Implementation of the Reverse Shortest Path Problem in Unit Graph Disks

Rafał Kajca

2024

Abstract

Given a set P of n points in the plane, the unit disk graph $G_r(P)$ for a given r is an undirected graph, where the vertices are points of P, and there exists an edge connecting two points $p, q \in P$, if the Euclidean distance between p and q is at most r. The reverse shortest path problem is, for a given $\lambda \in \mathbb{N}$ and two points $s, t \in P$, to compute the smallest value of r such that the length of the shortest path between s and t in $G_r(P)$ is at most λ. In this paper we present two algorithms, one that runs in $O(n \log(n) + n \log(r))$, and the second one in $O(\lfloor \lambda \rfloor n \log(n))$. We showcase how the algorithms work, their implementation and compare their performance.

1 Introduction

Given a set P of n points in the plane, the unit disk graph $G_r(P)$ for a given r is an undirected graph, where the vertices are points of P, and there exists an edge connecting two points $p, q \in P$, if the Euclidean distance between p and r is at most r. The unit disk graph can also be viewed as an intersection graph of disks centered at each point of P with a radius of $r/2$. So if the distance between two points is at most r, that means that their respective disks intersect.

In this paper we consider the following *reverse shortest path* problem, where for a given $\lambda \in \mathbb{N}$ and two points $s, t \in P$, we try to compute the smallest r such that the length of the shortest path between s and t in $G_r(P)$ is at most λ. In the rest of the paper we use r^* to denote the optimal value of r. Therefore the problem is to find the value of r^*. In this paper we showcase three algorithms in total:

- A Brute-Force Algorithm that runs in $O(n^2 \log(n))$ time,
- An algorithm based on a binary search using a decision algorithm created by Chan and Skrepetos [1]. This procedure is of complexity $O((\log(r^*) + \log(n) + \theta)n)$, where θ denotes bin-search iterations.
- An $O(\lambda n \log(n))$ algorithm, that was first introduced in [2].

Out of those three only the latter two are of any interest, the Brute-Force one was implemented only as a baseline. We will refer to the bin-search algorithm as the first algorithm, and to $O(\lambda n \log(n))$ as the second one. All three has been implemented in this paper, and in section 6 we will go through the implementation details. The C++ code is available in a public repository [1] . First we discuss the motivation of this paper, as well as some preliminaries that need to be understood, to analyze those algorithms.

1.1 Motivation

Assume that we have n transmitters, that can communicate with one another. We now want to transmit messages between two chosen transmitters, a starting point, and the end point. To do that the output power of those transmitters needs to be configured, so that a message can be send through those transmitters. All transmitters have the same output power, that allows them to output signals in a disk of a radius r. A transmitter can send a signal to another transmitter only if it has that other transmitter inside its output disk, and a signal can be send between two transmitters only a set number of times. The question is, what should be the output power of the transmitters to send a message from the starting transmitter, to the end transmitter?

As we can see the reverse shortest path problem (RSP) can be found in real-life problems. That is why RSP and similar problems have been extensively studied. There have been many algorithms to solve RSP, but a considerable number of them are far too complex, and impractical to implement. In this paper we focus on two algorithms, those that we believe are easier in practice, but still are of usable time complexity.

2 Preliminaries

Lets first take a look at the decision algorithm showed by Chan and Skrepetos. Given a value r, the *decision problem* is to decide whether $r \geq r^*$. We can see that $r \geq r^*$ if and only if the distance between s and t in $G_r(P)$ is at most λ.

Most papers [1, 2] work on the theoretical *real* RAM model. In that model, exact computations on real numbers are possible, without limitation. Unfortunately our computers do not have that power, but with real-world computing we will utilize approximations, as we will do in the first algorithm in section 4.

2.1 CS algorithm

Both algorithms make use of the $O(n)$ decision algorithm created by Chan and Skrepetos [1]. Throughout the paper we will refer to that method as the CS algorithm. It solves a decision problem, so for a given set of points P, given as a list sorted by the $x-$coordinate, and another list sorted by the $y-$coordinate, λ, r, a starting point s and an end point t, it returns whether $r \geq r^*$ or $r \leq r^*$.
It works in the following way:

1. Construct a grid of size $\frac{r}{\sqrt{2}}$ on a plane,

[1]The C++ implementation is available in a public GitHub repository: `https://github.com/rafak1/Unit-Disk-RSP-Implementation`

2. Starting from the point s run a BFS algorithm on that grid

3. If BFS visits t then $r \geq r^*$

4. If BFS does $\lambda + 1$ steps without visiting t then $r < r^*$

We next preview the CS algorithm. For more details visit the original papers [1, 2].

2.1.1 Building the Grid

In the rest of the paper, we will let $||p - q||$ denote the euclidean distance between the point p and the point q.

The first step of the algorithm is to compute a grid $\psi_r(P)$, of square cells whose sides are of length $\frac{r}{\sqrt{2}}$. The side length is not arbitrary, as it ensures that for two points contained in one cell, p and q, $||p - q|| \leq r$. For a given cell C, C' is a *neighbour* of that cell if and only if the distance between any point of C, and a point of C' is at most r. It is easy to see that a cell has $O(1)$ *neighbours*, as shown in Figure 1.

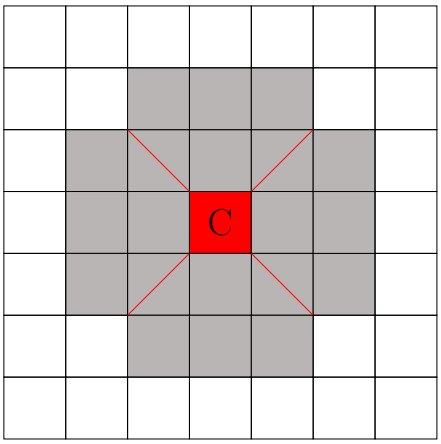

Figure 1: The grey cells represent the neighbour cells of C, and red lines of length r are originating from each corner of C

Let P' be the subset of P with the following property: $\forall p \in P'$ there exists a path from s to s in $\psi_r(P)$. The computation of the grid is in detail explained in [2], and it is centered on the following lemma:

Lemma 2.1. *Given two lists of points P, one sorted by the $x-$coordinate, and the other one by the $y-$coordinate. Than the following can be achieved:*

1. *P' , $\mathscr{C}$ - the set of cells, and the partition lines of $\psi_r(P)$, can be obtained in $O(n)$ time.*

2. *With $O(n)$ preprocessing, for a point $p \in P'$, the cell that contains p can be obtained in $O(1)$ time.*

3. *With $O(n)$ preprocessing, for a cell $C \in \mathscr{C}$, the neighbour set of $N(C)$ can be obtained in $O(|N(C)|)$ time.*

4. *With $O(n)$ preprocessing, for a cell $C \in \mathscr{C}$, the points contained in that cell $P(C)$, can be obtained in $O(|P(C)|)$ time.*

We will expand on the implementation details of building the grid in chapter 5.

2.1.2 Running BFS

The next step of the CS algorithm is to execute a breadth-first search with the help of a grid $\psi_r(P)$. Let us define a set S_i of points from P whose distance from the starting point s is exactly i. S_0 consists only of the point s. Now the task is to calculate points of S_i using S_{i-1}. As in a classic BFS we denote q as *visited* when $q \in \bigcup_{i=0}^{i-1} S_i$. Lets take a look at some point in S_{i-1}. Point p lies in a cell C. As stated before we know that for every $q \in C \implies ||p-q|| < r$. So we may add all not *visited* points in C to S_i. Now to find all points outside of C that will go to S_i: We know that for a cell C all possible candidates to be in S_i need to be in some neighbouring cell C'. Now we have the following lemma:

Lemma 2.2 (sub-problem 1)**.** *Given a horizontal line L, a x-presorted set of $|n_r|$ red points that are below L and a x-presorted set of $|n_b|$ blue points above L, we are able to determine for each blue point whether there is a red point at distance at most r from it in $O(|n_r| + |n_b|)$ time.*

Proof. In our case, n_r - *red* points refer to points of C, n_b - *blue* points however are points of a neighbour C', and L refers to the boundary of C that is in the direction of C', as shown in Figure 2.

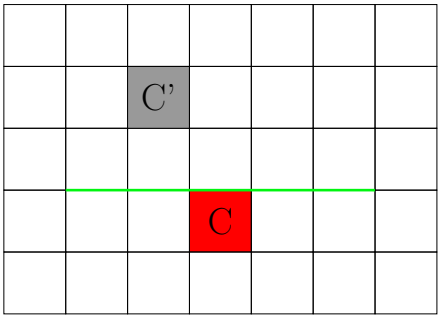

Figure 2: Relative positions of C, and C', and their corresponding green line L

We now move on to quickly prove that lemma.

Circle of radii r center at a *red* point has at most one arc above L. Let Γ denote the set of those arcs of all of n_r. The upper envelope of Γ divides the plane into two parts. If a *blue* point p is below that envelope, there exists a *red* point q, so that $||p-q|| < r$. If a *blue* point p is above, then there are no such *red* points. To calculate the upper envelope of Γ we make the following observations:

1. Every two arcs of Γ intersect at most once,
2. Arcs above L are x-monotone,
3. As *red* points are already sorted, we can calculate Γ in $O(n_r)$ using an algorithm similar in spirit to a Graham's scan [6]

To calculate the upper envelope we use the notion of pseudo-lines. We can extend each arc, to a curve from $x = -\infty$ to $x = \infty$ such that it intersects only once with the pseudo-line of another unit disk. Moreover we want to ensure that the order of the pseudo-lines $x = \infty$ coincide with the order of x-coordinates of the corresponding disk centers, as shown in Figure 3.

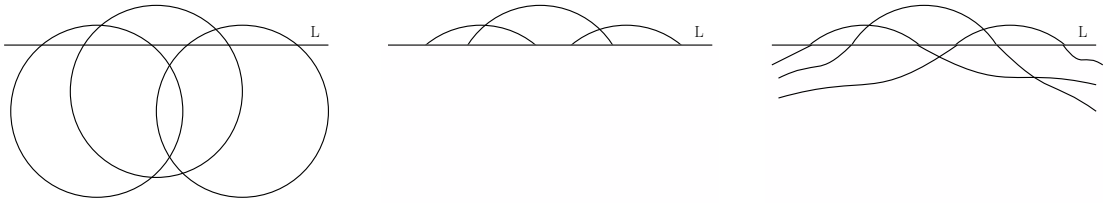

Figure 3: The first image depicts three unit disks, and a line L; the second one arcs created by those disks, and the third - pseudo-lines for those arcs.

Then we apply the algorithm described in [7], where a pseudo-code is given for computing the upper envelope of such pseudo-lines. We now can find in $O(n_r)$ all intersection points of the upper envelope (i.e. points above L, where two pseudo-lines intersect), lest call them U. Then we sort by the x coordinate points of U with the *blue* points, as this will allow us to determine for a point in n_b the envelope arc that may be the one to cover it. This also takes $O(n_r)$ time. □

We can generalize this lemma to also work for the y coordinate, as this will be needed to cover neighbours above cell C. We will further describe this procedure along its implementation in chapter subsubsection 6.3.1.

After solving sub-problem 1 for all cells, and all their neighbours, we will have calculated the set S_i. When we reach the end point t, do λ steps, or find that $S_i = \emptyset$ we can stop the algorithm, and return true or false accordingly. The above BFS algorithm takes $O(n)$ time as each cell from $\psi_r(P)$ is present in at most two steps of the BFS, and each cell has a constant amount of neighbours.

2.2 Parametric Search

The technique of Parametric Search was first introduced by Megiddo in [4]. The idea behind that technique is further explained, and expanded in [3, 5]. It has been widely used in efficient algorithms for geometric problems.

Assume that we have a decision problem B that is true for some interval $[r^*, \infty)$. The problem we want to solve is to determine the value of r^*, minimum for which B is true. To do that we need two algorithms: a sequential decision algorithm C for problem B, that for a given r returns whether $r \geq r^*$ or $r < r^*$, and a parallel algorithm A - which is driven by *'comparisons'*, that may depend on the parameter r. The idea then is to sequentially simulate the execution of

A on the unknown parameter r^*. As we do not know r^* we resolve the *'comparisons'* using the decision algorithm C. As a single *'comparison'* is dependent on r we may check weather $r \geq r^*$ or $r < r^*$ using C, therefore we will know the result of this *'comparison'* as it would have been executed with the parameter r^*. After executing A we are able to extract the value r^* as a sort of by-product. We will expand on the idea of Parametric Search further in subsubsection 5.3.2.

2.3 Sorted Matrix Search

Assume that we have a sorted matrix M of size $n \times m$ where $n \geq m$, so both the columns and rows of M are sorted (i.e. $M[i,j] \geq M[i,j+1]$ and $M[i,j] \geq M[i-1,j]$). It is easy to see that the largest value of such a matrix would be $M[n-1,0]$, and the smallest $M[0,m-1]$.

The goal is to find the smallest value found in such a matrix for which some decision algorithm returns true. The decision algorithm runs in time $O(n)$ (as our CS algorithm). A sorted matrix search technique for that special case was shown by Frederickson and Johnson in [9], later generalized in [8]. They managed to accomplish $O(n\log(n))$ time complexity.

First, we artificially extend the matrix, so that it is a square matrix, of size $2^s \times 2^s$ where $2^s \geq n$. We can keep the matrix sorted by assigning new cells the value of $-\infty$. The main idea is to now split the matrix, run the decision algorithm, discard unimportant sub-matrices, and repeat, until only one cell is left. We will call a sub-matrix that is of *interest* (i.e. it may contain the value we are looking for) an *active* sub-matrix. Initially only M is *active*. One iteration of the algorithm goes as follows:

1. Select the upper median x_s (ranking $\lfloor (h+1)/2 \rfloor$ from h elements) of the smallest elements of every *active* sub-matrix.
2. Select the lower median x_l (ranking $\lceil (h+1)/2 \rceil$ from h elements) of the largest elements of every *active* sub-matrix.
3. Test both x_s and x_l for feasibility with the decision algorithm.
4. If x_s is feasible discard all *active* sub-matrices whose smallest elements are greater than or equal to x_s, but keep one sub-matrix which has the smallest element equal to x_s.
5. If x_s is not feasible discard all *active* sub-matrices whose largest elements are less than or equal to x_s.
6. Do similar operations in regard to x_l.
7. Split every remaining *active* matrices into four sub-matrices of equal size, if the size of the *active* matrix is different than 1×1.
8. If there is only one *active* matrix of size 1×1, return that value.

By *discarding*, we mean that a sub-matrix is no longer *active*, and is not included in next steps. It is proven in [9] that such a procedure yields a $O(n\log(n))$ time complexity.

This search algorithm can also be slightly modified to look for the biggest value for which the decision algorithm returns false.

3 The Brut-Force Algorithm

It is trivial to see that r^* must be a distance between two points of P. We can take advantage of that to calculate the exact value of r^* in $O(n^2 \log(n))$ time. We can calculate all the different distances between every two different points, sort them, and do a binary search to find the smallest value r that has the desired path. There are $O(n^2)$ such distances and we can sort them in $O(n^2 \log(n))$ time. The decision algorithm that we used in the binary search, works by constructing the intersection graph $G(E, V)$ for a given r, and then running a standard BFS algorithm, to check whether the length of the shortest path between s and t is at most λ. A single decision algorithm takes $O(n^2)$ time, as graph G has at most $O(n^2)$ edges. We implemented that algorithm to use as a baseline, and compare results from other algorithms, as the Brute-Force Algorithm is really easy to implement.

4 First Algorithm

The first algorithm that we implemented, uses the fact that for all $r \geq r^*$ the CS algorithm will return *true*, but for all $r < r^*$ the CS algorithm returns *false*. Because of that we will do a binary search on real numbers, to find the smallest r, on which the CS algorithm returns *true*. Using binary search comes with disadvantages, as now, we will only calculate an approximation of r*. To run the CS algorithm in $O(n)$ time we need to sort the points P beforehand. We can use a standard $O(n \log(n))$ sorting algorithm to do that. First the algorithm, calculates an interval $r^* \in [2^i, 2^{i+1}]$, which can be done in $O(\log(r^*))$. We denote θ as *precision*, which will mean the number of iterations of binary search that looks for the optimal value in that calculated interval $[2^i, 2^{i+1}]$. Putting it all together we get an algorithm that approximates r^* in $O((\log(r^*) + \log(n) + \theta)n)$ time. We expand more on the implementation of that algorithm, especially the CS algorithm, in subsection 6.2.

5 Second Algorithm

The biggest downside of the First Algorithm is that we don't get the exact value of r^*, but an approximation that affects time complexity.

The second algorithm that was implemented for this paper is the algorithm presented by Wang and Zhao [2]. It's more complex, and more problematic to implement then the First Algorithm, but it may be of interest as an alternative. It was originally created on the theoretical *real RAM model*, to give a precise value of r^*, that is why the approximating First Algorithm was not taken into consideration, despite having a seemingly better complexity. The Second Algorithm runs in $O(\lambda n \log(n))$ time. This algorithm is only of interest when λ is relatively small. In the case that $\lambda = \theta(n)$ the running time becomes $O(n^2 \log(n))$ which is no better then the Brute-Force Algorithm Given λ and $s, t \in P$ our goal is to again calculate the optimal r^* radius of the disks, for which the length of the shortest path between s and t in $G_{r^*}(P)$ is at most *lambda*.

5.1 Basic Idea

The Second Algorithm, presented in [2], uses parametric search, but rather an unorthodox one. Instead of using a parallel algorithm, the algorithm here is purely sequential. This helps a lot with implementation. The basic idea of the Second Algorithm is to simulate the execution of the CS algorithm [1] on the unknown value of r^*. We do that by keeping an interval $(r_1, r_2]$, and an invariant that $r^* \in (r_1, r_2]$. At each step of our algorithm we shrink that interval, using the actual CS decision algorithm, so that our invariant holds. After the algorithm terminates after λ steps, or after t is reached we will be able to get the optimal value r^*. Summarizing the basic idea of the algorithm is to simulate the breadth-first-search (BFS) algorithm on the grid as it would have been executed on the sought after value r^*.

We initially set our interval to $(0, \infty)$, as it is clear that $r^* \in (0, \infty)$. Before any actual shrinking, input points need to be sorted. We can obtain two lists, first sorted by the x coordinate, and the second by y in $O(n \log(n))$ time. Throughout the rest of this paper, by shrinking the interval, we mean *'trying'* to shrink the interval. So if we *'try'* to shrink interval $(r_1, r_2]$ with $(a_1, a_2]$ then the resulting interval is $(\max(r_1, a_1), \min(r_2, a_2)]$.

5.2 Building The Grid

The first step is to build a grid. The problem now is, that we do not know the value of r^*, so we do not know for which value we should build said grid. That is why, we first want to shrink our initial interval to $(r_1, r_2]$, where for each $r \in (r_1, r_2)$ the *combinatorial structure* is the same, and $r^* \in (r_1, r_2]$. By two grids having the same *combinatorial structure* we mean that both grids have the same amount of vertical, and horizontal lines, and every point p is in a cell with row i, and column j in both grids.

To accomplish that we have the following lemma:

Lemma 5.1. *An interval $(r_1, r_2]$ containing r^*, so that if $r^* \neq r_2$ the combinatorial structure of $\psi_r(P)$ for every $r \in (r_1, r_2)$ stays the same, can be computed in $O(n \log(n))$ time*

The proof in more detail is also present in [2].

Proof. We let P_1 denote the set of points to the right of our starting point s including s. We define $S = p_1, p_2, p_3, ..., p_m$ to be the list of P_1 sorted by the x coordinate, with $m = |P_1|$ and $p_1 = s$. There is a vertical line running through p_1 and there are at most $2m$ vertical lines [2]. We then define a matrix M of size $m \times 2m$, where for all $1 \leq i \leq m$ and $1 \leq j \leq 2m$

$$M[i, j] = \sqrt{2} \cdot \frac{x(p_i) + x(p_1)}{j}$$

It can be verified that such a matrix is a sorted matrix, thus using a sorted matrix search technique of [8, 9] explained in subsection 2.3, and the CS algorithm as our decision algorithm we can find a value r_1 that is the biggest value in M that CS returns false on, and a value r_2 - the smallest value with which CS returns true. We then shrink our initial interval with $(r_1, r_2]$.

A more detailed proof of correctness is present in [2]. But the intuition behind the matrix is that those are the values where the *combinatorial structure* of the grid changes. That is why for our new interval, for every $r \in (r_1, r_2)$, the *combinatorial structure* of $\psi_r(P)$ doesn't change in respect to horizontal lines to the right of s. We ensure that $r^* \in (r_1, r_2]$ because $CS(r_1)$ returns false, and $CS(r_2)$ returns true.

This handles only the horizontal grid lines to the right of s. Now we define $P_2 = (P \setminus P_1) \cup s$. Again define $S = p_1, p_2, p_3, ..., p_m$, and analogously execute the shrinking procedure. The same thing has to be done for vertical lines, so now we use the list of points sorted by the y coordinate, and do comparisons on the ordinate.

After all is done, so eight matrix searches in total, we obtain an interval where for each $r \in (r_1, r_2)$ *combinatorial structure* of $\psi_r(P)$ stays the same. □

We pick any value $r \in (r_1, r_2)$ from Theorem 5.1, and then proceed to build the grid $\psi_r(P)$ for that value. We will later address said grid as $\psi(P)$. By building the grid we mean to compute the information of Theorem 2.1. All this is done in $O(n \log(n))$, as building the grid with presorted point takes $O(n)$ time. We hold onto that grid, as it will be crucial to run a *'parametric BFS'* using said grid.

5.3 BFS

We proceed similarly in nature to the breadth first search in the CS algorithm which we explained in subsubsection 2.1.2. The difference now is that to run it on the unknown value r^* we have to keep shrinking the interval on every step. We denote $S_i(r)$ as points from P whose distance from the starting point s is exactly i, assuming that the radius of disks is r. In the following we maintain an invariant that for every $r \in (r_1, r_2)$ (in the interval also computed in the i-th step) $S_i = S_i(r) = S_i(r^*)$. So with each step we try to shrink the interval so that $S_i(r)$ will remain the same for every r in our interval. We start with the interval $(r_1, r_2]$ computed while constructing the grid. As previously set $S_0 = \{s\}$.

On the i-th step, we have at our hand S_{i-1}, and an interval $(r_1, r_2]$ that hold all of the invariants we talked about above. Using previously calculated grid $\psi(P)$ we gain the cell information for each point in S_{i-1}. For each such cell C we make the observation that all points in C should be in S_i. This is proven in [2]. So as in the CS algorithm we can add to S_i points of $P(C)$ that have not yet been *discovered*. Next for each cell, and for its each neighboring cells, we have to solve the sub-problem discussed in Theorem 2.2. Recall that in the original CS algorithm this requires three steps:

1. Build the upper envelope of *red* points,
2. Sort vertices of the envelope with *blue* points,
3. For each blue point decide whether it is below, or above the envelope.

Where *red* points are point of S_{i-1} in C, and *blue* points are the points of a neighbouring cell C'. In this situation, not knowing the precise value of r^* we want to compute with, we have to parameterize each subroutine.

5.3.1 Computing the Upper Envelope

As in the CS algorithm, lets define $\Gamma(r)$ as a set of arcs above L made from disks of radius r, and $U(r)$ as the upper envelope of $\Gamma(r)$. The goal of the first step is to shrink the interval, so that $r^* \in (r_1, r_2]$ and if $r^* \neq r_2$then for every $r \in (r_1, r_2)$ the *combinatorial structure* of $U(r)$ is the same as $U(r^*)$. By having the same *combinatorial structure* we mean that the set of *red* points that define arcs of $U(r)$ is exactly the same as in $U(r^*)$ with the same order.

To do this, we need to figure out when the structure of $U(r)$ changes. For an arbitrary r $U(r)$ changes when an arc disappears, or rather is 'covered' by other arcs. Imagine a situation: We have three points p_1, p_2 and p_3. We are trying to see at what parameter r will p_2 disappear from $U(r)$. At the moment when p_2 disappears from $U(r)$ all three circles intersect at a common point q. Since q is on $U(r)$, there is no closer point to q than p_1, p_2, p_3. This situation is show in Figure 4.

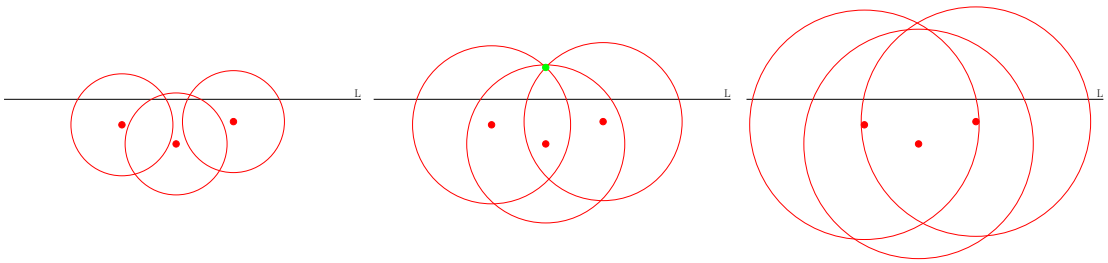

Figure 4: The first image depicts three unit disks, created by p_1, p_2 and p_3; The second shows the situation for the *critical value* of r, and the green point - q.

Knowing that q is equidistant from all those three points, we can draw a conclusion that q is a vertex of a Voronoi Diagram [6] of *red* points. This implies that the *combinatorial structure* of $U(r)$ changes only when r is the same as $||p - q||$, where q is a Voronoi vertex, and p is the closest point to q.

That is why we build the Voronoi Diagram of *red* points, which takes $O(n_r \log(n_r)$. For each vertex v of the diagram we add to a set Q a value $||v - q||$, which we call a *critical value*, where q is the closest point to v. Next, sort Q in $O(|n_r| \log(|n_r|)$ (as $|Q| = |n_r|$). By using binary search we are able to obtain the smallest value r_1 of Q that the CS algorithm returns true, and the biggest value r_2 on which the CS algorithm will return false. By that we ensure that $r^* \in (r_1, r_2]$ and the structure of $U(r)$ does not change.

This procedure will take $O(n \log(n))$ time, as the CS algorithm takes linear time. Now, assuming that in a single BFS step we handle I cells, summing up this step would take $O(I \cdot n \log(n))$, which is problematic as $I = O(n)$. That is why we first collect into Q *critical values* of all I instances of this problem, sort them together, and then search for the new interval. That way in a single BFS step we keep the complexity to $O(n \log(n))$.

5.3.2 Sorting the upper envelope vertices with blue points

As in the original CS algorithm, we now want to sort vertices of the upper envelope and *blue* points together. So far, we have calculated an interval $(r_1, r_2]$ in which the structure of the upper envelope is the same as it would be for r^*. We now want to shrink the interval even more, so that now if $r^* \neq r$ then for each $r \in (r_1, r_2)$ the list sorted by x of $U(r)$ and *blue* points has the same structure as a sorted list of $U(r^*)$ and *blue* points.

In the original paper [2] authors here propose to use Cole's parametric search [3], which will solve this problem in $O(n \log(n))$. However, this algorithm is fairly complicated, especially when implemented with the use of an AKS sorting network [10]. An easier to implement algorithm was presented by Goodrich and Pszona in [5]. This algorithm runs in time $O(n \log(n))$ with

high probability, as it is similar in spirit to a quicksort. In this paper we implemented that parametric sort presented in that paper [5].

The practical Cole's parametric search is based on a sorting algorithm called boxsort [11]. As stated it is similar to quicksort, but instead of there being only one pivot in a step, boxsort picks $\sqrt{n}$ pivots for an array of n elements. Going into more detail, here are the steps of a boxsort algorithm:

1. Randomly choose $\sqrt{n}$ elements, set those elements as *marked*,
2. Sort the *marked* elements,
3. Use the sorted *marked* elements to split an array into $\sqrt{n}+1$ sub-problems in a way so that every element from the i-th sub-problem is bigger than the $(i-1)$-th *marked* element, but smaller than the (i)-th *marked* element.
4. Recursively call boxsort on each one of $\sqrt{n}+1$ sub-problems.

The third step is realized by $n-\sqrt{n}$ binary searches through the *marked* items. It takes $O(n\log(\sqrt{n})$ time, but we may think of it as *marked* elements forming a balanced binary search tree. Locating a sub-problem for an item is then done by *routing* the element through the BST. The tree has $\log(\sqrt{n})$ levels, and as each *routing* are independent of one another, the third step can be realized in $\log(\sqrt{n})$ parallel steps.

We look at this algorithm as the parallel algorithm of parametric search, as explained in subsection 2.2. The CS algorithm will be the decision algorithm used. Let us introduce the notion of a comparison, and its root. In this case we cannot compare the x coordinate of some points, as the position of vertices of $U(r)$ is dependant on r. We want to determine the result of a comparison under the assumption that the radius of the unit-disks is r^*. To do that for each comparison we can find a *root* (if it exists). A *root* of a comparison is the value of the radius in which points we are comparing are equal. For example, lets say we want to compare the x coordinate of to vertices of the upper envelope. The *root* of that comparison is a value of r in which both the vertices have the same x coordinate. Having the *root* r, we can 'solve' (i.e. determine the result of that comparison when the radius is r^*) a comparison using the CS algorithm, as we will know whether $r \geq r^*$ or if $r < r^*$.

Solving comparisons in such a way is costly, as a single execution of the CS algorithm takes $O(n)$ time. The goal is to try and minimize the number of CS algorithms we need to execute. To do that we want to 'batch' as many comparisons as possible. That is why we view the boxsort algorithm as a parallel one, to 'batch' comparisons, as they do not depend on each other if they are to be executed simultaneously. When we accumulate a satisfying number of comparisons, and their *roots*, by solving a median comparison (i.e. the comparison with the median root) we can resolve at least a half of those comparisons. If the median root r_m is feasible, that means that every $r \geq r_m \geq r^*$. If r_m is not feasible than every comparison with a root smaller than the median satisfies $r \leq r_m < r^*$. Either way, we know the result to a half of comparison we batched. We will identify each comparison $x_i < x_j$ as a comparison of its root with r^* as $r_{ij} < r^*$.

Cole in his paper on parametric search [3] introduced a weighting scheme for comparisons. We assign *weight* to every comparison, and instead of taking the median comparison and resolving it, we take a weighted median [12], which can be done in $O(n)$ time. The weighing scheme of Godrich and Pszona [5] is a little different than that of Cole [3]. Lets introduce the concept

of an *active* comparison. This idea is straightforward in the case of a sorting network. We connect certain comparisons with directed *wires*. If a comparison has *wires* connected to it, it means that the comparison is dependent on previous comparisons. We also say that a *wire* is *active* when the origin comparison is resolved. Initially, only the input wires are *active*. We say that a comparison is *active*, when all its input wires are *active*, and the comparison itself is not resolved. When a comparison gets resolved, then it is no longer *active*, and its output *wires* become *active*, possibly activating next comparisons.

No we assign *weight* to each comparison. It makes sense here to think of boxsort as a sorting network. To correctly distribute *weight*, we need to add *virtual* comparisons to our 'network' (*Virtual* comparisons are only so that the *weighting scheme* works correctly, they are not necessary in implementation). This notion is further explained in [5]. The *weighting scheme* works as follows:

When a comparison C with *weight* w gets resolved, and causes m comparisons to become *active*, every one of them is assigned *weight* $\frac{w}{2m}$

Every comparison is assigned *weight* only once, from its last resolved parent comparison. With that information, the boxsort works as follows:

1. Randomly set $\sqrt{n}$ elements as *marked*,
2. Create $\frac{\sqrt{n}(\sqrt{n}-1)}{2}$ comparisons for sorting the *marked* elements, each of weight w.
3. When the last comparison finishes *activate* comparisons to *route* the elements to their corresponding sub-problem, each comparison of weight $\frac{w}{2n^2}$
4. Recurse into sub-problems, and assign weight $\frac{w}{4n^{4.5}}$ to initial comparisons in every one of them.

For more details check [3, 5]. We will also expand on the implementation, and calculating root values further in subsubsection 6.4.4.

Again, to keep the complexity of $O(n\log(n))$ we need to sort all of the points, of all the instances of sub-problem together. When sorting we look for two roots (r_1, r_2) such that the r_1 was the smallest present root of the comparison where the CS algorithm returned true, and r_2 was the largest were it returned false.

5.3.3 Deciding whether a blue point is below, or above the envelope

From the previous steps, we have calculated an interval $(r_1, r_2]$, with which, if $r^* \neq r_2$ then for each $r \in (r_1, r_2)$ structure of $U(r)$ and the order of vertices of $U(r)$ alongside *blue* points does not change. This guarantees that for every r in this interval, for each *blue* point q, q is spanned by a certain arc (lets call the arc $\alpha_q(r)$). Now the task is to shrink the interval, so that $(r_1, r_2]$ still contains r^* and if $r^* \neq r_2$ then for each $r \in (r_1, r_2)$ the relative position of every q (below or above) with respect to $\alpha_q(r)$ stays the same.

Observe, that the *blue* point does not change. It is only $\alpha_q(r)$ that changes with r. Thus, for each *blue* q, we need to calculate a *critical value* when the arc $\alpha_q(r)$ is on that point. We store all of those *critical value* in a set Q. Now, similarly as in the previous steps, we sort values of Q, and do a binary search using the CS algorithm, to find two values: r_1 and r_2. r_1 is the largest value on which the CS algorithm returns false, and r_2 is the smallest value when the result is true. We then shrink the interval to $(r_1, r_2]$ As the relative position of any q and $\alpha_q(r)$

changes only when r is on one of those specific values of Q, we have obtained an interval so that if $r^* \neq r_2$ then for each $r \in (r_1, r_2)$ relative position of every q and $\alpha_q(r)$ is fixed.

The above algorithm will take $O(n \log(|n_b|))$ time. We again run into problems with complexity, when looking at all I instances of this problem. Here again as in the subsubsection 5.3.1 we need to take into consideration *critical values* of those instances all together. We then sort them together, and find the new r_1 and r_2 in the resulting set.

5.3.4 Terminating the breadth first search

After shrinking the interval in all those steps, we are ready to conduct a single step in the BFS algorithm. It is done in the same way, as it would have been done in the original CS algorithm, now for any $r \in (r_1, r_2)$.

We stop the algorithm if we conduct λ steps, or we find the end point t in any S_i. Wang and Zhao end this part with following lemmas alongside their proofs [2].

Lemma 5.2. *If $t \in S_i$ and $i \leq \lambda$, then $r^* = r_2$.*

Lemma 5.3. *If $t \notin S_i$ and $i \geq \lambda$, then $r^* = r_2$.*

This ends this section, bringing us to the conclusion:

Theorem 5.4. *The reverse shortest problem for the unweighted unit-disk graphs can be solved in $O(\lfloor \lambda \rfloor \cdot n \log(n))$ time.*

We will expand on the implementation of that algorithm in the following chapter.

6 Implementations

In this section we will discuss the implementation of both algorithms. They are available in a public repository, written in C++. It also makes use of an external CGAL library - The Computational Geometry Algorithms Library.

6.1 Endpoints

The implementation consists of three endpoints in template classes:

```
T brut<T>::solve(std::vector<Point<T>> points, int lambda,
    int s_i, int t_i);
T alg_1<T>::solve(std::vector<Point<T>> points, int lambda,
    int s_i, int t_i, int precision = 1000);
T alg_2<T>::solve(std::vector<Point<T>> points, int lambda,
    int s_i, int t_i);
```

Where **Point<T>** is a template class representing a two coordinate point. **points** refer to a list of centers of unit disks, and **s_i** and **t_i** are indexes of start, and end points in that list. All three function return the value of r^*. The next parts dive deeper into the implementation of the First, and the Second Algorithm.

6.2 The First Algorithm

As discussed in section 4 the First Algorithm is a binary search on real numbers, using the CS algorithm as our decision algorithm. That is why in this section we will focus on the implementation of the CS algorithm. To run the BFS algorithm, we need to first build the grid:

6.2.1 Building the Grid

By building the grid, we mean to compute the information of Theorem 2.1. We first calculate the vertical partition lines, and then the horizontal ones. Lets focus on the vertical lines, as calculating the other lines is very similar. The first step is to obtain two lists of points: $l_1, l_2, l_3, ..., l_m$ of points to the left of s sorted by the x coordinate and, a list of points to the right of s $p_1, p_2, p_3, ..., p_k$. Both lists contain s, so $s = l_1 = p_1$, and both lists consist only of points that have a chance to be reached from s. By having a chance we mean that $\forall_{i>j} x(p_i) - x(p_j) <= r$ and $\forall_{i>j} x(l_i) - x(l_j) <= r$. We start with a vertical line going through s. And we add a lines every $\frac{r}{\sqrt{2}}$, until there exists a line to the right of p_k. We do the same to the left side of s, adding lines every $-\frac{r}{\sqrt{2}}$ until there is a line to the left of l_m. We repeat that procedure for vertical lines.

The next step is to assign points its corresponding cell and grid coordinates (i.e. (i, j) where i means that the i-th horizontal line is the closest horizontal line to the left of a point, and j is the j-th vertical line, which is the closest line below that point). To assign the coordinates, we can iterate through the sorted points, firstly by x coordinate and check if we passed another vertical line. Then we do the same by the y coordinate. We cannot keep every cell, as pessimistically there could be way too many. We want to keep only those, that have at least one point as there are $O(n)$ of them. In a grid structure we keep a list of cells satisfying that property. To find all those non-empty cell we create a list of pairs (i, j) that correspond to a grid coordinates of each point. We now sort them using a radix sort as both i and j are bounded by n. This sorts the array in $O(n)$. Then iterating that sorted array we can find duplicate cells, and for each cell create a list of points inside it.

The last step is to calculate neighbouring cells of each cell. This can be done in $O(n)$ time by a scan through non-empty cells. Recall that $N(C)$ are in at most five rows of the grid. We take advantage of that during the scan, and maintain four cells - for a cell C with grid indexes (i, j) we keep four neighbouring cells C' with indexes (i', j') where $i' \in \{i-2, i-1, i+1, i+2\}$, and j' to be the closest to j (i.e. $|j' - j|$ is minimized). Then we can look for neighbours near that four cells. It may happen that there is no cell C' with a ordinate grid index that we are looking for, then we skip that cell. Summarizing the scan looks like this:

1. Move to the next non-empty cell C,
2. Move every one of the four cells C', so that they satisfy the conditions stated earlier.
3. For every cell C' try moving through the list of non empty cells starting at the index of that cell looking for neighbours of C. There is only $O(1)$ indexes that should be checked.
4. Repeat from step 1, until all non-empty cells have their neighbours assigned.

This computes the grid information in $O(n)$ time complexity.

6.3 Running BFS

This section will explain the implementation of a BFS step explained in subsubsection 2.1.2. For each point we keep its distance from the starting point s (Initially set to -1 for all points expect s where its 0), in a classic breadth first search fashion. Our goal is to calculate S_i based on S_{i-1}. As S_{i-1} is a set of points, our first goal is to find a set of cells C_S that have at least one point from S_{i-1}. To do that we do the following for each point p in S_{i-1}:

1. Iterate through points of a cell C that p is in:
   (a) If the distance of a point is -1 set it to i, and add the point to S_i,
   (b) Else if the distance of a point is $i-2$ or $i-1$, set the distance to $-i-2$,
2. If there was a point in a cell that satisfied either of conditions listed above, add the cell to C_S.

This allows us to obtain a list of cells containing points of S_{i-1} without duplicates, and ensures that the points of S_{i-1} will not be used again in S_i, as two steps may contain points from the same cell. Now to obtain the points of S_{i-1} grouped by cells, we iterate through the points of a cell from C_S, and get only the points where the distance is $-i-2$.

The next step is to solve the sub-problem 1 of Theorem 2.2, for each neighbor of each cell of C_S. Lets focus on one instance of this sub-problem. The points of our cell C present in S_{i-1} that we have calculated earlier will be our *red* points n_r and all points of the neighbour cell C' will be the *blue* points. The line that is the 'wall' of C in the direction of the neighbour is our L. In the implementation we are only choosing the vertical 'walls' of C only when C' has the same y grid coordinate as C. As stated in Theorem 2.2 the first task is to calculate the upper envelope of arcs centered at points of n_r.

6.3.1 Graham scan of pseudo-lines

The implementation of this part is very similar to the pseudo code given in Skrepetos's work [7], but with a little modification. The pseudo-code is presented in Algorithm 1.

The question is however, what is meant by 'intersecting' or the 'intersection point' of those pseudo-lines as we do not want to compute them explicitly. By the intersection point, we mean an intersection point of corresponding circles that have a greater y coordinate. And to check if two arcs intersect, we check whether the previously defined intersection point is above L. The only difference of the code in Algorithm 1 and the original pseudo-code [7] is at lines 4-7. That is because our arcs can only not be intersecting when they are 'inside' one other, as every two *red* are in the same cell. This is shown in Figure 5.

In that situation we only want to keep the arc that is above the other, so the one with its y coordinate of its center greater. That explains how we calculate the upper envelope of arcs above L centered at *red* points.

6.3.2 Adding points to the next step

After obtaining the upper envelope the goal is to decide which *blue* points are under that envelope. As in subsubsection 2.1.2 we sort the vertices of the envelope with *blue* points. To get the vertices of the envelope we calculate the intersection points of every two consecutive

Algorithm 1 Graham scan for arcs

Input: An array P of n arcs, ordered with respect to x-values of their centers
Output: A stack S of the arcs of P that participate in the upper envelope of its arrangement

```
1:  S.PUSH(P[0])
2:  for i = 1 to n-1 do
3:      while true do
4:          if P[i] and S.TOP do not intersect then
5:              if P[i] is above S.TOP then
6:                  S.POP
7:                  S.PUSH(P[i])
8:              break
9:          else
10:             if S.NEXT_TOP = NULL then
11:                 S.PUSH(P[i])
12:                 break
13:             else
14:                 v = intersection point of S.TOP and S.NEXT_TOP
15:                 w = intersection point of S.TOP and P[i]
16:                 if w lies to the right of v then
17:                     S.PUSH(P[i])
18:                     break
19:                 else
20:                     S.POP
21: end for
22: return S
```

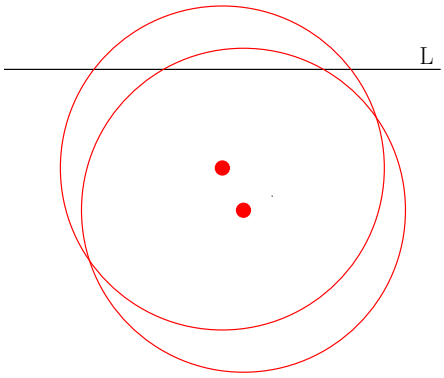

Figure 5: Two arcs 'inside' one another

arcs. We than can merge them with the *blue* points in $O(n)$ time as both the vertices, and the points of C' are sorted. Than we need to iterate through this newly created list, and remember the last seen vertex (initially the first arc of the upper envelope), because the second arc that

created this vertex is the closest arc to the *blue* points that will be present in this array before the next vertex. For each *blue* point we check whether the distance to the center of the disk that defined that arc is at most r. When that is the case, and the *blue* point is not yet visited (i.e. the distance is set to -1) We add the point to S_i.

This whole procedure also needs to be adapted for the case when the neighbouring cell has the same y grid coordinate as C. Then L is vertical, and we need to compare by the y coordinate. The next modification will need to be made when C' is below C. Then e.g. the intersection point of two arcs will be the one with smaller y coordinate.

If it happens that we find the end point t in S_i we end the algorithm, and return true. We do λ steps of such BFS in total, when the algorithm ends without founding t, or we find that $S_i = \emptyset$ we return false. This ends the implementation of the first algorithm.

6.4 The Second Algorithm

The implementation of the second algorithm takes major inspiration from the First, as many steps are very similar. As explained in section 5 we maintain an interval $(r_1, r_2]$ initially set to $(0, \infty]$, so in our case: (0,std::numeric_limits<T>::max()]. We keep two lists of sorted input points, by the x coordinate, and by y.

6.4.1 Building the Grid

The first task is to shrink the interval and build the grid as in subsection 5.2. We do that by utilizing the sorted matrix search techniques of [8, 9]. The implementation of such a search is straightforward. We identify a sub-matrix with two pairs of indexes, one pair for the largest element, and one pair for the smallest. We than do the procedure described in subsection 5.2 8 times in total, as we have to find values of $(r_1, r_2]$ (so two searches, one for the smallest feasible value, and one for the largest non feasible value) four different times:

- For points to the right of s, by the x-coordinate,
- For points to the left of s, by the x-coordinate,
- For points above s, by the y-coordinate,
- For points below s, by the y-coordinate

For each case we create a list of points including s as in previous chapters.

After shrinking the interval four times, we build the grid for any value $r \in (r_1, r_2)$. In the implementation we have built it for $\frac{r_1+r_2}{2}$

6.4.2 Parametric breadth first search

Now the goal is to run a parametric BFS with the help of that grid. Similarly as before, we have to calculate S_i using S_{i-1} while shrinking the interval. As in the implementation of the first algorithm in subsection 6.3, for every point we keep distance from s, and with that, we can obtain a set of cells C_S that have a point in S_{i-1}. Then for each neighbour C' of each cell C we solve the sub-problem of Theorem 2.2. As discussed in section 5, we need to 'batch' the *critical values* from all instances of this sub-problem, to ensure that each step is of $O(n \log(n))$ time complexity.

6.4.3 Computing the upper envelope

The first task is to shrink the interval so that if $r^* \neq r_2$ then for every $r \in (r_1, r_2)$ the *combinatorial structure* of the upper envelope stays the same. To do that we calculate a set Q (start with $Q = \emptyset$). For each cell C of C_S do the following:

1. Obtain the points of C that are in S_{i-1},
2. Calculate a Voronoi diagram for those points.
3. For each vertex v of the Voronoi diagram, and its nearest point q, add to Q the value of $||q - v||$.

To calculate the Voronoi diagram we used the C++ class **CGAL::Voronoi_diagram_2** available in the CGAL library.

After the procedure is done for all cells of C_S. We can sort Q, (using any $O(n \log(n))$ sorting algorithm), and do a binary search using the CS algorithm to find our desired values of $(r_1, r_2]$. r_1 as the smallest feasible value of Q, and r_2 as the largest non-feasible.

6.4.4 Sorting the upper envelope

The second task is to shrink the interval so that if $r^* \neq r_2$ then for every $r \in (r_1, r_2)$ the order of vertices of the upper envelope and *blue* points stays the same. We use the Cole's parametric search [3, 5], explained in subsubsection 5.3.2.

As in the previous step, we collect all points that we want to compare into a set Q. For each cell C of C_S:

1. Obtain the points of C that are in S_{i-1},
2. For each neighbouring cell C'
   (a) Calculate the upper envelope of points of C, using $r \in (r_1, r_2)$ (e.g. $r = \frac{r_1+r_2}{2}$),
   (b) Add to Q all the vertices of the upper envelope, and all the points of C'

Now we perform the *parametric sorting*.

Lets first discuss the mater of comparison *roots*. It is a value of r, for which the two points we are comparing are equal in the compared coordinate (i.e. $x(p_1) = x(p_2)$). We have four types of comparisons:

1. A comparison between two *blue* points,
2. A comparison between two vertices of the upper envelope from the same instance of subproblem 1,
3. A comparison between a vertex, and a *blue* point,
4. A comparison between two vertices of the upper envelope from different instances,

The first type, as *blue* points does not depend on the value of r, are trivial to handle, by comparing the coordinates of those points.

For the second type, as vertices from the same instance have the same *combinatorial structure* for every $r \in (r_1, r_2)$, we can calculate the vertex points for any set r. In the implementation we chose $r = \frac{r_1+r_2}{2}$. Then we compare the coordinates of those vertices calculated for that r.

The next types are more intricate. For a comparison between a vertex and a *blue* point, we need to observe that a vertex, for all r, lies on a perpendicular bisector between centers of circles, that defined the arcs that made up said vertex, as shown on Figure 6.

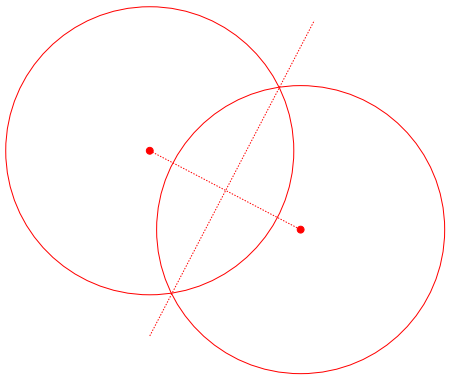

Figure 6: As r changes, the position of the intersection point of two unit-disks stays on the bisector between the two centers

The vertex 'moves' on that bisector when r changes. To find the *root* we have to find the value of r for which the vertex has the same x coordinate as the *blue* point. To do that, first obtain the formula of that bisector, and then calculate the value of that function with x equal to that of the *blue* point's abscissa.

For the fourth type, we have two vertices, that are 'moving' along their corresponding bisectors while r changes. A vertex of the upper envelope is the intersection point of two arcs. The *root* is the value of r, for which those intersection points will have the same x-coordinate, as shown in Figure 7

We can calculate the intersection point of two circles using geometry. Now we equate the x coordinate of both vertices, and from there we calculate a formula for r. The calculations are pretty complex, so for more detail, check the source code of the implementation.

It may also happen that for some of those types there is no *root*. While implementing, calculations for all those types need to by adapted to work for each relative position of C and C'.

As stated previously in the subsubsection 5.3.2, the task now is to sequentially simulate the parallel execution of the boxsort sorting algorithm. We start with two values $(r_1 = 0, r_2 = \infty)$, which we will shrink during the sorting. When we resolve a comparison i.e. we know whether the root $r < r^*$ or $r \geq r^*$, we shrink our internal interval accordingly.

To simulate the parallel execution of the algorithm, we need to keep a list of sub-tasks, so for each recurrence step we keep a list of points to be sorted, and the initial comparison weight. Assuming we have z sub-tasks with sizes n_i for $1 \leq i \leq m$ (Initially there is one task of size n), and initial weights w_i for $1 \leq i \leq m$, a single step of our algorithm is as follows:

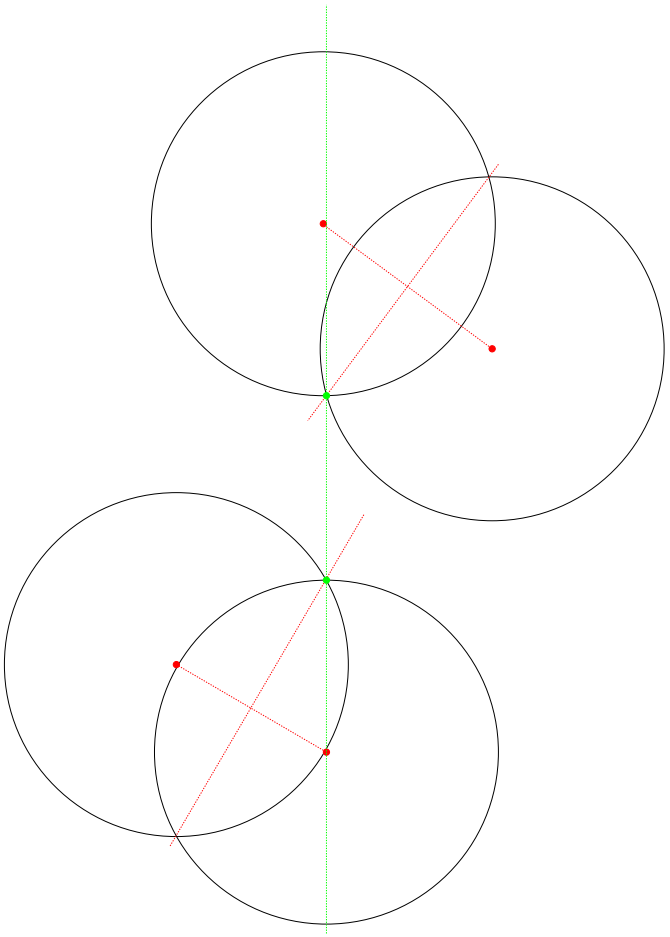

Figure 7: Comparison of the fourth type, of two vertices of different instances. The green points show the intersection points that have the same x-coordinate.

1. For each sub-task randomly choose $\sqrt{n_i}$ elements, set those elements as marked, and create a list of them M.
2. For each sub-task create $\sqrt{n_i} \times (\sqrt{n_i} - 1)$ comparisons for sorting M. If a comparison does not depend on the value of r, resolve it in $O(1)$ time.
3. Resolve the comparisons:
   (a) Find the weighted median of all comparisons,
   (b) Using the CS algorithm resolve the median comparison (using its root r_m), shrink the interval accordingly.
   (c) If r_m is feasible, solve all comparisons where the root is no smaller than r_m, and remove them from the list of comparisons.

(d) If r_m is not feasible, solve all comparisons where the root is no bigger than r_m, and remove them from the list of comparisons.

(e) If there are still comparisons left, go back to step (a)

4. For each sub-task use a sorting algorithm (e.g. bubble sort) to sort M using the comparison results from previous steps.

5. For each sub-task create a binary search tree T_i using the sorted list of M, and create comparisons of every non-marked element with the root element of T_i, with weight $\frac{w_i}{2n^2}$

6. Resolve those comparisons and route the non-marked elements through T_i:

   (a) Solve comparison similarly to step 3,

   (b) If a comparison is resolved, add the next comparison to active comparisons according to the T_i, with weight following the *weight rule*.

   (c) For each item a find the corresponding bucket i.e. two elements in T_i: x and y, where x is the 'smallest' element 'bigger' than a, and y - the 'biggest' element 'smaller' than a.

7. For each sub-task, for each bucket re-curse into a new sub-tasks with initial weight $\frac{w_i}{4n^{5.5}}$

As stated previously, to find the weighted median we use the algorithm of [12]. It runs in $O(n)$ time, and is similar in spirit to a classic *SELECT* algorithm.

To create a BST tree from a sorted list, we pick the middle element as a root, and then do the same for the left child of the root (elements to the left of root element), and the right child of the root (elements to the right of the root element)

At the end we shrink the main interval, using the one calculated in this step. This ends this subsection.

6.4.5 Deciding whether a blue point is below, or above the envelope

Next, the goal is to shrink the interval, so that the relative positions of all *blue* points and the upper envelopes stays the same. Again, we create a set Q, and try to accumulate *critical values* from all instances of sub-problem 1. We do the same procedure as in a normal CS algorithm, but instead of adding points to S_i, we add the *critical values* as explained in subsubsection 5.3.3. Some steps, and calls to other function require the value of the radius r, when this is necessary we can use any value $r \in (r_1, r_2)$. We used $\frac{r_1+r_2}{2}$. For each cell C of C_S do the following:

1. Obtain the points of C that are in S_{i-1},

2. For each neighbouring cell C' of C:

   (a) Calculate the upper envelope of *red* points as in subsection 6.3,

   (b) Find the vertices of the upper envelope as in subsection 6.3,

   (c) Merge the list of vertices and *blue* points into a sorted list,

   (d) Iterate through this array and for each *blue* point p add to Q the value of r for which the arc, (with its disk center at point q) that is 'covering' the *blue* point, is on that *blue* point (i.e the value of $||p - q||$).

This procedure as before needs to be adapted for the case when C has the same ordinate grid coordinate as C', or when C' is below C.

After doing that for every C in C_S, we sort Q and find the values of r_1 and r_2 by a binary search using the CS algorithm as our decision algorithm. We look for the smallest value where the CS algorithm returns true, and the largest where it returns false. We then shrink the main interval.

Now to conduct the 'real' step after shrinking the interval, we again set $r = \frac{r_1+r_2}{2}$ and conduct a step as it would be conducted in the CS algorithm on the value of r set earlier.

This ends the section on implementation. For more details, please visit the public repository where source code is present.

7 Performance

In the last section we compare the performance of the three implementations presented above.

The public repository contains a test generator, which stores the case scenario as follows: In the first line four numbers separated by a white-space: n, λ, s_i, t_i, where n is the number of points, and λ, s_i and t_i is defined as in the RSP. Than in next n lines, there are pairs of floats. The i-th line describes the coordinates of the i-th line.

A built repository contains three executable files, one for each algorithm, which parses the case scenario, and outputs the result, alognside the execution time.

We prepared a big test ($n = 5000$), executed those three executables, and recorded the execution time for each of them. We run the test for various values of λ, as the Second Algorithms complexity depend on the value. Time is measured by utilizing the C++ **chrono** library. For those measurements we used $\theta = 50$ for the first algorithm.

	$\lambda = 1$	$\lambda = 2$	$\lambda = 5$	$\lambda = 100$
The First Algorithm	14154 ms	13539 ms	13770 ms	12724 ms
The Second Algorithm	798 ms	6924 ms	68651 ms	73259 ms
Brute-Force Algorithm	19287 ms	17888 ms	17195 ms	16195 ms

As we can see the second algorithm has unmatched performance when the value of λ is low. As expected the value of λ does not drastically change the execution speed of the first algorithm and the brute-force algorithm. The second algorithm quickly becomes the slowest, which can be attributed to a massive constant coming from a difficult implementation.

We now move to compare the execution speed of the first algorithm depending on the value of θ - the binary search precision. We prepared an even bigger test ($n = 7000$), and again measured the time now for different values of θ. The brute force algorithm finished the execution in **18012** milliseconds, with $r^* = 2.83686$ and the first algorithm results are in the following table:

	$\theta = 5$	$\theta = 10$	$\theta = 20$	$\theta = 50$
The First Algorithm time	2425 ms	7188 ms	12599 ms	34742 ms
The First Algorithm result	3	2.83789	2.83686	2.83686

As presented, the optimal value of θ for our approximation is between 10 and 20, where the first algorithm is still way faster than the brute force one.

There are improvements that can be applied to those implementations. Let's hope the future generations are not afraid to make them.

References

[1] Chan, T.M., & Skrepetos, D. (2016) All-Pairs Shortest Paths in Unit-Disk Graphs in Slightly Subquadratic Time. *International Symposium on Algorithms and Computation.*

[2] Haitao Wang, Yiming Zhao (2021) Reverse Shortest Path Problem for Unit-Disk Graphs

[3] Richard Cole (1987) Slowing down sorting networks to obtain faster sorting algorithms.

[4] Nimrod Megiddo (1983) Applying Parallel Computation Algorithms in the Design of Serial Algorithms

[5] Michael T. Goodrich, Paweł Pszona (2013) Cole's Parametric Search Technique Made Practical

[6] Franco P. Preparata and M. Ian Shamos. (1985) Computational Geometry: An Introduction.

[7] Dimitrios Skrepetos (2018) Shortest Paths in Geometric Intersection Graph

[8] Greg N. Frederickson and Donald B. Johnson (1984) Generalized Selection And Ranking: Sorted Matrices

[9] Greg N. Frederickson and Donald B. Johnson (1983) Finding k-th Paths and p-Centers by Generating and Searching Good Data Structures

[10] Miklós Ajtai, János Komlós, and Endre Szemerédi (1983) An O(n log n) sorting network. In Proceedings of the 15th Annual ACM Symposium on Theory of Computing (STOC), pages 1–9

[11] R. Reischuk. (1985) Probabilistic parallel algorithms for sorting and selection. SIAM J. Comput., 14(2):396–409

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. (2009) Introduction to Algorithms. The MIT Press, 3rd edition.