University of Southern Denmark

Department of Mathematics and Computer Science

DM819: Computational Geometry

Exam Project, Part 2

Authors

Mikkel Brix Nielsen
mikke21

Steffen Nørgaard Bach
stbac21

December 4, 2024

Contents

1 Introduction

The point location problem involves being able to answer which region of a planar subdivision a query point lies within. In this project, the goal is to efficiently solve the point location problem using trapezoidal maps. Specifically, given a set of n non-adjacent and non-intersecting line segments, where no two coordinates (`x` or `y`) are identical, efficiently construct a trapezoidal map and accompanying search structure in $O(n \text{ log } n)$ expected time. This search structure, once built, should allow for the efficient answering of which trapezoid contains the query point in expected $O(\log n)$ time.

Throughout this report, whenever "the book" is written or referred to, the specific book in mention is "Computational Geometry, Algorithms and Applications", unless otherwise specified.

2 Program Description

This section will briefly describe the content of the different files and folders, as well as their contribution to the functionality of the program.

2.1 Testing

The folder `Tests` contains 21 files that have been used to test different situations and ensure correctness in various scenarios. The tests from `test1.txt` through `test13.txt` concern testing the program in generous position, while the tests from `test4.txt` through `test21.txt` concern format testing of the input file and will result in various errors being printed to `stdout`.

2.2 Location of the source files

The `src` folder contains the main program files necessary for running the program. It includes the core algorithm, utility functions, and data structures used in the program. These include the following: `SS.py`, `main.py`, `algorithm.py`, `utils.py`, `objects.py`, which will be described below.

2.2.1 Representing trapezoids, line segments and points

The file `objects.py` implements an object representation for points, consisting of an `x`- and a `y`-value, line segments, consisting mainly of two points (`start`, `end`), and trapezoids, consisting of two line segments (`upper`, `lower`) and two points (`leftp`, `rightp`) along with various methods to graphically or textually display them.

2.2.2 The search structure

This is implemented in the file `SS.py`, which firstly defines the `Node` class. This class has three attributes, namely `data`, `left`, and `right`, and is implemented with the flexibility to contain any of the three types defined in `objects.py`. Replacement of a node's data can thereby be done with the method `_overwrite()`, which replaces the data and assigns children according to its given parameters.

The implementation of the `SearchStructure` class is responsible for the majority of the actual work that the program does, as this is where the process of inserting a line segment is defined. To achieve this, several auxiliary functions have been defined for functionalities like querying $\mathcal{D}$, creating and merging trapezoids, assigning neighbors, and updating $\mathcal{D}$ accordingly with each insertion.

2.2.3 The implementation of the algorithm

The file `algorithm.py` implements the functions used to extract points and create the corresponding line segments from the input, creating the bounding box, initializing the search structure $\mathcal{D}$, generating a random permutation of the line segments, and then sequentially inserting them into $\mathcal{D}$.

2.2.4 The utility functions

The file `utils.py` implements various methods used in `main.py` to check that the program is called correctly, to read and verify that the user input follows the specified format, convert the input to line segments and a point, run the algorithm, display the trapezoidal map and the query point, write output to a text file, and create and save a graphical representation of the trapezoidal map.

2.2.5 Tying it all together

Finally, the file `main.py` uses the methods defined in `utils.py` to first check that the program is called correctly and that the input is formatted correctly. Then it creates objects for the point and the line segments specified in the input file, and the algorithm is then run with these as input.

3 Correctness Analysis

In this section, the correctness of various aspects of the program will be analyzed, particularly how line segments are inserted. In this section, an "outermost" neighbor refers to a trapezoid that is not crossed by the inserted segment, but which is a neighbor to one of the trapezoids that are, and is thus relevant when assigning neighbors for the trapezoids created during the insertion of a line segment. A line segment intersecting a trapezoid splits it into two new trapezoids; one above the segment and one below the segment, which will be referred to as `above` and `below`, respectively.

3.1 Finding Trapezoid(s)

The initial step of inserting a line segment into the search structure $\mathcal{D}$ is to retrieve a list of all directly affected trapezoids in $\mathcal{T}$, which is done by `_find_intersected_trapezoids()`.

The first trapezoid Δ_0 is found by searching $\mathcal{D}$ based on the start point, p_i, of a given line segment s_i. If compared to a point, navigate by `x`-value of p_i, if compared to a line segment, navigate by its relative position to p_i. When a node containing a trapezoid is encountered, the traversal stops and the node is returned.

If the segment s_i continues into a neighboring trapezoid, which is determined by whether its endpoint q_i is further right than the right-defining point of the current trapezoid, the remaining trapezoids are found in $\mathcal{T}$ by sequentially navigating to the next right-neighboring trapezoid with respect to whether s_i enters the upper or lower right-neighbor, until q_i is no longer to the right of the next trapezoid's left-defining point, i.e. Δ_k is found. Thus, the returned list of trapezoid-nodes is $\{\Delta_0, \Delta_1, ..., \Delta_k\}$, corresponding to the trapezoid-nodes which s_i intersects.

3.2 Building Trapezoids and Maintaining Neighbors ($\mathcal{T}$)

The structure of $\mathcal{T}$ is built and maintained by assigning neighbor-relations to each trapezoid being created with each insertion of a line segment. The currently established neighbor-relations must therefore also be updated when a node containing a trapezoid is changed to contain something else. The purpose of maintaining $\mathcal{T}$ is to determine which trapezoids are affected by the insertion of a line segment, and make the appropriate changes to $\mathcal{D}$.

3.2.1 Case 1

When a line segment is inserted within a single trapezoid, four new trapezoids are created. These four new trapezoids consists of one to the left, one to the right, one above, and one below s_i, namely `A`, `B`, `C`, and `D`. This follows the example seen on figure 1.

The trapezoids `C` and `D` both have the left- and right-defining point being the start- and endpoint of s_i. They inherit the top- and bottom-defining line segments from Δ_0 accordingly, and they both get `A` as left- and `B` as right-neighbor.

The trapezoid `A` is created with p_i as right-defining point and inherits the left-defining point, the left-neighbors, the upper-, and the lower-defining line segments of Δ_0, and its right-neighbors are assigned as `C` and `D`. The left-neighbors of Δ_0 are then updated to reflect the change made to Δ_0, and their respective right-neighbor corresponding to Δ_0 is reassigned to `A`. Vice versa for `B` and the right-neighbors of Δ_0.

3.2.2 Case 2+

When inserting a line segment that is not contained within a single trapezoid, the directly affected trapezoids $\Delta_0, \Delta_1, ..., \Delta_k$ found and returned by `_find_intersected_trapezoids()` are handled first by creating trapezoids, defining their dimensions, and assigning their neighbors within Δ_0 and Δ_k, respectively. The neighbor-relations for the external neighbors for Δ_0 and Δ_k are also updated accordingly.

In this case, `A`, `B`, and `C` are the three trapezoids created within Δ_0 around p_i, where `A` is to the left of p_i, `B` is above s_i and right of p_i, and `C` is below s_i and right of p_i.

A key difference in this case is the fact that Δ_0 can have two right-neighbors, but only one of those are included as one of the trapezoids from $\Delta_1, ..., \Delta_{k-1}$. Vice versa for Δ_k. The left-neighbor for the outermost right-neighboring trapezoid of Δ_0 is therefore reassigned to either `B` or `C` depending on whether the right-defining point of Δ_0 is above or below s_i, respectively. Similarly, either `B` or `C` gets the outermost trapezoid assigned as its right-neighbor. The same thing is done vice versa when creating trapezoids within Δ_k around q_i. An example of this can be seen on figure 9 where `R9` gets `R3` as its left-neighbor and `R3` gets `R9` as its right-neighbor.

After this, each trapezoid $\Delta_i \in \{\Delta_1, ..., \Delta_{k-1}\}$ is sequentially split into two new trapezoids, namely `above` and `below`. At each split of a Δ_i, if it has two neighbors at either side, the neighbor relation between the outermost trapezoid to the left and to the right of Δ_i and the trapezoids created within Δ_i above and below s_i is assigned in both direction with respect to whether the right- and left-defining point of Δ_i is above or below s_i, respectively.

This mutual assignment happens every time a trapezoid above or below s_i is NOT merged, and this case is always flipped. Hence, when merging above the inserted segment, neighbor relations are assigned between the trapezoid created below s_i and the outermost neighbors of Δ_i, namely its lower-left- and lower-right-neighbors. Vice versa when merging below the inserted segment.

Deciding whether to merge the `above` or `below` trapezoid of Δ_{i-1} with the `above` or `below` trapezoid of Δ_i is merely a matter of whether they share the same upper- or lower-defining line segment, respectively. When merging happens, the trapezoid to the left extends through the trapezoid to the right by inheriting its right-defining point and its right-neighbors. When merging does not happen, neighbor relations between the two trapezoids are assigned while keeping the possible preexisting neighbor-relation and ordering them accordingly with respect to whether these two trapezoids are above or below s_i. An example of how this happens can be seen in section 5.2.

3.3 Building and Maintaining the Search Structure ($\mathcal{D}$)

Construction and maintenance of $\mathcal{D}$ is done by overwriting the data of nodes and assigning their children with respect to where in $\mathcal{T}$ and under which circumstance a line segment is inserted.

The Search Structure $\mathcal{D}$ starts with the bounding box as root, and every insertion of a line segment at this point onward will fall into one of the two cases described previously. The construction of $\mathcal{D}$ follows the way that has been presented and described in the book. The purpose of $\mathcal{D}$ is to maintain a structure that correctly lets the algorithm navigate through points and line segments to ultimately find the trapezoid that contains the queried point in $\mathcal{T}$.

The neighbor-relations between trapezoids is completely irrelevant to $\mathcal{D}$, so, in this subsection, we are only interested in nodes, their children and what data (`Point`, `LineSegment`, `Trapezoid`) it contains.

3.3.1 Case 1

As described in the previous subsection, when the inserted line segment is contained within a single trapezoid, the four new trapezoids `A`, `B`, `C`, and `D` are created. Since the segment is known in the `insert()` method, the node containing the trapezoid, which has now been split into four, can easily be overwritten to contain the sub-graph induced by the new relationship, which in turn updates the structure of $\mathcal{D}$.

In particular, when a segment is inserted into a single trapezoid, the node containing the trapezoid is overwritten to contain the point p_i. This node gets `A` as left-child and a new node containing q_i as right-child. The node for q_i gets a new node containing s_i as left-child and `B` as right-child. The s_i node gets `C` as left-child and `D` as right-child. The search structure $\mathcal{D}$ now contains the additional sub-graph, a textual representation of which can be seen on figure 3, where “root” replaces the node of the trapezoid wherein this insertion was made.

3.3.2 Case 2+

When a line segment intersects multiple trapezoids, the algorithm incrementally splits, fixes neighbor relations, and overwrites $\mathcal{D}$ as it progressively handles each intersected trapezoid.

The first update is done by an overwrite performed after creating the three new trapezoids in both Δ_0 and Δ_k that yields A_{Δ_0}, B_{Δ_0}, and C_{Δ_0} as the three trapezoids around

p_i within Δ_0, as well as A_{Δ_k}, B_{Δ_k}, and C_{Δ_k} as the three trapezoids around q_i within Δ_k. Please note that the `A`'s are the trapezoids to the left/right of p_i/q_i, and `B`'s and `C`'s are the new trapezoids created above and below s_i at either end, respectively.

This overwrite is made to the node containing Δ_0, replacing its data with p_i, assigning its left-child as A_{Δ_0} and its right-child as a node with s_i as data, left-child as B_{Δ_0}, and right-child as C_{Δ_0}.

The next sequence of updates to $\mathcal{D}$ is made with an overwrite for each of the trapezoids $\Delta_i \in \{\Delta_1, ..., \Delta_{k-1}\}$. This happens after each merge attempt for `B` with `above` and `C` with `below` by replacing the data of the Δ_i node with s_i, and assigning its children to either `B` and `below` or to `above` and `C` as left- and right-child depending on whether the merge occurred above or below the inserted segment within that trapezoid, respectively.

The last update to $\mathcal{D}$ is made with an overwrite to Δ_k's node by replacing its data with q_i, assigning the right-child to A_{Δ_k}, and the left-child to a new node with s_i as data, B_{Δ_k} as its left-child, and C_{Δ_k} as its right-child.

4 Complexity analysis

The `main.py` file calls a few methods in `utils.py`, including `check_usage` which takes constant time, as well as `get_content`, `format_content`, and `create_lines_and_point`, each of which take linear time based on the size of the input, and lastly `run_algorithm`, which is quite a bit more intricate.

4.1 The method to run the algorithm

The method `run_algorithm` first call the method `build_TM_and_SS`, which implements construction and maintenance of both the search structure, $\mathcal{D}$ and the trapezoidal map, $\mathcal{T}$ and runs in $O_E(n \log n)$ time. After having built $\mathcal{T}$ and $\mathcal{D}$ it queries $\mathcal{D}$ for the location of the given query point, which takes $O_E(\log\ n)$ time. Then, if the relevant flags are set it creates a plot in $O(n)$ time that shows which trapezoid the query point lies within or writes the textual representation of the trapezoid, which contains the query point, to `output.txt` in $O(1)$ time. Optionally, a textual representation of $\mathcal{D}$ can be added to `output.txt` and an image of $\mathcal{T}$ can be created and saved as `TrapezoidalMapPlot.png`. This, however, increases the time complexity of reporting to $O_E(n)$, since the creation of $\mathcal{D}$'s string representation takes linear time in its size, and the generation of the image of $\mathcal{T}$ also takes linear time in its size. Hence, the time required to construct the necessary structures and report the trapezoid wherein a query point is located is $O_E(n \log n)$.

4.2 The method `build_TM_and_SS`

This method runs in $O_E(n \log n)$ time and starts by initializing an instance of the search structure with a bounding box that is able to contain the query point and all the line segments. This structure is used to represent and maintain the structure of the digraph representing the search structure as well as the trapezoidal map, which takes constant time to initialize once the bounding box has been created, hence is $O(1)$. A random permutation of the line segments is generated, and each one of these are then sequentially inserted into $\mathcal{D}$, while maintaining its and $\mathcal{T}$'s structure. After this process, it returns the final trapezoidal map, which takes $O_E(n)$ time to extract from $\mathcal{D}$. The specific methods responsible for performing these tasks and the their respective time complexities will be described in the following sections.

4.3 Creating the bounding box

This method starts by extracting all $m = 2n$ points from the n line segments. It then iterates through each of them to find the minimum and maximum x- and y-values, which take time $O(m)$. These are then compared with the x- and y-value of the query point, $O(1)$, to ensure that it lies within the bounding box as well. This is done by making the bounding box a trapezoid that is slightly larger than a box defined by the minimum and maximum found x- and y-values, $O(1)$. Thus, the time complexity of creating the bounding box is of order $O(m) = O(2n) \in O(n)$.

4.4 Generating a random permutation of the line segments

This method uses Python's built-in `random.sample` method, which takes $O(m \log n)$ time, where m is the number of elements to choose and n is the number of choices, so in this case where a random permutation of the entire list of the n line segments is needed, the complexity becomes $O(n \log n)$.

4.5 Processing the line segments

This step of the algorithm is exclusively handled by the insert methods defined in `SS.py`, which starts by querying the search structure $\mathcal{D}$ for the trapezoid that contains the start point p_i of the line segment s_i. This takes takes $O_E(\log n)$. This trapezoid is then used to find all other trapezoids that are intersected by s_i, a process which takes time $O_E(\log n)$ and is expected to yield a list of trapezoids of length log n.

For the case where a line segment intersects multiple trapezoids, the work done for each of these trapezoid is constant and will involve handling the first and last trapezoids, splitting and merging the intermediate trapezoids along the segment, and overwriting nodes in $\mathcal{D}$ to maintain its structure.

If a line segment is fully contained within a trapezoid, the work done for this trapezoid involves splitting it into four, updating its neighbor's relations to the new trapezoids, and overwriting the the corresponding node to maintain the structure of $\mathcal{D}$.

4.6 Line segment intersects multiple trapezoids

When a line segment intersects multiple trapezoids the first trapezoid, Δ_0, and the last trapezoid, Δ_k, are each split into three, which requires defining three new trapezoids based on the one being split and the line segment, which takes constant time for both, $O(1)$. The neighbors of Δ_0 and Δ_k are then updated to reflect their relation to the new trapezoids that Δ_0 and Δ_k were split into, respectively. This also takes constant time, $O(1)$.

Handling the intermediate trapezoids, $\{\Delta_1, \Delta_2, ..., \Delta_{k-1}\}$, means splitting every Δ_i into an `above` and a `below` trapezoid, and each require a constant time check to see if they can be merged with their respective left-neighboring trapezoid for both of the new trapezoids. If it is possible to merge them, some constant time operations will merge them, update neighbors, and overwrite Δ_i's node's data with s_i in $\mathcal{D}$, maintaining $\mathcal{D}$'s structure. Otherwise, if they cannot be merged, their neighbors are updated in constant time. This constant time work $O(1)$ is done for all of the $O_E(\log n)$ trapezoids, so the time it takes to insert a line segment that intersects multiple trapezoids is of order $O_E(\log n)$.

4.7 Line segment contained within single trapezoids

When a line segment, s_i, is contained entirely within a single trapezoid, Δ_i, it is very easy to process as it is split into four parts based on Δ_i's `upper`, `lower`, `leftp` and `rightp`. These four trapezoids have their internal neighbor relations set. Thereafter, the external neighbors are updated, which takes constant time. After having split Δ_i and updated its neighbors, $\mathcal{D}$ is updated to reflect this change by overwriting the data of Δ_i's node with s_i, which also takes constant time. Thus, it takes time $O(1)$ to process a line segment that is entirely contained within a trapezoid after having found said trapezoid.

4.8 Querying the search structure

When querying the search structure the same method, as when finding what trapezoid contains the starting point of a line segment, is used. The query time for this method follows directly from the correctness analysis, which compares this implementation to that of the book. Thus, by Theorem 6.3, the time it takes to query $\mathcal{D}$ for the trapezoid containing a query point q is $O_E(\log n)$. The search structure $\mathcal{D}$ follows from the same theorem, and is $O_E(n)$.

4.9 Extracting the trapezoidal map from the search structure

To extract the trapezoidal map, $\mathcal{T}$, from the search structure, $\mathcal{D}$ a linear time tree-like traversal in the size of $\mathcal{D}$ is performed where only the "leaves" are collected (nodes without left- and right-children). Since $\mathcal{D}$ is a digraph, then in order to not collect the same trapezoid multiple times, an attribute on each trapezoid called `collected` is set to `True` the first time it is visited before collecting it. Afterward, if a trapezoid that has already been collected is visited again through some different path, it is not collected again. Thus is takes time $O_E(n)$ to extract $\mathcal{T}$ from $\mathcal{D}$. The Size Complexity of the trapezoidal map $\mathcal{T}$ follows from Lemma 6.2, which states that a trapezoidal map contains at most $3n + 1$ trapezoids in general position. Since any trapezoid can have at most 2 neighbors to either side (4 total), then the total number of assigned neighbor relation cannot exceed $4(3n+1)$, which is $O(n)$.

4.10 Overall algorithmic complexity

To summarize; before running the algorithm, some constant time methods are executed, followed by the creation of the bounding box, and search structure initialization, which takes linear time $O(n)$. A random permutation of the line segments is generated in time $O(n \log n)$. Processing all line segments takes time $O_E(n \log n)$. Extracting the trapezoidal map takes $O_E(n)$. Querying the search structure takes time $O_E(\log n)$. Reporting takes expected linear time to generate output (using flag `-o`), constant time to only return query result (using flag `-qo`), or linear time to produce the plot (using flag `-p`). Thereby, it takes time $O_E(n \log n)$ for the algorithm to construct the search structure, construct the trapezoidal map and query the search structure for the specified query point.

5 Testing

This section outlines which types of tests were performed to verify the functionality and correctness of the program, to ensure it behaves as expected under various conditions, with respect to generous position. The following subsections provide an overview of a few examples of the tests performed and their corresponding outcomes.

5.1 The new segment s_i lies completely in trapezoid Δ

On figure 1, the insertion of a line segment fully contained within a single trapezoid is illustrated with its effect on $\mathcal{T}$ as well as on $\mathcal{D}$. A similar situation occurs when prompting our program with the example provided in `Tests/test1.txt`, which yields the plot on figure 2. A textual representation of the structure of $\mathcal{D}$ and the neighbor relations of $\mathcal{T}$ can be found in appendix A.

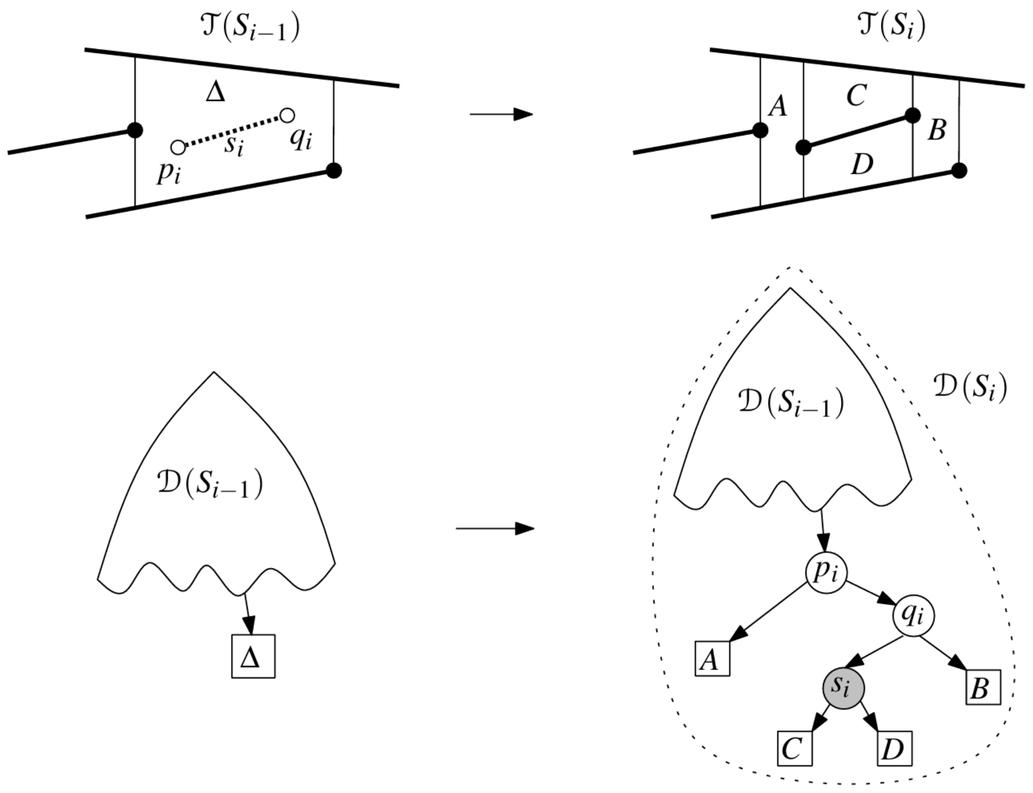


Figure 1: Figure 6.7 on page 131 in the book "Computational Geometry, Algorithms and Applications"

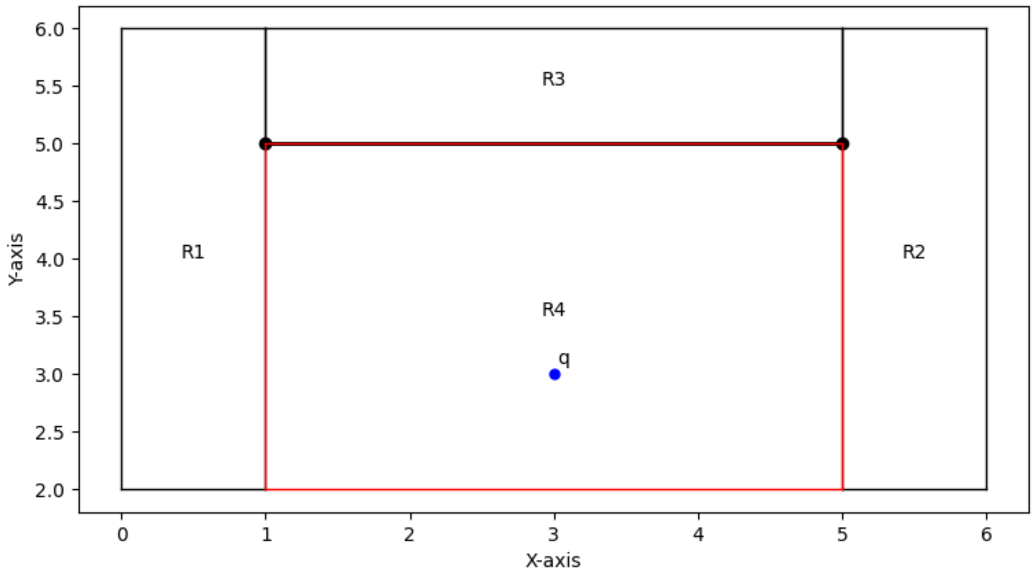


Figure 2: Plot resulting from running `Tests/test1`

```
Root: Point(x: 1.0, y: 5.0):
    L: Trapezoid:
        label: R1
        rightN: ['R3', 'R4']
    R: Point(x: 5.0, y: 5.0):
        L: LineSegment(from: Point(x: 1.0, y: 5.0), to: Point(x: 5.0, y: 5.0)):
            L: Trapezoid:
                label: R3
                leftN: ['R1']
                rightN: ['R2']
            R: Trapezoid:
                label: R4
                leftN: ['R1']
                rightN: ['R2']
        R: Trapezoid:
            label: R2
            leftN: ['R3', 'R4']
```

Figure 3: Abbreviated version of `output.txt` from running `Tests/test1.txt`

As seen on figure 3, the root is the point in $(1, 5)$, which is the start point of the inserted line segment and corresponds to p_i on figure 1.

The left-child of point $(1, 5)$ is `R1`, which is a trapezoid with `R3` and `R4` as right-neighbors, respectively corresponding to `A`, `C`, and `D`, on figure 1. Its right-child is the point in $(5, 5)$, corresponding to q_i.

The left-child of point $(5, 5)$ is the line segment from $(1, 5)$ to $(5, 5)$, corresponding to s_i. This line segment has `R3` as left- and `R4` as right-child, both of which trapezoids with `R1` as left- and `R2` as right-neighbor. Its right-child is `R2`, which is a trapezoid with `R3` and `R4` as left-neighbors, corresponding to `B` on figure 1.

5.2 Line segment intersects at least three trapezoids

Considering the two following figures 4 and 6; during the insertion of the final line segment from $(1, 2)$ to $(15, 2)$, `R1` and `R8` are both split into three new trapezoids each. Thus, `R1` is replaced by `R21`, `R22`, and `R23`, then `R8` is replaced by `R24`, `R25`, and `R26`. A split and a merge attempt is then made with the two new trapezoids created above and below the segment to the left of the current trapezoid for each of the trapezoids crossed by the line segment. So, `R4` is split into `R27` above and `R28` below, followed by `R23` merging into `R28`. The trapezoid `R13` is then split into `R29` and `R30`, with `R23` then merging into `R30`. Similarly, `R15` is split into `R31` and `R32`, and `R29` then merges into `R31`. This procedure continues until the trapezoid created in `R8` are reached, at which point a final merge attempt is made.

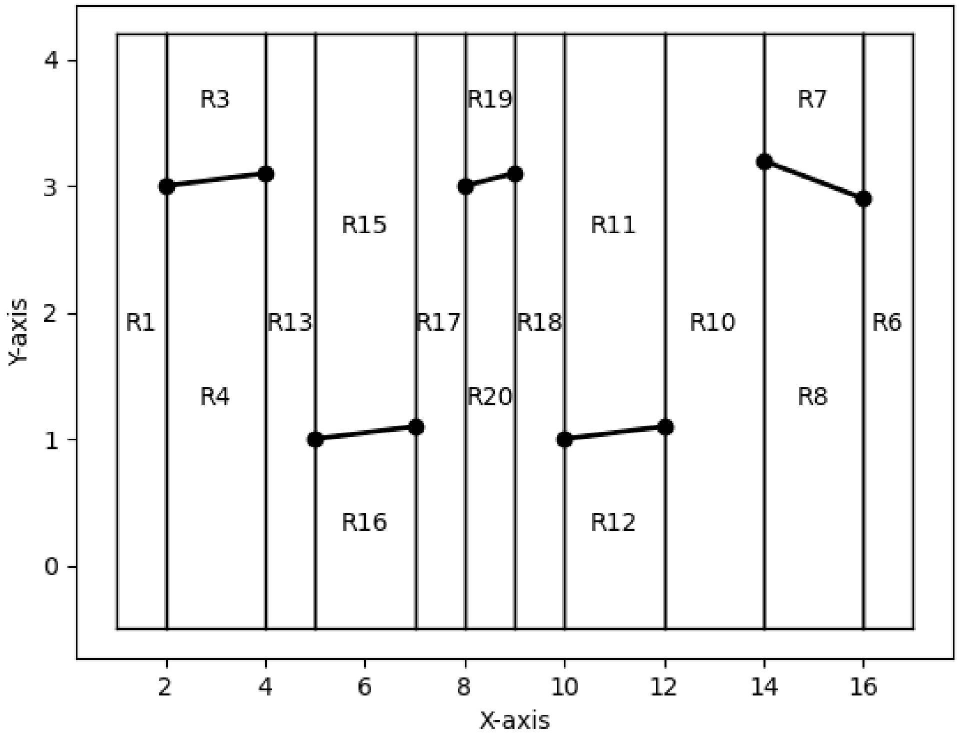


Figure 4: Plot resulting from running `Tests/test10.txt`, but prior to insertion of last segment from $(1, 2)$ to $(15, 2)$

To verify the correctness of neighbor assignments during this process, it is sufficient to assert whether `R29` and `R34` have the correct neighbors to both sides. Inspecting the (reduced version figure 5) output yielded from running this test shows the following for those two trapezoids:

```
Root: Point(x: 2.0, y: 3.0):
   R: |--->Point(x: 4.0, y: 3.1):
      R: |--->Point(x: 14.0, y: 3.2):
         L: |--->Point(x: 10.0, y: 1.0):
            L: |--->Point(x: 5.0, y: 1.0):
               L: |--->LineSegment((x: 1.0, y: 2.0), (x: 15.0, y: 2.0)):
                  L: |--->Trapezoid R29:
                         leftN: ['R3', 'R27']
                         rightN: ['R19', 'R35']
               R: |--->Point(x: 7.0, y: 1.1)
                  R: |--->Point(x: 8.0, y: 3.0)
                     L: |--->LineSegment((x: 1.0, y: 2.0), (x: 15.0, y: 2.0))
                        R: |--->Trapezoid R34:
                               leftN: ['R32', 'R16']
                               rightN: ['R40', 'R12']
```

Figure 5: Reduced version of output produced by `Tests/test10.txt`

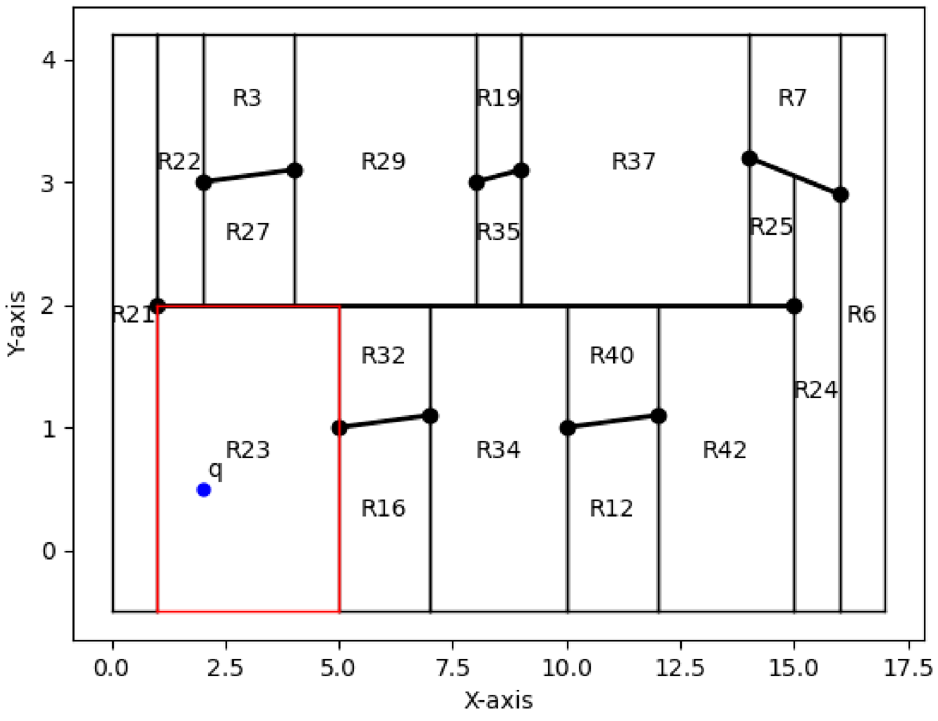


Figure 6: Plot resulting from running `Tests/test10.txt`

5.3 Line segment intersects two trapezoids

When a line segment only intersects two trapezoids, no intermediate merging will take place. Here, when inserting the second line segment it intersects `R4` and `R2`, shown on figure 7, both of which are split into three new trapezoids each, resulting in the creation of `R5`, `R6`, `R7`, `R8`, `R9` and `R10`, as shown on figure 8. Merging `R7` and `R10` then results in the trapezoidal map shown on figure 9, for which a textual representation can be found in appendix C. To verify that the line segment has been inserted correctly, inspecting the neighbor lists of the various trapezoids in appendix C reveals that `R1`'s lower right-neighbor has become `R5`, `R5`'s left-neighbor has become `R1`, `R3`'s right-neighbor has been updated to `R9`, `R9`'s upper-left-neighbor is `R3` and its lower neighbor is `R6`, and `R6` also has `R9` as its right-neighbor. Lastly, merging `R7` over `R10` updates `R8`'s left-lower neighbor to `R7`, which matches the trapezoids visual location in figure 9.

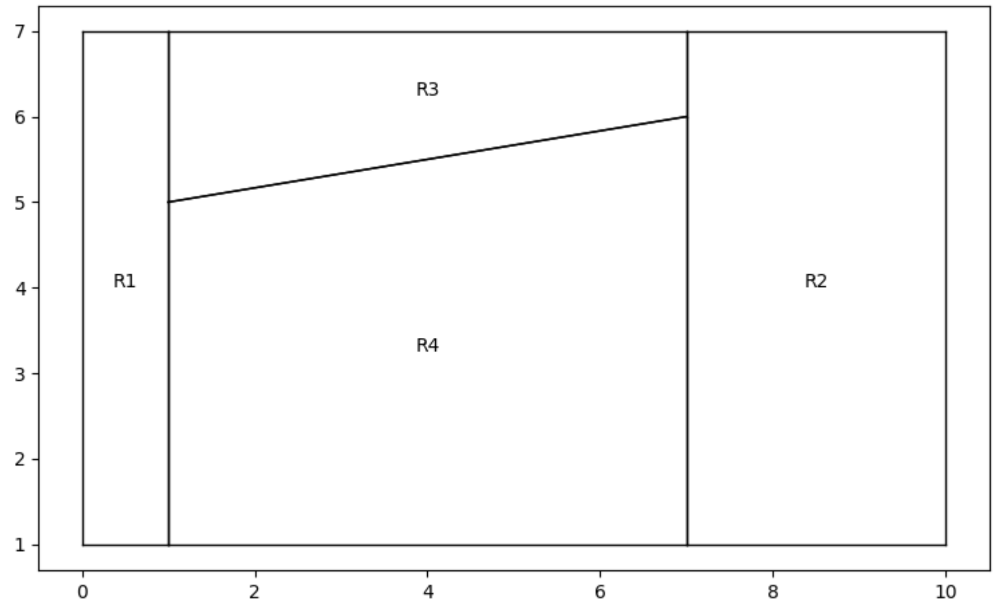


Figure 7: Trapezoidal map after inserting the first line segment from `Tests/test2.txt`

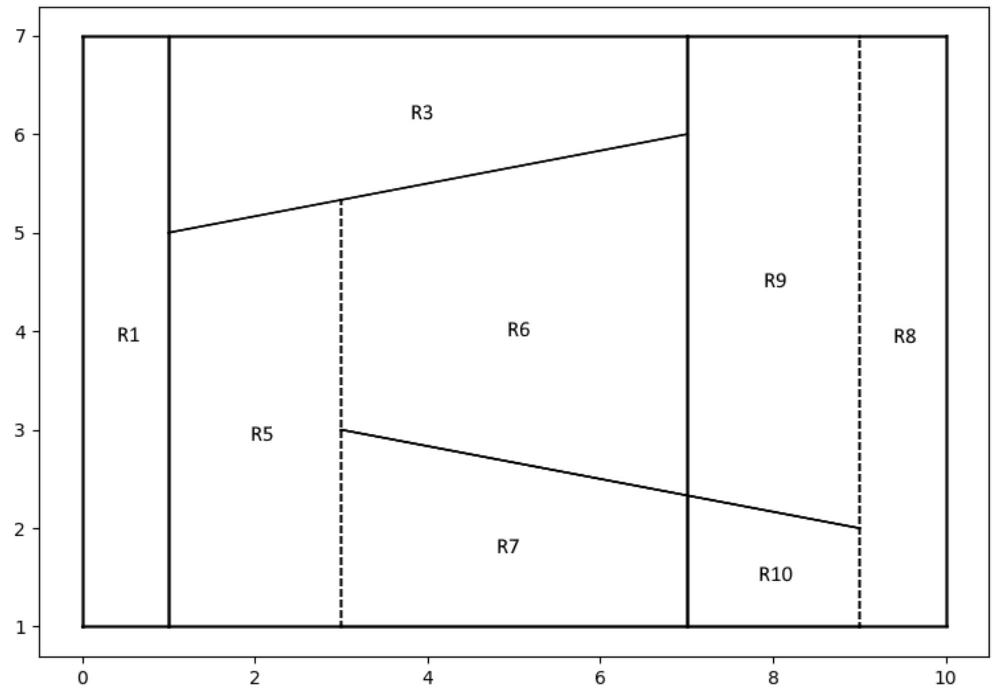


Figure 8: Trapezoidal when splitting `R4` and `R2` `Tests/test2.txt`

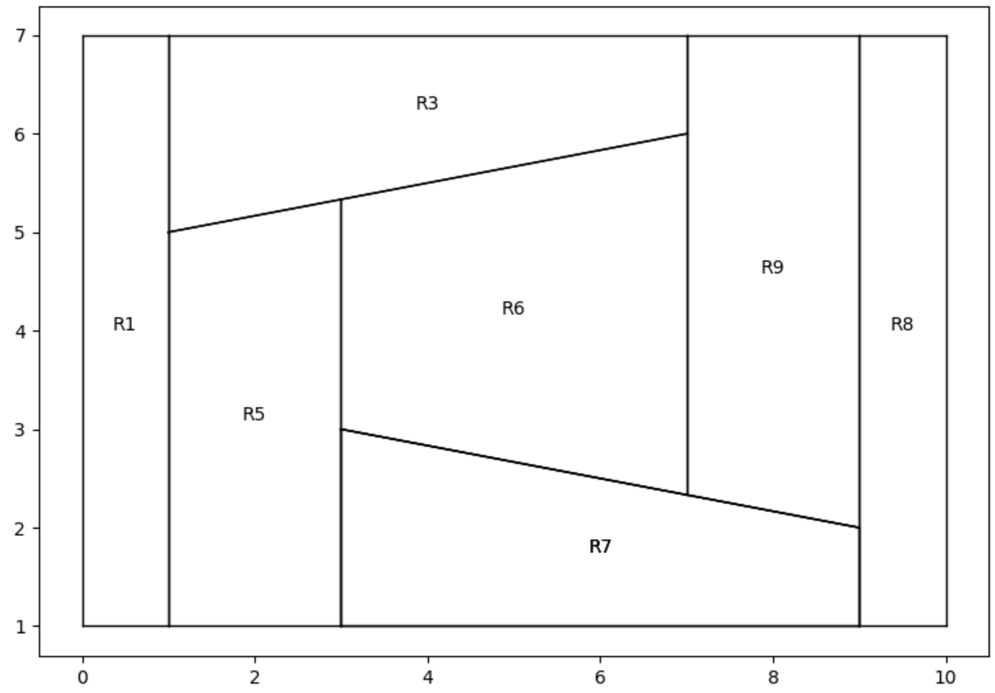


Figure 9: Trapezoidal map after inserting the second line segment from `Tests/test2.txt`

6 Manual

This section will briefly describe and cover the format for valid input and output files, how to run the program, get a textual output in the file `output.txt` along with a visual representation of the trapezoidal map in `TrapezoidalMapPlot.png`, and how to display the output graphically. A more detailed explanation of what will be covered here can be found in the provided `README.md` file or in the appendix where a PDF version of the full `README.md` is also located.

6.1 Input Format

For input data, a line segment going from `(x1, y1)` to `(x2, y2)` is represented as `x1 y1 x2 y2` terminated by one newline. Similarly, a point is represented by using two coordinates instead of four and there must be exactly one.

A test file should have the point specified on the first line followed by a sequence of disjoint line segments, where no two coordinate's `x`- or `y`-values are identical, on the subsequent lines. Positive and negative integer and float coordinates are allowed. The sign, however, is required to be immediately in front of the number.

Additionally, before `x1`, in between two coordinates, and after `y2` (but before the newline), any number of spaces or tabs are allowed.

An example of what a valid input file could look like is the following:

```
2.5 0.5
2 2 17 2
14 3.2 16 2.9
15 6 18 5
```

6.2 Output Format

The files `output.txt` and `TrapezoidalMapPlot.png` will be created in the current directory from which you run the program. E.g., If you run the program using `python ./src/main.py -o ./Tests/test1.txt` command, in the project folder, both files will be placed alongside the `README.md` and `requirements.txt` files in the project folder. If you navigate to the `src` folder and run the program from here using the same or a similar command, both files will be placed in the `src` folder. Similarly, if the flag `-qo` is used instead only the output file will be generated but its location will be determined exactly as previously described.

The format of `output.txt`, when using the `-o` flag will contain exactly one sentence describing which trapezoid contains the query point along with a textual representation of the search structure. Using, instead, the `-qo` flag will result in only the sentence describing the query points location and the trapezoid containing it in `output.txt`. An example of what `output.txt` could look like can be found in appendix A, when `-o` is used and B when `-qo` is used.

6.3 Running the code

To run the code, ensure first that you are at the root of the project folder. I.e., you should be able to see the following folders and files: `src`, `Images`, `Tests`, `README.md`, `requirements.txt`, and `report.pdf`. Ensure then that you have at least Python 3.8.10 and pip 20.0.2 installed. This can be done by using the commands `python -version`, which should output 3.8.10 or higher, and `pip -version`, which should output 20.0.2 or higher.

Then create a virtual environment using the command `python -m venv myvenv` replacing `myvenv` with your preferred name. To activate the virtual environment use the command `./myvenv/Scripts/activate` for Windows machines or the command `source myvenv/bin /activate` on MacOS and Linux machines. Then to install the required packages use the command `pip install -r requirements.txt`, which works for both Windows, MacOS, and Linux machines.

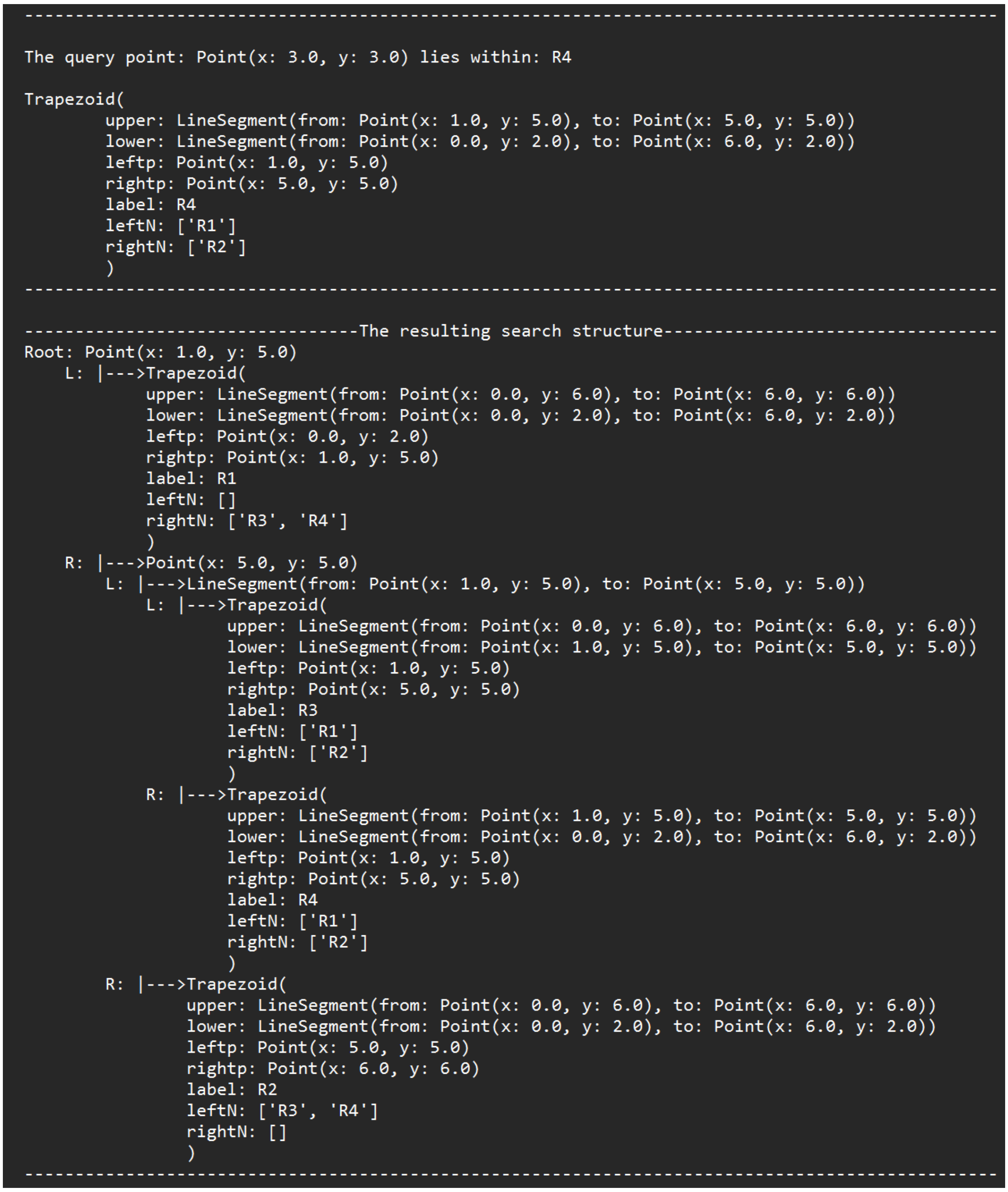
Then, to run the program, replace `<path_to_file>` with either the path to one of your test files or the path to one of the provided ones in `Test`, e.g., `./Test/test1.txt`, and use the command `python ./src/main.py [ optional -d -p -o -qo] <path_to_file>`, where `-d` enables debug mode, `-p` enables graphical output, `-o` creates the file `output.txt` and `TrapezoidalMapPlot.png`, and `-qo` only creates `output.txt`, where the programs output will be written as described here 6.2.

7 Conclusion

The program functions correctly with respect to the tests in the test suite, all of which produce a consistent and correct search structure and trapezoidal map based on the given line segments, regardless of permutation. To ensure that they correspond to what is presented in the book (disregarding its allowance of degenerate cases), the construction and maintenance of both the search structure and the trapezoidal map have been carefully tested, inspected, and verified in various scenarios with several different permutations for each. After verifying correctness, it was ensured that the program met the required performance standards, specifically by confirming that no part of the program exceeds the time complexity of $O_E(n \log n)$, nor that the size complexity exceed $O_E(n)$, and that the query time for a point in the map is $O_E(\log n)$.

8 Appendix

A Example output Tests/test1.txt with `-o` flag.



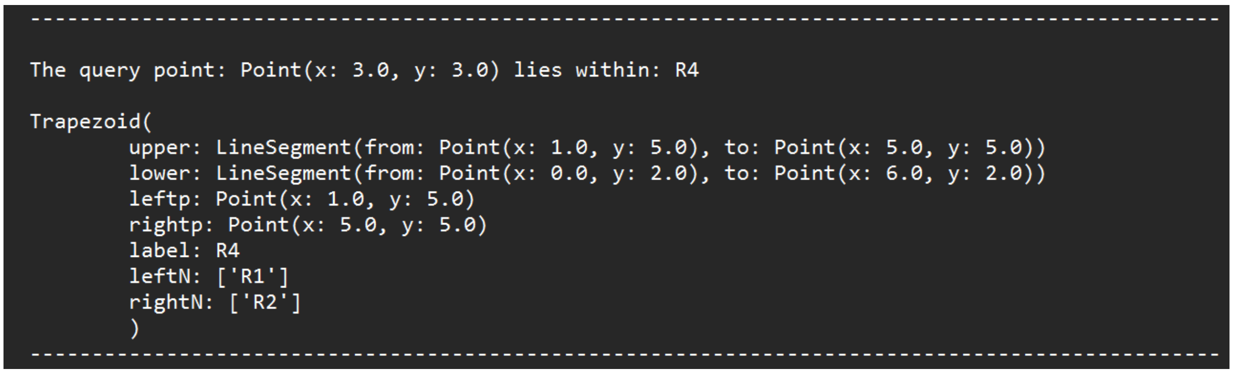
```
-----------------------------------------------------------------------------------------------

The query point: Point(x: 3.0, y: 3.0) lies within: R4

Trapezoid(
        upper: LineSegment(from: Point(x: 1.0, y: 5.0), to: Point(x: 5.0, y: 5.0))
        lower: LineSegment(from: Point(x: 0.0, y: 2.0), to: Point(x: 6.0, y: 2.0))
        leftp: Point(x: 1.0, y: 5.0)
        rightp: Point(x: 5.0, y: 5.0)
        label: R4
        leftN: ['R1']
        rightN: ['R2']
        )
-----------------------------------------------------------------------------------------------

---------------------------------The resulting search structure--------------------------------
Root: Point(x: 1.0, y: 5.0)
    L: |--->Trapezoid(
            upper: LineSegment(from: Point(x: 0.0, y: 6.0), to: Point(x: 6.0, y: 6.0))
            lower: LineSegment(from: Point(x: 0.0, y: 2.0), to: Point(x: 6.0, y: 2.0))
            leftp: Point(x: 0.0, y: 2.0)
            rightp: Point(x: 1.0, y: 5.0)
            label: R1
            leftN: []
            rightN: ['R3', 'R4']
            )
    R: |--->Point(x: 5.0, y: 5.0)
        L: |--->LineSegment(from: Point(x: 1.0, y: 5.0), to: Point(x: 5.0, y: 5.0))
            L: |--->Trapezoid(
                    upper: LineSegment(from: Point(x: 0.0, y: 6.0), to: Point(x: 6.0, y: 6.0))
                    lower: LineSegment(from: Point(x: 1.0, y: 5.0), to: Point(x: 5.0, y: 5.0))
                    leftp: Point(x: 1.0, y: 5.0)
                    rightp: Point(x: 5.0, y: 5.0)
                    label: R3
                    leftN: ['R1']
                    rightN: ['R2']
                    )
            R: |--->Trapezoid(
                    upper: LineSegment(from: Point(x: 1.0, y: 5.0), to: Point(x: 5.0, y: 5.0))
                    lower: LineSegment(from: Point(x: 0.0, y: 2.0), to: Point(x: 6.0, y: 2.0))
                    leftp: Point(x: 1.0, y: 5.0)
                    rightp: Point(x: 5.0, y: 5.0)
                    label: R4
                    leftN: ['R1']
                    rightN: ['R2']
                    )
        R: |--->Trapezoid(
                upper: LineSegment(from: Point(x: 0.0, y: 6.0), to: Point(x: 6.0, y: 6.0))
                lower: LineSegment(from: Point(x: 0.0, y: 2.0), to: Point(x: 6.0, y: 2.0))
                leftp: Point(x: 5.0, y: 5.0)
                rightp: Point(x: 6.0, y: 6.0)
                label: R2
                leftN: ['R3', 'R4']
                rightN: []
                )
-----------------------------------------------------------------------------------------------
```

B Example output Tests/test1.txt. with `-qo` flag

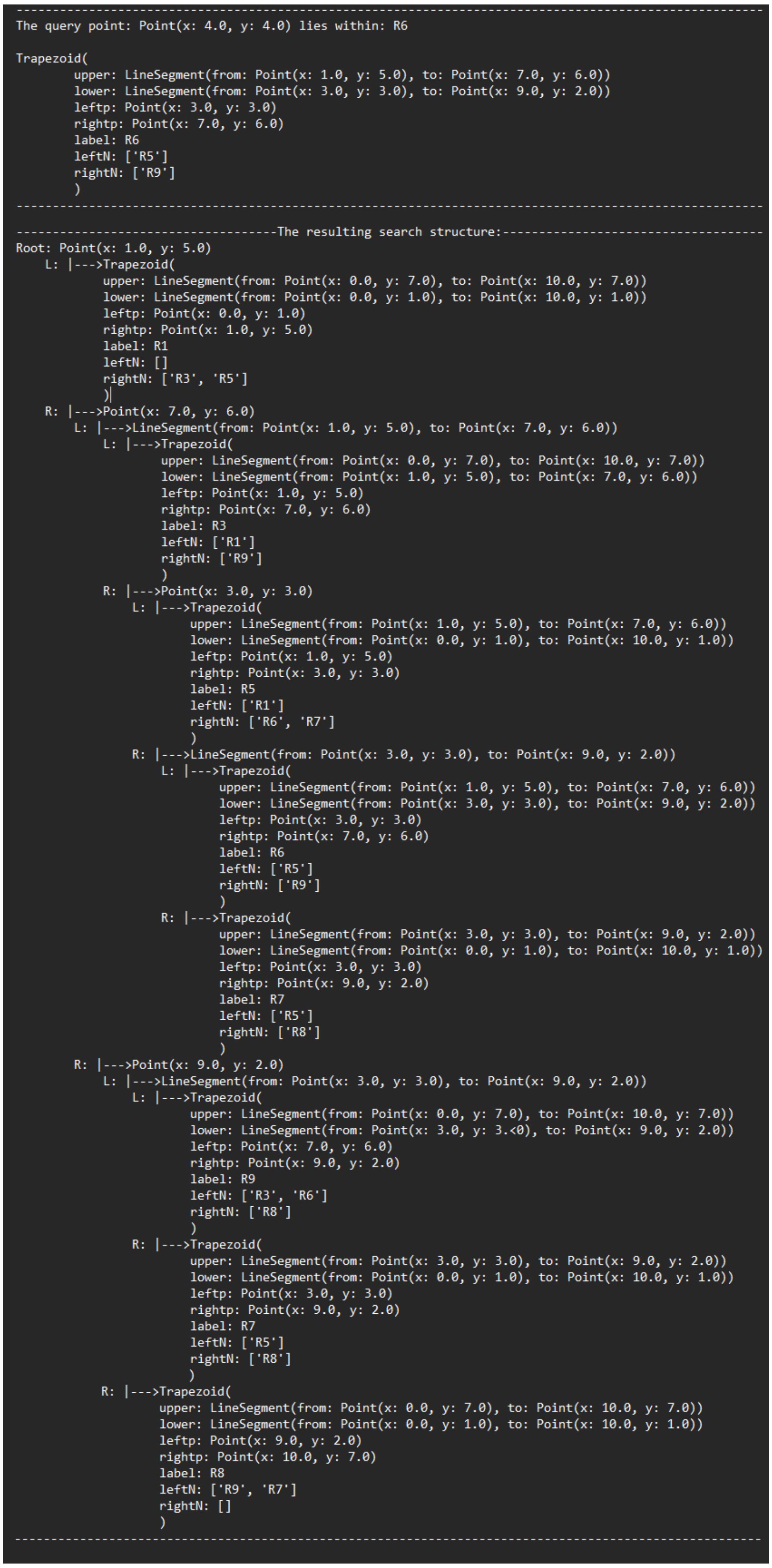


```
---------------------------------------------------------------------------------------------

The query point: Point(x: 3.0, y: 3.0) lies within: R4

Trapezoid(
        upper: LineSegment(from: Point(x: 1.0, y: 5.0), to: Point(x: 5.0, y: 5.0))
        lower: LineSegment(from: Point(x: 0.0, y: 2.0), to: Point(x: 6.0, y: 2.0))
        leftp: Point(x: 1.0, y: 5.0)
        rightp: Point(x: 5.0, y: 5.0)
        label: R4
        leftN: ['R1']
        rightN: ['R2']
        )
---------------------------------------------------------------------------------------------
```

C Example output Tests/test2.txt. with `-o` flag



```
-------------------------------------------------------------------------------------------------------
The query point: Point(x: 4.0, y: 4.0) lies within: R6

Trapezoid(
        upper: LineSegment(from: Point(x: 1.0, y: 5.0), to: Point(x: 7.0, y: 6.0))
        lower: LineSegment(from: Point(x: 3.0, y: 3.0), to: Point(x: 9.0, y: 2.0))
        leftp: Point(x: 3.0, y: 3.0)
        rightp: Point(x: 7.0, y: 6.0)
        label: R6
        leftN: ['R5']
        rightN: ['R9']
        )
-------------------------------------------------------------------------------------------------------

------------------------------------The resulting search structure:------------------------------------
Root: Point(x: 1.0, y: 5.0)
    L: |--->Trapezoid(
            upper: LineSegment(from: Point(x: 0.0, y: 7.0), to: Point(x: 10.0, y: 7.0))
            lower: LineSegment(from: Point(x: 0.0, y: 1.0), to: Point(x: 10.0, y: 1.0))
            leftp: Point(x: 0.0, y: 1.0)
            rightp: Point(x: 1.0, y: 5.0)
            label: R1
            leftN: []
            rightN: ['R3', 'R5']
            )|
    R: |--->Point(x: 7.0, y: 6.0)
        L: |--->LineSegment(from: Point(x: 1.0, y: 5.0), to: Point(x: 7.0, y: 6.0))
            L: |--->Trapezoid(
                    upper: LineSegment(from: Point(x: 0.0, y: 7.0), to: Point(x: 10.0, y: 7.0))
                    lower: LineSegment(from: Point(x: 1.0, y: 5.0), to: Point(x: 7.0, y: 6.0))
                    leftp: Point(x: 1.0, y: 5.0)
                    rightp: Point(x: 7.0, y: 6.0)
                    label: R3
                    leftN: ['R1']
                    rightN: ['R9']
                    )
            R: |--->Point(x: 3.0, y: 3.0)
                L: |--->Trapezoid(
                        upper: LineSegment(from: Point(x: 1.0, y: 5.0), to: Point(x: 7.0, y: 6.0))
                        lower: LineSegment(from: Point(x: 0.0, y: 1.0), to: Point(x: 10.0, y: 1.0))
                        leftp: Point(x: 1.0, y: 5.0)
                        rightp: Point(x: 3.0, y: 3.0)
                        label: R5
                        leftN: ['R1']
                        rightN: ['R6', 'R7']
                        )
                R: |--->LineSegment(from: Point(x: 3.0, y: 3.0), to: Point(x: 9.0, y: 2.0))
                    L: |--->Trapezoid(
                            upper: LineSegment(from: Point(x: 1.0, y: 5.0), to: Point(x: 7.0, y: 6.0))
                            lower: LineSegment(from: Point(x: 3.0, y: 3.0), to: Point(x: 9.0, y: 2.0))
                            leftp: Point(x: 3.0, y: 3.0)
                            rightp: Point(x: 7.0, y: 6.0)
                            label: R6
                            leftN: ['R5']
                            rightN: ['R9']
                            )
                    R: |--->Trapezoid(
                            upper: LineSegment(from: Point(x: 3.0, y: 3.0), to: Point(x: 9.0, y: 2.0))
                            lower: LineSegment(from: Point(x: 0.0, y: 1.0), to: Point(x: 10.0, y: 1.0))
                            leftp: Point(x: 3.0, y: 3.0)
                            rightp: Point(x: 9.0, y: 2.0)
                            label: R7
                            leftN: ['R5']
                            rightN: ['R8']
                            )
        R: |--->Point(x: 9.0, y: 2.0)
            L: |--->LineSegment(from: Point(x: 3.0, y: 3.0), to: Point(x: 9.0, y: 2.0))
                L: |--->Trapezoid(
                        upper: LineSegment(from: Point(x: 0.0, y: 7.0), to: Point(x: 10.0, y: 7.0))
                        lower: LineSegment(from: Point(x: 3.0, y: 3.<0), to: Point(x: 9.0, y: 2.0))
                        leftp: Point(x: 7.0, y: 6.0)
                        rightp: Point(x: 9.0, y: 2.0)
                        label: R9
                        leftN: ['R3', 'R6']
                        rightN: ['R8']
                        )
                R: |--->Trapezoid(
                        upper: LineSegment(from: Point(x: 3.0, y: 3.0), to: Point(x: 9.0, y: 2.0))
                        lower: LineSegment(from: Point(x: 0.0, y: 1.0), to: Point(x: 10.0, y: 1.0))
                        leftp: Point(x: 3.0, y: 3.0)
                        rightp: Point(x: 9.0, y: 2.0)
                        label: R7
                        leftN: ['R5']
                        rightN: ['R8']
                        )
            R: |--->Trapezoid(
                    upper: LineSegment(from: Point(x: 0.0, y: 7.0), to: Point(x: 10.0, y: 7.0))
                    lower: LineSegment(from: Point(x: 0.0, y: 1.0), to: Point(x: 10.0, y: 1.0))
                    leftp: Point(x: 9.0, y: 2.0)
                    rightp: Point(x: 10.0, y: 7.0)
                    label: R8
                    leftN: ['R9', 'R7']
                    rightN: []
                    )
-------------------------------------------------------------------------------------------------------
```

D README.md file as PDF.

Starting on the next page is a PDF version of the `README.md` file.

README

This README will contain a description of how to setup a virtual environment, how to get the program running in the virtual environment on both Windows, MacOS/Linux, as well as IMADA LAB caveats and all.

Creating a Virtual Environment

To create a virtual environment for this project, follow these steps:

1. Open your terminal of choice and check that you have Python 3.8.10 or higher, as follows:

```
python --version
# should output 3.8.10 or higher
```

   - Or maybe something similar (IMADA LAB):

```
python3 --version
```

2. Navigate to the project folder where this README is located.

3. Run the following command to create a virtual environment (replace `myvenv` with your preferred name):

```
python -m venv myvenv
```

   - Or on IMADA LAB use:

```
python3 -m pip install virtualenv
python3 -m virtualenv myenv
```

4. Activate the virtual environment:

   - **Windows**:

```
.\myvenv\Scripts\activate
```

   - **MacOS/Linux/IMADA LAB**:

```
source myvenv/bin/activate
```

Installing Requirements

Once the virtual environment is activated, install the required packages by running:

- **Windows/MacOS/LINUX**

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

- **IMADA LAB**

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

Usage

To use the program, ensure you are inside the project folder and have Python 3.8.10 and Pip 20.0.2 or higher installed. To check this you can run the following commands:

- **Python**

  ```
  python --version      # should output 3.8.10 or higher
  ```

- **Pip**

  ```
  pip --version         # should output 20.0.2 or higher
  ```

Then to run the program use the following command:

```
python ./src/main.py [ optional -d -p -o -qo ] <path_to_file>
```

Parameters

- **Optional Flags**:
  - `-d`: Enable debug mode.
  - `-p`: Plot the output (region containing query point is colored red, query point is colored blue).
  - `-o`: Output a textual result in `output.txt` and a visual result in `TrapezoidalMapPlot.png`.
  - `-qo`: Output only the result of the query in `output.txt` (No search structure or image).

- **File Argument**:
  - You can specify either:
    - `./Test/<file_name>` to use the provided test files, or
    - `<path_to_file>` to use your own test files.

Input Fromat

For input data, represent a line segment going from `(x1, y1)` to `(x2, y2)` by `x1 y1 x2 y2` terminated by one newline.

Similarly, a point is represented by using two coordinates instead of four and there should be exactly one.

A test file should have the point specified on the first line followed by a sequence of disjoint line segments on the subsequent lines, where no two coordinate's `(x or y)` are identical.

Positive and negative integer and float coordinates are allowed, the sign is however required to be immediately in front of the integer/float.

Additionally, before `x1`, in between two coordinates, and after `y2` (but before the newline), any number of spaces or tabs are allowed, so an input file could look like this (or worse):

```
            2.5         0.5
2               2 17              2
            1 3      4                   3.1
5               1 7         1.1
8            3 9                    3.1
                 10         1       12       1.1
14              3.2 16               2.9
                     3 6 11 4
15 6 18 5
6               -1 13 -0.5
    2.25 -1.25 19               -1.5
            0.5                 7 16.5 4
8.5 0.5         0 0.1
                        18.5 -0.1 11.5 0.5
```

Input Errors

In regards to input not following the specified format various errors can arise and will be printed to standard output. These errors include:

- Multiple points defined in input.
- No point defined in input.
- No line segments defined in the input file.
- Point not defined first in input file.
- Input file empty.
- Points or line segments being wrongly formatted.
- A line segment is vertical i.e. `x1 == x2`

Textual / Pysical Output

The files `output.txt` and `TrapezoidalMapPlot.png` will be created in the current directory from which you run the program. For example:

- If you run the program, using this command, in the project folder, the files will be placed alongside the `README.md` and `requirements.txt` files in the project folder.
- If you navigate to the `src` folder and run the program from here, the files will be placed in the `src` folder and so on.

The format of `output.txt` will have a sentence describing which trapezoid the query point is contained within, along with the data describing the trapezoid followed by a textual representation of the search structure, which could look like the following:

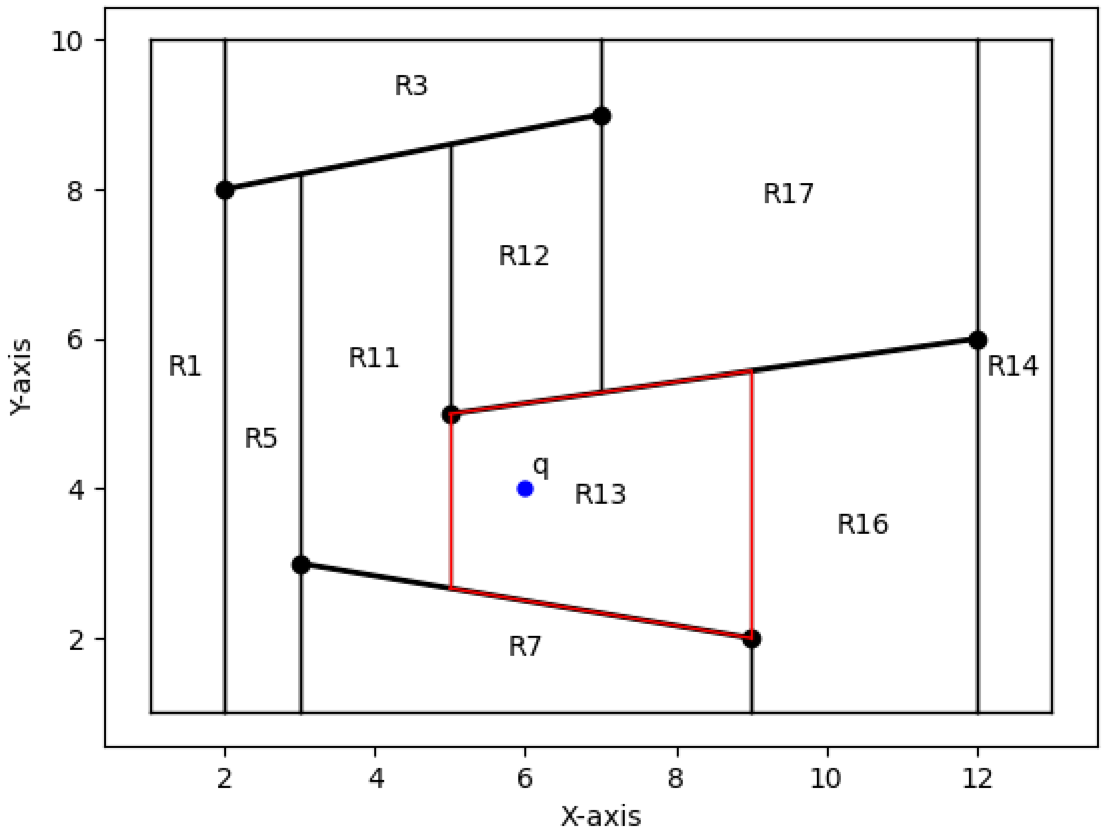
```
-----------------------------------------------------------------------------------
The query point: Point(x: 3.0, y: 3.0) lies within: R4

Trapezoid(
    upper: LineSegment(from: Point(x: 1.0, y: 5.0), to: Point(x: 5.0, y: 5.0))
    lower: LineSegment(from: Point(x: 0.0, y: 2.0), to: Point(x: 6.0, y: 2.0))
    leftp: Point(x: 1.0, y: 5.0)
    rightp: Point(x: 5.0, y: 5.0)
    label: R4
    leftN: ['R1']
    rightN: ['R2']
    )
-----------------------------------------------------------------------------------


--------------------------The resulting search structure--------------------------
Root: Point(x: 1.0, y: 5.0)
    L: |--->Trapezoid(
        upper: LineSegment(from: Point(x: 0.0, y: 6.0), to: Point(x: 6.0, y: 6.0))
        lower: LineSegment(from: Point(x: 0.0, y: 2.0), to: Point(x: 6.0, y: 2.0))
        leftp: Point(x: 0.0, y: 2.0)
        rightp: Point(x: 1.0, y: 5.0)
        label: R1
        leftN: []
        rightN: ['R3', 'R4']
        )
    R: |--->Point(x: 5.0, y: 5.0)
        L: |--->LineSegment(from: Point(x: 1.0, y: 5.0), to: Point(x: 5.0, y:
5.0))
            L: |--->Trapezoid(
                upper: LineSegment(from: Point(x: 0.0, y: 6.0), to: Point(x: 6.0,
y: 6.0))
                lower: LineSegment(from: Point(x: 1.0, y: 5.0), to: Point(x: 5.0,
y: 5.0))
                leftp: Point(x: 1.0, y: 5.0)
                rightp: Point(x: 5.0, y: 5.0)
                label: R3
                leftN: ['R1']
                rightN: ['R2']
                )
            R: |--->Trapezoid(
                upper: LineSegment(from: Point(x: 1.0, y: 5.0), to: Point(x: 5.0,
y: 5.0))
                lower: LineSegment(from: Point(x: 0.0, y: 2.0), to: Point(x: 6.0,
y: 2.0))
                leftp: Point(x: 1.0, y: 5.0)
                rightp: Point(x: 5.0, y: 5.0)
                label: R4
                leftN: ['R1']
                rightN: ['R2']
                )
        R: |--->Trapezoid(
            upper: LineSegment(from: Point(x: 0.0, y: 6.0), to: Point(x: 6.0, y:
6.0))
            lower: LineSegment(from: Point(x: 0.0, y: 2.0), to: Point(x: 6.0, y:
2.0))
            leftp: Point(x: 5.0, y: 5.0)
            rightp: Point(x: 6.0, y: 6.0)
            label: R2
            leftN: ['R3', 'R4']
            rightN: []
            )
---------------------------------------------------------------------------------
```

Graphical Output

For graphical output, the `-p` option enables the program to generate a visual representation of the results. When this flag is used, Matplotlib is employed to display the trapezoidal map by plotting the trapezoids constructed during its creation along with the line segments specified in the input file as well as the query point. The trapezoid containing the query point will be colored red and the query point colored blue, which could look like the following figure:



Notes on The Provided Tests

There are provided 21 tests of which the test files `test14.txt` through `test21.txt`, will fail and output something resembling the following errors to stdout:

- `test14.txt`: `Error: File '.\Test\test14.txt' is empty.`
- `test15.txt`: `Error: No line segments provided in test file.`
- `test16.txt`: `Error: No point provided in test file.`
- `test17.txt`: `Error: The point is not defined first in input file.`
- `test18.txt`: `Error: Multiple points defined in input file, line 4.`
- `test19.txt`: `Error: Line 1 in input file was not formatted correctly, was expecting 2 coordinates but recieved 1 ('0.0').`
- `test20.txt`: `Error: Line 4 in input file was not formatted correctly, was expecting 4 coordinates but recieved 3 ('2.0', '4.0', '6.0').`
- `test21.txt`: `Error: Vertical line defined on line 2 in input file.`

For all other test files, when the `-qo` flag is used, all test files should output what region contains the query point is output.txt. Additionally, if the `-o` flag is used a representation of the search structure will also be written to output.txt along with what region contains the query point as well as a picture of the map in TrapezoidalMapPlot.png. If the `-p` flag is used, the output should be displayed graphically. In any case, the program should run without errors.