Department of Mathematics and Computer Science

University of Southern Denmark

DM819: Exam project one

Author
Mikkel Nielsen
mikkn21@student.sdu.dk

Author
Malthe H. Petersen
malpe21@student.sdu.dk

April 9, 2025

Contents

1 Problem description

The geometric problem we are solving is problem 2.14 from our course textbook "*Computational geometry: algorithms and applications*"[1]

> 2.14 Let S be a set of n disjoint line segments in the plane, and let p be a point not on any of the line segments of S. We wish to determine all line segments of S that p can see, that is, all line segments of S that contain some point q so that the open segment $\overline{pq}$ doesn't intersect any line segment of S. Give an $O(n \log n)$ time algorithm for this problem that uses a rotating half-line with its endpoint at p.

2 special cases

The degenerate cases/special cases that we considered are the following:

Figure 1 is the special case where points lay on the initial sweepline. This creates a special case, as the points that intersect this line have to be treated differently. If a point is the start of a line segment, nothing changes, and it is handled like any other point at the start of a line segment would. But if the point is the end of a line segment, then the point should be removed after all points intersecting the initial sweepline have entered the status . This way, the point will be reported as being seen if it was the point in front of the other points on the sweep line, but it will not stay in the status.

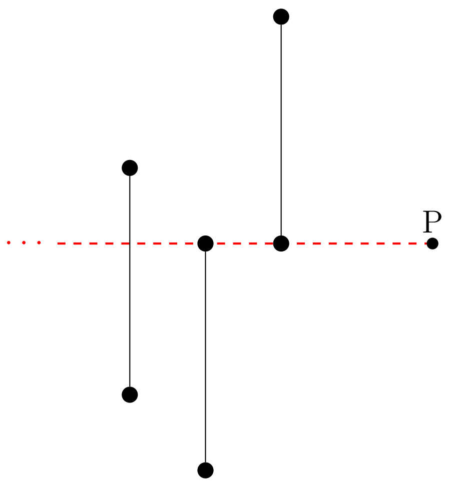


Figure 1: Special case 1: The red line is the sweep line that continues infinitely to the left and P is the query point

Figure 2 is the second special case where the degree of the line segments is the same. Here the distance to the query point P is used to add the closest points to the status first and remove the furthest first.

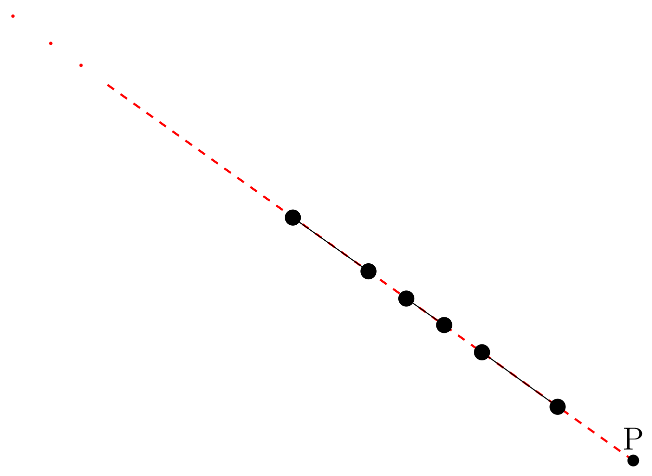


Figure 2: Special case 2: The red line is the sweep line that continues infinitely to the left and P is the query point

3 geometric calculations

Angles: We make use of the observation that the absolute difference between the angles Θ formed by the line segments $\overline{PE}_{\text{start}}$ and $\overline{PE}_{\text{end}}$. Which is the line segments from the query point P and the endpoints of any line segment. Relative to the sweep line l, in general position is less than 180 degrees when the line segment does not intersect the initial sweep line and is greater than 180 degrees when it intersects the initial sweep line. This observation is depicted on Figure 3. Then as described in Figure 1 there is a special case when one of these angles is equal to zero.

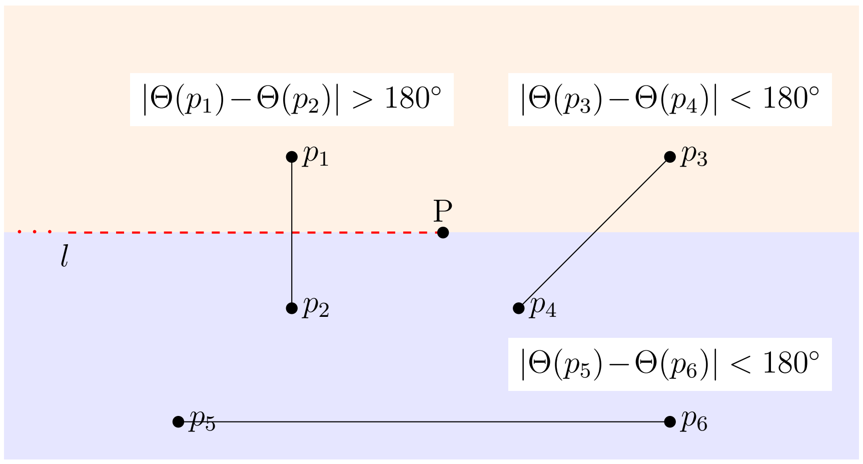

$|\Theta(p_1) - \Theta(p_2)| > 180°$
$|\Theta(p_3) - \Theta(p_4)| < 180°$
p_1
p_3
P
l
p_2
p_4
$|\Theta(p_5) - \Theta(p_6)| < 180°$
p_5
p_6

Figure 3: **Observation of angles**:. Let $\Theta(p_x)$ denote the angle between the line segment $\overline{p_x P}$ and l where l is the sweep line going from the query point P to infinity. Then we depict the two cases in general position of angles less than 180 degrees and greater than 180 degrees.

Extension point: When finding the intersection between two line segments, to compare the two line segments to check for an intersection between a line segment and sweep line we calculate an "extension point", by calculating a direction vector from the query point to the current event point. We normalize and extend this vector by a dynamic factor. This dynamic factor is the line segment's furthest point from the query point times two. This creates a line segment $\overline{PP_{\text{Extension}}}$ that is long enough to intersect the current line segment and the line segments currently in the status.

Parallel, Coincident & Collinear: Creating a line segment from the query point to the extension point gives us a line segment that we can use to check intersections with the other line segments currently in the status to determine the line segment closer to the query point.

The one interesting case is when the determinant is zero between these two lines. If this is the case, the lines are either parallel or coincident. A second determinant calculation is done to verify collinearity to confirm if the points from both lines are aligned. For collinear lines, a 1D overlap check is performed on the x- and y-coordinates to determine if the line segments overlap. If they do, we always take the endpoint of the line that is closest to the query point as our "intersection point".

4 asymptotic complexity

We did not implement a search tree ourselves, instead, we used the library `bintrees`[2] and their implementation of a self-balancing red-black tree. However, in the documentation[2] the asymptotic complexity of the insert operation is not specified, so we make the assumption that the asymptotic complexity for insert is $O(\log n)$, as this is the typical complexity of this operation.

When it comes to asymptotic complexity, there are two parts to consider, initialization and the actual algorithm:

Initialization: The initialization loops over all the line segments, already putting the initialization at $O(n)$ asymptotic complexity. For every line, some angle and distance calculations are made, but these are all constant operations. The rest of the initialization is then adding events to the event queue, and adding any lines that intersect the initial sweep line to the status. As our status is a red-black tree, inserting an element into the status takes $O(\log n)$ time. Lastly, the event queue is sorted giving us roughly $n \log n + n \log n \in O(n \log n)$

which means initialization has an asymptotic complexity of $O(n \log n)$.

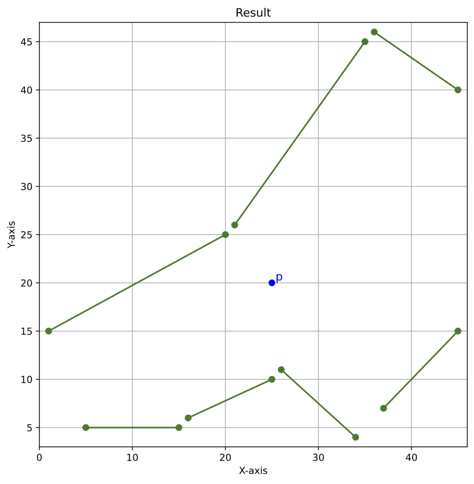
Algorithm: The algorithm itself loops over all event points in the event queue, causing an $2n$ complexity. As we have classified our event points as either “start” or “end” points, the action the algorithm performs is either an insert or remove operation, respectively, on the status. Both of these operations take $\log n$ time to perform, and since the algorithm performs one of these operations on every event point, this raises the complexity to $2n \log n$. After either inserting or removing a line segment from the status, the algorithm checks what the smallest element of the tree is, to check whether a new line segment can be seen, as the tree is ordered by distance to the query point. Finding the smallest line segment of the status is a $\log n$ operation [2], which brings the entire algorithm to a $2n \log n + \log n$ complexity, or in asymptotic complexity: $O(n \log n)$.

thus the overall run time of the algorithm is $O(n \log n) + O(n \log n) \in O(n \log n)$.

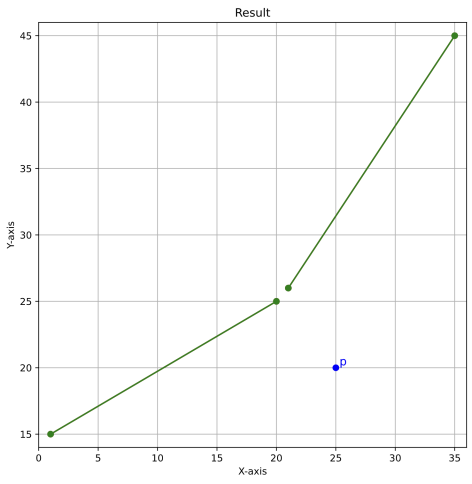
5 Test suite

We have developed a wide range of tests to verify our implementation works correctly both in general position and with special cases.
For testing general position, we have created the test files `t1.txt`, `t2.txt`,

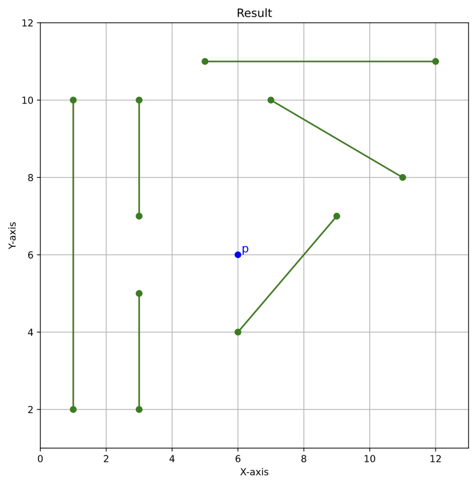
t3.txt and big_test.txt, that contain different amounts of line segments, that are all expected to be seen by the query point. The results of these tests can be seen in Figure 4.



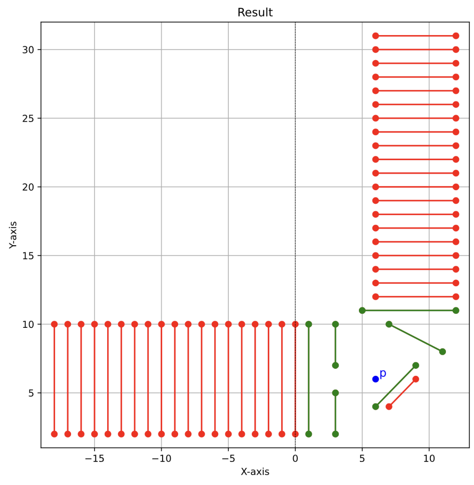
(a) Visual result of `t1.txt`



(b) Visual result of `t2.txt`



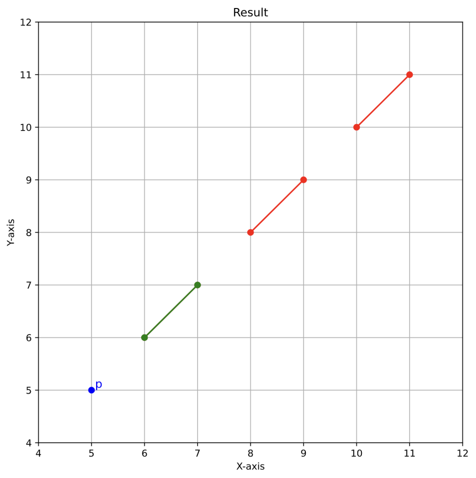
(c) Visual result of `t3.txt`



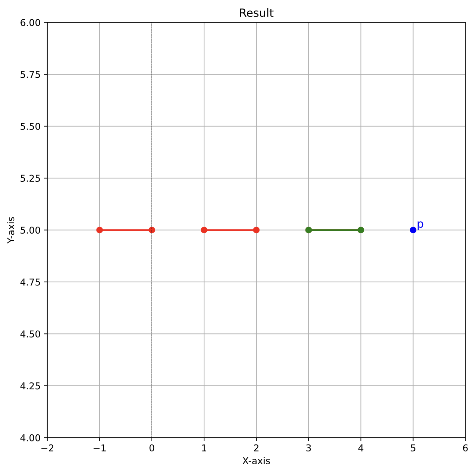
(d) Visual result of `big_test.txt`

Figure 4: General position tests

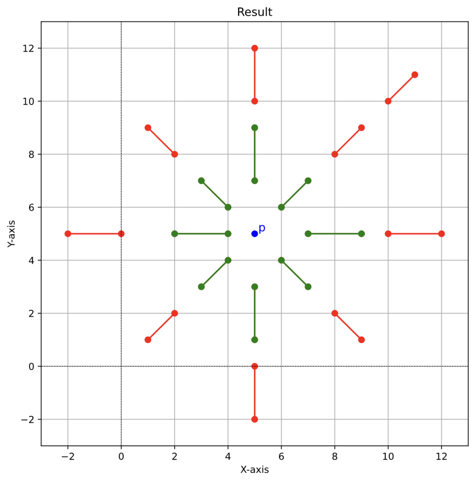
Then, to test for collinearity, we have the test `ColinearLine.txt`, that simply contains some collinear lines, `ColinearStart.txt` which tests for collinear lines that lay on the initial sweepline, and `ManyColinearLines.txt` which tests for several colinear lines throughout a coordinate system. The results of these tests can be seen at Figure 5.



(a) Visual result of `ColinearLine.txt`



(b) Visual result of `ColinearStart.txt`

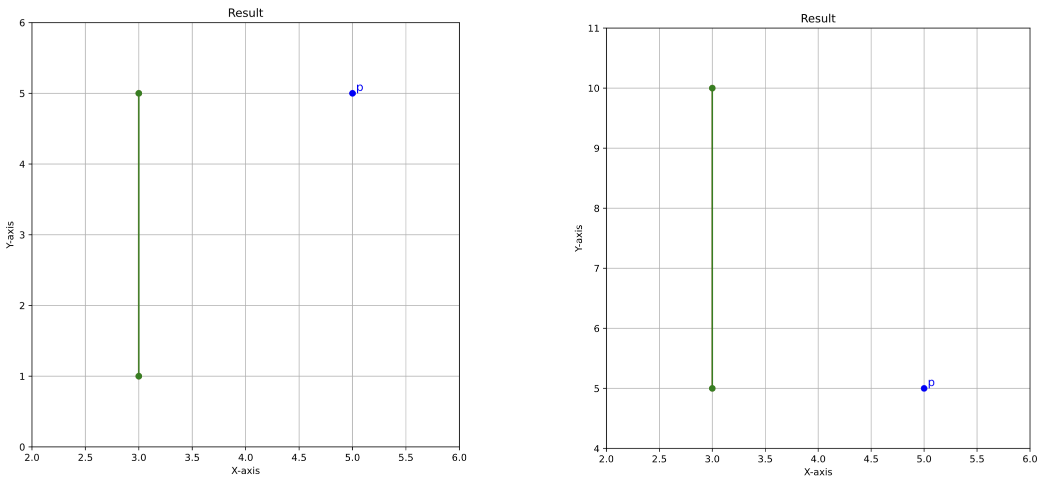


(c) Visual result of `ManyColienarLines.txt`

Figure 5: Collinearity tests

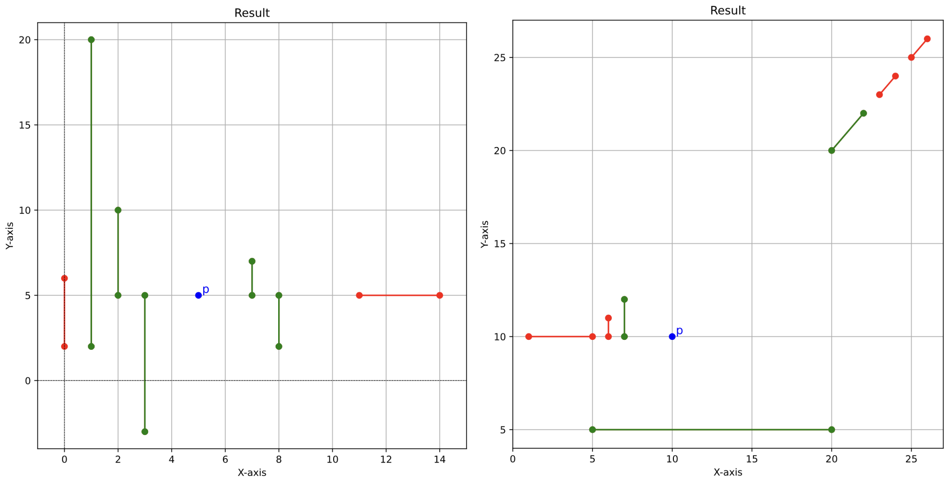
Testing for line segments with a start or end point on the initial sweepline, we have `PointOnSweepLineDownwards.txt` testing for line segments with an end point on the initial sweepline,
`PointOnSweepLineUpwards.txt` testing for line segments with a start point on the initial sweepline, and `PointsOnSweepLine.txt` testing for several cases of both previous tests together with line segments that simply intersect the initial sweepline without having any points on it. Additionally, we have made a test combining the two special cases called `OnSweepLine.txt`, the

results of these four tests can be seen at Figure 6.



(a) Visual result of `PointOnSweepLineDownwards.txt`

(b) Visual result of `PointOnSweepLineUpwards.txt`



(c) Visual result of `PointsOnSweepLine.txt`

(d) Visual result of `OnSweepLine.txt`

Figure 6: Testing for points on the initial sweepline

6 Manual

Note that these instructions work on IMADA's Virtual Computer Lab and of course on your own machine.

Input: The input to the program is a txt file, that contains a single query point and however many line segments the user wishes. The format of the txt file is the following:

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

Where the delimiter between the line segments or query point can be anything except for numbers. This allows the user to write line segments $(x_1, y_1)(x_2, y_2)$ as "$a\ x1\ a\ y1\ ax2\ a\ y2$" or "$(x1, .ay1\{\}\{\{x2y1$". 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 line segment and query point needs to end with a new line character ("$\backslash n$"). Further, the query point can be anywhere in the file. However, there must be one single query point and the line segments must contain two sets of coordinates. Additionally, empty lines in the text file are ignored.

Output: The output is given both textually and visually. It is printed to the terminal to give a textual representation of the result, these are all the line segments that can be seen from the query point. The format of the textual representation is: "$x1\ y1\ x2\ y2\ \backslash n$". The result is also visualized and saved as a PDF. In this visualization, the green lines are the line segments that can be seen from the query point, and the red lines are the line segments that can not. The query point is also shown as a blue dot with the name "p".

Setup: To make it easy to run the program, we have a "`requirements.txt`" containing all the relevant Python packages needed. Further, we also made a "`setup.py`" script that will simply run the bash command: "`Python -m pip install -r requirements.txt`" for Linux \Windows and "`Python3 -m pip install -r requirements.txt`" for macOS. This script can simply be run in the following way:

```
> Python setup.py
```

Note that depending on your OS, this might be `Python3`. Once done, all necessary packages are installed.

Running the program: The program can be run in the two following ways, shown in Figure 7. Option one is to give the path to the data file as a command line argument to the program. The other is that you can run the program, and it will ask the user for the path to the data file. In both cases is the path either absolute or relative to the program.

```
> Python prog.py data.txt
```

```
> Python prog.py
> File name: data.txt
```

Figure 7: The two ways of running our plane sweep algorithm. Note that depending on your operating system, it might be `Python` or `Python3`. Further, the "`data.txt`" is the path to your data file absolute or relative to "`prog.py`".

7 File overview

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

data: Folder containing several .txt files, all of which is a test used to verify that our program performs correctly.

line_segment.py: File containing all our data structures, and a few helper functions used by the data structures.

line_segment_plotter.py: File containing the code responsible for creating the visual output of our program.

prog.py: File responsible for initialising our program and taking the input from the user.

requirements.txt: File containing all the python packages and their exact version used in the project.

setup.py Run this file to easily setup for our project, installing all packages with the correct version needed for our project to run.

sweep_line.py: File containing our implementation of the sweep line algorithm the project concerns.

Bibliography

[1] Mark De Berg. *Computational geometry: algorithms and applications.* Springer Science & Business Media, 2000.

[2] Manfred Moitzi. bintrees: Fast binary and avl tree implementations. `https://pypi.org/project/bintrees/`, 2024. Version 2.2.0, Python package.