Two Optimization Problems for Unit Disks

Sergio Cabello*[1] and Lazar Milinković[2]

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.

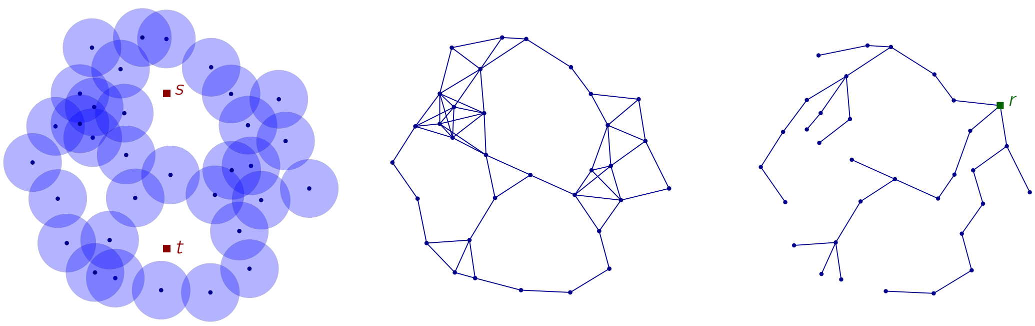

Figure 1 Left: unit disks and two additional points s and t. Middle: intersection graph of the disks. Right: a shortest-path tree in the graph.

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.

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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

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

Unit disks are the most standard model used for wireless sensor networks; see for example [7, 9, 17]. 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 [11, 14], 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 [16]. The minimum-separation problem and variants thereof have been considered in [3, 8, 15]. The problem is dual to the barrier-resilience problem considered in [1, 10, 12, 13]. 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 Jejčič [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 $G_{\leqslant 1}(P)$ 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 $G_{\leqslant 1}(P)$. 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 Jejčič [4] 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 [4]. 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 Jejčič [4] 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. Standards tools in Computational Geometry imply that $T_{DT}(n) = O(n \log n)$, $T_c(n) = O(nqlogn)$ and $T_q(n) = O(\log n)$. This leads to the following.

▶ **Theorem 2.1** (Cabello and Jejčič [4]). *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 s in the unweighted graph $G_{\leqslant 1}(P)$.*

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 Giannopouolos [3] 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 advantage and a disadvantage: it works for any reasonable shapes, like segments or ellipses, and not just unit disks. This means that it is very generic, which is good, but it cannot exploit any properties of unit disks.

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

```
UnweightedShortestPath(P, r)
build the Delaunay triangulation DT(P)
for p ∈ P
    dist[p] = ∞
    π[p] = NIL
dist[r] = 0
W_0 = {r}
i = 1
while W_{i−1} ≠ ∅
    build data structure for nearest neighbour queries in W_{i−1}
    Q = W_{i−1}      // candidate points
    W_i = ∅
    while Q ≠ ∅
        q an arbitrary point of Q
        remove q from Q
        for qp edge in DT(P)
            w = nearest neighbour of p in W_{i−1}
            if dist[p] = ∞ and |pw| ⩽ 1
                dist[p] = i
                π[p] = w
                add p to Q
                add p to W_i
    i = i + 1
return dist[·] and π[·]
```

Figure 2 Algorithm from [4] to compute a shortest path tree in the unweighted case.

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

2.2.1 Generic algorithm specialized for unit disks

Let us first introduce some notation. Recall that s and t are the two points to separate. Each walk W in the graph $G = G_{\leqslant 1}(P)$ defines a planar curve in the obvious way: we connect the points of P with segments in the order given by W. We will relax the notation slightly and denote also by W the curve itself. For any spanning tree T of G and any edge $e \in E(G) \setminus E(T)$, let $\mathit{cycle}(T, e))$ be the unique cycle in $T + e$. Finally, for any walk in $G(P)$, let $\mathtt{cr}_2(st, W)$ be the modulo 2 value of the number of crossings between the segment st and (the curve defined by) W.

The following property is implicit in [3] and explicit in [5]:

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

A consequence of this is that finding a minimum separation amounts to finding a shortest cycle 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. For each root r let us fix a shortest-path tree T_r from r in G. When there are many, the choice of T_r is irrelevant. Then, we can restrict our search to

$$\{\mathit{cycle}(T_r, e) \mid r \in P,\ e \in E(G) \setminus E(T_r)\}.$$

This follows from the co-called 3-path condition; see [3] for the ideas, and appendix **??** for a self-contained proof. APPENDIX

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

We then have for each T_r

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

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

GenericMinimumSeparation(P, s, t)

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

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

Let us look into the time complexity of the algorithm. For each point $r \in P$ we have to compute a shortest-path tree in G. This can be done in $O(n \log n)$ in our case, as discussed 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 $\tau \geqslant 0$. Thus, the segment st is vertical and t is above s. The implementation also makes this assumption. 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 2.2.** *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 $\{b \in B \mid ab \text{ intersects } \sigma \text{ and } |ab| \leqslant 1\}$ is empty. A similar data structure with the same guarantees handles queries to know whether $\{b \in B \mid ab \text{ does not intersect } \sigma \text{ and } |ab| \leqslant 1\}$ is empty.*

Proof. We are going to use point-line duality and range trees. These are standard concepts in Computational Geometry; wee for example [2, Chapters 5 and 8]. We assume that the reader is familiar with the topic.

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

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

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

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

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:

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

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

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

This means that we get the set $\{b \in B \mid ab \text{ intersects } \sigma\}$ as the union of canonical subsets $P(v_1), \ldots, P(v_k)$ for $k = O(\log^2 n)$ nodes in the secondary structure 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 $\{b \in B \mid ab \text{ intersects } \sigma \text{ and } |ab| \leqslant 1\}$ 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-dimensionsal range tree takes $O(\log^2 n)$ time to find the $O(\log^2 n)$ nodes $v_1, \ldots, v_k$ 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 $\{b \in B \mid ab \text{ does not intersect } \sigma \text{ and } |ab| \leqslant 1\}$ is done similarly (and the same data structure works), we just have to query for 2 of the other quadrants. (We do not need to query for the other 3 quadrants because one of them is always empty.) ◀

From the theoretical perspective is would be more efficient to compute the union of the $|B|$ regions

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

and make point location there. Since the regions cannot have many crossings, good assymptotic 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 $\pi[\,]$, $dist[\,]$ and $N[\,]$, as discussed in Section 2.2.1. We group the points by their distance from r:

$$W_i \;=\; \{p \in P \mid dist[p] = i\}, \quad i = 0, 1, \ldots, n$$

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 $W_{dist[p]-1} \cup W_{dist[p]} \cup W_{dist[p]+1}$. 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

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

We are interested in edges pq of G such that $N[p] + N[q] + \mathtt{cr}_2(st, pq) = 1 \pmod 2$. 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 $p \in L_i^j \cup R_i^j$, $q \in L_i^{1-j} \cup R_i^{1-j} \cup L_{i-1}^{1-j} \cup R_{i-1}^{1-j}$, $|pq| \leqslant 1$, and pq does not cross st;
- for some $i \in \mathbb{N}$ and some $j \in \{0,1\}$, we have $p \in L_i^j \cup R_i^j$, $q \in L_i^j \cup R_i^j \cup L_{i-1}^j \cup R_{i-1}^j$, $|pq| \leqslant 1$, 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^j. If for some p we get that $|pq_p| \leqslant 11$, then we have obtained a closed walk through r of length $i+i'+1$ crossing st an odd number of times. 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 $R_{i'}^{1-j}$ as discussed in Lemma 2.2 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 we get a closed walk through r of length $i+i'+1$ that crosses st an odd number of times. 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 2.2 into a data structure. Then, for each point $p \in L_i^j$ we query the data structure (for not crossing st). If we get some nonempty set, then we get a cycle of length (at most) $i+i'+1$ that crosses st an odd number of times. The overall running time, if $m = |L_i^j| + |R_{i'}^{1-j}|$, is $O(m \log^3 m)$.

We conclude that each of the cases can be done in $O(m \log^3 m)$ 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 2.3.** *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 $G = G_{\leqslant 1}(P)$. For each root $r \in P$ we build the shortest-path tree and the sets $W_i, L_i^0, L_i^1, R_i^0, L_i^1$ 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. ◀

The work for each root is described schematically in Figure 4. A slightly more precise version of the algorithm is given as SeparationUnitDisksCompact in Figure 7 of the Appendix. 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 $2i < best$. Moreover (and this is not described in the algorithm), 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 a closed walk of inside the while-loop we can finish the work for the root r, and move onto the next root.

SEPARATIONUNITDISKSCOMPACTROOT$(P, s, t, r, best)$

$(dist[\,], \pi[\,])$ = shortest path tree from r in $G_{\leqslant 1}(P)$
Compute the levels $W_0, W_1, \dots$
for $i = 0 \dots n$
    Compute $N[p]$ for each p in W_i
    Compute L_i^0, L_i^1, R_i^0, R_i^1
$i = 1$
while $2i < best$
    // within each side of the y-axis
    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 crosing σ
    search candidates crossing σ 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 crosing σ
    search candidates not crossing σ 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 $L_i^1 \times R_i^0$
    $i = i + 1$

Figure 4 Work for each vertex in the news algorithm for minimum separation with unit disks.

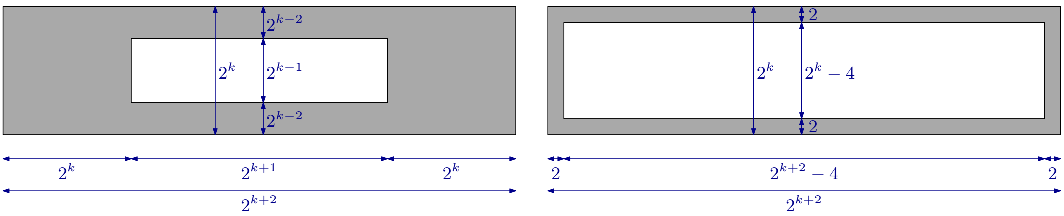

Figure 5 Data generation with a small hole (left) and a large hole (right).

3 Implementation and experiments

We have implemented the algorithms in Section 2 using CGAL ([?] because it provides the more complex procedures we need: Delaunay triangulations, range trees, and nearest neighboours. Although in some cases we had to make small modifications, it was very helpful to have the CGAL code available as a starting point. version

CGAL provides procedures to compute the Delaunay triangulation []. For nearest neighbours (in the plane) we used the library by []. Although in principle one could use also point location on the Voronoi diagram, our earlier tests suggested using We did not test other libraries.

For range trees we extended the structure presented in [].

Exact/non-exact arithmetic

Data was generated uniformly at random in polygons...

All the experiments were carried out in a Computer, environment.

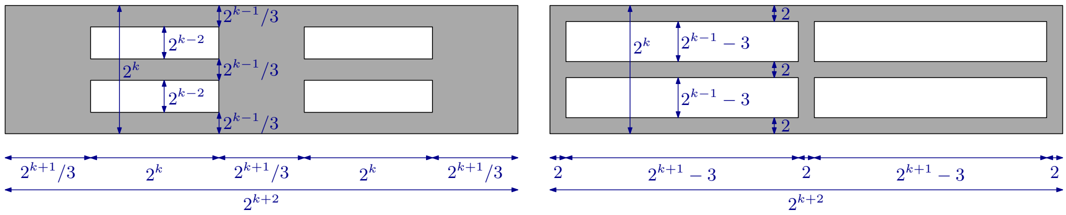

Figure 6 Data generation with four small holes (right) and four large holes (right).

3.1 Shortest-path tree in unit-disk graphs

Alternative constructions. Let us mention two obvious alternatives that we use in our comparison.

The first alternative is to build the graph $G = G_{\leqslant 1}(P)$ explicitly. Thus, for each pair of points p, q we check whether their distance is at most once 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.

The second option 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 data. When processing a point p in a cell c, we have to treat all the points in c and its adjacent cells as candidate points. The preprocessing is linear, and the time spent to compute a shortest-path tree depends on the density of the graph G. The number of candidates that are checked is proportional to the number of edges of the graph.

Are perhaps points deleted from the list once they are assigned a level?

3.2 Minimum separation with unit-disk

4 Conclusions

References

1 S. Bereg and D. G. Kirkpatrick. Approximating barrier resilience in wireless sensor networks. In *Proc. 5th ALGOSENSORS*, volume 5804 of *LNCS*, 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. Jejčič. 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 *LNCS*, 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 S. Kloder and S. Hutchinson. Barrier coverage for variable bounded-range line-of-sight guards. In *Proc. ICRA*, pages 391–396. IEEE, 2007.

11 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.

12 S. Kumar, T. H. Lai, and A. Arora. Barrier coverage with wireless sensors. In *Proc. 11th MobiCom*, pages 284–298. ACM, 2005.

13 S. Kumar, T.-H. Lai, and A. Arora. Barrier coverage with wireless sensors. *Wireless Networks*, 13(6):817–834, 2007.

14 Z. Lotker and D. Peleg. Structure and algorithms in the sinr wireless model. *SIGACT News*, 41(2):74–84, 2010.

15 R. Penninger and I. Vigan. Point set isolation using unit disks is NP-complete. *CoRR*, abs/1303.2779, 2013.

16 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.

17 F. Zhao and L. Guibas. *Wireless Sensor Networks: An Information Processing Approach.* Elsevier/Morgan-Kaufmann, 2004.

A Missing proofs

3-path condition

B Full new algorithm for separation with unit disks

```
SeparationUnitDisksCompact(P, s, t)
best = n + 1 // length of the best separation so far
for r ∈ P
    (dist[ ], π[ ]) = shortest path tree from r in G_{≤1}(P)
    // Compute the levels W_i
    for i = 0 ... n
        W_i = new empty list
    for p ∈ P
        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
    N[r] = 0
    for i = 1 ... n
        for p ∈ W_i
            N[p] = N[π[p]] + cr_2(st, pπ[p])  (mod 2)
            if p to the left of the y-axis
                add p to L_i^{N[p]}
            if p to the right of the y-axis
                add p to R_i^{N[p]}
    i = 1
    while 2i < best
        // length 2i; within each side of the y-axis
        search candidates in L_i^0 × L_{i-1}^1
        search candidates in L_i^1 × L_{i-1}^0
        search candidates in R_i^0 × R_{i-1}^1
        search candidates in R_i^1 × R_{i-1}^0
        // length 2i; across y-axis crosing σ
        search candidates in L_i^0 × R_{i-1}^0 crossing σ
        search candidates in L_i^1 × R_{i-1}^1 crossing σ
        search candidates in L_{i-1}^0 × R_i^0 crossing σ
        search candidates in L_{i-1}^1 × R_i^1 crossing σ
        // length 2i; across y-axis not crosing σ
        search candidates in L_i^0 × R_{i-1}^1 not crossing σ
        search candidates in L_i^1 × R_{i-1}^0 not crossing σ
        search candidates in L_{i-1}^0 × R_i^1 not crossing σ
        search candidates in L_{i-1}^1 × R_i^0 not crossing σ
        // length 2i + 1; within each side of the y-axis
        search candidates in L_i^0 × L_i^1
        search candidates in R_i^0 × R_i^1
        // length 2i + 1; across y-axis crosing σ
        search candidates in L_i^0 × R_i^0 crossing σ
        search candidates in L_i^1 × R_i^1 crossing σ
        // length 2i + 1; across y-axis not crosing σ
        search candidates in L_i^0 × R_i^1 not crossing σ
        search candidates in L_i^1 × R_i^0 not crossing σ
        i = i + 1
return best
```

Figure 7 New algorithm for minimum separation with unit disks.