

Two Optimization Problems for Unit Disks

Sergio Cabello*¹ and Lazar Milinković²

¹ FMF, University of Ljubljana, and Institute of Mathematics, Physics and Mechanics, Slovenia

² FMF and FRI, University of Ljubljana, Slovenia

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.

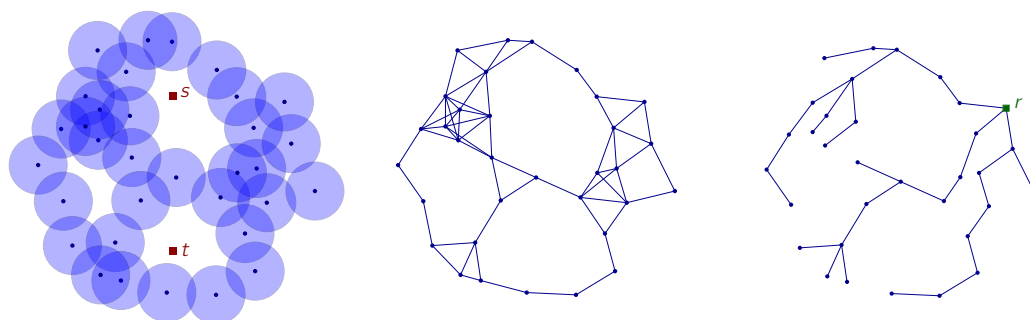


Figure 1 Left: unit disks and two additional points s and t . Middle: intersection graph of the disks. Right: a shortest-path tree in the graph.

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

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

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 [7, 9, 21]. 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 [12, 15], but obtaining efficient algorithms for them is much more difficult.

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 [20]. The minimum-separation problem and variants thereof have been considered in [3, 8, 17]. The problem is dual to the barrier-resilience problem considered in [1, 11, 13, 14]. 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 two algorithms to compute shortest-path trees in unit disk graphs in $O(n \log n)$ worst-case time: one by Cabello and Jeřič [4] and one Efrat, Itai and Katz [6]. Here we report on an implementation of a modification of the algorithm in [4], 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 [6] would be substantially harder to implement and it has worse constants hidden in the O -notation.

As mentioned before, it is not obvious that the minimum-separation problem can be solved in polynomial time. A 2-approximation algorithm is given by Gibson, Kanade, and Varadarajan [8]. Cabello and Giannopoulos [3] 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 have to be employed to reduce the worst-case running time. 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.

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

70 **2 Description of algorithms**

71 **2.1 Shortest-path tree in unit-disk graphs**

72 We describe here the algorithm of Cabello and Jejčič [4] to compute a shortest path tree in
 73 G from a given root point $r \in P$. As it is usually done for shortest path algorithms, we use
 74 tables $\text{dist}[\cdot]$ and $\pi[\cdot]$ indexed by the points of P to record, for each point $p \in P$, the distance
 75 $d_G(s, p)$ and the ancestor of p in a shortest (s, p) -path.

76 The pseudocode of the algorithm, which we call UNWEIGHTEDSHORTESTPATH, is in
 77 Figure 2. We explain the intuition, taken almost verbatim from [4]. We start by computing
 78 the Delaunay triangulation $DT(P)$ of P . We then proceed in rounds for increasing values of
 79 i , where at round i we find the set W_i of points at distance exactly i in G from the source r .
 80 We start with $W_0 = \{r\}$. At round i , we use $DT(P)$ to grow a neighbourhood around the
 81 points of W_{i-1} that contains W_i . More precisely, we consider the points adjacent to W_{i-1}
 82 in $DT(P)$ as candidate points for W_i . For each candidate point that is found to lie in W_i ,
 83 we also take its adjacent vertices in $DT(P)$ as new candidates to be included in W_i . For
 84 checking whether a candidate point p lies in W_i we use a data structure to find a nearest
 85 neighbour of p in W_{i-1} . If the distance from p to its nearest neighbour w in W_{i-1} is smaller
 86 than 1, then the shortest path tree is extended by connecting p to w .

87 Cabello and Jejčič [4] show that the algorithm correctly computes the shortest-path tree
 88 from r . If for nearest neighbors we use a data structure that, for n points, has construction
 89 time $T_c(n)$ and query time $T_q(n)$, and the Delaunay triangulation is computed in $T_{DT}(n)$ time,
 90 then the algorithm takes $O(T_{DT}(n) + T_c(n) + nT_q(n))$ time. Standard tools in Computational
 91 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
 92 leads to the following.

93 ► **Theorem 1** (Cabello and Jejčič [4]). *Let P be a set of n points in the plane and let r
 94 be a point from P . In time $O(n \log n)$ we can compute a shortest path tree from r in the
 95 unweighted graph $G_{\leq 1}(P)$.*

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

98 **2.2 Minimum separation with unit-disk**

99 Cabello and Giannopoulos [3] present an algorithm for the minimum separation problem
 100 that in the worst-case runs in cubic-time. The algorithm has one feature that is both an
 101 advantage and a disadvantage: it works for any reasonable shapes, like segments or ellipses,
 102 and not just unit disks. This means that it is very generic, which is good, but it cannot
 103 exploit any properties of unit disks.

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

```

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 [4] to compute a shortest path tree in the unweighted case.

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

114 2.2.1 Generic algorithm specialized for unit disks

115 Let us first introduce some notation. Recall that s and t are the two points to separate. Each
 116 walk W in the graph $G = G_{\leq 1}(P)$ defines a planar polygonal curve in the obvious way: we
 117 connect the points of P with segments in the order given by W . We will relax the notation
 118 slightly and denote also by W the curve itself. For any spanning tree T of G and any edge
 119 $e \in E(G) \setminus E(T)$, let $cycle(T, e)$ be the unique cycle in $T + e$. Finally, for any walk in $G(P)$,
 120 let $\text{cr}_2(st, W)$ be the modulo 2 value of the number of crossings between the segment st and
 121 (the curve defined by) W . The following property is implicit in [3] and explicit in [5]:

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

A consequence of this is that finding a minimum separation amounts to finding a shortest cycle in G that crosses the segment st an odd number of times. Moreover, one can show that we can restrict our search to a very concrete family cycles, as follows. Consider any optimal cycle W^* and let r^* be any vertex in W^* . Fix a shortest-path tree T_{r^*} from r^* in G . When there are many, the choice of T_{r^*} is irrelevant. Then, the set of cycles $\{cycle(T_{r^*}, e) \mid e \in E(G) \setminus E(T_{r^*})\}$ contains an optimal solution. This follows from the co-called 3-path condition; see [3] for the ideas, and Appendix A.1 for a self-contained proof. Since we do not know r^* , we just try all possible roots. (This leads to the option of having a randomized algorithm, by selecting some roots at random, for the case where the optimal solution is large.) In summary, the size of the optimal solution is given by

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

124 The values $\text{cr}_2(st, cycle(T_r, e))$ can be computed in constant amortized time per edge
 125 with some easy bookkeeping. Consider a fixed tree T_r . For each point $p \in P$ we store $N[p]$
 126 as the parity of the number of crossings of the path in T_r from r to p . When p is not the
 127 root, the value $N[p]$ can be computed from the value of its parent $\pi[p]$ in T_r using that
 128 $N[p] = N[\pi[p]] + \text{cr}_2(st, p\pi[p])$. In the algorithm we have written it this way (lines 4–6), but
 129 one can also compute the values at the time of computing the shortest path tree T_r .

We then have for each shortest-path tree T_r

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

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

134 Let us look into the time complexity of the algorithm. For each point $r \in P$ we have to
 135 compute a shortest-path tree in G . This can be done in $O(n \log n)$ in our case, as discussed
 136 in Section 2.1. Then, for each edge pq of G some constant amount of work is done. Thus for

each point r we spend $O(n \log n + |E(G)|)$. This is cubic in the worst-case. We could get an improved running time if we can treat all the edges of G compactly. This is what we explain next.

2.2.2 Compact treatment of edges

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

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

► **Lemma 2.** *Let B be a set of n points with positive x -coordinates. We can preprocess B in $O(n \log^3 n)$ time such that, for any query point a with negative x -coordinate, we 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 empty. The same data structure can handle queries to know whether the set $\{b \in B \mid ab \text{ does not intersect } \sigma \text{ and } |ab| \leq 1\}$ is empty.*

Proof. We are going to use point-line duality and range trees. These are standard concepts in Computational Geometry; see for example [2, Chapters 5 and 8]. We assume that the reader is familiar with the topic. For readers that are not very familiar with point-line duality, Figure 7 in the appendix may be helpful.

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

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

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

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

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

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

In the dual space, L_p^* is a segment with endpoints $(\varphi_1(p), 0)$ and $(\varphi_2(p), -\tau)$, for some 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:

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

163 Given a point a to the left of the y axis, the set of points $b \in B$ with the property that ab
 164 intersects σ corresponds to the points b with $\varphi(b)$ in the bottom-right quadrant with apex
 165 $\varphi(a)$.

166 We can use a 2-dimensional range tree to store the point set $\varphi(B)$, where each point
 167 $b \in B$ is identified with its image $\varphi(b)$. Moreover, for each node v in the secondary level of
 168 the range tree, we store a data structure for nearest neighbours for the canonical set $P(v)$ of
 169 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

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

170 This means that we get the set $\{b \in B \mid ab \text{ intersects } \sigma\}$ as the union of canonical subsets
 171 $P(v_1), \dots, P(v_k)$ for $k = O(\log^2 n)$ nodes in the secondary levels of the 2-dimensional range
 172 tree. For each such canonical subset $P(v_i)$, we query for the nearest neighbour of a . If
 173 for some v_i we get a nearest neighbour at distance at most 1 from a , then we know that
 174 $\{b \in B \mid ab \text{ intersects } \sigma \text{ and } |ab| \leq 1\}$ 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

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

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

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

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

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

183 and make point location there. Since the regions cannot have many crossings, good asymptotic
 184 bounds can be obtained. However, such approach seems to be only of theoretical interest
 185 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

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

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

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 2 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 2 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 3.** *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 $W_i, L_i^0, L_i^1, R_i^0, R_i^1$ for all i in $O(n \log n)$ time. We then have at most n iterations where, in 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. ◀


```

// Work for root  $r \in P$ 
1  ( $dist[\ ], \pi[\ ]$ ) = shortest path tree from  $r$  in  $G_{\leq 1}(P)$ 
2  Compute the levels  $W_0, W_1, \dots$ 
3  for  $i = 0 \dots n$ 
4      Compute  $N[p]$  for each  $p$  in  $W_i$ 
5      Compute  $L_i^0, L_i^1, R_i^0, R_i^1$ 
6       $i = 1$ 
7      while  $2i < best$  and  $W_i \neq \emptyset$ 
          // within each side of the  $y$ -axis
8          search candidates in
               $L_i^0 \times L_{i-1}^1, L_i^1 \times L_{i-1}^0, R_i^0 \times R_{i-1}^1, R_i^1 \times R_{i-1}^0, L_i^0 \times L_i^1$  and  $R_i^0 \times R_i^1$ 
          // across  $y$ -axis crossing  $\sigma$ 
9          search candidates crossing  $\sigma$  in
               $L_i^0 \times R_{i-1}^0, L_i^1 \times R_{i-1}^1, L_{i-1}^0 \times R_i^0, L_{i-1}^1 \times R_i^1, L_i^0 \times R_i^0$  and  $L_i^1 \times R_i^1$ 
          // across  $y$ -axis not crossing  $\sigma$ 
10         search candidates not crossing  $\sigma$  in
               $L_i^0 \times R_{i-1}^1, L_i^1 \times R_{i-1}^0, L_{i-1}^1 \times R_i^0, L_{i-1}^0 \times R_i^1, L_i^0 \times R_i^1$  and  $L_i^1 \times R_i^0$ 
11          $i = i + 1$ 

```

■ **Figure 4** Work for each vertex in the new algorithm for minimum separation with unit disks.

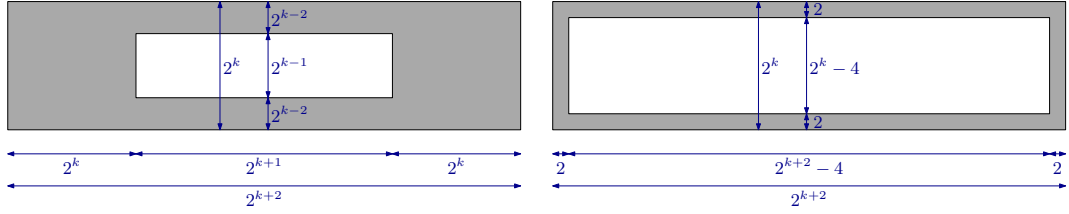
229 The work for each root is described schematically in Figure 4. A slightly more precise
230 version of the algorithm is given as SEPARATIONUNITDISKS COMPACT in Appendix A.1. As
231 before, the variable *best* stores the length of the shortest cycle (or actually rooted closed
232 walk) that we have found so far. We can start setting $best = n + 1$ at start. If eventually
233 we finish with the value $best = n + 1$, it means that there is no feasible solution for the
234 separation problem. When we consider a root r we are interested in closed walks rooted
235 at r and length at most $best$. Since any closed walk through a vertex of W_i has length at
236 least $2i$, we only need to consider indices i such that $2i < best$. Moreover (and this is not
237 described in the algorithm, but it is done in the implementation), we can consider first the
238 pairs that give walks for length $2i$ first, like for example $L_i^0 \times L_{i-1}^1$ and then the ones that
239 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
240 in the while-loop, we can finish the work for the root r , and move onto the next root.

241 3 Implementation and experiments

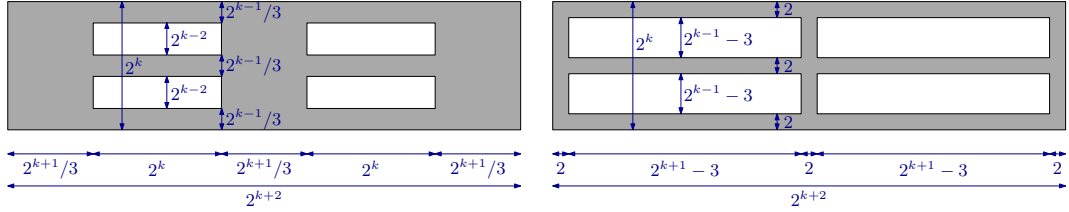
242 We have implemented the algorithms of Section 2 in C++ using CGAL version 4.6.3 [19]
243 because it provides the more complex procedures we need: Delaunay triangulations and
244 Voronoi diagrams [10], range trees [16], and nearest neighbours [18]. Although in some cases
245 we had to make small modifications, it was very helpful to have the CGAL code available as
246 a starting point. The coordinates of the points were Cartesian doubles.

247 Experiments were carried out in a laptop with CPU i5-5200U at 2.20 Ghz, 8GB of RAM,
248 and Windows 10. *All times we report are in seconds.*

249 **Data generation.** Data points were generated uniformly at random in the following
250 polygonal domains: rectangles without holes, rectangles with a “small” rectangular hole,



■ **Figure 5** Data generation with a small hole (left) and a large hole (right).



■ **Figure 6** Data generation with four small holes (left) and four large holes (right).

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 5 and 6. We generated 1K, 2K, 5K, 10K, 20K and 50K points for the cases where the outer rectangle has sizes 4×1 , $8 \times 2, \dots, 128 \times 32$. The data was generated once and stored. For the minimum-separation problem s was placed in the middle of a hole and t vertically above s in the outer face. Some of these domains are not meaningful for the minimum-separation problem because the disks centered at the points cover s .

Shortest-path tree in unit-disk graphs. We have implemented the algorithm described in Section 2.1. For the shortest-path tree we used the Delaunay triangulation as provided by CGAL. The data structure for nearest neighbour queries is a small extension of the one provided by [10]. In the tables we refer to this algorithm as **SSSP**.

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

The second alternative we consider is to use a unit-length grid. Two points (x, y) and (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 points of a grid cell c in a list $\ell(c)$. The non-empty lists $\ell(c)$ are stored in a dictionary, where the bottom-left corner of the cell is used as key. We can then run some sort of BFS using this structure. The list $\ell(c)$ for a cell c maintains the points that have not been visited by the BFS tree yet. When processing a point p in a cell c , we have to treat all the points in the lists of c and its 8 adjacent cells as candidate points. Any point that is adjacent to p is then removed from the list of its cell. The preprocessing is linear, and the time spent to compute a shortest-path tree depends on the distribution of the points. It is easy to produce cases where the algorithm would need quadratic time. For each shortest-path tree we compute the lists and the dictionary anew. (This step is very fast in any case.) In the tables we refer to this algorithm as **grid**.

The measured times are in Tables 1–4. For SSSP and BFS we report the preprocessing

Rectangle without holes		20K points					
size rectangle		4×1	8×2	16×4	32×8	64×16	128×32
SSSP preprocessing		0.03	0.03	0.02	0.03	0.03	0.03
SSSP average/root		0.13	0.13	0.13	0.13	0.13	0.14
BFS preprocessing		25.06	20.43	17.77	17.73	17.35	17.18
BFS average/root		3.41	1.36	0.40	0.09	0.03	0.01
grid		1.61	1.65	0.69	0.23	0.09	0.05
		50K points					
SSSP preprocessing		0.09	0.09	0.07	0.07	0.07	0.07
SSSP average/root		0.59	0.56	0.38	0.38	0.37	0.37
BFS preprocessing		>3min	159.81	144.97	140.40	131.85	132.48
BFS average/root	memory limit		9.38	2.79	0.58	0.14	0.04
grid		11.57	13.66	4.59	1.35	0.43	0.19

■ **Table 1** Times for shortest paths in rectangles without holes.

Rectangle 1 small hole		10K points					
size rectangle	4×1	8×2	16×4	32×8	64×16	128×32	
SSSP preprocessing	0.021	0.012	0.012	0.012	0.012	0.012	
SSSP average/root	0.104	0.059	0.060	0.061	0.064	0.070	
BFS preprocessing	8.500	4.300	4.100	4.100	4.000	4.000	
BFS average/root	1.183	0.318	0.091	0.026	0.008	0.003	
grid	0.486	0.513	0.168	0.072	0.035	0.026	
		20K points					
SSSP preprocessing	0.027	0.026	0.025	0.025	0.025	0.025	
SSSP average/root	0.142	0.137	0.136	0.136	0.138	0.145	
BFS preprocessing	24.813	19.817	18.396	17.976	17.542	17.313	
BFS average/root	3.253	1.328	0.467	0.108	0.031	0.011	
grid	2.181	2.627	0.668	0.262	0.104	0.060	

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

time that is independent of the source (like building the Delaunay triangulation or building the graph) and the average time spent for a shortest-path tree over 50 choices of the root. For grid we just report the total running time; assigning points to the grid cells and putting them into a dictionary is almost negligible. As it can be seen, the results for sssp are very much independent of the shape and, for dense point sets it outperforms the other algorithms.

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

Minimum separation with unit-disk. We have implemented the algorithm GENER-ICMINIMUMSEPARATION and the new algorithm based on a compact treatment of the edges. The shortest-path trees are constructed using the algorithm of Section 2.1. The table $N[\]$ 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.

Rectangle 4 small holes size rectangle	10K points			20K points		
	32 × 8	64 × 16	128 × 32	32 × 8	64 × 16	128 × 32
SSSP preprocessing	0.021	0.018	0.018	0.027	0.026	0.026
SSSP average/root	0.056	0.058	0.064	0.125	0.126	0.131
BFS preprocessing	6.291	6.102	6.364	18.325	17.887	17.256
BFS average/root	0.033	0.010	0.004	0.102	0.031	0.010
grid	0.064	0.031	0.023	0.230	0.096	0.055

■ **Table 3** Times for shortest paths in rectangles with 4 small holes.

Rectangle 4 large holes size rectangle	5K points			10K points		
	32 × 8	64 × 16	128 × 32	32 × 8	64 × 16	128 × 32
SSSP preprocessing	0.007	0.009	0.009	0.012	0.018	0.019
SSSP average/root	0.027	0.028	0.026	0.057	0.054	0.053
BFS preprocessing	1.420	1.420	1.390	5.740	5.660	5.720
BFS average/root	0.007	0.003	0.002	0.028	0.013	0.007
grid	0.019	0.013	0.010	0.060	0.038	0.027

■ **Table 4** Times for shortest paths in rectangles with 4 large holes.

296 In the data structure of Lemma 2, we do use a 2-dimensional tree as the primary structure,
 297 making some modifications of [16]. In the secondary structure, for nearest neighbour, instead
 298 of using Voronoi diagrams, we used a small modification of the *kd*-trees implemented in [18].
 299 In some preliminary experiments this seemed to be a better choice. In our modification, we
 300 make a range search query for points at distance at most 1, and finish the search whenever
 301 we get the first point. In the new algorithm, before calling to the function to candidates pairs,
 302 like for example $L_i^0 \times R_1^i$, we test that both sets are non-empty. This simple test reduced the
 303 time by 30-50% in our test cases.

304 Besides the new algorithm we also implemented the generic algorithm of Section 2.2.1.
 305 The measured times are in Tables 5–6. For the case of 4 holes we always put t above the
 306 rectangle and s in one hole. We also constructed an instance with the rectangle of size 32×8 ,
 307 one small hole, the original 2K points, and 1K extra points quite cluttered around the portion
 308 of st in the domain. The generic algorithm took 427 seconds and the new algorithm took
 309 259 seconds.

Rectangle 1 small hole size rectangle	2K points			
	8 × 2	16 × 4	32 × 8	64 × 16
new separation algorithm	65	64	53	38
generic algorithm	215	87	43	29

■ **Table 5** Times for minimum separation with 1 hole.

Rectangle 4 holes size rectangle	2K points, small holes			5K points, large holes		
	32 × 8	64 × 16	128 × 32	32 × 8	64 × 16	128 × 32
new separation algorithm	29	35	35	413	451	388
generic algorithm	80	40	30	416	266	206

■ **Table 6** Times for minimum separation with 4 holes.

References

- 1 S. Bereg and D. G. Kirkpatrick. Approximating barrier resilience in wireless sensor networks. In *Proc. 5th ALGOSENSORS*, volume 5804 of *LNCIS*, pages 29–40. Springer, 2009.
- 2 M. d. Berg, O. Cheong, M. v. Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 3rd ed. edition, 2008.
- 3 S. Cabello and P. Giannopoulos. The complexity of separating points in the plane. *Algorithmica*, 74(2):643–663, 2016.
- 4 S. Cabello and M. Ježić. Shortest paths in intersection graphs of unit disks. *Comput. Geom.*, 48(4):360–367, 2015.
- 5 S. Cabello and M. Kerber. Semi-dynamic connectivity in the plane. In *Algorithms and Data Structures - 14th International Symposium, WADS 2015. Proceedings*, volume 9214 of *Lecture Notes in Computer Science*, pages 115–126. Springer, 2015.
- 6 A. Efrat, A. Itai, and M. J. Katz. Geometry helps in bottleneck matching and related problems. *Algorithmica*, 31(1):1–28, 2001.
- 7 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.
- 8 M. Gibson, G. Kanade, and K. Varadarajan. On isolating points using disks. In *Proc. 19th ESA*, volume 6942 of *LNCIS*, pages 61–69. Springer, 2011.
- 9 M. L. Huson and A. Sen. Broadcast scheduling algorithms for radio networks. In *IEEE MILCOM '95*, volume 2, pages 647–651 vol.2, 1995.
- 10 M. Karavelas. 2D voronoi diagram adaptor. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.6 edition, 2015.
- 11 S. Kloder and S. Hutchinson. Barrier coverage for variable bounded-range line-of-sight guards. In *Proc. ICRA*, pages 391–396. IEEE, 2007.
- 12 F. Kuhn, R. Wattenhofer, and A. Zollinger. Ad-hoc networks beyond unit disk graphs. In *Proceedings of the 2003 Joint Workshop on Foundations of Mobile Computing, DIALM-POMC '03*, pages 69–78, 2003.
- 13 S. Kumar, T. H. Lai, and A. Arora. Barrier coverage with wireless sensors. In *Proc. 11th MobiCom*, pages 284–298. ACM, 2005.
- 14 S. Kumar, T.-H. Lai, and A. Arora. Barrier coverage with wireless sensors. *Wireless Networks*, 13(6):817–834, 2007.
- 15 Z. Lotker and D. Peleg. Structure and algorithms in the SINR wireless model. *SIGACT News*, 41(2):74–84, 2010.
- 16 G. Neyer. dD range and segment trees. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.6 edition, 2015.
- 17 R. Penninger and I. Vigan. Point set isolation using unit disks is NP-complete. *CoRR*, abs/1303.2779, 2013.
- 18 H. Tangelder and A. Fabri. dD spatial searching. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.6 edition, 2015.
- 19 The CGAL Project. *CGAL User and Reference Manual*. CGAL Editorial Board, 4.6 edition, 2015.
- 20 Y. Wang, J. Gao, and J. S. Mitchell. Boundary recognition in sensor networks by topological methods. In *Proceedings of the 12th Annual International Conference on Mobile Computing and Networking, MobiCom '06*, pages 122–133, 2006.
- 21 F. Zhao and L. Guibas. *Wireless Sensor Networks: An Information Processing Approach*. Elsevier/Morgan-Kaufmann, 2004.

A

 Appendix

A.1 Additional material

► **Lemma 4.** *Let W^* be a shortest cycle in G that crosses the segment st an odd number of times and let r^* be any vertex in W^* . Fix a shortest-path tree T_{r^*} from r^* in G . Then, the 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 st an odd number of times.*

Proof. The proof is a standard consequence of the 3-path condition and very similar to that given in [3]. We spell it out.

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:

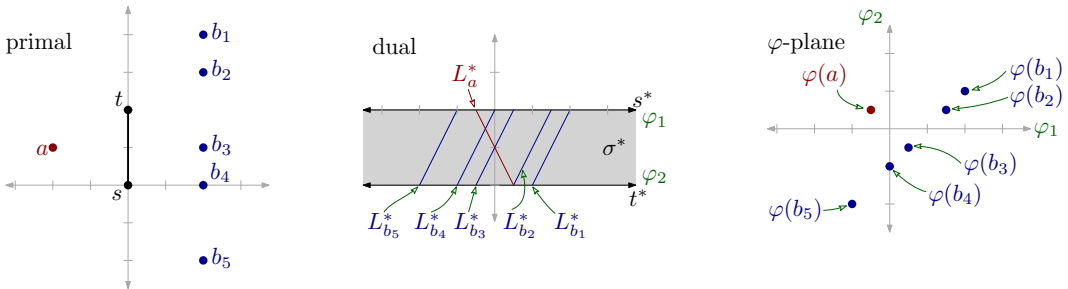
$$\begin{aligned}
 \sum_{pq \in W^*} \text{cr}_2(st, \text{walk}(T_{r^*}, pq)) &= \sum_{pq \in W^*} (\text{cr}_2(st, T_{r^*}[r^* \rightarrow p]) + \text{cr}_2(st, pq) + \text{cr}_2(st, T_{r^*}[q \rightarrow r^*])) \\
 &= \sum_{pq \in W^*} \text{cr}_2(st, pq) \\
 &= \text{cr}_2(st, W^*) \\
 &= 1.
 \end{aligned}$$

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

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

Since r^* is a vertex of W^* and p_0q_0 is an edge of W^* , the length of W^* is at least the 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, this second part is exactly the length of $\text{walk}(T_{r^*}, p_0q_0)$, which is at least the length of $\text{cycle}(T_{r^*}, p_0q_0)$.

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



■ **Figure 7** Transformation in the proof of Lemma 2.

```

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              $N[p] = N[\pi[p]] + \text{cr}_2(st, p\pi[p]) \pmod{2}$ 
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  $best$ 

```

Rectangle 1 large hole size rectangle	5K points			10K points		
	32×8	64×16	128×32	32×8	64×16	128×32
SSSP preprocessing	0.006	0.006	0.006	0.012	0.013	0.014
SSSP average/root	0.027	0.027	0.027	0.057	0.055	0.054
BFS preprocessing	1.010	0.994	0.983	4.043	4.070	4.105
BFS average/root	0.008	0.004	0.002	0.031	0.015	0.008
grid	0.043	0.021	0.017	0.573	0.541	0.172

■ **Table 7** Times for shortest paths in rectangles with a large hole.