Data Structures and Algorithms (INFO-F413)

Assignment 1: Binary Space Partitions

Joan S. Gerard S.
Computer Science Student
Id Number 000471612
jgerards@ulb.ac.be
8 November, 2018

Contents

List of Figures

1 Introduction

We have proved the following theorem during classes:

Theorem 1. *The expected size of the partition produced by the randomized algorithm is bounded from above by* $n + 2nH_n = O(nlogn)$.

In order to demonstrate this empirically a program that creates a BSP tree was written in Python language. Such program receives as an input a text-file which contains a set of n segments $\{S_1, S_2, ..., S_n\}$, the program builds another set (size $n!$) containing the possible permutations of the original set, then it creates a BSP tree for each permutation, it calculates its size and finally it creates four graphics:

- the set of segments in the plane.
- the minimum and maximum size obtained after running BSP.
- the list of the first segments chose by the BSP algorithm that produced the smallest and biggest BSP tree.

2 Description of the program

In this section we will discuss some problems that were faced during the development of the program.

2.1 Upper-Down segment division

When the BSP algorithm choses a segment and projects a line over this segment, this line could intersect some other segments and these should be cut into two different ones and make the upper-down division in order to keep iterating with the rest of segments until only one of them exists in the remaining plane.

When the BSP algorithm starts it will chose the first segment S_1 from the set of segments. S_1 has the coordinates $(x_1, y_1), (x_2, y_2)$ in the (x, y) plan so we can easily determine the slope of the line, which we will call m given by:

$$m = \frac{(y_2 - y_1)}{(x_2 - x_1)} \tag{1}$$

Then the equation of the line is given by:

$$y = mx + b \tag{2}$$

In order to obtain b we can simply replace (x_1, y_1) in equation 2 and we will get concrete values for m and b. This calculation was made in the *Segment.py* (A.3) file by the following method:

```
    def line_projection(self):
        """
        y = mx + b
        :return: Line
        """
        epsilon = 8e-7
        m = (self.end.y - self.start.y)/(self.end.x - self.start.x + epsilon)
        b = self.end.y - m * self.end.x
        return Line(m, b)
```

Listing 1: Line Projection Code

Note that in order to avoid division by 0 a value, epsilon, very small is added into m equation.

Once we have the line projection it is easy to determine which other segments are above, below or intersected by the line projection of the chosen segment. For instance, given the segment S_2 with points $(x_{21}, y_{21}), (x_{22}, y_{22})$ and the line projection of the segment S_1 defined by the equation 2, we want to determine the relative position of S_2 to the line projection of S_1, we simply replace x_{21} and x_{22} in equation 2 and we obtain y_1, y_2 respectively. The result of these values can be interpreted as follows:

- If $y_{21} > y_1$ and $y_{22} > y_2$: the segment S_2 is above S_1.
- If $y_{21} < y_1$ and $y_{22} < y_2$: the segment S_2 is below S_1.
- If $y_{21} > y_1$ and $y_{22} < y_2$ (or viceversa): the segment S_2 intersects the line projection of S_1.
- If $y_{21} = y_1$ and $y_{22} = y_2$: the segment S_2 is on the same line projection of S_1.

This is calculated by the following code in the *BSPManager.py* (A.4) file:

```
    def _compare(self, segment, to_segment):
        """
        Upper-down calculation between segments.
        :param segment:     Segment
        :param to_segment:  Segment
        :return:            string
        """
        y1, y2 = self._get_y_pos_based_on(segment, to_segment)

        """to_segment is up to the segment"""
        if to_segment.start.y > y1 and to_segment.end.y > y2:
            return 'up'
        elif to_segment.start.y < y1 and to_segment.end.y < y2:
            return 'down'
        else:
            return 'between'
```

Listing 2: Upper-Down segment division

Note that the case when S_2 is on the same line projection of S_1 was not considered in this algorithm.

2.2 Segment permutations

Initially the program read a file which contains a set of segments, then it generates all the possible segment permutations because we want to be sure that a permutation that generates a BSP tree whose size exceeds $O(nlogn)$ does not exist. Thus, the program will generate a BSP tree for each set of segments that belongs to the permutations list.

The file *Main.py* (A.5) contains the code that generates the permutations, builds the BSP tree and obtain their sizes:

```
        # Get all possible permutations of the segments
        list_permuted_segments = itertools.permutations(segments)

        for set_segments in list_permuted_segments:
            bsp = self.bsp.build(list(set_segments))
            size = self.bsp.size(bsp)
            sizes.append(size)

            # if it is the first time it reads this size
            if size not in segment_classifier_size:
                segment_classifier_size[size] = []

            # it saves a record of the BSP tree size obtained
```

```
            # given the first segment selected from the permutation
            segment_classifier_size[size].append(bsp.get_value().name)
            i += 1
        return sizes, segment_classifier_size
```

Listing 3: Segment Permutations Code

2.3 BSP algorithm

The method is recursive and basically it chooses the first segment of the set, makes the upper-down division based on the line projection and calls the same algorithm for the subset of segments until there is only one segment in a cell. The file *BSPManager.py* (A.4) contains the following code for building the BSP tree recursively given a set of segments.

```
    def _build_bsp_recursive(self, segments):
        """
        It builds the BSP tree recursively.
        :param segments: list
        :return: Node
        """
        # node is empty/null
        if len(segments) == 0:
            return None
        # there is only one segment
        if len(segments) == 1:
            leaf = Node()
            leaf.add_value(segments.pop())
            return leaf
        # there are some segments to be processed
        else:
            up_segments = []
            down_segments = []
            segment_separator = segments.pop(0)
            for segment in segments:
                comparison_result = self._compare(segment_separator, segment)
                # segment is above the line separator
                if comparison_result == 'up':
                    up_segments.append(segment)
                # segment is below the line separator
                elif comparison_result == 'down':
                    down_segments.append(segment)
                # segment intersects the line separator
                elif comparison_result == 'between':
                    y1, y2 = self._get_y_pos_based_on(segment_separator, segment)
                    intersection_point = self._get_intersection_point(segment_separator, segment)
                    # cut the segment in two and put one part
                    # into the up list segments and the other into the
                    # down list segment.
                    if y1 < segment.start.y and y2 > segment.end.y:
                        segment_a = segment.cut_to(intersection_point)
                        segment_b = segment.cut_from(intersection_point)
                        up_segments.append(segment_a)
                        down_segments.append(segment_b)
                    elif y1 > segment.start.y and y2 < segment.end.y:
                        segment_a = segment.cut_to(intersection_point)
                        segment_b = segment.cut_from(intersection_point)
                        up_segments.append(segment_b)
                        down_segments.append(segment_a)
            # Recursive call
            right_node = self._build_bsp_recursive(up_segments)
            left_node = self._build_bsp_recursive(down_segments)

```

```
        # Node creation
        root = Node()
        root.add_value(segment_separator)
        root.add_left(left_node)
        root.add_rigth(right_node)
        return root
```

Listing 4: BSP Algorigthm

3 Analysis of data

The program was tested with three different set of segments obtaining the same result.

3.1 First case

The first case corresponds to the following set of segments:

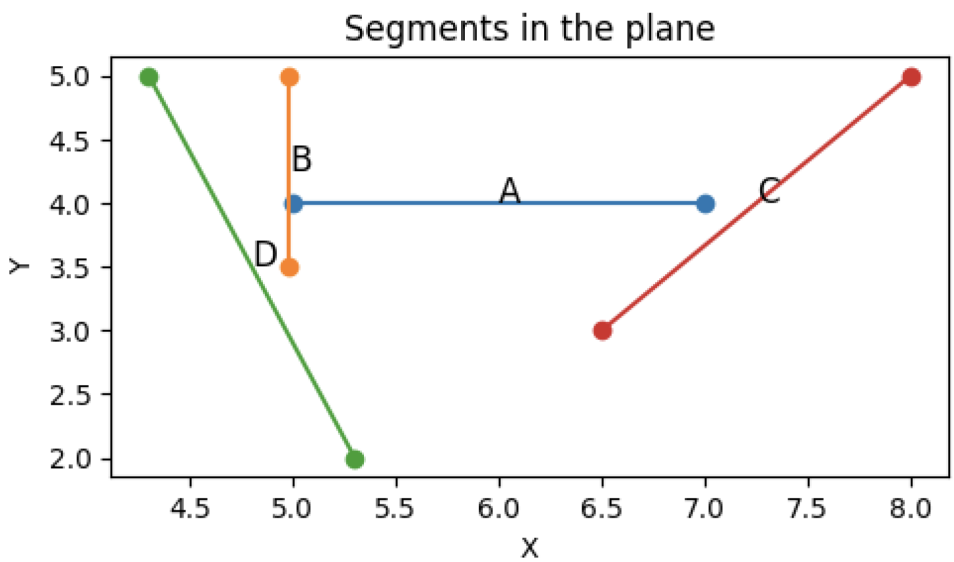

Figure 1: Set of segments from assignemnt

Figure 2 shows BSP tree size of all the possible permutations of the set of segments after running the algorithm.

Given the number of segments ($n = 4$), we have 4! possible combinations. Figure 2 shows that there were 3 permutations of segments that build a BSP tree with a size equals to 4, 8 of size 5, 6 of size 6, 4 of size 7 and 3 of size 8.

Figure 3 shows that permutations running BSP starting with segment C (1 permutation) or D (2 permutations) obtained a tree with size equals to 4 which corresponds to the minimum size (called minimal occurrence here).

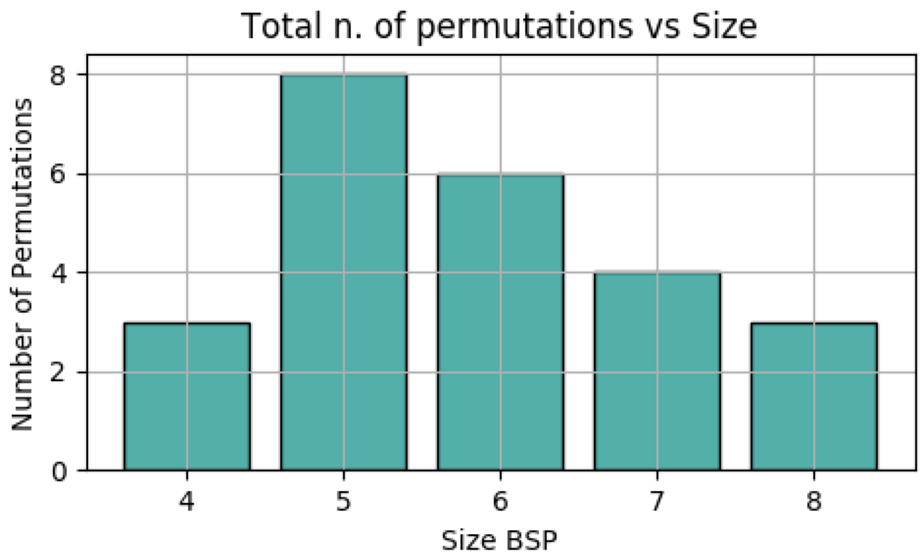

Figure 2: Permutations vs Size for first case

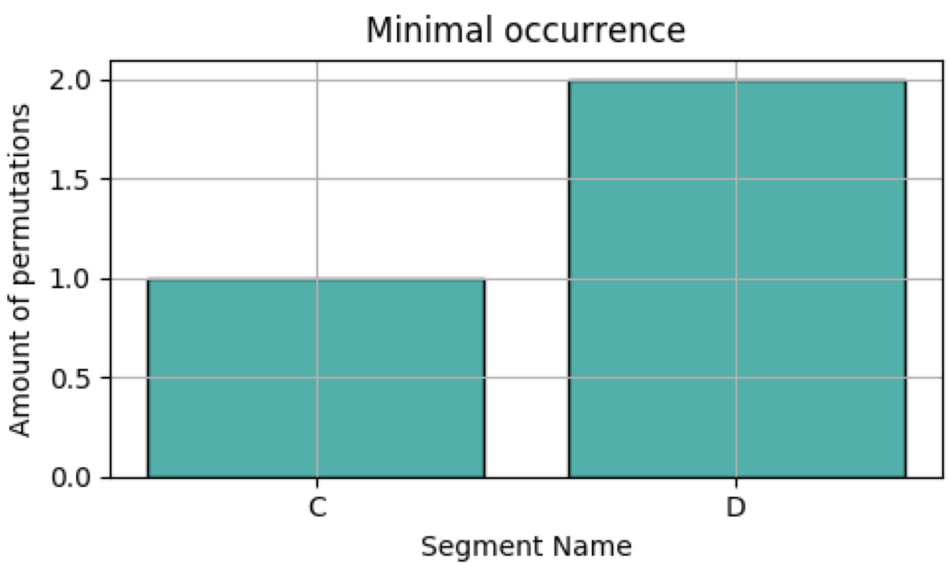

Figure 3: Best choices for first case

The figure 4 shows that those permutations that run BSP starting with segment A (3 permutations) obtained a tree with a size equals to 8 which corresponds to the maximum size (called maximal occurrence here).

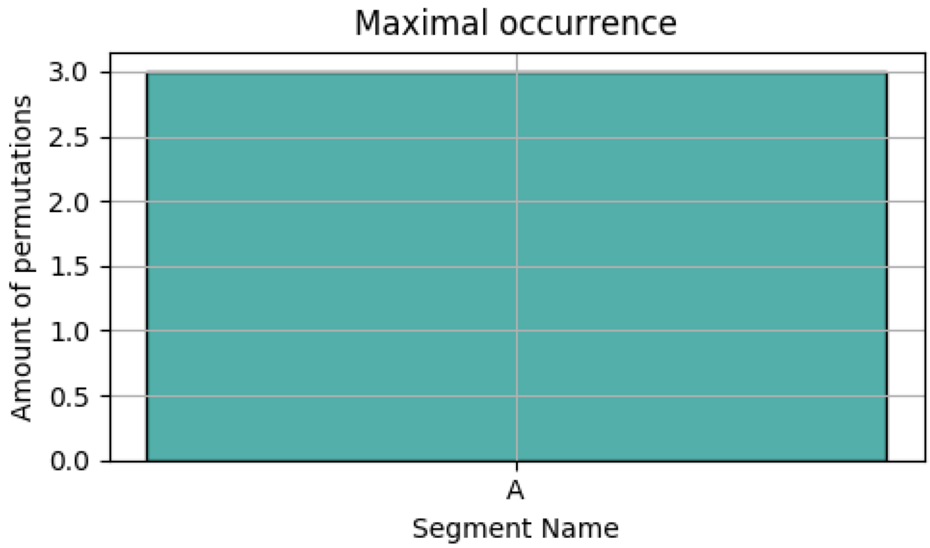

Figure 4: Worst choices for first case

The worst-case scenario generated a BSP tree of size 8 and it did not exceed the upper bound limit of $O(nlogn)$. Additionally, we also know that line projections over C and D do not intersect any other segments while the one over A intersects all others, this fact corresponds to the minimal and maximal occurrence respectively.

3.2 Second case

Figure 5 shows a different set of sets ($n = 8$). The maximum size for the BSP tree is 15 for this case which is again lower than the upper bound. Note also that some permutations that run BSP starting with segments B or C obtained a tree with a size equals to 8 which is the best case possible. However, those starting with A, E, G, H obtained a BSP of size 15 which is the worst case possible. Furthermore, the projection of B and C does not intersect with any other segment, this is not the case for A, E, G and H.

3.3 Third case

Here $n = 8$ and we have some parallel segments and only one not-parallel segment.

Figure 6 shows that the maximum size for a BSP tree given the set of segments is 15 which is again lower than the upper bound. Additionally, the amount of permutations to get a BSP of size 8 is higher so the probability of choosing a permutation that generates a smaller BSP is also high. Note also that there are more permutations starting with segments B or F, which are closer to the segment A, that generates a minimum-size BSP tree.

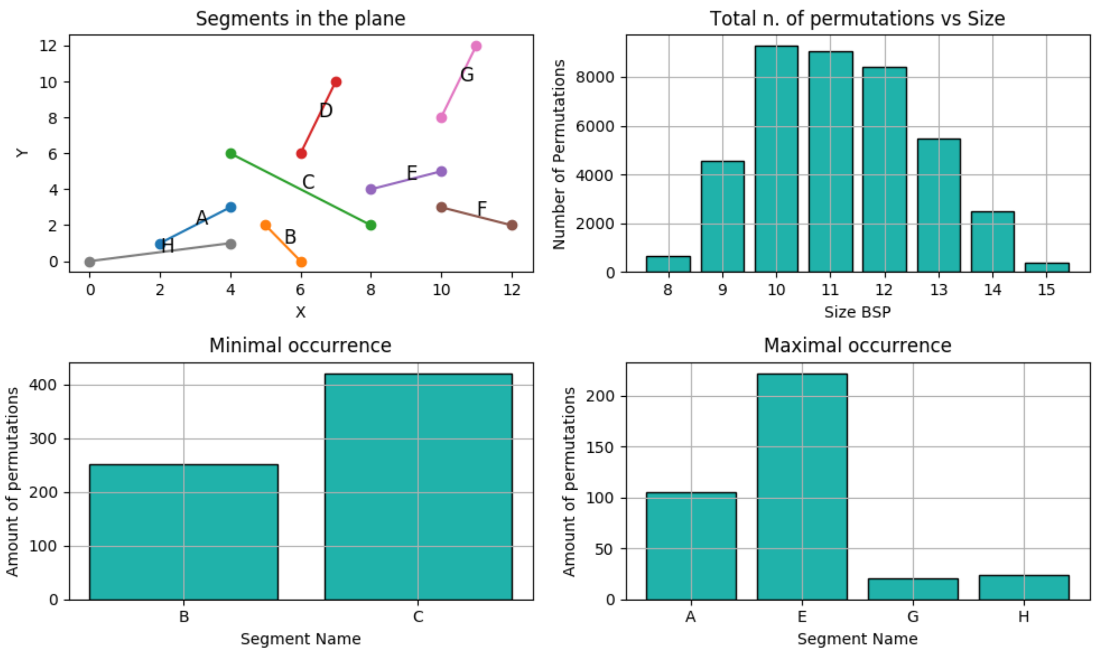

Figure 5: Data analysis for the second case

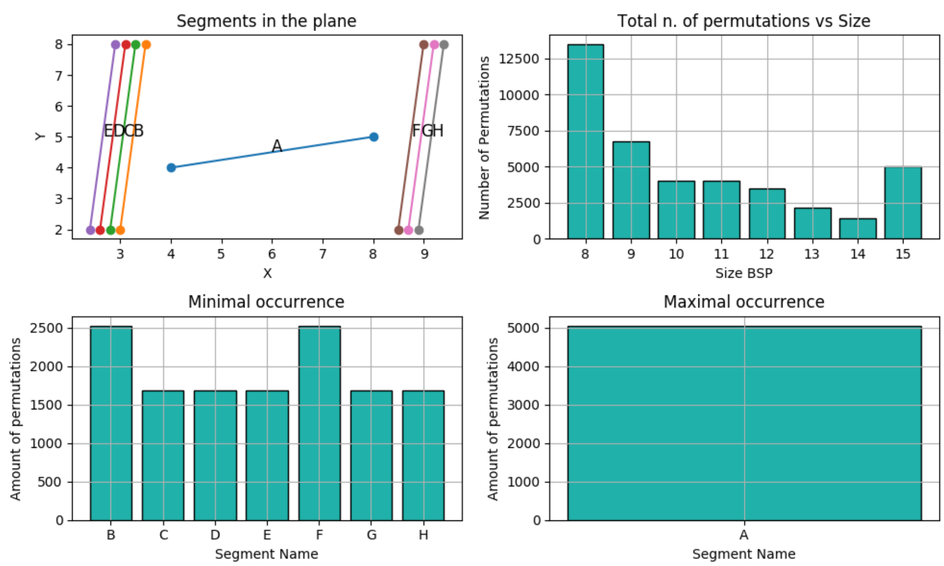

Figure 6: Data analysis for the third case

4 Conclusions

Based on the three cases presented above, we can see that permutations that generated the smallest binary partition for each case would be the ones starting with the segments whose line projections do not cut any other segment.

The three cases presented on this report were chosen between other test cases and, as it was expected, none of them overpassed the upper bound limit given by theorem 1.

A Code

Here is all the code used for this assignment.

A.1 Node.py

```
class Node:
    def __init__(self):
        self._value = None
        self._right = None
        self._left = None

    def add_rigth(self, node):
        self._right = node

    def add_left(self, node):
        self._left = node

    def add_value(self, value):
        self._value = value

    def has_children(self):
        return self._right is not None or self._left is not None

    def get_left(self):
        return self._left

    def get_right(self):
        return self._right

    def get_value(self):
        return self._value
```

A.2 Line.py

```
class Line:
    def __init__(self, m, b):
        """
        y = mx + b
        :param m: float
        :param b: float
        """
        self.m = m
        self.b = b

    def calculate_y(self, x):
        """
        Calculate y given x
        :param x: float
        :return:
        """
        return self.m * x + self.b
```

A.3 Segment.py

```
from Point import Point
from Line import Line


class Segment:
    def __init__(self, start, end, name='R'):
        """
        given 2 points
```

```
        :param start: Point
        :param end:   Point
        """
        self.start = start
        self.end = end
        self.name = name

    def line_projection(self):
        """
        y = mx + b
        :return: Line
        """
        epsilon = 8e-7
        m = (self.end.y - self.start.y)/(self.end.x - self.start.x + epsilon)
        b = self.end.y - m * self.end.x
        return Line(m, b)

    def cut_to(self, point):
        """
        Cuts segment from origin to the point
        :param point: Point
        :return: Segment
        """
        return Segment(self.start, point)

    def cut_from(self, point):
        """
        Cuts segment from point to the end
        :param point: Point
        :return: Segment
        """
        return Segment(point, self.end)
```

A.4 BSPManager.py

```
from Node import Node
from Point import Point


class BSPManager:

    def build(self, segments):
        """
        It calls the recursive method.
        :param segments: list
        :return: Node
        """
        binary_tree = self._build_bsp_recursive(segments)
        return binary_tree

    def _build_bsp_recursive(self, segments):
        """
        It builds the BSP tree recursively.
        :param segments: list
        :return: Node
        """
        # node is empty/null
        if len(segments) == 0:
            return None
        # there is only one segment
        if len(segments) == 1:
            leaf = Node()
            leaf.add_value(segments.pop())
            return leaf
        # there are some segments to be processed
```

```
        else:
            up_segments = []
            down_segments = []
            segment_separator = segments.pop(0)
            for segment in segments:
                comparison_result = self._compare(segment_separator, segment)
                # segment is above the line separator
                if comparison_result == 'up':
                    up_segments.append(segment)
                # segment is below the line separator
                elif comparison_result == 'down':
                    down_segments.append(segment)
                # segment intersects the line separator
                elif comparison_result == 'between':
                    y1, y2 = self._get_y_pos_based_on(segment_separator, segment)
                    intersection_point = self._get_intersection_point(segment_separator, segment)
                    # cut the segment in two and put one part
                    # into the up list segments and the other into the
                    # down list segment.
                    if y1 < segment.start.y and y2 > segment.end.y:
                        segment_a = segment.cut_to(intersection_point)
                        segment_b = segment.cut_from(intersection_point)
                        up_segments.append(segment_a)
                        down_segments.append(segment_b)
                    elif y1 > segment.start.y and y2 < segment.end.y:
                        segment_a = segment.cut_to(intersection_point)
                        segment_b = segment.cut_from(intersection_point)
                        up_segments.append(segment_b)
                        down_segments.append(segment_a)
            # Recursive call
            right_node = self._build_bsp_recursive(up_segments)
            left_node = self._build_bsp_recursive(down_segments)

            # Node creation
            root = Node()
            root.add_value(segment_separator)
            root.add_left(left_node)
            root.add_rigth(right_node)
            return root

    def _compare(self, segment, to_segment):
        """
        Upper-down calculation between segments.
        :param segment:     Segment
        :param to_segment:  Segment
        :return:            string
        """
        y1, y2 = self._get_y_pos_based_on(segment, to_segment)

        """to_segment is up to the segment"""
        if to_segment.start.y > y1 and to_segment.end.y > y2:
            return 'up'
        elif to_segment.start.y < y1 and to_segment.end.y < y2:
            return 'down'
        else:
            return 'between'

    def _get_y_pos_based_on(self, segment, to_segment):
        """
        Calculate position of to_segment relative to segment.
        :param segment:     Segment
        :param to_segment:  Segment
        :return: int, int
        """
```

```
        line_separator = segment.line_projection()
        y1 = line_separator.calculate_y(to_segment.start.x)
        y2 = line_separator.calculate_y(to_segment.end.x)
        return y1, y2

    def _get_intersection_point(self, segment, to_segment):
        """
        Calculate intersection of segment with the line projection.
        :param segment:     Segment
        :param to_segment:  Segment
        :return: Point
        """
        line1 = segment.line_projection()
        line2 = to_segment.line_projection()
        x = (line1.b - line2.b) / (line2.m - line1.m)
        y = line1.m * x + line1.b
        return Point(x, y)

    def size(self, bsp):
        """
        Call the recursive call that calculates the size of the tree.
        :param bsp: Node
        :return: int
        """
        return self._size(bsp, 0)

    def _size(self, node, counter):
        """
        Calculates the size of the tree.
        :param node: Node root node at the very first call
        :param counter: int
        :return:
        """
        # Node is empty, do nothing.
        if node is None:
            return counter
        # It is a leaf
        if not node.has_children():
            return counter + 1
        # Recursive calls
        else:
            counter = self._size(node.get_left(), counter)
            counter = self._size(node.get_right(), counter)
            return counter + 1
```

A.5 Main.py

```
from Segment import Segment
from Point import Point
from BSPManager import BSPManager
from Graphic import Graphic
import itertools
import sys


class Main:
    def __init__(self):
        self.bsp = BSPManager()

    def extract_segments_from_file(self):
        """
        It reads the segments from a file with the follow structure:
        x1, y1, x2, y2, name
        :return: list
        """
```

```
        segments = []
        file_name = 'test_samples.txt'
        if len(sys.argv) > 1:
            file_name = sys.argv[1]
        file = open(file_name, 'r')
        for line in file:
            points = line.split(' ')
            point_start = Point(float(points[0]), float(points[1]))
            point_end = Point(float(points[2]), float(points[3]))
            name = points[4].replace('\n', '')
            segments.append(Segment(point_start, point_end, name))
        return segments

    def execute(self):
        """
        It generates all the possible permutations given a set of
        segments and saves it into a list called list_permuted_segments
        then it builds the BSP tree for each set of segments and calculates
        its size.

        I
        :return: list, dictionary
        """
        i = 0
        sizes = []
        segments = self.extract_segments_from_file()
        segment_classifier_size = {}

        # Get all possible permutations of the segments
        list_permuted_segments = itertools.permutations(segments)

        for set_segments in list_permuted_segments:
            bsp = self.bsp.build(list(set_segments))
            size = self.bsp.size(bsp)
            sizes.append(size)

            # if it is the first time it reads this size
            if size not in segment_classifier_size:
                segment_classifier_size[size] = []

            # it saves a record of the BSP tree size obtained
            # given the first segment selected from the permutation
            segment_classifier_size[size].append(bsp.get_value().name)
            i += 1
        return sizes, segment_classifier_size

    def execute_once(self):
        """
        Executes the BSP algorithm for a given set of segments.
        """
        segments = self.extract_segments_from_file()
        bsp = self.bsp.build(list(segments))
        print('Size of BSP tree: %i' % self.bsp.size(bsp))


main = Main()
graphic = Graphic()
segments = main.extract_segments_from_file()
main.execute_once()
sizes, segment_classifier_size = main.execute()

graphic.graphic(segments, sizes, segment_classifier_size)
```