DM819
Exam Project Part 2

Jonas Bork Jørgensen - jonjo21
Mikkel Bie Madsen - mikkn21

8th of December, 2024

Contents

1 Problem Description

This project aims to implement Fortune's algorithm[1] to compute Voronoi diagrams in time $O(n \log n)$. To simplify the problem we assume the input points to be in *general position*. Specifically, the input points satisfy the following conditions:

- All points are distinct.
- No four points lie on a common circle.
- No two points share the same x-coordinate.
- No two points share the same y-coordinate.
- All points are in the plane.

As we assume the input to be in a general position, we do not have any *special cases* to consider. Note that the program might still support these cases but will not be guaranteed. As per the rules of the project[8], we assume our search tree to be balanced, when we analyze the asymptotic complexity of our implementation in section 3.

2 Implementation

This section will cover the interesting details and quirks of our implementation of Fortune's Algorithm.

The algorithm requires different information to be stored in internal nodes and leaves of the status tree so we created our own tree implementation to have better control over this. Additionally, we needed to tightly control how to add arcs to the tree which was made easier by using our own implementation of a tree. This, however, also means that the tree in our implementation is not balanced but we will be assuming it's balanced during the running time analysis of the algorithm.

We do not store the breakpoints in the internal nodes nor do we store the parabolas in the leaves since they change every iteration so we instead store the information needed to calculate them and then calculate them when they are needed.

2.1 Geometric Calculations

Bounding box: When constructing the bounding box, we need to handle the infinite half-edges by extending or contracting them to fit into the bounding box. An infinite half-edge will always have one endpoint inside the bounding box (because we have defined the bounding box as such) and another endpoint exactly on the border of the bounding box since the edge is infinite and has an endpoint within the box. To fit the edge inside the bounding box, the infinite end of the edge needs to be moved onto the border of the bounding box based on the direction of the edge. The infinite end of the half-edge is a breakpoint in the algorithm and thus is stored in the status. Therefore, we have two points on the infinite half-edge and can define a line, $y = ax + b$, between them which can be used to find more points on the edge. Let (x_1, y_1) and (x_2, y_2) be two points on the edge, then a is:

$$a = \frac{y_2 - y_1}{x_2 - x_1}$$

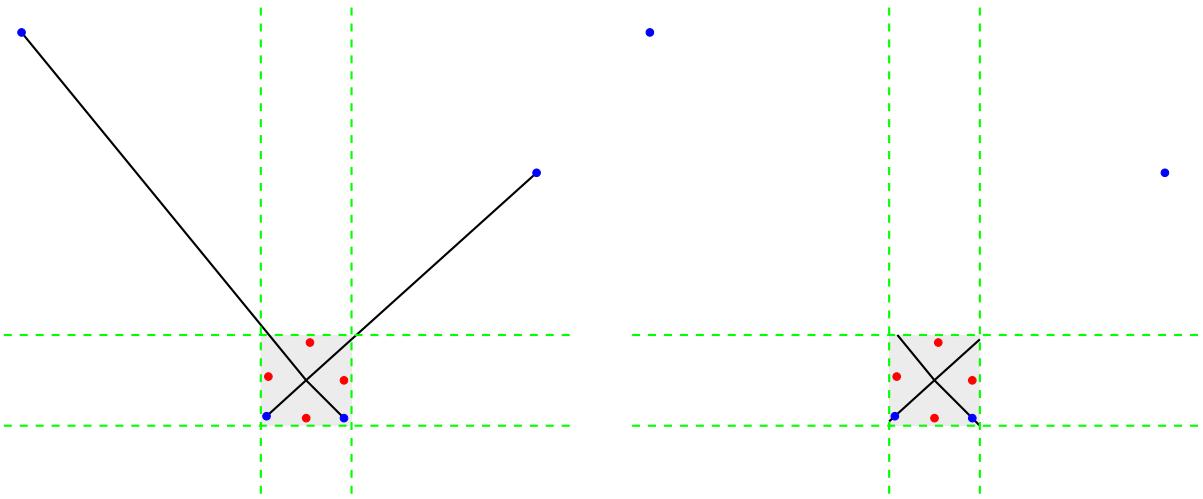
Once a is known, the y-intercept b can be calculated by rearranging the line equation $y = ax + b$:

$$b = y_1 - a \cdot x_1$$

This gives us the line, $y = ax + b$, which is the extension of the infinite half-edge.

The candidates for where the infinite endpoint of the edge should be placed are where the line intersects with the border of the bounding box, where each side of the border of the bounding box has been extended to infinite lines. The line can intersect with all four sides of the bounding box since the line does not encode the direction of the edge. Thus, by considering the direction of the edge, we can prune two of these candidates which are the sides of the bounding box that are not in the direction of the edge. The remaining candidates are true intersections of the infinite half-edge so to fit the edge to the bounding box, we need to find which intersection happens first. The intersection that happens first is the one that has the smallest distance to the origin of the edge.

The bounding box is defined by a window of $[x, x']$ and $[y, y']$ where $x < x'$ and $y < y'$ such that x defines the left side of the bounding box, x' the right side, y the bottom and y' the top. Thus, to find the intersection between the line of the edge and one of the sides of the bounding box, you insert the value of the side into the line equation and solve for the remaining unknown variable. An example of this process is visualised on Figure 1.



(a) Before updating half-edges

(b) After updating of half-edges

Figure 1: Visualization of the *bounding box* and how half-edges are updated to fit within the bounding box. Figure 1a depicts the output of Fortunes algorithm where a bounding box have been calculated. Figure 1a depicts how half-edges have been adjusted using the bounding box, for a more accurate visualize output. Note that blue vertices are an "infinite" direction and thus the edges incident to these vertices are half-edges. The red vertices are sites. The gray shaded area is the area shown in our visualized output.

Define circle from 3 points: To determine the center and radius of a circle passing through three non-colinear points, we set up a system of equations based on the general equation of a circle

centered at (h, k) with radius r:

$$(x-h)^2 + (y-k)^2 = r^2$$

By substituting the coordinates of the three points $(x_1, y_1), (x_2, y_2)$ and (x_3, y_3) into this equation, we obtain three equations:

$$\begin{aligned}(x_1-h)^2 + (y_1-k)^2 &= r^2,\\ (x_2-h)^2 + (y_2-k)^2 &= r^2,\\ (x_3-h)^2 + (y_3-k)^2 &= r^2,\end{aligned}$$

Subtracting the second equation from the first and the third equation from the second eliminates r_2, resulting in two linear equations in h and k:

$$\begin{aligned}2h(x_1-x_2) + 2k(y_1-y_2) &= x_1^2 - x_2^2 + y_1^2 - y_2^2,\\ 2h(x_2-x_3) + 2k(y_2-y_3) &= x_2^2 - x_3^2 + y_2^2 - y_3^2,\end{aligned}$$

These equations can be simplified by dividing both sides by 2:

$$\begin{aligned}h(x_1-x_2) + k(y_1-y_2) &= \frac{x_1^2 - x_2^2 + y_1^2 - y_2^2}{2},\\ h(x_2-x_3) + k(y_2-y_3) &= \frac{x_2^2 - x_3^2 + y_2^2 - y_3^2}{2}.\end{aligned}$$

We now have a system of two linear equations with two unknowns h and k. We solve this using *Cramer's Rule*[10] to obtain the center $C(h, k)$. Note that if the determinant is zero, the points are colinear, and a unique circle cannot be defined.

Once the center $C(h, k)$ is found, the radius r is calculated by plugging the center back into the circle equation with any of the original points:

$$r = \sqrt{(x_i-h)^2 + (y_i-k)^2}, \quad \text{for } i = 1, 2, \text{ or } 3.$$

Find breakpoints: In Fortunes algorithm, each parabola of the beach line is defined such that from any point on the parabola to the *site* point is equal to the distance from that point to the *sweep line*. This corresponds to a parabola defined by its *focus* point and a *directrix*[2]. Each parabola can then be described as follows:

Definition 1 Let $s(h, k)$ denote the focus point and $y = b$ the directrix. Then the equation of the parabola for s is:

$$y = \frac{(x-h)^2}{2(k-b)} + \frac{(k+b)}{2}$$

Figure 2 depicts an overview of all the cases to consider when calculating breakpoints between two arcs. Since we are in *general* position only the cases marked with a green box are possible, and thus these are the cases we will discuss. Note however on Figure 2 there are some cases that are not possible, even if we don't assume general position, but nevertheless one should consider the case under implementation.

For the case depicted in Figure 2c we differentiate between the case of the figure and the analogous symmetric case. That is whether or not $S_1(h_1, k_1)$ or $S_2(h_2, k_2)$ is the vertical line. Then

using the equation from definition 1, we find the y-value of the intersection between the vertical line and the site as follows:

$$y = \frac{(h_1 - h_2)^2}{2(k_2 - b)} + \frac{(k_2 + b)}{2}$$

where the vertical line e.g., could be S_1 and the beach line $y = b$. The x-value of the intersection point is simply the x-value of the vertical line e.g., S_1. This gives us the following intersection point $P(h_1, y)$.

For the last case depicted in Figure 2e, the sweep line is below both parabolas defined by the sites $S_1(h_1, k_1)$ and $S_2(h_2, k_2)$. Then to find the breakpoint(s) between two arcs we define the parabolas corresponding to the two sites S_1 and S_2 using the equation from definition 1. The breakpoints occur at the intersection of these two parabolas, which requires solving for x in the following quadratic equation:

$$\frac{(x - h_1)^2}{2(k_1 - b)} + \frac{k_1 + b}{2} = \frac{(x - h_2)^2}{2(k_2 - b)} + \frac{k_2 + b}{2}.$$

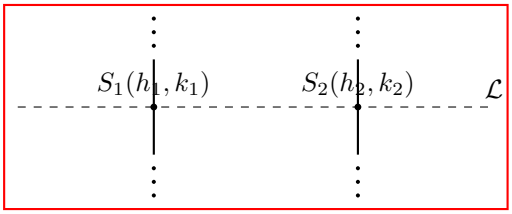
This equation is solved using the python library *SymPy*[9]. Only real solutions are retained, as they correspond to actual intersection points. For each real x-coordinate, the corresponding y-coordinate is computed as one would expect, by substituting x back into one of the parabola equations. Thus resulting in the coordinates of the breakpoint(s).

2.2 Checking For Circle Events

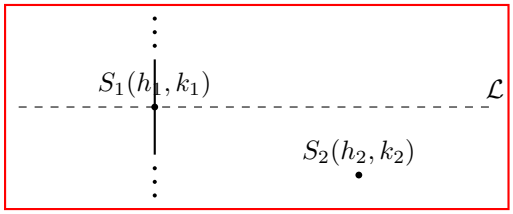
A circle event happens for three arcs if and only if the two breakpoints between these arcs converge. If they converge, then the point that they converge is the centre of the circle whose border intersects the three sites that created the arcs. The existence of this circle does not necessarily mean that the breakpoints converge but if it does not exist, then the breakpoints do not converge. This is because there can be two breakpoints where a circle can be defined (and thus a convergence point exists) but the breakpoints move away from the convergence point. This means that they will never converge. Thus, we need to additionally check whether the breakpoints move towards the convergence point or they move away from it.

Checking whether the breakpoints move towards the convergence point requires us to know the direction in which each breakpoint moves. This is, however, not an obvious property of the breakpoints since they are only points and therefore don't have a direction. A useful property of the breakpoints though is that the direction that a given breakpoint moves is constant throughout the algorithm. Thus, to find the direction of a breakpoint, we simulate the sweep line being slightly lower than it already is and then re-calculate the breakpoint. Now we have the original breakpoint and the new breakpoint which together defines the direction that the breakpoint moves, meaning we can check whether the breakpoint moves towards the convergence point. If they converge, we add the circle event to the event queue.

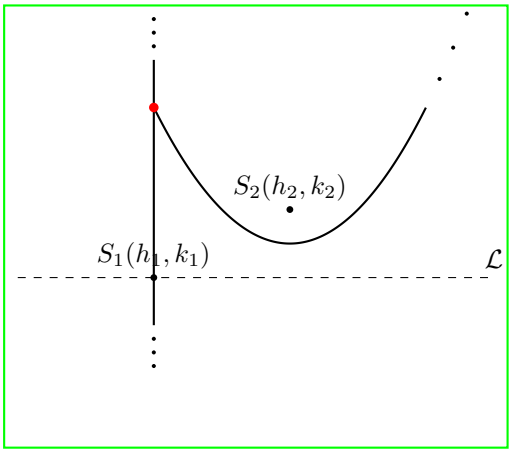
Thus, to check whether a circle event occurs, we first define a circle based on the sites that define the three arcs and then verify that the breakpoints between these arcs move towards the convergence point (the centre of the circle). Defining the circle takes $O(1)$ time since we have access to the sites from the three arcs but we don't have trivial access to the breakpoints between the arcs. The breakpoints correspond to specific nodes in the tree so from the leaves of the tree that correspond to the arcs, we need to search upwards to find the correct nodes. This takes $O(\log n)$ time. Once these are found, we can in $O(1)$ time find the direction of the breakpoints and check



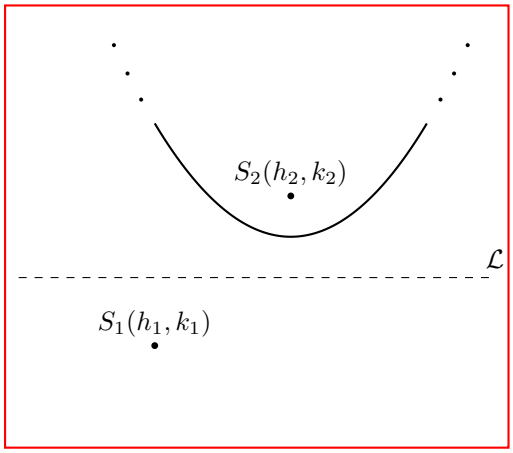
(a) *Case 1*: Both parabolas are vertical lines.



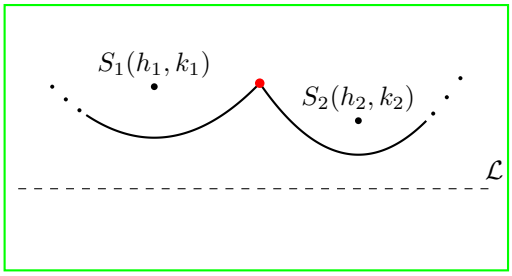
(b) *Case 2*: S_1's parabola is a vertical line on the sweep line and S_2's parabola is below the sweep line.



(c) *Case 3*: S_1's parabola is a vertical line on the sweep line and S_2's parabola is above the sweep line.



(d) *Case 4*: S_1's parabola does not exist above the sweep line



(e) *Case 5*: Both parabolas are valid and above the sweep line

Figure 2: The five cases to consider when calculating the breakpoints between the two arcs defined by the sites $S_1(H_1, K_1)$ and $S_2(H_2, K_2)$. The valid cases for general position are marked with a green box and the invalid are marked with a red one. The breakpoints between the two sites are depicted as a red dot. The sweep line $\mathcal{L}$ is depicted as a dotted line. Note that for the cases Figure 2b, Figure 2c and Figure 2d there is an analogous symmetric case.

whether they move towards the convergence point. Adding the circle event to the event queue (if the breakpoints converge) takes $O(\log n)$ time [5]. Thus, the total time to check whether a circle event happens and add it to the event queue is $O(\log n)$.

2.3 Update Breakpoints

In subsection 2.1, we discussed how we created the bounding box which required an infinite half-edge to have two points, i.e., the origin and some point on the infinite end of the half-edge. The latter does not, however, necessarily exist. During the algorithm, whenever we calculate a breakpoint, we update the origin of the corresponding edge to that breakpoint. Since the infinite end of the half-edge is a breakpoint in the status, this point will be updated if the breakpoint is calculated. If, however, the breakpoint is never calculated, the infinite half-edge will not have a point on the infinite end of it so the direction of the edge cannot be found (since we only have one point on the edge). To fix this issue, at the end of the algorithm, we move the sweep line slightly down and go through all the nodes in the tree and update their breakpoints, thus also updating the origin of all the edges in the diagram. Therefore, we are guaranteed that when we need to find the direction of any infinite half-edges, we have two points on the edge.

2.4 HandleSiteEvent and HandleCircleEvent

In our implementation, the majority of the HandleSiteEvent functionality that is described in the book is implemented in our tree, rather than a separate function that uses the tree. This is because HandleSiteEvent mostly manipulates the tree so it was easier to integrate it into the tree since we created our own implementation of the tree anyway. Ideally, we would have integrated HandleCircleEvent into the tree as well but due to the time constraint of the project, we prioritised getting other tasks completed in the project. Therefore, HandleSiteEvent is the `add` function of the tree which calls the `add` functions of the internal nodes and the leaves, where most of the functionality of HandleSiteEvent is in the `add` function on the leaves.

3 Asymptotic Complexity

Algorithm 1 Fortunes Algorithm

```
1: Let EVENT_QUEUE be all site events
2: Let STATUS be a balanced binary search tree
3: while ¬ EVENT_QUEUE.IS_EMPTY do
4:     event ← EVENT_QUEUE.POP
5:     if event is site event then
6:         HANDLE_SITE_EVENT(event)
7:     else
8:         HANDLE_CIRCLE_EVENT(event)
9:     end if
10: end while
11: LOWER_SWEEP_LINE_BY(10)                ▷ Simulate the sweep line lower
12: STATUS.UPDATE_BREAKPOINTS
```

Algorithm 1 is the general idea of our implementation of Fortune's algorithm. The while loop on line 3 runs over all n sites and a maximum of $2n-5$ additional circle events can be added to `EVENT_QUEUE` throughout the algorithm, per Theorem 7.3 and Lemma 7.8[1]. Additionally, per iteration the loop performs a pop operation on the `EVENT_QUEUE` this operation requires $O(\log n)$ work[6] as the `EVENT_QUEUE` is implemented as a sorted list from the python library *SortedContainers*. Thus, the while-loop on line 3 will do a maximum of $(2n - 5 + n) \cdot \log n = (3n - 5) \cdot \log n \in O(n \log n)$ work.

HANDLE_SITE_EVENT: Firstly we search our `STATUS` to find the leaf α that lies vertically above the site P being processed. This step requires at most $O(\log n)$ work. Next, we check for "false alarms", i.e., sites already associated with a circle event. Using P's reference to the circle event, we determine whether it is a false alarm. If so, the circle event is removed from the `EVENT_QUEUE`, which incurs an additional $O(\log n)$ work[4]. We then replace α in `STATUS` with a subtree and create the necessary edges. This step involves creating objects and updating pointers, which takes $O(1)$ work. Finally, we check the triple of consecutive arcs to see if the breakpoints converge in both directions. This requires identifying the two arcs immediately to the left and the two to the right of P. Each lookup in the tree takes at most $O(\log n)$, resulting in four tree searches with a total cost of $O(\log n)$. For each direction, we determine whether the arcs define a circle event, a process that also takes $O(\log n)$ as explained in subsection 2.2. In total, the complexity of `HANDLE_SITE_EVENT` is $O(\log n)$.

HANDLE_CIRCLE_EVENT: Firstly, we update the internal nodes of `STATUS`, to reflect the deletion of the site P that represents the disappearing arc α. This requires searching through `STATUS` to locate and update the relevant internal nodes, taking $O(\log n)$ work. Next, we delete all circle events involving α from `EVENT_QUEUE`. This step incurs an addition $O(\log n)$[4] work. We then compute the circle causing the circle event and determine its centre. A vertex is created at the circle's centre, and the necessary pointers are updated. This takes in total $O(1)$ work. Finally, we check the new triple of consecutive arcs. For the triple where the former left neighbour of α is the middle arc, we check whether its breakpoints converge. Similarly, we perform the same check for the triple where the former right neighbour is the middle arc. As explained in subsection 2.2 each of these checks takes $O(\log n)$ work. In total, the complexity of `HANDLE_CIRCLE_EVENT` is $O(\log n)$.

STATUS.UPDATE_BREAKPOINTS: As we go through the entire `STATUS`, we spend an additional $O(n)$ work.

Total Asymptotic Complexity: To recap we have:

- Loop $O(n)$ times
  - Pop from event queue $\in O(\log n)$
  - Handle site event $\in O(\log n)$ or handle circle event $\in O(\log n)$
- Updating breakpoints $\in O(n)$

Giving our implementation of Fortune's algorithm a total asymptotic complexity of $O(n \log n)$. Recall that we assume our `STATUS` to be a *balanced* search tree.

4 Test Suite

To test the correctness of our implementation we have made a suite of unit tests alongside a suite of end-to-end tests. As our implementation involves multiple data structures and numerous helper functions, we chose to unit test the most critical components, such as finding a breakpoint.

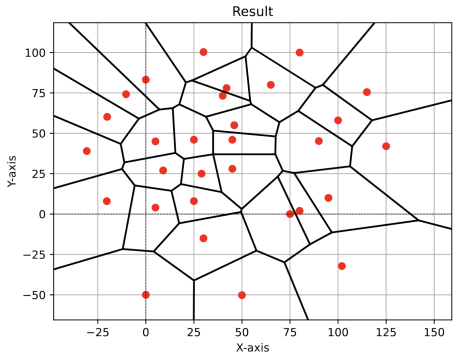
The unit tests were implemented using the python library *pytest*[7]. The tests can be run using the following command:

```
> pytest
```

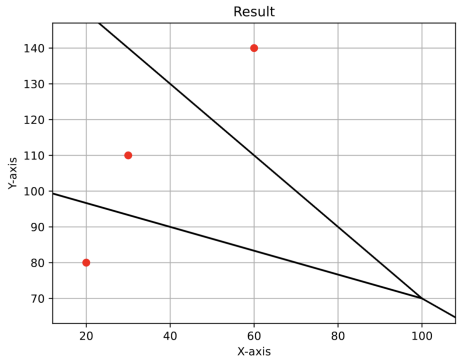
Additionally, pytest enabled us to configure a simple GitHub Action workflow. This ensures that our tests are automatically executed every time changes are pushed to the repository. Thereby ensuring that when changes are made, the critical components are not affected.

The end-to-end tests are simply a collection of `.txt` files containing sets of points. Since the input is in general position, there is a limited number of meaningful test cases to consider. Figure 3 depicts an overview of the visual results of the meaningful end-to-end test we concluded, these consist of:

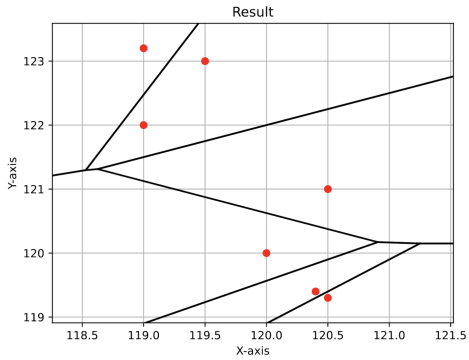
- test `test-big.txt` which results is depicted at Figure 3a. This is our big end-to-end test containing 30 points with a mixture of integers and floats.
  - Our big test also contains the case where one point $p_1(a, b)$'s x coordinate is the y coordinate of another point $p_2(c, a)$.
- test `test-circle.txt` which result is depicted at Figure 3b is a single circle event.
- test `test-close.txt` which results is depicted at Figure 3c is a small set of sites with x and y values that are close i.e., some points have a difference of 0.1.
- test `test-square.txt` which results is depicted at Figure 3d is four sites in a square that is slightly rotated to be in general position.



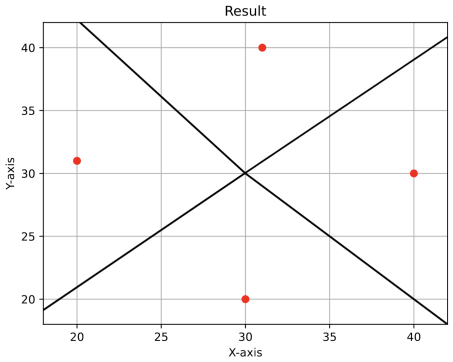
(a) Visual result of `test-big.txt`



(b) Visual result of `test-circle.txt`



(c) Visual result of `test-close.txt`



(d) Visual result of `test-square.txt`

Figure 3: Overview of results of end-to-end test. Note that sites are depicted as red dots

5 Manual

Note that these instructions have not been tested on IMADA's Virtual Computer Lab, but they will work on your own machine, given that you have Python installed.

Input: The input to the program is a set of points in *general position*. These points can be provided in two ways: either through a `.txt` file or via standard input (stdin). The format for the points is as follows:

```
x1 y1 \n
x2 y2 \n
...
```

Each line represents a point, where x and y are numerical values separated by a delimiter. The delimiter can be anything except for numerical values. This allows the user to write points x_1 y_1 as "a $x1$ a $y1$" or "$\{a$ $x1$ $\}(!$ $y1$". Note how the delimiters do not need to be the same nor do we

require a single delimiter, the only requirement for the user is that each point needs to end with a new line character ("\n"). Additionally, empty lines in the text file are ignored.

Output: The output is given both textually and visually. It is printed to standard out (stdout) to provide a textual representation of the result, i.e., all the edges and vertices of the doubly connected edge list which is the Voronoi diagram. The format of the textual representation is:

```
Edges:
(100.00, 70.00), inf(105.71, 66.19)
inf(105.71, 66.19), (100.00, 70.00)
(100.00, 70.00), inf(-305.47, 475.47)
inf(-305.47, 475.47), (100.00, 70.00)
(100.00, 70.00), inf(-134.39, 148.13)
inf(-134.39, 148.13), (100.00, 70.00)

Vertices:
(100.00, 70.00)
```

Where $inf(x, y)$ indicates that this is an infinite half-edge in the direction of the point (x, y). Note that the textual representation is only showing precision up to two decimal points. The result is also visualized and saved as a PDF "*result.pdf*", which is generated using the python library *matplotlib*[3]. The bounding box of our visualized voronoi diagram is calculated as the maximum and minimum x and y values of the coordinates of all sites and vertices of the doubly connected edge list. In addition, a small padding is calculated as a percentage of the bounding box which is added, such that it scales nicely for examples where sites and vertices are close. Sites are visualized as a red dot. Vertices are the points for which a maximum of 3 edges meet.

Setup: To make it easy to run the program, we have a "`requirements.txt`" containing all the relevant Python packages needed. Then one can simply run the command once in the same folder as `requirements.txt` file:

```
pip install -r requirements.txt
```

Once done, all necessary packages are installed.

Running the program: The program can be run in the two following ways, shown in Figure 4. Option one is to direct the path to the data file to the program using the `<` operator. The other is that you can run the program, and provide the data directly through stdin, and once done entering the data terminate the input. Depending on your operating system this is either `Ctrl+D` or `Ctrl+Z`.

```
> python prog.py  < data.txt
```

```
> python prog.py
> x1 y1
> x2 y2
> ...
> ^Z
```

Figure 4: The two ways of running our implementation of Fortune's algorithm. Note that depending on your operating system, it might be `python` or `python3` and `^Z` or `^D`.
Further, the "`data.txt`" is the path to your data file.

6 File Overview

This section briefly explains the contents of each file contained in the project

- **data:** Directory containing several `.txt` files, for our end-to-end tests.
- **src:** Directory containing the actual implementation
  - **visualization**: Directory for our visualization code.
    - **dcel_plot.py:** File containing the code responsible for creating the visual output of our program.
  - **dcel.py:** File containing the data structure *Doubly-Connected Edge List*.
  - **events.py:** File containing classes for our site and circle events.
  - **geometry.py:** File containing our geometric calculations.
  - **fortunes.py**: File containing our implementation of Fortune's algorithm.
  - **point.py:** File containing the class for a point.
  - **prog.py:** File responsible for initialising our program and taking the input from the user.
  - **tree.py:** File containing our status, which is a binary search tree.
- **test:** Directory containing all our unit tests.
- **requirements.txt:** File containing all libraries and dependencies.
- **pytest.ini:** Contains `pytest` configuration.

Bibliography

[1] Mark de Berg et al. *Computational Geometry: Algorithms and Applications*. 3rd. Springer, 2008. ISBN: 978-3-540-77973-5. DOI: 10.1007/978-3-540-77974-2. URL: https://doi.org/10.1007/978-3-540-77974-2.

[2] GeeksforGeeks. *Focus and Directrix of a Parabola*. Accessed: 2024-12-01. 2024. URL: https://www.geeksforgeeks.org/focus-and-directrix-of-a-parabola/.

[3] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[4] Grant Jenks. *Sorted Containers Documentation*. Accessed: 2024-12-06. URL: https://grantjenks.com/docs/sortedcontainers/sortedlist.html#sortedcontainers.SortedList.remove.

[5] Grant Jenks. *SortedList Add Documentation*. Accessed: 2024-12-06. URL: https://grantjenks.com/docs/sortedcontainers/sortedlist.html#sortedcontainers.SortedList.add.

[6] Grant Jenks. *SortedList Add Documentation*. Accessed: 2024-12-06. URL: https://grantjenks.com/docs/sortedcontainers/sortedlist.html#sortedcontainers.SortedList.pop.

[7] Holger Krekel et al. *pytest 8.3.3*. Accessed: 2024-12-04. pytest Development Team, 2004–. URL: https://pytest.org.

[8] Kim Skak Larsen. *DM819 Rules and Guidelines*. https://imada.sdu.dk/u/kslarsen/dm819/rules.php. Last visited: 2024-12-05.

[9] Aaron Meurer et al. "SymPy: symbolic computing in Python". In: *PeerJ Computer Science* 3 (Jan. 2017), e103. ISSN: 2376-5992. DOI: 10.7717/peerj-cs.103. URL: https://doi.org/10.7717/peerj-cs.103.

[10] OpenStax. *Solving Systems with Cramer's Rule*. Accessed: 2024-12-04. 2023. URL: https://math.libretexts.org/Bookshelves/Precalculus/Precalculus_1e_(OpenStax)/09%3A_Systems_of_Equations_and_Inequalities/9.08%3A_Solving_Systems_with_Cramer's_Rule.