}^{1-j}$ for nearest neighbours. Then, for each point p in L_i^j , we query the data structure to find its nearest neighbour q_p in $L_{i'}^{1-j}$. If for some p we get that $|pq_p| \leq 1$, then we have obtained an edge pq_p of G with $\text{cr}_2(\text{cycle}(T_r, pq_p)) = 1$ and $\text{dist}[p] + \text{dist}[q_p] + 1 = i + i' + 1$. If for each p we have $|pq_p| > 1$, then $L_i^j \times L_{i'}^{1-j}$ does not contain any edge of G . The overall running time, if $m = |L_i^j| + |L_{i'}^{1-j}|$, is $O(m \log m)$.
- If we want to search the candidates $(p, q) \in L_i^j \times R_{i'}^j$ such that pq crosses st , we first preprocess $R_{i'}^j$ as discussed in Lemma 3 into a data structure. Then, for each point $p \in L_i^j$ we query the data structure (for crossing st). If we get some nonempty set, then there is an edge pq of G with $p \in L_i^j$, $q \in R_{i'}^j$, $\text{cr}_2(\text{cycle}(T_r, pq)) = 1$ and $\text{dist}[p] + \text{dist}[q] + 1 = i + i' + 1$. Otherwise, there is no edge $pq \in L_i^j \times R_{i'}^j$ that crosses st . The overall running time, if $m = |L_i^j| + |R_{i'}^j|$, is $O(m \log^3 m)$.

• If we want to search the candidates $(p, q) \in L_i^j \times R_{i'}^{1-j}$ such that pq does not cross st , we first preprocess $R_{i'}^{1-j}$ as in Lemma 3 into a data structure. Then, for each point $p \in L_i^j$ we query the data structure (for not crossing st). The remaining discussion is like in the previous item.

We conclude that each of the cases can be done in $O(m \log^3 m)$ worst-case time, where m is the number of points involved in the case. Iterating over all possible values i , it is now easy to convert this into an algorithm that spends $O(n \log^3 n)$ time per root r . We summarize the result we have obtained. This improves for the case of unit disks the previous, generic algorithm.

Theorem 4. *The minimum-separation problem for n unit disks can be solved in $O(n^2 \log^3 n)$ time.*

Proof. Let P be the centers of the disks and, as before, consider the graph $G = G_{\leq 1}(P)$. For each root $r \in P$ we build the shortest-path tree and the sets $W_i, L_i^0, L_i^1, R_i^0, L_i^1$ for all i in $O(n \log n)$ time. We then have at most n iterations where, at iteration i , we spend $O(|W_i \cup W_{i-1}| \log^3 |W_i \cup W_{i-1}|)$ time. Since the sets W_i are disjoint, adding over i , this means that we spend $O(n \log^3 n)$ time per root $r \in P$.

Correctness follows from the foregoing discussion and the fact that the algorithm is computing the same as the generic algorithm. \square

The resulting new algorithm is given in Figure 5. As before, the variable *best* stores the length of the shortest cycle (or actually rooted closed walk) that we have found so far. We can start setting *best* = $n + 1$ at start. If eventually we finish with the value *best* = $n + 1$, it means that there is no feasible solution for the separation problem. When we consider a root r we are interested in closed walks rooted at r and length at most *best*. Since any closed walk through a vertex of W_i has length at least $2i$, we only need to consider indices i such that $2i < \text{best}$. Moreover (and this is not described in the algorithm, but it is done in the implementation), we can consider first the pairs that give walks for length $2i$ first, like for example $L_i^0 \times L_{i-1}^1$ and then the ones that give length $2i + 1$, like for example $L_i^0 \times L_i^1$. If we use this order, as soon as we find an edge in the while-loop, we can finish the work for the root r , and move onto the next root.

3 Implementation and experiments

We have implemented the algorithms of Section 2 in C++ using CGAL version 4.6.3 [20] because it provides the more complex procedures we need: Delaunay triangulations and Voronoi diagrams [12], range trees [17], and nearest neighbours [19]. Although in some cases we had to make small modifications, it was very helpful to have the CGAL code available as a starting point. The coordinates of the points were Cartesian doubles.

Experiments were carried out in a laptop with CPU i7-6700HQ at 2.60 Ghz, 8GB of RAM, and Windows 10. *All times we report are in seconds.*

Data generation Data points were generated uniformly at random in the following polygonal domains: rectangles without holes, rectangles with a "small" rectangular hole, rectangles with a "large" rectangular hole, rectangles with 4 "small" rectangular holes, and rectangles with 4 "large" rectangular holes. The precise proportions of the domains with holes are in Figures 6 and 7. We generated 1K, 2K, 5K, 10K, 20K and 50K points for the cases where the outer rectangle has sizes $4 \times 1, 8 \times 2, \dots, 128 \times 32$. The data was generated

```

SEPARATIONUNITDISKSCOMPACT( $P, s, t$ )
1   $best = n + 1$  // length of the best separation so far
2  for  $r \in P$ 
3      ( $dist[\ ], \pi[\ ]$ ) = shortest path tree from  $r$  in  $G_{\leq 1}(P)$ 
      // Compute the levels  $W_i$ 
4      for  $i = 0 \dots n$ 
5           $W_i =$  new empty list
6      for  $p \in P$ 
7          add  $p$  to  $W_{dist[p]}$ 
      // Compute  $N[\ ]$  for the elements of  $W_i$  and
      // and construct  $L_i^0, L_i^1, R_i^0, R_i^1$ 
8       $N[r] = 0$ 
9      for  $i = 1 \dots n$ 
10         for  $p \in W_i$ 
11              $N[p] = N[\pi[p]] + \mathbf{cr}_2(st, p\pi[p]) \pmod{2}$ 
12             if  $p$  to the left of the  $y$ -axis
13                 add  $p$  to  $L_i^{N[p]}$ 
14             if  $p$  to the right of the  $y$ -axis
15                 add  $p$  to  $R_i^{N[p]}$ 
16      $i = 1$ 
17     while  $2i < best$  and  $W_i \neq \emptyset$ 
        // length  $2i$ ; within each side of the  $y$ -axis
18     search candidates in  $L_i^0 \times L_{i-1}^1$ 
19     search candidates in  $L_i^1 \times L_{i-1}^0$ 
20     search candidates in  $R_i^0 \times R_{i-1}^1$ 
21     search candidates in  $R_i^1 \times R_{i-1}^0$ 
        // length  $2i$ ; across  $y$ -axis crossing  $\sigma$ 
22     search candidates in  $L_i^0 \times R_{i-1}^0$  crossing  $\sigma$ 
23     search candidates in  $L_i^1 \times R_{i-1}^1$  crossing  $\sigma$ 
24     search candidates in  $L_{i-1}^0 \times R_i^0$  crossing  $\sigma$ 
25     search candidates in  $L_{i-1}^1 \times R_i^1$  crossing  $\sigma$ 
        // length  $2i$ ; across  $y$ -axis not crossing  $\sigma$ 
26     search candidates in  $L_i^0 \times R_{i-1}^1$  not crossing  $\sigma$ 
27     search candidates in  $L_i^1 \times R_{i-1}^0$  not crossing  $\sigma$ 
28     search candidates in  $L_{i-1}^0 \times R_i^1$  not crossing  $\sigma$ 
29     search candidates in  $L_{i-1}^1 \times R_i^0$  not crossing  $\sigma$ 
        // length  $2i + 1$ ; within each side of the  $y$ -axis
30     search candidates in  $L_i^0 \times L_i^1$ 
31     search candidates in  $R_i^0 \times R_i^1$ 
        // length  $2i + 1$ ; across  $y$ -axis crossing  $\sigma$ 
32     search candidates in  $L_i^0 \times R_i^0$  crossing  $\sigma$ 
33     search candidates in  $L_i^1 \times R_i^1$  crossing  $\sigma$ 
        // length  $2i + 1$ ; across  $y$ -axis not crossing  $\sigma$ 
34     search candidates in  $L_i^0 \times R_i^1$  not crossing  $\sigma$ 
35     search candidates in  $L_i^1 \times R_i^0$  not crossing  $\sigma$ 
36      $i = i + 1$ 
37 return  $best$ 

```

Figure 5: New algorithm for minimum separation with unit disks.

330 once and stored. For the minimum-separation problem s was placed in the middle of a
 331 hole and t vertically above s in the outer face. Some of these domains are not meaningful
 332 for the minimum-separation problem because the disks centered at the points cover s .

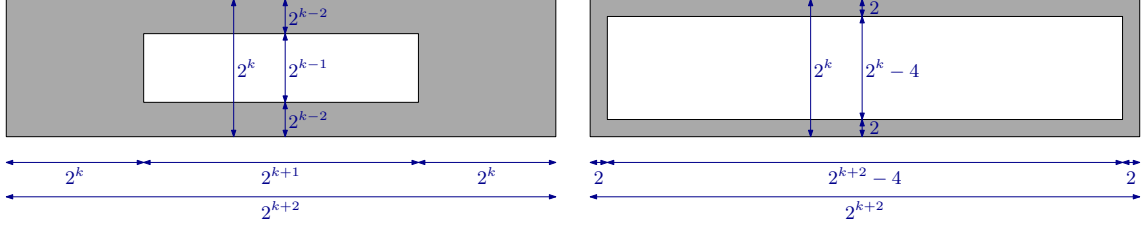


Figure 6: Data generation with a small hole (left) and a large hole (right).

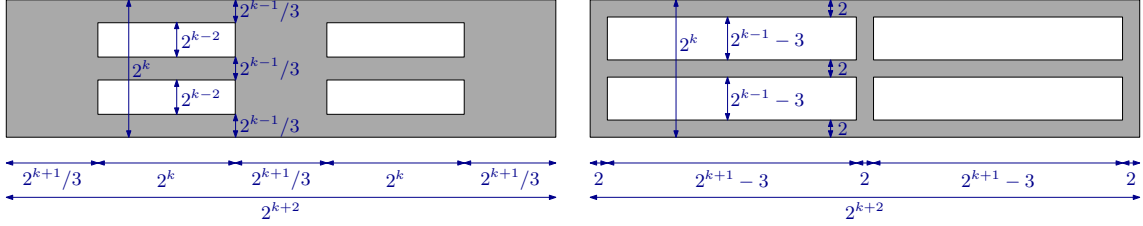


Figure 7: Data generation with four small holes (left) and four large holes (right).

333 **Shortest-path tree in unit-disk graphs** We have implemented the algorithm de-
334 scribed in Section 2.1. For the shortest-path tree we used the Delaunay triangulation as
335 provided by CGAL. The data structure for nearest neighbour queries is a small extension
336 of the one provided by [12], which in turn is based on the Delaunay triangulation. When
337 making a query for nearest neighbour of p in W_{i-1} (line 17 in Figure 2), we have the option
338 to provide an extra parameter that acts as some sort of hint: if the nearest neighbour is
339 near the hint, the algorithm is faster. For our implementation, we exploit this as follows.
340 Consider an iteration of the while loop (lines 13–22). If the point q is from W_{i-1} then we
341 use a point in a face of $DT(W_{i-1})$ incident to q as the hint for all the points p considered in
342 the iteration. If the point q is not from W_{i-1} , then we already know that $q \in W_i$ and thus
343 $\pi[q] \in W_{i-1}$. In this case we use a point in a face of $DT(W_{i-1})$ incident to $\pi[q]$ as hint
344 for all the points p considered in the iteration. Using such hints reduced the running time
345 substantially, so we used this feature in the implementation. Note that this improvement
346 does not come with guarantees in the worst-case. In the tables we refer to this algorithm
347 as **SSSP**.

348 We compared the implementation with two obvious alternative algorithms to compute
349 shortest-path trees. The first alternative is to build the graph $G = G_{\leq 1}(P)$ explicitly. Thus,
350 for each pair of points p, q we check whether their distance is at most one and add an edge
351 to a graph data structure. We can then use breadth-first-search (BFS) from the given root
352 r . The preprocessing is quadratic, and the time spent to compute a shortest-path tree
353 depends on the density of the graph G . In the tables we refer to this algorithm as **BFS**.

354 The second alternative we consider is to use a unit-length grid. Two points (x, y) and
355 (x', y') are in the same grid cell if and only if $(\lfloor x \rfloor, \lfloor y \rfloor) = (\lfloor x' \rfloor, \lfloor y' \rfloor)$. We store all the
356 points of a grid cell c in a list $\ell(c)$. The non-empty lists $\ell(c)$ are stored in a dictionary,
357 where the bottom-left corner of the cell is used as key. We can then run some sort of BFS
358 using this structure. The list $\ell(c)$ for a cell c maintains the points that have not been
359 visited by the BFS tree yet. When processing a point p in a cell c , we have to treat all
360 the points in the lists of c and its 8 adjacent cells as candidate points. Any point that is
361 adjacent to p is then removed from the list of its cell. The preprocessing is linear, and
362 the time spent to compute a shortest-path tree depends on the distribution of the points.
363 It is easy to produce cases where the algorithm would need quadratic time. For each

364 shortest-path tree we compute the lists and the dictionary anew. (This step is very fast in
365 any case.) In the tables we refer to this algorithm as *grid*.

366 As mentioned earlier, we did not implement the algorithm of Chan and Skrepetos [5]
367 because of time constraints. We expect that it would work good.

368 The measured times are in Tables 1–5. For SSSP and BFS we report the preprocessing
369 time that is independent of the source (like building the Delaunay triangulation or building
370 the graph) and the average time spent for a shortest-path tree over 50 choices of the root.
371 For grid we just report the total running time; assigning points to the grid cells and putting
372 them into a dictionary is almost negligible. As it can be seen, the results for SSSP are
373 very much independent of the shape and, for dense point sets it outperforms the other
374 algorithms.

375 While the algorithm SSSP has guarantees in the worst case, for BFS and grid one
376 can construct instances where the behavior will be substantially bad. For example, to
377 the instance with 10K points in a rectangle of size 32×8 with a small hole we added 1K
378 points quite cluttered. The increase in time with respect to the original instance was for
379 SSSP 9,7% (preprocessing) and 13,6% (one root), for BFS it was 21,9% (preprocessing)
380 and 56,5% (one root), and for grid it was 25%.

Rectangle without holes		20K points					
size rectangle		4×1	8×2	16×4	32×8	64×16	128×32
SSSP preprocessing		0.018	0.018	0.018	0.018	0.019	0.021
SSSP average/root		0.011	0.012	0.012	0.012	0.013	0.013
BFS preprocessing		18.70	13.46	12.03	11.40	11.32	11.13
BFS average/root		2.437	1.018	0.321	0.069	0.017	0.005
grid		1.309	1.130	0.474	0.160	0.060	0.035
		50K points					
SSSP preprocessing		0.051	0.050	0.053	0.051	0.051	0.053
SSSP average/root		0.034	0.037	0.037	0.036	0.035	0.036
BFS preprocessing		>2min	86.12	74.76	74.15	72.41	71.49
BFS average/root	memory limit		6.524	2.422	0.510	0.119	0.035
grid		6.297	7.125	3.188	0.923	0.301	0.139

Table 1: Times for shortest paths in rectangles without holes.

381 **Minimum separation with unit-disk** We have implemented the algorithm GENER-
382 ICMINIMUMSEPARATION and the new algorithm based on a compact treatment of the edges.
383 The shortest-path trees are constructed using the algorithm of Section 2.1. The table $N[]$
384 and the sets $L_i^0, L_i^1, R_i^0, R_i^1$ are constructed at the time of computing the shortest-path tree.

385 In the data structure of Lemma 3, we do use a 2-dimensional tree as the primary
386 structure, making some modifications of [17]. In the secondary structure, for nearest
387 neighbour, instead of using Voronoi diagrams, we used a small modification of the *kd*-trees
388 implemented in [19]. In some preliminary experiments this seemed to be a better choice.
389 In our modification, we make a range search query for points at distance at most 1, and
390 finish the search whenever we get the first point. In the new algorithm, before calling
391 to the function to candidates pairs, like for example $L_i^0 \times R_1^i$, we test that both sets are
392 non-empty. This simple test reduced the time by 30-50% in our test cases.

393 Besides the new algorithm we also implemented the generic algorithm of Section 2.2.1.
394 The measured times are in Tables 6–7. For the case of 4 holes we always put t above the

Rectangle 1 small hole		10K points				
size rectangle	4×1	8×2	16×4	32×8	64×16	128×32
SSSP preprocessing	0.011	0.012	0.009	0.010	0.010	0.009
SSSP average/root	0.004	0.005	0.005	0.005	0.006	0.006
BFS preprocessing	3.724	3.033	2.890	2.826	2.874	2.841
BFS average/root	0.587	0.248	0.078	0.021	0.006	0.002
grid	0.258	0.313	0.119	0.049	0.022	0.015

		20K points				
SSSP preprocessing	0.019	0.019	0.019	0.019	0.018	0.023
SSSP average/root	0.010	0.012	0.011	0.012	0.013	0.013
BFS preprocessing	15.22	13.47	11.51	11.66	11.73	11.38
BFS average/root	2.402	1.045	0.369	0.088	0.023	0.006
grid	1.122	1.339	0.461	0.181	0.074	0.036

Table 2: Times for shortest paths in rectangles with a small hole.

Rectangle 1 large hole		5K points			10K points		
size rectangle	32×8	64×16	128×32	32×8	64×16	128×32	
SSSP preprocessing	0.004	0.005	0.005	0.009	0.010	0.010	
SSSP average/root	0.002	0.002	0.002	0.005	0.005	0.005	
BFS preprocessing	0.751	0.767	0.742	2.783	3.175	2.804	
BFS average/root	0.006	0.003	0.002	0.025	0.012	0.006	
grid	0.018	0.012	0.008	0.053	0.032	0.022	

Table 3: Times for shortest paths in rectangles with a large hole.

rectangle and s in one hole. It seems that the choice of the hole does not substantially affect the experimental time in our setting.

To show that our new algorithm can work substantially faster than the generic algorithm, we created an instance where we expect so. For this we take the a rectangle of size 32×8 with one small hole, the original 2K points, and add 1K extra points quite cluttered around the portion of st in the domain. The generic algorithm took XXX seconds and the new algorithm took XXX seconds. If instead we add 2K points...

References

- [1] S. Bereg and D. G. Kirkpatrick. Approximating barrier resilience in wireless sensor networks. *Proc. 5th ALGOSENSORS*, pp. 29–40. Springer, LNCS 5804, 2009, http://dx.doi.org/10.1007/978-3-642-05434-1_5.
- [2] S. Cabello and P. Giannopoulos. The complexity of separating points in the plane. *Algorithmica* 74(2):643–663, 2016, <http://dx.doi.org/10.1007/s00453-014-9965-6>.
- [3] S. Cabello and M. Jeřič. Shortest paths in intersection graphs of unit disks. *Comput. Geom.* 48(4):360–367, 2015, <http://dx.doi.org/10.1016/j.comgeo.2014.12.003>.
- [4] S. Cabello and M. Kerber. Semi-dynamic connectivity in the plane. *Algorithms and Data Structures - 14th International Symposium, WADS 2015. Proceedings*, pp. 115–126. Springer, Lecture Notes in Computer Science 9214, 2015, http://dx.doi.org/10.1007/978-3-319-21840-3_10.

Rectangle 4 small holes		10K points			20K points		
size rectangle	32 × 8	64 × 16	128 × 32	32 × 8	64 × 16	128 × 32	
SSSP preprocessing	0.010	0.011	0.009	0.018	0.018	0.019	
SSSP average/root	0.005	0.006	0.007	0.012	0.013	0.014	
BFS preprocessing	2.925	2.861	2.866	11.97	11.93	11.59	
BFS average/root	0.020	0.006	0.002	0.085	0.022	0.006	
grid	0.048	0.024	0.016	0.190	0.070	0.040	

Table 4: Times for shortest paths in rectangles with 4 small holes.

Rectangle 4 large holes	5K points			10K points		
	size rectangle	32 × 8	64 × 16	128 × 32	32 × 8	64 × 16
SSSP preprocessing	0.004	0.005	0.005	0.013	0.015	0.009
SSSP average/root	0.003	0.003	0.003	0.006	0.005	0.005
BFS preprocessing	0.715	0.734	0.717	2.897	2.910	3.182
BFS average/root	0.005	0.002	0.001	0.019	0.008	0.004
grid	0.013	0.010	0.008	0.045	0.026	0.020

Table 5: Times for shortest paths in rectangles with 4 large holes.

- [5] T. M. Chan and D. Skrepetos. All-pairs shortest paths in unit-disk graphs in slightly subquadratic time. *27th International Symposium on Algorithms and Computation, ISAAC 2016*, pp. 24:1–24:13, LIPIcs 64, 2016, <http://dx.doi.org/10.4230/LIPIcs.ISAAC.2016.24>.
- [6] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 3rd ed. edition, 2008, <http://dx.doi.org/10.1007/978-3-540-77974-2>.
- [7] A. Efrat, A. Itai, and M. J. Katz. Geometry helps in bottleneck matching and related problems. *Algorithmica* 31(1):1–28, 2001, <http://dx.doi.org/10.1007/s00453-001-0016-8>.
- [8] J. Gao and L. Guibas. Geometric algorithms for sensor networks. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 370(1958):27–51, 2011, <http://dx.doi.org/10.1098/rsta.2011.0215>.
- [9] M. Gibson, G. Kanade, R. Penninger, K. Varadarajan, and I. Vigan. On isolating points using unit disks. *J. of Computational Geometry* 7(1):540–557, 2016, <http://dx.doi.org/10.20382/jocg.v7i1a22>.
- [10] M. Gibson, G. Kanade, and K. Varadarajan. On isolating points using disks. *Proc. 19th ESA*, pp. 61–69. Springer, LNCS 6942, 2011, http://dx.doi.org/10.1007/978-3-642-23719-5_6.
- [11] M. L. Huson and A. Sen. Broadcast scheduling algorithms for radio networks. *IEEE MILCOM '95*, vol. 2, pp. 647–651 vol.2, 1995, <http://dx.doi.org/10.1109/MILCOM.1995.483546>.
- [12] M. Karavelas. 2D voronoi diagram adaptor. *CGAL User and Reference Manual*, 4.6 edition. CGAL Editorial Board, 2015, <http://doc.cgal.org/4.6/Manual/packages.html#PkgVoronoiDiagramAdaptor2Summary>.

Rectangle 1 small hole	2K points			
	size rectangle	8×2	16×4	32×8 64×16
new separation algorithm		41	41	32 30
generic algorithm		155	60	26 21
cycle length		9	20	46 126

Table 6: Times for minimum separation with 1 hole.

Rectangle 4 holes	2K points, small holes			5K points, large holes		
	size rectangle	32×8	64×16	128×32	32×8	64×16 128×32
new separation algorithm		20	18	5.6	233	259 240
generic algorithm		26	17	4.1	269	168 120
cycle length		23	68	201	29	77 342

Table 7: Times for minimum separation with 4 holes.

- [13] S. Kloder and S. Hutchinson. Barrier coverage for variable bounded-range line-of-sight guards. *Proc. ICRA*, pp. 391-396. IEEE, 2007, <http://dx.doi.org/10.1109/ROBOT.2007.363818>.
- [14] F. Kuhn, R. Wattenhofer, and A. Zollinger. Ad-hoc networks beyond unit disk graphs. *Proceedings of the 2003 Joint Workshop on Foundations of Mobile Computing*, pp. 69–78, DIALM-POMC '03, 2003, <http://doi.acm.org/10.1145/941079.941089>.
- [15] S. Kumar, T.-H. Lai, and A. Arora. Barrier coverage with wireless sensors. *Wireless Networks* 13(6):817–834, 2007, <http://doi.org/10.1007/s11276-006-9856-0>.
- [16] Z. Lotker and D. Peleg. Structure and algorithms in the SINR wireless model. *SIGACT News* 41(2):74–84, 2010, <http://doi.acm.org/10.1145/1814370.1814391>.
- [17] G. Neyer. dD range and segment trees. *CGAL User and Reference Manual*, 4.6 edition. CGAL Editorial Board, 2015, <http://doc.cgal.org/4.6/Manual/packages.html#PkgRangeSegmentTreesDSummary>.
- [18] R. Penninger and I. Vigan. Point set isolation using unit disks is NP-complete. *CoRR* abs/1303.2779, 2013, <http://arxiv.org/abs/1303.2779>.
- [19] H. Tangelder and A. Fabri. dD spatial searching. *CGAL User and Reference Manual*, 4.6 edition. CGAL Editorial Board, 2015, <http://doc.cgal.org/4.6/Manual/packages.html#PkgSpatialSearchingDSummary>.
- [20] The CGAL Project. *CGAL User and Reference Manual*. CGAL Editorial Board, 4.6 edition, 2015, <http://doc.cgal.org/4.6/Manual/packages.html>.
- [21] Y. Wang, J. Gao, and J. S. B. Mitchell. Boundary recognition in sensor networks by topological methods. *Proceedings of the 12th Annual International Conference on Mobile Computing and Networking*, pp. 122–133, MobiCom '06, 2006, <http://doi.acm.org/10.1145/1161089.1161104>.
- [22] F. Zhao and L. Guibas. *Wireless Sensor Networks: An Information Processing Approach*. Elsevier/Morgan-Kaufmann, 2004.