Advanced Analysis of Algorithms Assignment
University of the Witwatersrand

Tevlen Naidoo (2429493)

May 2023

Contents

Chapter 1

Introduction

Orthogonal rectangles are commonly used in computer graphics, architecture, and engineering to represent objects and spaces in two-dimensional planes. One important aspect of working with orthogonal rectangles is being able to determine their adjacencies, which can be used to analyze and design complex systems. However, manually identifying the adjacencies of a large number of rectangles can be time-consuming and error-prone. Therefore, the development of an algorithm or program that can automatically determine adjacencies of orthogonal rectangles is essential for streamlining various applications.

The purpose of this assignment is to develop an algorithm that can accurately and efficiently determine the vertical adjacencies of orthogonal rectangles. The algorithm will use various techniques, such as geometric analysis and data structures, to identify the vertical edges of each rectangle and compare them to those of adjacent rectangles. The program will be tested on a variety of datasets to evaluate its effectiveness and efficiency.

The results of this assignment will provide a foundation for the development of a more efficient and accurate algorithm for identifying adjacencies of orthogonal rectangles.

Chapter 2

Aim

This assignment is intended to give some exposure to the experimental nature of Computer Science – specifically the concept of measuring the performance of different algorithms that solve the same problem and relating these measurements to the theoretical analysis of the algorithms. This will be accomplished by performing an experiment and preparing a document discussing the design, implementation and evaluation of the experiment.

A second aim of the assignment is to introduce a new technique for solving problems more efficiently than the obvious *brute force* approach.

Chapter 3

The Problem

3.1 Determining Vertical Adjacencies Between Orthogonal Rectangles

In a particular application, it is important to be able to find the adjacencies (shared edges or parts of edges) between a set of orthogonal rectangles in the plane efficiently. An orthogonal rectangle is a rectangle whose edges are aligned with the x and y axes.

In determining the adjacencies between the rectangles, horizontal and vertical adjacencies can be treated as separate cases. In this assignment, only the case of vertical adjacencies will be tackled. Horizontal adjacencies can be treated analogously.

[8] The tasks in this assignment are to

1. develop a brute force or naïve algorithm to solve this problem, and then
2. to use knowledge about the problem to develop an improved algorithm

3.2 Requirements

Any orthogonal rectangle R can be defined by the x and y coordinates of its bottom-left and top-right corners. The input to the algorithm/program will be a list of rectangles plus their coordinates.
Our program will accept input in the form of

$$\text{Number}, x_1, y_1, x_2, y_2$$

For example, our program would accept an input of

$$1, 0, 0, 20, 30$$

which would mean: Rectangle 1 with bottom-left corner at $(0, 0)$, and top-right corner at $(20, 30)$.

The algorithm thus requires a data structure which contains the rectangle number and these coordinates as well as the ability to keep track of other information calculated in the algorithm. This other information is the number of rectangles which are adjacent to the right-hand side of the rectangle being considered and a list of these rectangles plus the x coordinate and bottom and top y coordinates of the respective adjacencies. These lists of adjacencies (one list per rectangle) are, in fact, the required output from this algorithm and are used later in the application.

Chapter 4

The Input

For this experiment, we will be generating our own input to test.

4.1 Generating the Input

4.1.1 Algorithm for Generating the Input

The following is the algorithm for generating the input data necessary for this experiment.

1. We create an empty list which will be where we store all the rectangles.
2. We start with an initial, single rectangle. This rectangle will be the area within which, we generate non-overlapping, adjacent, orthogonal rectangles.
3. We add this initial rectangle to the rectangle list.
4. We split this initial rectangle into four smaller rectangles.
5. We remove the initial rectangle from the rectangle list and add the four smaller rectangles to the rectangle list.
6. While the rectangle list has fewer rectangles than what we want, we do the following:
   - 6.1. Choose a random rectangle from the rectangle list.
   - 6.2. Split this random rectangle into four smaller rectangles.
   - 6.3. We remove the random rectangle from the rectangle list and we add the four smaller rectangles to the list.
7. If the rectangle list contains more rectangles than we want, we randomly choose and remove a rectangle from the rectangle list until it contains the number of rectangles we want.

This algorithm was inspired by the code provided by Dr Ian Sanders [10].

4.1.2 Pseudo-code for Generating the Input

Algorithm 1: Generating Input

```
Output: list of rectangles
Set $num\_rectangles$
Initialise an empty list of rectangles
Generate an initial rectangle $r$
Split $r$ into 4 smaller rectangles
Add the 4 smaller rectangles to the rectangle list
while length of rectangle list $\leq num\_rectangles$ do
    Choose a random rectangle from the rectangle list
    Split that rectangle into 4 smaller rectangles
    Remove the random rectangle from the list
    Add the 4 smaller rectangles to the list
end
while length of rectangle list $\neq num\_rectangles$ do
    Choose a random rectangle $r$ from the rectangle list
    Remove $r$ from the rectangle list
end
return list of rectangles
```

4.2 Understanding the Input

The input is structured as follows:

$$\text{Number}, x_1, y_1, x_2, y_2$$

or, in text form (as seen as the headers of the input .csv files):

```
Number,x1,y1,x2,y2
```

Where

Number is the rectangle number

(x_1, y_1) are the coordinates of the bottom-left corner of the rectangle

(x_2, y_2) are the coordinates of the top-right corner of the rectangle

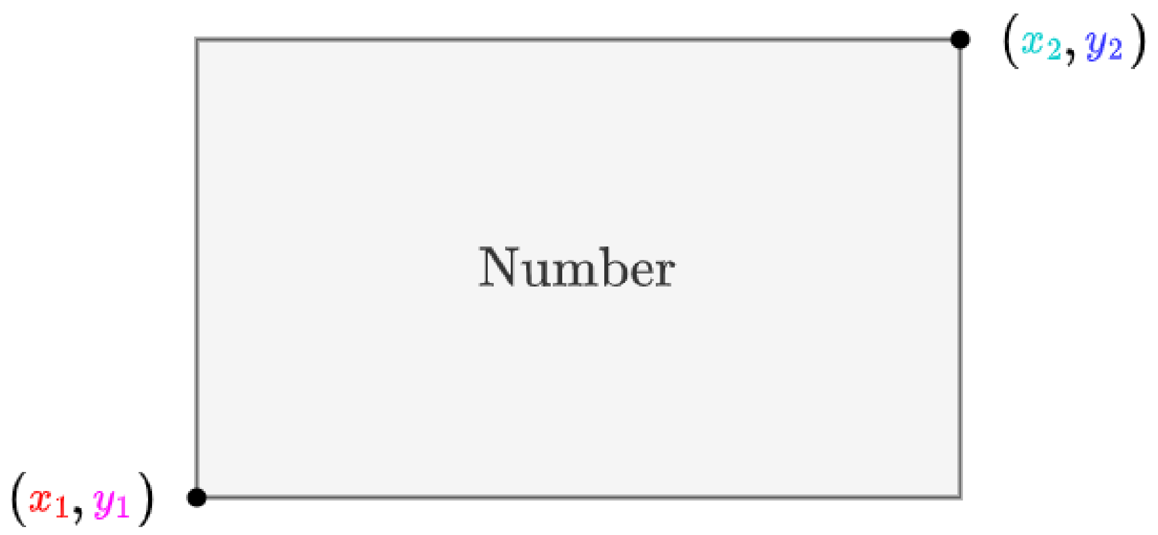


Figure 4.1: Visual Representation of the Input Structure

4.3 Visualising the Input

We will use the data from the *sample_input.csv* file.

To see the contents of *sample_input.csv*, refer to Figure B.1

Number	x_1	y_1	x_2	y_2
1	5	5	16	22
2	9	26	16	35
3	16	5	20	9
4	16	11	32	15
5	16	18	25	29
6	25	18	30	22
7	25	25	27	29
8	27	28	31	31
9	32	12	36	19
10	36	13	39	21
11	39	14	43	22

Table 4.1: Better Formatted Data from the *sample_input.csv* File

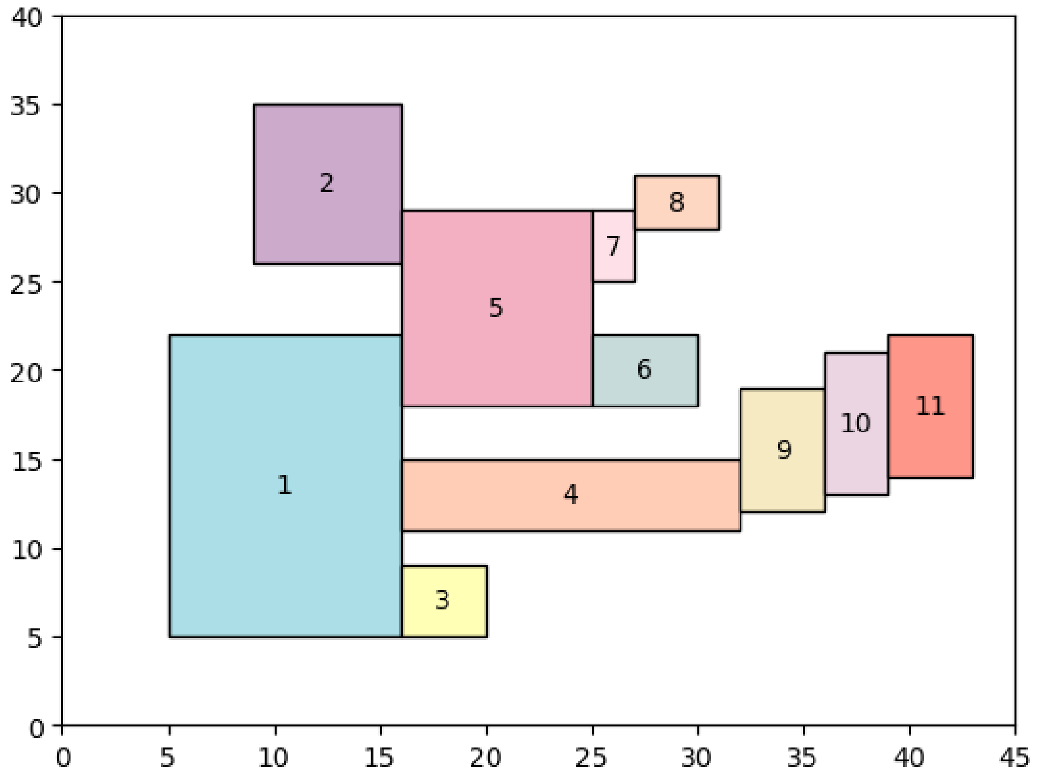


Figure 4.2: Visual Representation of the Data from *sample_input.csv*

Visual Representation of the Generated Input Data

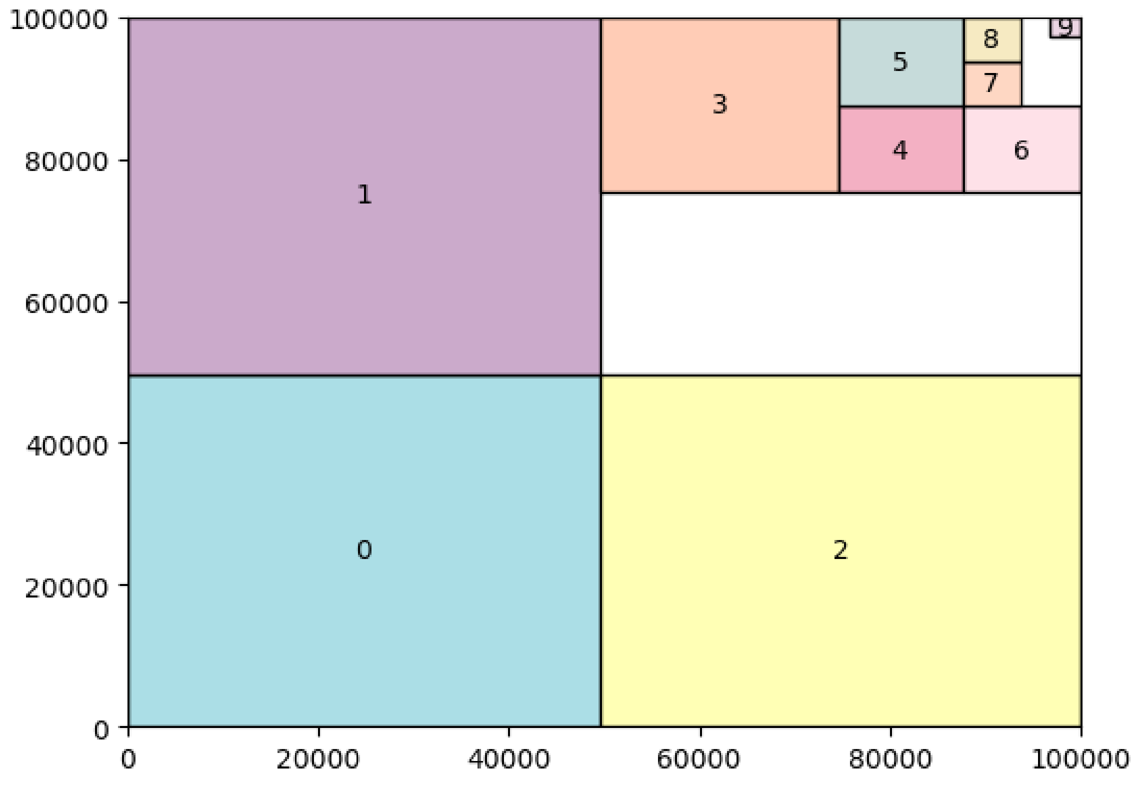


Figure 4.3: Visual Representation of the Data from *in10.csv*

Chapter 5

Vertical Adjacency

In order to identify cases of vertical adjacency, we first need to understand:

- What is vertical adjacency?
- How do we determine vertical adjacency?
- What are the details of the vertical adjacency?

Note: For this assignment, we will only be looking for adjacencies to the right of a given rectangle.

5.1 What is Vertical Adjacency?

Vertical adjacency refers to the relationship between two orthogonal rectangles that share a common vertical edge. In other words, two rectangles are vertically adjacent if they are located directly above or below each other and share a common vertical edge that is parallel to the y-axis.

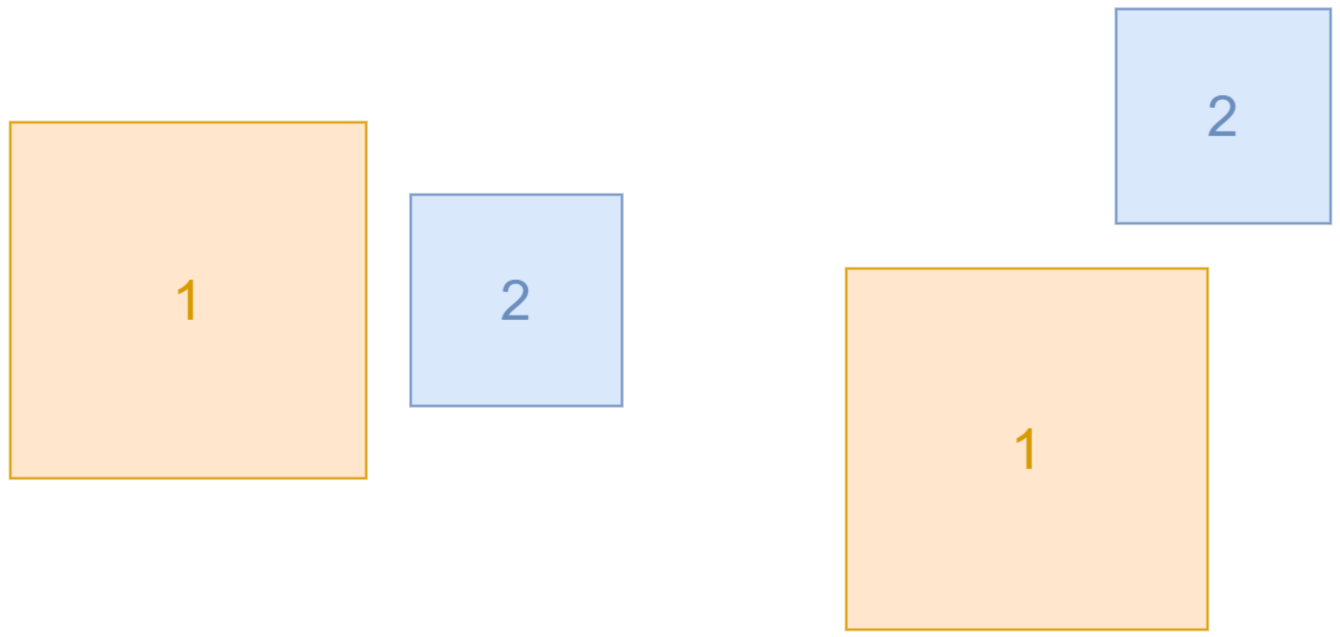


Figure 5.1: Rectangles that are NOT vertically adjacent

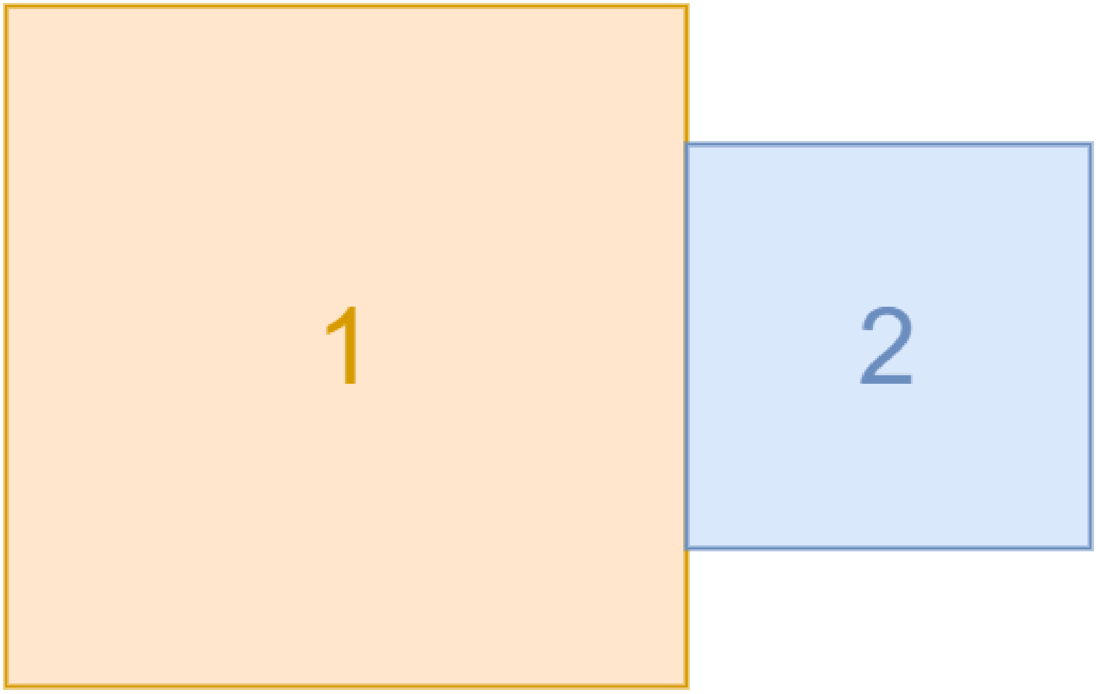


Figure 5.2: Rectangles that are vertically adjacent

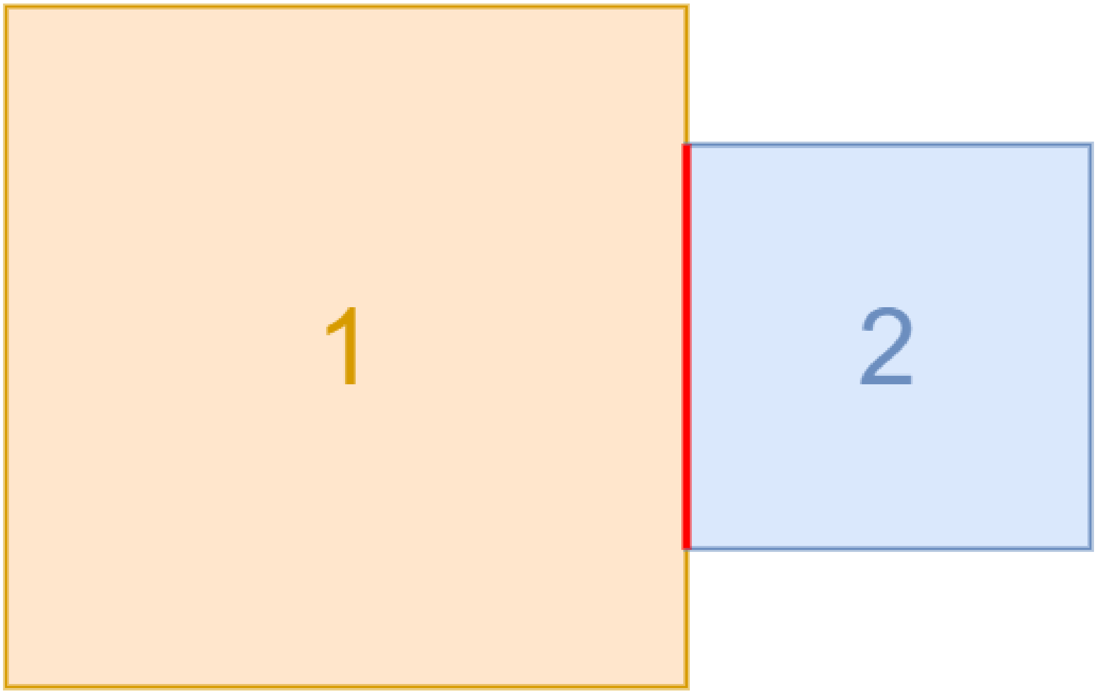


Figure 5.3: Indicating the region of vertical adjacency

5.2 Determining Vertical Adjacency

We start by comparing two rectangles. Lets call them Rectangle 1 and Rectangle 2. We are trying to see if Rectangle 2 is vertically adjacent, to the right, to Rectangle 1.

In order to do this, we need four corners to compare:

- The top-right and bottom-right corners of Rectangle 1
- The top-left and bottom-left corners of Rectangle 2

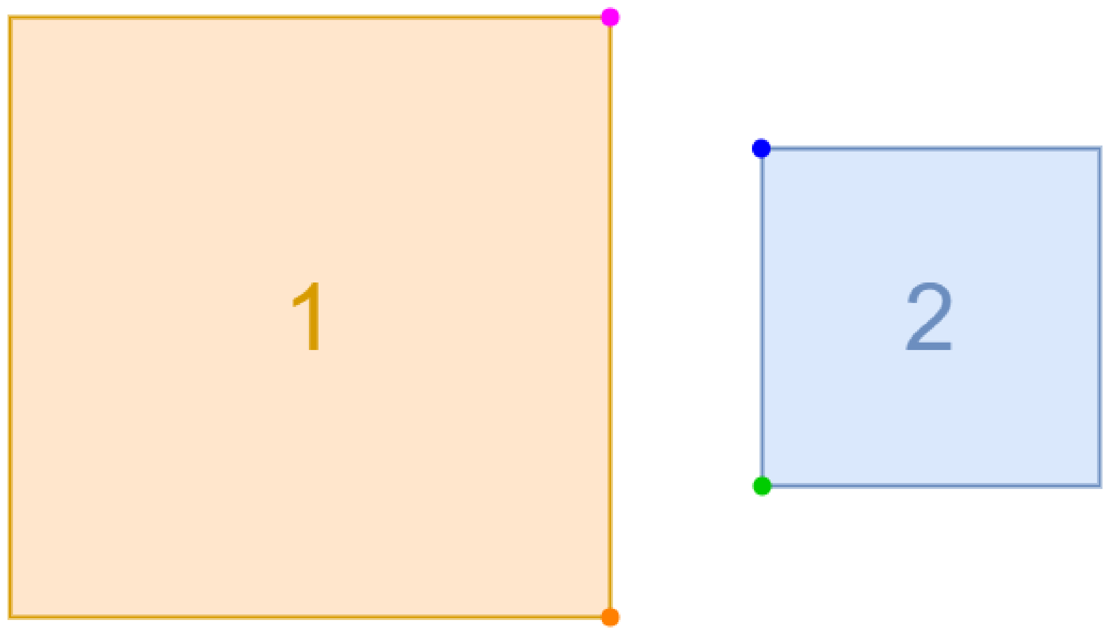


Figure 5.4: Two rectangles with the corners of importance highlighted

The top-right corner of Rectangle 1 is shown by ●
The bottom-right corner of Rectangle 1 is shown by ●

The top-left corner of Rectangle 2 is shown by ●
The bottom-left corner of Rectangle 2 is shown by ●

Let the coordinates of the top-right and bottom-right corners, of Rectangle 1, be (x_{TR}, y_{TR}) and (x_{BR}, y_{BR}) respectively.

Let the coordinates of the top-left and bottom-left corners, of Rectangle 2, be (x_{TL}, y_{TL}) and (x_{BL}, y_{BL}) respectively.

5.2.1 Comparing x-values

The first step, to see if Rectangle 2 is vertically adjacent to Rectangle 1, is to compare the x-values of Rectangle 1's top-right corner and Rectangle 2's top-left corner. For Rectangle 2 to be vertically adjacent to Rectangle 1, these x-values must be the same. In other words, the top-right corner of Rectangle 1 and top-left corner of Rectangle 2 must lie on the same vertical line.

Particularly,

$$x_{TR} = x_{TL}$$

Note: We can also compare the x-values of Rectangle 1's bottom-right corner and Rectangle 2's bottom-left corner

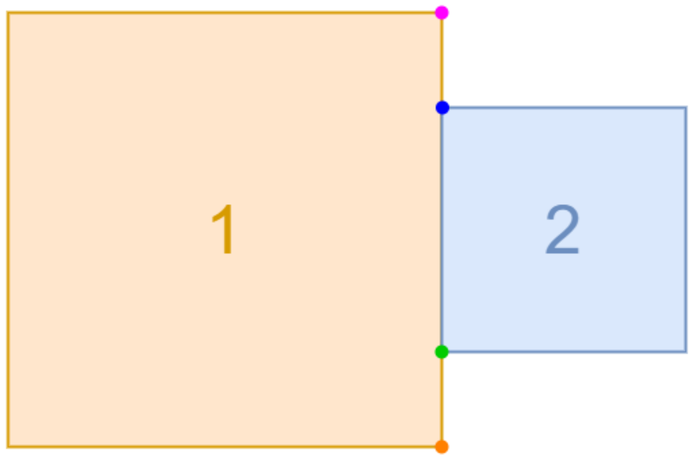
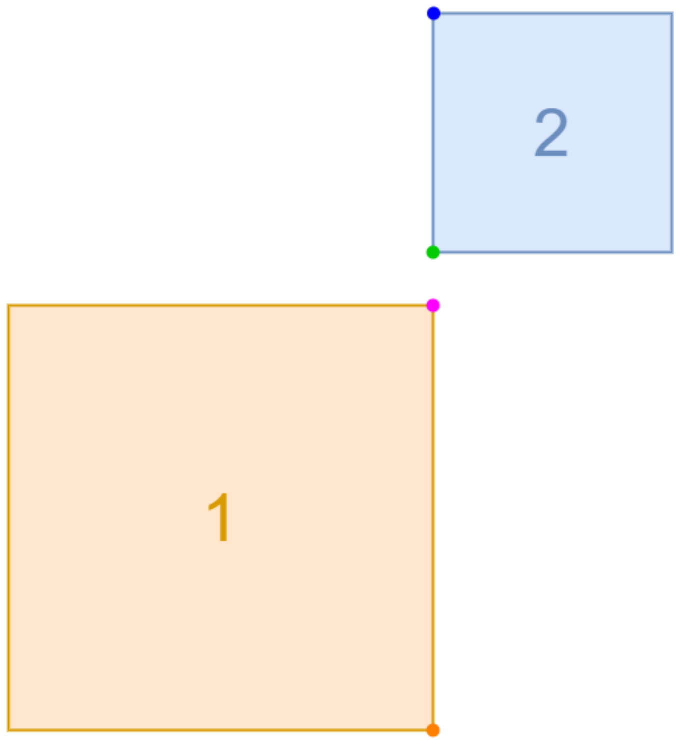


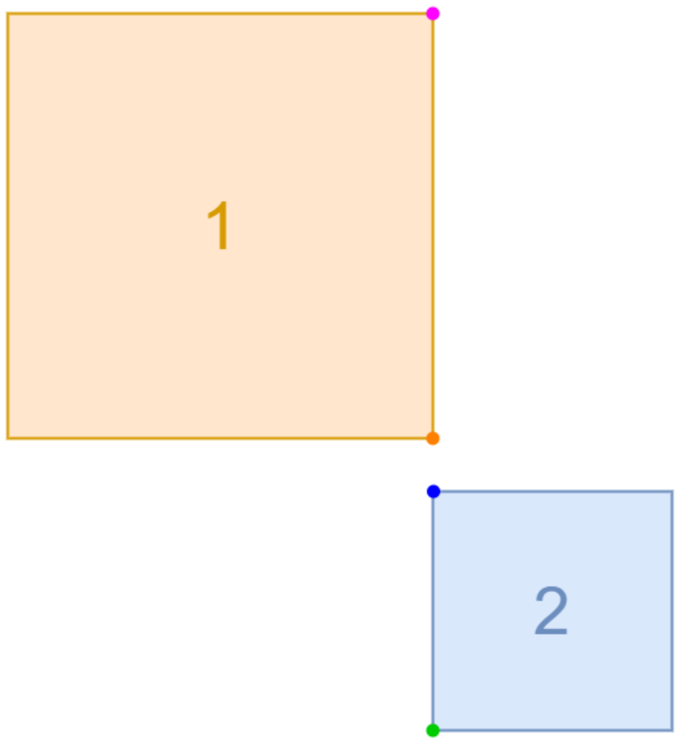
Figure 5.5: Example where the x-values of the top-right corner and the top-left corner are the same

5.2.2 Comparing y-values

Just having matching x-values does not guarantee vertical adjacency. There are cases where the x-values of the top-right corner and the top-left corner are the same but Rectangle 1 and Rectangle 2 are not vertically adjacent.



(a) Rectangle 2 is above Rectangle 1



(b) Rectangle 2 is below Rectangle 1

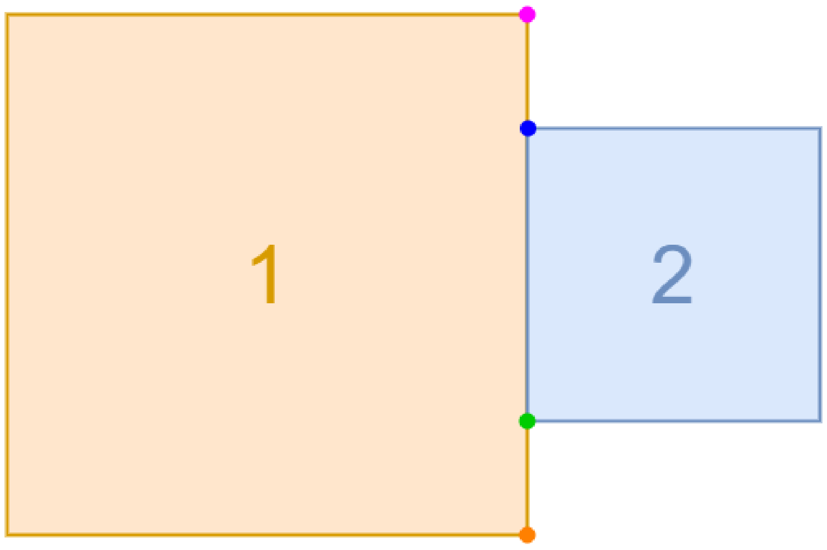
Figure 5.6: Examples where the x-values of the top-right corner and the top-left corner are the same but the rectangles are NOT vertically adjacent

To get rid of these cases, we must now compare the y-values of Rectangle 1's right corners and Rectangle 2's left corners.

Let the coordinates of the top-right and bottom-right corners, of Rectangle 1, be (x_{TR}, y_{TR}) and (x_{BR}, y_{BR}) respectively.

Let the coordinates of the top-left and bottom-left corners, of Rectangle 2, be (x_{TL}, y_{TL}) and (x_{BL}, y_{BL}) respectively.

Case 1: Both of Rectangle 2's left corners are within the range of Rectangle 1's right corners

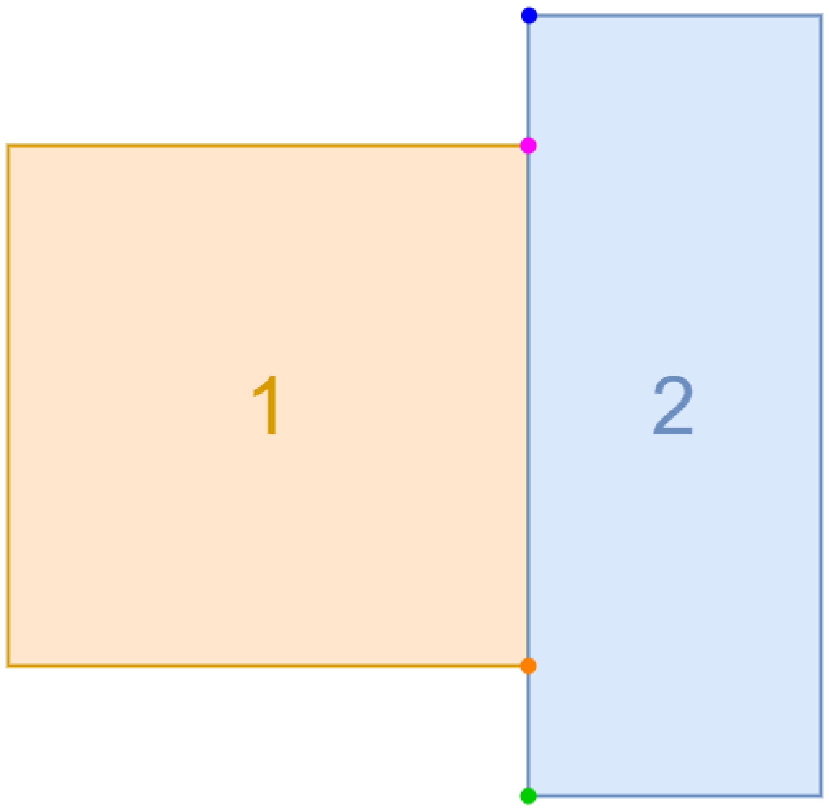


$$y_{BR} \leq y_{BL} < y_{TL} \leq y_{TR}$$

or

$$y_{BL}, y_{TL} \in [y_{BR}, y_{TR}], \text{ where } y_{BL} < y_{TL}$$

Case 2: Both of Rectangle 1's right corners are within the range of Rectangle 2's left corners

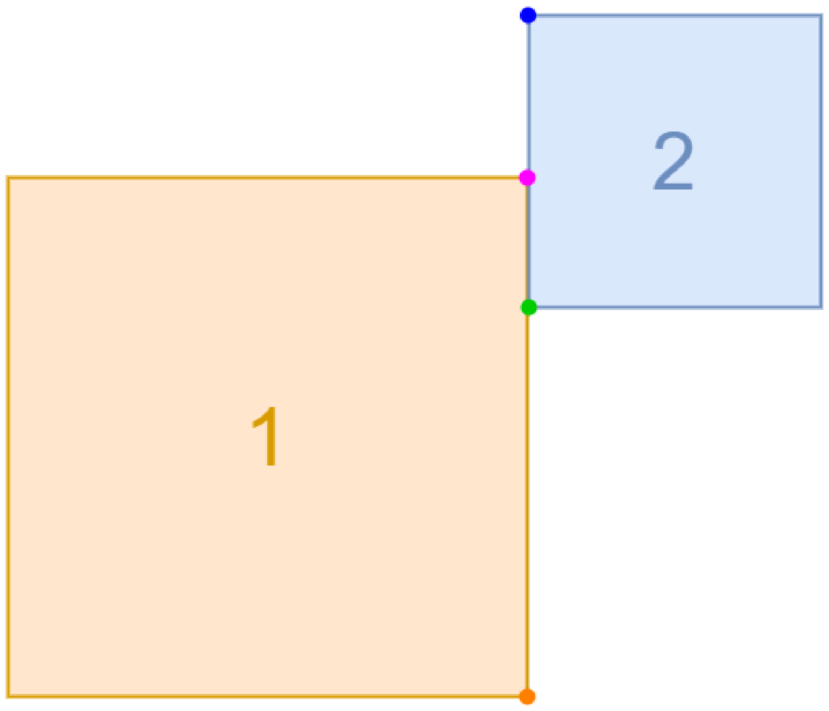


$$y_{BL} \leq y_{BR} < y_{TR} \leq y_{TL}$$

or

$$y_{BR}, y_{TR} \in [y_{BL}, y_{TL}], \text{ where } y_{BR} < y_{TR}$$

Case 3: Rectangle 2's bottom-left corner lies within the range of Rectangle 1's right corners

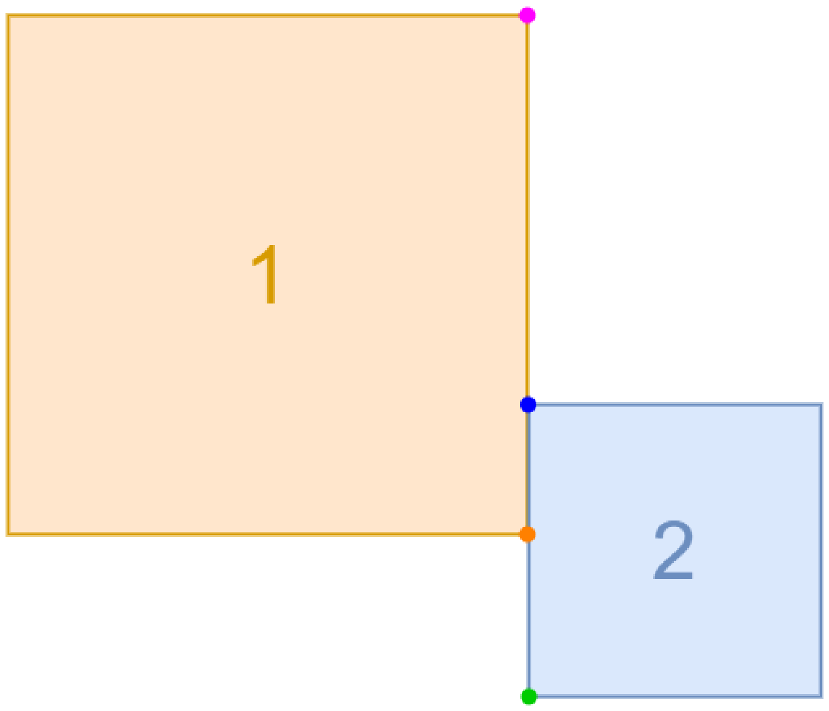


$$y_{BR} < y_{BL} < y_{TR}$$

or

$$y_{BL} \in (y_{BR}, y_{TR})$$

Case 4: Rectangle 2's top-left corner lies within the range of Rectangle 1's right corners



$$y_{BR} < y_{TL} < y_{TR}$$

or

$$y_{TL} \in (y_{BR}, y_{TR})$$

5.3 Details of the Vertical Adjacency

Once we have confirmed that Rectangle 2 is vertically adjacent to Rectangle 1, we need to record the details of the vertical adjacency.

The details of the vertical adjacency are:

1. The x-value where Rectangle 1 and Rectangle 2 meet
2. The range of which Rectangle 1 and Rectangle 2 are adjacent.
   In particular:
   - The y-value for the top of the adjacency, y_t
   - The y-value for the bottom of the adjacency y_b

   So Range $= [y_b, y_t]$

Detail 1 is simple. Since we know that the two rectangles are vertically adjacent, the x-value is simply the Rectangle 1's top-right corner.

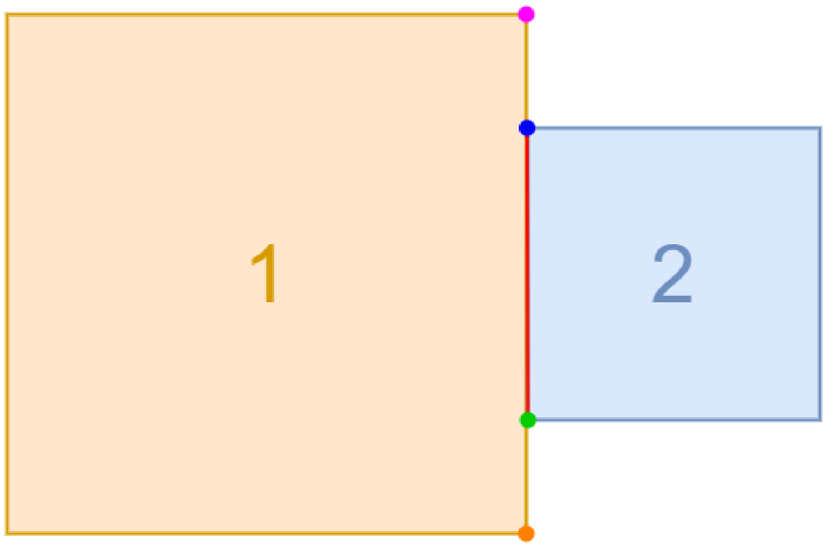
$$x = x_{TR}$$

Note: We can also use the bottom-right corner or Rectangle 1, or the left corners of Rectangle 2, to get the x-value where the two rectangles meet.

Detail 2, the range of which Rectangle 1 and Rectangle 2 are adjacent, changes for each case of y-values

Note: The region of adjacency is given by the red line ———

Case 1: Both of Rectangle 2's left corners are within the range of Rectangle 1's right corners



In this case, the range is simply the range of Rectangle 2's left corners.

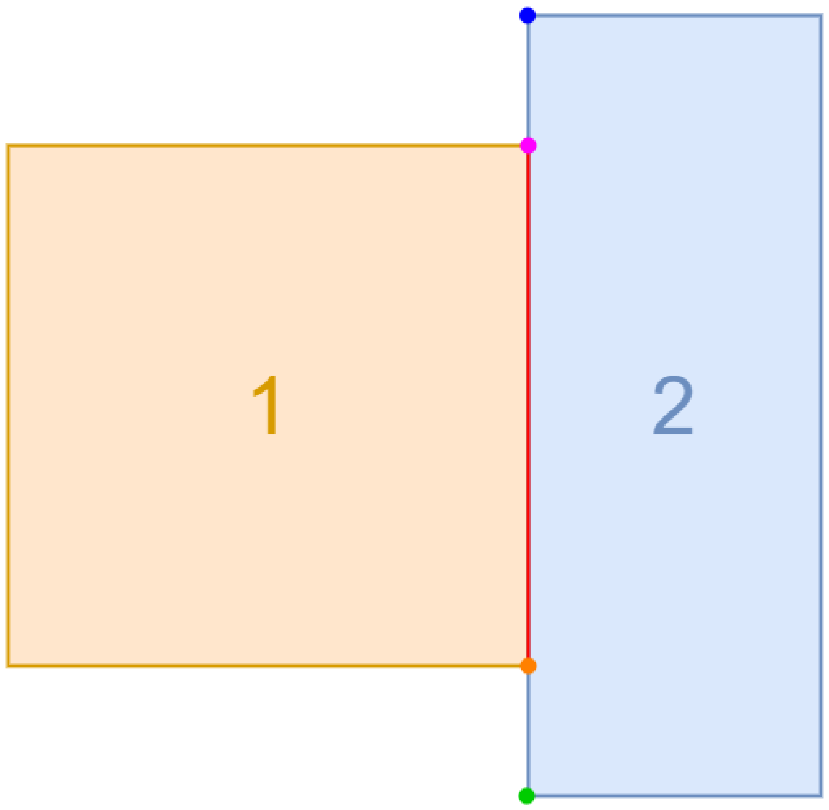
$$\text{Range} = [y_{BL}, y_{TL}]$$

In other words,

y_t is the y value of the top-left corner of Rectangle 2
$y_t = y_{TL}$

y_b is the y value of the bottom-left corner of Rectangle 2
$y_b = y_{BL}$

Case 2: Both of Rectangle 1's right corners are within the range of Rectangle 2's left corners



In this case, the range is simply the range of Rectangle 1's right corners.

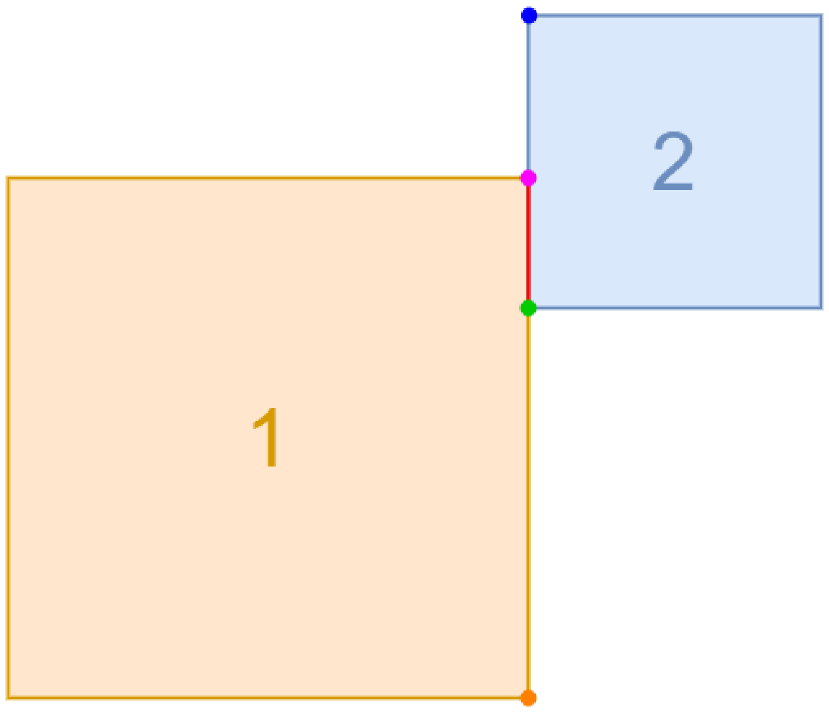
$$\text{Range} = [y_{BR}, y_{TR}]$$

In other words,

y_t is the y value of the top-right corner of Rectangle 1
$y_t = y_{TR}$

y_b is the y value of the bottom-right corner of Rectangle 1
$y_b = y_{BR}$

Case 3: Rectangle 2's bottom-left corner lies within the range of Rectangle 1's right corners



In this case, the range is the range between the top-right corner of Rectangle 1 and the bottom-left corner of Rectangle 2.

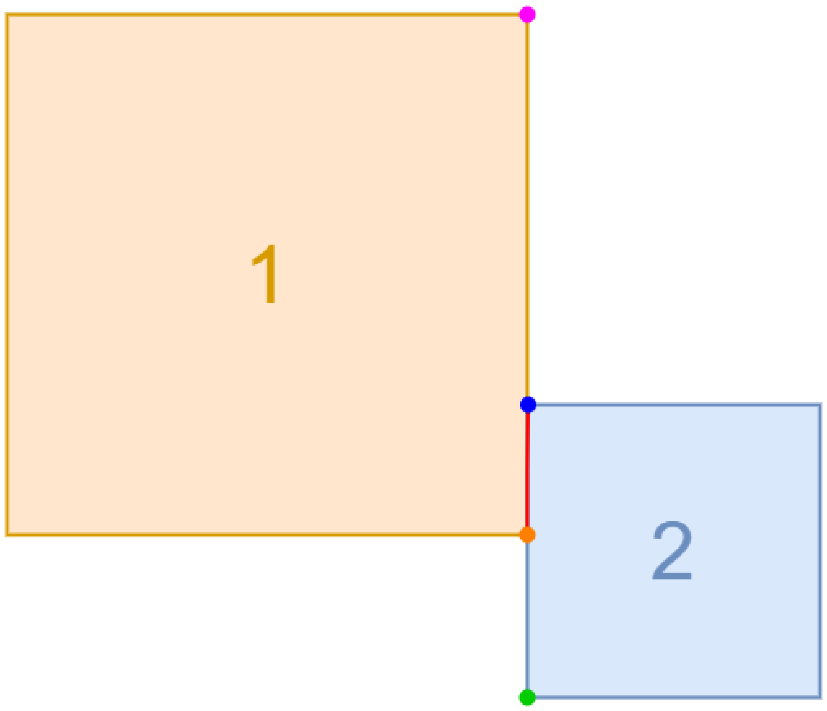
$$\text{Range} = [y_{BL}, y_{TR}]$$

In other words,

y_t is the y value of the top-right corner of Rectangle 1
$y_t = y_{TR}$

y_b is the y value of the bottom-left corner of Rectangle 2
$y_b = y_{BL}$

Case 4: Rectangle 2's top-left corner lies within the range of Rectangle 1's right corners



In this case, the range is the range between the top-left corner of Rectangle 2 and the bottom-right corner of Rectangle 1.

$$\text{Range} = [y_{BR}, y_{TL}]$$

In other words,
y_t is the y value of the top-left corner of Rectangle 2
$y_t = y_{TL}$

y_b is the y value of the bottom-right corner of Rectangle 1
$y_b = y_{BR}$

Chapter 6

The Brute Force Algorithm

We are tasked with coming up with a Brute Force algorithm to solve the problem presented to us. This algorithm is supposed to be the quick-and-simple approach to solving the problem. Not designed to be efficient but merely to be the obvious answer.

6.1 The Logic of the Brute Force Algorithm

To solve the problem that was proposed to us, the Brute Force Approach adheres to the following logic:

Essentially, we will be performing a linear search [4]. A linear search is the simplest approach employed to search for an element in a data set. It examines each element until it finds a match, starting at the beginning of the data set, until the end. The search is finished and terminated once the target element is located [1]. We will be modifying the linear search to solve the problem presented to us. Instead of searching for an element, we will be comparing a rectangle R to every other rectangle r in the rectangle list to see if rectangle r is vertically adjacent, to the right, to rectangle R.

For each rectangle R in the list of rectangles, we compare R to every other rectangle. For a rectangle r to be vertically adjacent to R, we first check if the x-coordinate of R's top-right corner is the same as the x-coordinate of r's top-left corner. If they are the same, we then check for one of four cases:

1. The y-values for r's top-left and bottom-left corners lie between the y-values for R's top-right and bottom-right corners

2. The y-values for R's top-right and bottom-right corners lie between the y-values for r's top-left and bottom-left corners

3. The y-value for r's bottom-left corner lies between the y-values for R's top-right and bottom-right corners

4. The y-value for r's top-left corner lies between the y-values for R's top-right and bottom-right corners

If any of these four cases are true, then r is vertically adjacent to the right of R, and we can add r to the adjacency list for R along with the details of the adjacency.

For a more in-depth explanation for determining vertical adjacency and the details of the vertical adjacency, see **chapter 5**

6.2 Implementation of the Brute Force Algorithm

To implement this Brute Force algorithm, follow this pseudo-code:

Key:
TR - Top Right
BR - Bottom Right
TL - Top Left
BL - Bottom Right

Note: $r.TR.x$ means the x-coordinate of top-right corner of rectangle r

Algorithm 2: Brute Force Algorithm

```
Input: list of rectangles
Output: list of adjacency lists
foreach rectangle $current$ in the rectangle list do
    foreach rectangle $other$ in the rectangle list do
        if $current \neq other$ then
            if $current.TR.x = other.BL.x$ then
                $adjacent \leftarrow$ False
                if $current.BR.y \leq other.BL.y < other.TL.y \leq current.TR.y$ then
                    $adjacent \leftarrow$ True
                    $y_t \leftarrow other.TL.y$
                    $y_b \leftarrow other.BL.y$
                end
                else if $other.BL.y \leq current.BR.y < current.TR.y \leq other.TL.y$ then
                    $adjacent \leftarrow$ True
                    $y_t \leftarrow current.TR.y$
                    $y_b \leftarrow current.BR.y$
                end
                else if $current.BR.y \leq other.BL.y \leq current.TR.y$ then
                    $adjacent \leftarrow$ True
                    $y_t \leftarrow current.TR.y$
                    $y_b \leftarrow other.BL.y$
                end
                else if $current.BR.y \leq other.TL.y \leq current.TR.y$ then
                    $adjacent \leftarrow$ True
                    $y_t \leftarrow other.TL.y$
                    $y_b \leftarrow current.BR.y$
                end
                if $adjacent$ is True then
                    Add $other$ to adjacency list for $current$
                end
            end
        end
    end
end
return list of adjacency lists
```

For this experiment, we implemented our Brute Force algorithm in `C++`. This implementation can be found in the file `BruteForce.cpp`.

6.3 Theoretical Analysis of the Brute Force Algorithm

6.3.1 Concluding the Time Complexity

Following the logic and the pseudo-code of the Brute Force Algorithm, we can see that we have a nested `for` loop. Meaning we have a `for` loop inside of a `for` loop. The outer loop iterates through the entire list of n rectangles. The inner loop also iterates through the entire list of n rectangles. So for every iteration up to n, we perform n iterations.

Therefore, the time complexity of this Brute Force algorithm is $O(n^2)$.

This time complexity is inefficient and less than ideal, but for the Brute Force algorithm, this is suitable and to be expected.

According to our implementation of the Brute Force Algorithm, we do not have a Best, Worst, and Average Case.

Theoretically, due to the nature of the implementation, this algorithm will compute in $O(n^2)$ time no matter what. Even when we determine that a rectangle is vertically adjacent to another, we still iterate through the entire list. There is no case where we do not iterate n^2 times.

6.3.2 Concluding the Space Complexity

This algorithm utilises two lists of size n where n is the number of rectangles. One list is the list that stores all the rectangles. The other list is the list that stores the output, the list of adjacency lists.

$$n + n = 2n \in O(n)$$

Therefore, the space complexity of this Brute Force algorithm is $O(n)$.

Chapter 7

The Improved Method Algorithm

We are tasked with coming up with a more efficient algorithm than the Brute Force algorithm to solve the problem presented to us. This algorithm is supposed to minimise computational time. Designed to be efficient and a far better approach than the obvious answer.

7.1 The Logic of the Improved Method Algorithm

To solve the problem that was proposed to us, the Improved Method Approach adheres to the following logic:

Essentially, we will be performing a line sweep [7]. In computational geometry, a line sweep algorithm, or plane sweep algorithm, is an algorithmic paradigm that uses a conceptual sweep line or sweep surface to solve various problems in Euclidean space. It is one of the critical techniques in computational geometry [2]. We will be modifying the line sweep algorithm to solve the problem presented to us.

For this algorithm, we utilise a data structure called an Event.

An event is structured as follows:

- **Rectangle** – This is a reference to the rectangle that the event was created from.
- **Type** – Indicates which corner of the rectangle this event represents. Events can only be one of two types.

$$\text{Type} \in \{\text{Bottom Left}, \text{Top Right}\}$$

- **Point** – The (x, y) coordinates of the corner of the rectangle that the event represents.

Now that we know what events are, we need to create a list of events. This list of events is how we are going to perform our line sweep.

We need to populate our list of events. We iterate through our list of rectangles. For each rectangle, in the rectangle list, we create two events:

1. A Bottom Left event
2. A Top Right event

We add both of these events to the events list.

The events list should be ordered in the following way:

- Event points with lesser x-values should come before event points with greater x-values.

  Let a and b be two different events.

  If $a.point.x < b.point.x$
  Then a comes before b in the events list.

- For event points with the same x-values, we look further to the y-values of the event points. Event points with lesser y-values should come before event points with greater y-values.

  Let a and b be two different events with $a.point.x = b.point.x$.

  If $a.point.y < b.point.y$
  Then a comes before b in the events list.

- For events with the same x- and y-values, we finally look at the types of the event points. Events that are of type Bottom Left should come before events that are of type Top Right.

  Let a and b be two different events with $a.point = b.point$.

  If $a.type =$ Bottom Left and $b.type =$ Top Right
  Then a comes before b in the events list.

Note: Since we know that the rectangles do not overlap, we know that every event is unique. In other words, there will never be a situation where two events are of the same type AND have the same coordinates.

Once we have our sorted events list, we create an empty list, that will store rectangles, called candidates, that will store all the rectangles that could potentially be adjacent. We now iterate through our events list.

This iteration through our events list is our "line sweep". Conceptually, we can imagine a vertical, straight-line 'sweeping' from left to right of our defined Euclidean space of rectangles. The sorted list of events makes it so that the events are arranged in the order in which the sweeping vertical line would have encountered the events.

For each event e in the events list, we do the following:
Let r be the rectangle that e references.

- If e is a Bottom Left event, then we add r to our candidates list.

- If e is a Top Right event, then we remove r from our candidates list. After that, we iterate through all the rectangles in the candidates list.

  For each candidate c in candidates, we check for one of four cases:

  1. The y-values for r's top-left and bottom-left corners lie between the y-values for c's top-right and bottom-right corners
  2. The y-values for c's top-right and bottom-right corners lie between the y-values for r's top-left and bottom-left corners

3. The y-value for r's bottom-left corner lies between the y-values for c's top-right and bottom-right corners
4. The y-value for r's top-left corner lies between the y-values for c's top-right and bottom-right corners

If any of these four cases are true, then c is vertically adjacent to the right of r, and we can add c to the adjacency list for r along with the details of the adjacency.

> For a more in-depth explanation for determining vertical adjacency and the details of the vertical adjacency, see **chapter 5**

The logic here is that only rectangles that are in the same region as the current rectangle (the rectangle that is currently being considered) could possibly be adjacent to the current rectangle. Rectangles that have already been encountered, or rectangles that are yet to be encountered, cannot be adjacent to the current rectangle, and subsequently should not be considered for adjacency.

This algorithm will only consider only the rectangles in the region, of the line sweep, for adjacency, decreasing our search space from the entire rectangle list to only rectangles that are in the same region as the current rectangle. In other words, our search space has been minimised significantly.

7.2 Implementation of the Improved Method Algorithm

Understanding Required Data Structures

Before we implement our Improved Method Algorithm, we first need to understand sets as a data structure. Sets are important as a data structure for several reasons:

- Uniqueness: Sets ensure that each element is unique within the collection.
- Efficient search: Sets provide efficient search operations. The underlying implementation of sets typically uses binary search trees to store elements. These data structures allow for fast lookup times, making sets suitable for applications that require frequent searches.
- Ordering: Sets offer an ordered collection of elements. The ordering can be based on a default comparison function (less-than operator, $<$) or a custom comparison function provided by the user.

  > Since we are using a custom data structure, we have to provide a custom comparison. For this experiment, the custom comparison's logic can be found in Section 7.1.

  This feature is beneficial when you need to iterate over the elements in a specific order or find the minimum or maximum element efficiently.
- Efficient insertion and deletion: Sets provide efficient insertion and deletion operations. The underlying data structures used by sets are designed to maintain balance and ensure fast modifications. As a result, adding or removing elements from a set has good time complexity, typically logarithmic $\big(O(\log n)\big)$.
- Iterator support: Sets offer iterators that allow you to traverse the elements in a sorted order. This iterator support enables you to perform operations on each unique element in a controlled and predictable manner.

Overall, sets are important as a data structure because they provide unique and ordered collections with efficient search, insertion, deletion, and set operations. They enable you to solve a wide range of problems efficiently and effectively.

In our implementation, we will be using sets instead of lists. Due to this change in the data structure, we do not need to sort our events, as they will be inserted into their correct position.

Implementation

To implement this Improved Method algorithm, follow this pseudo-code:

Algorithm 3: Improved Method Algorithm

```
Input: list of rectangles
Output: list of adjacency lists
Initialise the set events
foreach rectangle r in rectangle list do
    Create 2 Events. One for the bottom left corner of r, and another for the top right corner of r
    Add these 2 events to events
end
Initialise the set candidates
foreach event in events do
    if event is a Bottom Left event of a rectangle r then
        Add r to candidates
    end
    else event is a Top Right event of a rectangle r
        Remove r from candidates
        current ← r
        foreach candidate in candidates do
            adjacent ←False
            if $current.BR.y \leq candidate.BL.y < candidate.TL.y \leq current.TR.y$ then
                adjacent ← True
                $y_t \leftarrow candidate.TL.y$
                $y_b \leftarrow candidate.BL.y$
            end
            else if $candidate.BL.y \leq current.BR.y < current.TR.y \leq candidate.TL.y$ then
                adjacent ← True
                $y_t \leftarrow current.TR.y$
                $y_b \leftarrow current.BR.y$
            end
            else if $current.BR.y \leq candidate.BL.y \leq current.TR.y$ then
                adjacent ← True
                $y_t \leftarrow current.TR.y$
                $y_b \leftarrow candidate.BL.y$
            end
            else if $current.BR.y \leq candidate.TL.y \leq current.TR.y$ then
                adjacent ← True
                $y_t \leftarrow candidate.TL.y$
                $y_b \leftarrow current.BR.y$
            end
            if adjacent is True then
                Add candidate to adjacency list for current
            end
        end
    end
end
return list of adjacency lists
```

7.3 Theoretical Analysis of the Improved Method Algorithm

Following the logic of the Improved Method Algorithm, we can see that we do, in fact, have best, average, and worst case time complexities. Although there are best, average, and worst cases for our Improved Method Algorithm, the time complexities for all cases is $O(n \log n)$.

7.3.1 Concluding the Time Complexity

We initially traverse the rectangle list of n rectangles. This takes $O(n)$ time. For each rectangle in the rectangle list, we create two events and add these two events to the events set. Adding items to the set will initially take $O(1)$ time when the set is empty, and then will subsequently take $O(\log n)$ time. So, thus far, our traversal of the rectangles list and adding all the necessary events to the events set takes $O(n \log n)$ time.

Next, we iterate through our events set. Since for each rectangle, we create two events, if there are n rectangles, there are $2n$ events in the events set. Iterating through the event set takes $O(n)$ time since $f(n) = 2n \in O(n)$.
For each iteration, we do one of two operations:

Let m be the size of the candidates set.

- If the event is a Bottom Left event, we add the rectangle, that the event references, to the candidates set. Adding items to the set will initially take $O(1)$ time when the set is empty, and then will subsequently take $O(\log m)$ time.
- If the event is a Top Right event, we remove the rectangle, that the event references, from the candidates set. Removing items from the set will take $O(1)$ time if there is only one element in the set, otherwise, it will take $O(\log m)$ time.

  After we have removed the rectangle from the candidates set, we iterate through the candidates set to check if any of the candidates are adjacent to the rectangle. Iterating through the candidates set will take $O(m)$ time.

  Therefore this branch takes $m + \log m \in O(m)$ time.

Since the size of the candidates set will always be smaller than the size of the events set, we can assume that

$$\log m < m < \log n$$

Thus, we can say that the two operations, of the events set iteration, will not take more than $O(\log n)$ time.

Then, we are saying that we have an outer loop that runs in linear $\big(O(n)\big)$ time. Within the loop, we perform computations that runs in logarithmic $\big(O(\log n)\big)$ time. Therefore, iterating through the events set and performing the necessary operations, takes linearithmic $\big(O(n \log n)\big)$ time.

Therefore, overall, our Improved Method algorithm runs in $n \log n + n \log n = 2n \log n \in O(n \log n)$ time.

Best Case

The best case time complexity occurs when no rectangles, in the rectangle list, are adjacent.

In this, though, the time complexity remains $O(n \log n)$. The algorithm will simply complete slightly faster than average.

Average Case

The average case time complexity occurs a rectangle, in the rectangle list, could or could not have adjacencies.

In this, though, the time complexity remains $O(n \log n)$. This is the expected case, the case that will occur most often.

Worst Case

The worst case time complexity occurs when all rectangles, (excluding the rectangles on the extreme right end of the defined Euclidean space) in the rectangle list, have adjacencies.

In this, though, the time complexity remains $O(n \log n)$. The algorithm will simply complete slightly slower than average.

7.3.2 Concluding the Space Complexity

This algorithm utilises two lists of size n where n is the number of rectangles. One list is the list that stores all the rectangles. The other list is the list that stores the output, the list of adjacency lists.

Our Improved Method algorithm also utilises two sets. Our events set is of size $2n$. Our candidates is of size m where m is the number of items in the candidates set.

$$n + n + 2n + m = 4n + m < 4n + n = 5n \in O(n)$$

Therefore, the space complexity of our Improved Method algorithm is $O(n)$.

Chapter 8

The Output

The output that our Brute Force and Improved Method algorithms produce is in the following structure:

[Rectangle Number], [Number of Adjacent Rectangles], [Rectangle Number of Adjacent Rectangle], $[x]$, $[y_b]$, $[y_t]$

Lets look at the data from *sample_input.csv*

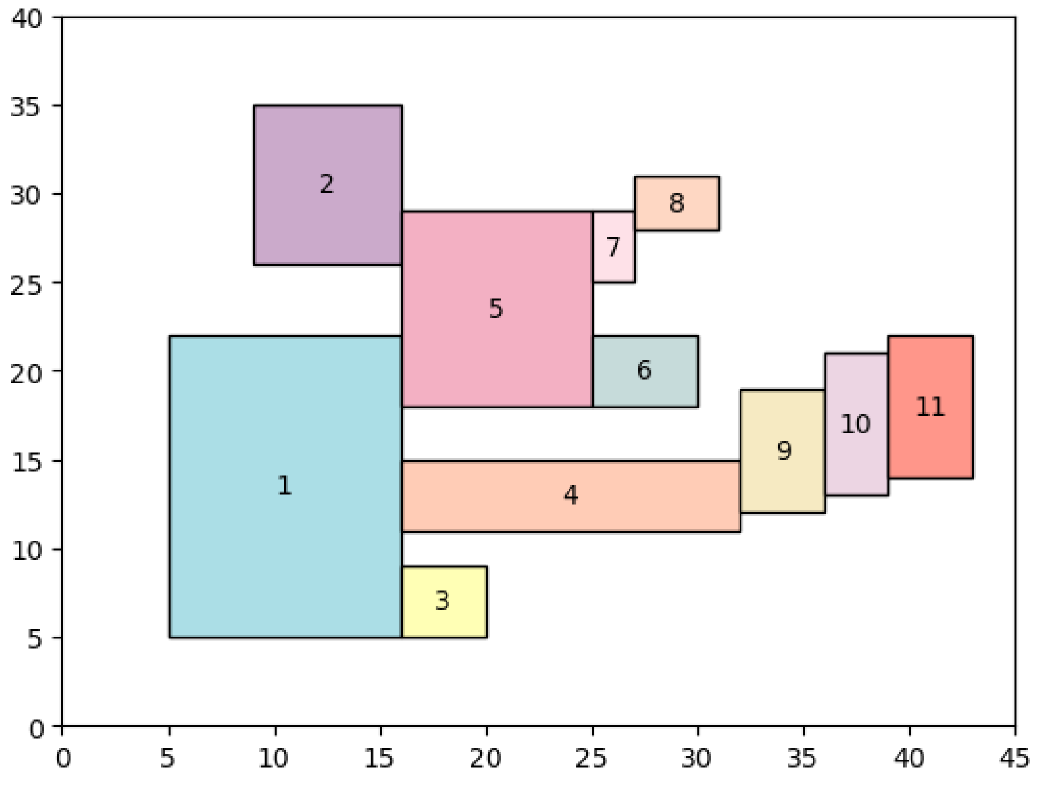


Figure 8.1: Visual Representation of the Data from *sample_input.csv*

Our programs took this input and produced the contents of *sample_output.csv*.

To see the contents of *sample_output.csv*, refer to Figure B.2

This output data can be interpreted as follows:

Rectangle 1 is adjacent to 3 rectangles: Rectangle 3 from (16, 5) to (16,9), Rectangle 4 from (16,11) to (16,15), Rectangle 5 from (16,18) to (16,22)
Rectangle 2 is adjacent to 1 rectangle: Rectangle 5 from (16,26) to (16,29)
Rectangle 3 has no adjacent rectangles
Rectangle 4 is adjacent to 1 rectangle: Rectangle 9 from (32,12) to (32,15)
Rectangle 5 is adjacent to 2 rectangles: Rectangle 6 from (25,18) to (25,22), Rectangle 7 from (25,25) to (25,29)
Rectangle 6 has no adjacent rectangles
Rectangle 7 is adjacent to 1 rectangle: Rectangle 8 from (27,28) to (27,29)
Rectangle 8 has no adjacent rectangles
Rectangle 9 is adjacent to 1 rectangle: Rectangle 10 from (36,13) to (36,19)
Rectangle 10 is adjacent to 1 rectangle: Rectangle 11 from (39,14) to (39,21)
Rectangle 11 has no adjacent rectangles

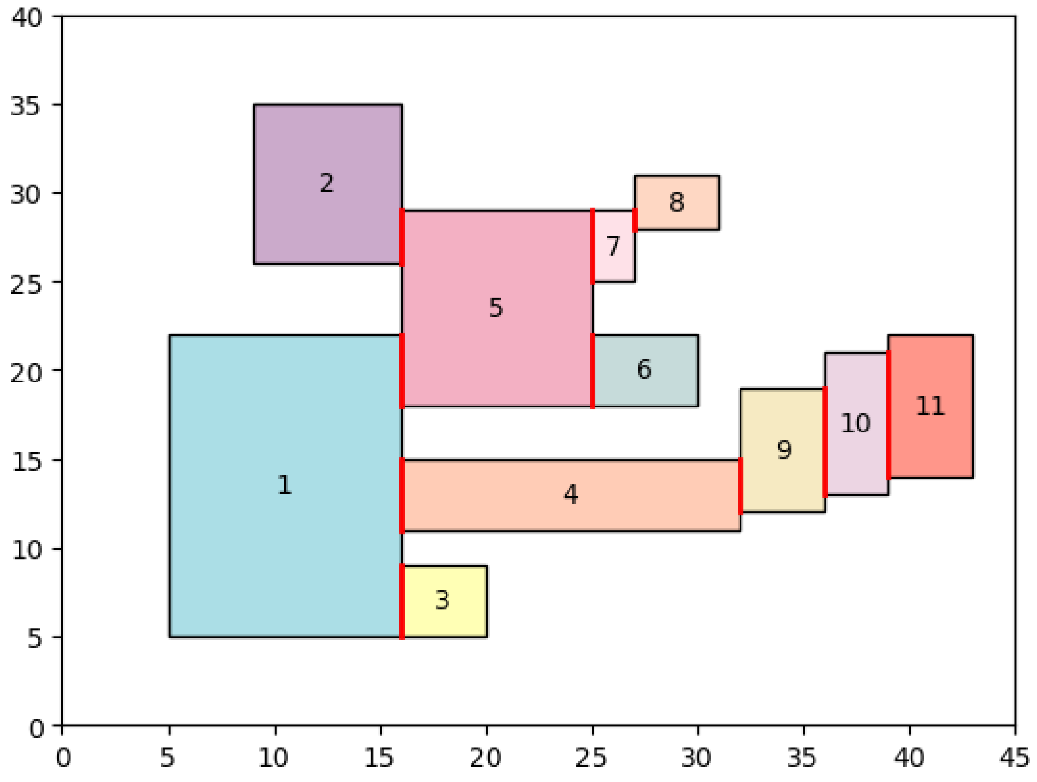


Figure 8.2: Visual Interpretation of the Data from *sample_output.csv*

Note: The region of adjacency is given by the red line

Chapter 9

Methodology

9.1 Strategy

9.1.1 Brute Force Strategy

The overall strategy for undergoing this experiment is as follows:

1. Inside the `GenerateRectangles.cpp` file, we specified the directory we want our input data files to be stored in. We also specified the number of input files we want to generate. For this experiment, we wanted to generate 18000 input files.

2. Once we made the necessary specifications in our `GenerateRectangles.cpp` code, we compiled our `C++` file to produce `Generate_Rectangles.exe`.

3. Now that we had our compiled application, `Generate_Rectangles`, we ran this program.

4. After the `Generate_Rectangles` application was done running and all the input data files had been generated and stored in the directory we specified, inside the `BruteForce.cpp` file, we specified the directory for where all the input files are stored, and we specified the directories for where we wanted our output data files to be stored as well as where we wanted our analysis file (stores the input size and how long the program took the produce an output) to be stored. We also specified the number of input files we wanted our Brute Force program to look at. In this experiment, we wanted our program too look at all 18000 input files.

5. Once we had made the necessary specifications in our `BruteForce.cpp` code, we compiled our `C++` file to produce `Brute_Force.exe`.

6. Now that we had our compiled application, `Brute_Force.exe`, we ran this program.

7. Once the Brute Force program was done running, we verified the results using random data files.

8. Once we determined the results were correct, we can move on to the analysis of the results.

9.1.2 Improved Method Strategy

The overall strategy for undergoing this experiment is as follows:

1. In Phase 1 of this assignment, we used the file `GenerateRectangles.cpp` to generate our input data. We generated and stored this data in a specified directory. Thus, we did not have to generate more input data files, we simply used the same input files that we used for the Brute Force program.

2. inside the `ImprovedMethod.cpp` file, we specified the directory for where all the input files are stored, and we specified the directories for where we wanted our output data files to be stored as well as where we wanted our analysis file (stores the input size and how long the program took the produce an output) to be stored. We also specified the number of input files we wanted our Improved Method program to look at. In this experiment, we wanted our program to look at all 18000 input files.
3. Once we had made the necessary specifications in our `ImprovedMethod.cpp` code, we compiled our `C++` file to produce `Improved_Method.exe`.
4. Now that we had our compiled application, `Improved_Method.exe`, we ran this program.
5. Once the Improved Method program was done running, we verified the results, of random data files, by looking at visual representations of the input, and also comparing the output of this program to the output of the Brute Force program.
6. Once we determined the results were correct, we can move on to the analysis of the results.

9.2 Hardware

These are the following hardware specifications that the programs were run on.

Device	HP ProBook 450 G5
Processor	Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz 1.99 GHz
Installed RAM	8,00 GB (7,89 GB usable)
System Type	64-bit operating system, x64-based processor

Hardware Specifications

9.3 Software

The following are the operating system specifications that the programs were run on.

Operating System	Windows 11
Edition	Windows 11 Pro
OS Build	22621.1555

Operating System Specifications

The following is the software that the programs were developed on.

Code Editor	Microsoft Visual Studio Code

Table 9.1: Software Specifications

9.4 Programming Language

[9] For this experiment, the programs for generating the input files, the Brute Force program, and the Improved Method program, were written and programmed in `C++`.
There are several reasons why we chose to implement this code in `C++`:

- Performance: `C++` is a compiled language, which means that it can be highly optimized for performance. `C++` code can often run faster than code written in interpreted languages like `Python` or `Ruby`.
- Low-level control: `C++` allows for low-level memory management and direct access to hardware, making it a good choice for systems programming and other applications where you need fine-grained control over system resources.
- Portability: `C++` code can be compiled to run on a wide range of platforms, including desktop computers, servers, mobile devices, and even embedded systems.
- Large community: `C++` has been around for over 30 years and has a large and active community of developers, which means that there are many resources available for learning and troubleshooting.
- Interoperability: `C++` can be easily integrated with other languages, including `C`, `Python`, and `Java`, making it a good choice for projects that require interoperability between different programming languages.

Overall, `C++` is a powerful and versatile language that can be a good choice for a wide range of applications, from high-performance computing to systems programming to game development.

9.5 Data Structures

The file for generating the input data, the file for performing the Brute Force Algorithm, and the file for performing the Improved Method Algorithm, were all written and programmed in `C++`. These files were `GenerateRectangles.cpp`, `BruteForce.cpp`, and `ImprovedMethod.cpp` respectively.

For all programs, we used Vectors [6] and custom Rectangle classes. Additionally, in the Improved Method program, we used a custom Event class and Sets [5].
We used vectors due to the flexibility of the data structure.

In `C++`, vectors and arrays are both used to store sequences of elements. However, there are several reasons why we might choose vectors over arrays:

- Dynamic size: Vectors can be resized dynamically at runtime, whereas arrays have a fixed size that is determined at compile time.
- Automatic memory management: Vectors manage their own memory allocation and deallocation, so you don't need to worry about memory management as you do with arrays.
- Easy to pass to functions: Vectors can be easily passed to functions as parameters, whereas arrays can be more complicated to pass.
- Range checking: Vectors perform bounds checking automatically, so you can avoid common errors like reading or writing outside the bounds of an array.
- Additional functionality: Vectors come with additional functionality, such as the ability to easily add or remove elements from the beginning or end of the sequence, sort elements, and perform other operations.

Overall, vectors are a more flexible and convenient option for storing sequences of elements in C++.

In `C++`, sets are a container class provided by the Standard Template Library (STL) that stores unique elements in a specific order. Here are a few reasons why we might choose to use sets in `C++`:

- Unique elements: Sets guarantee that each element is unique, meaning you cannot have duplicate values. If you need to maintain a collection of distinct elements and quickly check for uniqueness, sets are a suitable choice.
- Ordered collection: Sets maintain the elements in a specific order. By default, they use a strict weak ordering provided by the less-than operator (<). You can also customize the sorting order by providing your own comparison function or using a different container class, such as `std::set` or `std::unordered_set`.
- Fast search: Sets offer fast search operations using a binary search algorithm. The time complexity for searching an element in a set is logarithmic ($O(\log n)$), which makes it efficient even for large collections.
- Efficient insertion and deletion: Sets allow efficient insertion and deletion of elements. Adding an element to a set or removing it has a time complexity of logarithmic order ($O(\log n)$), similar to searching.
- Iterating over unique elements: Sets provide iterators that allow you to traverse the elements in ascending or descending order. This can be useful when you need to process each unique element in a sorted manner.

Overall, sets are valuable when you need to maintain a collection of unique elements, require them to be ordered, and frequently perform operations like searching, insertion, and deletion efficiently.

9.5.1 Program for Generating the Input

For the file `GenerateRectangles.cpp`, we used the following data structures:

- A custom rectangle class that stores the necessary information of the rectangle.

  Refer to Section C.1 to see the implementation of the custom rectangle class.

- A vector [6] for storing the list of rectangles which was to be the output for this program.

  ```
  std::vector<Rectangle> rectangles;
  ```

  This is a vector storing objects of the Rectangle class.

9.5.2 Brute Force Program

For the file `BruteForce.cpp`, we used the following data structures:

- A custom Rectangle class that stores the necessary information of the rectangle.

  Refer to Section C.2 to see the implementation of the custom rectangle class.

- Vectors [6] for
  - storing the list of rectangles which was to be the input for this program.

    ```
    std::vector<Rectangle> rectangles;
    ```

    This is a vector storing objects of the Rectangle class.

  - storing the adjacency lists for each of the rectangles which were to be the output for this program.

    ```
    std::vector<std::string> output;
    ```

    This is a vector storing the output of the Brute Force program.

9.5.3 Improved Method Program

For the file `ImprovedMethod.cpp`, we used the following data structures:

- A custom Rectangle class that stores the necessary information of the rectangle.

  Refer to Section C.3 to see the implementation of the custom rectangle class.

- A custom Event class that stores the necessary information of the event.

  Refer to Section D to see the implementation of the custom event class.

- Vectors [6] for
  - storing the list of rectangles which was to be the input for this program.

    ```
    std::vector<Rectangle> rectangles;
    ```

    This is a vector storing objects of the Rectangle class.
  - storing the adjacency lists for each of the rectangles which were to be the output for this program.

    ```
    std::vector<std::string> output;
    ```

    This is a vector storing the output of the Improved Method program.

- Sets [5] for
  - storing the list of events.

    ```
    std:set<Event> events;
    ```

    This is a set for storing objects of the Event class.
  - storing the candidates for adjacency. This is a set storing rectangles that could potentially be adjacent to other rectangles.

    ```
    std::set<Rectangle> candidates;
    ```

    This is a set for storing objects of the Rectangle class.
  - storing details of adjacencies.

    ```
    std::set<Rectangle::Details> adjacencies;
    ```

    This is a set for storing objects of the Detail subclass. This set is the adjacency list for a given Rectangle.
    Details is a subclass within the Rectangle class that stores the details of the adjacencies.

    Refer to Section C.3 to see the implementation of the Detail subclass.

9.6 Timing

Since both the Brute Force and Improved Method programs were coded in `C++`, we make use of the `Chrono` library for `C++` [3]. To use the `Chrono` library, in the headers section of our `C++` program, we included the heading **#include** <chrono>.

9.6.1 Brute Force Program

Before and after we execute our function that executes our Brute Force algorithm, and generates the data, we add the code

```
auto start = std::chrono::high_resolution_clock::now();
std::vector<std::string> output = getAdjacencies(rectangles);
auto stop = std::chrono::high_resolution_clock::now();
```

like so.

9.6.2 Improved Method Algorithm

Before and after we execute our function that executes our Improved Method algorithm, and generates the data, we add the code

```
auto start = std::chrono::high_resolution_clock::now();
std::vector<std::string> output = ImprovedMethod(rectangles);
auto stop = std::chrono::high_resolution_clock::now();
```

like so.

9.6.3 Recording the Timings

We record the execution times of the algorithms and add these timings to the respective analysis files. To add these execution times to our analysis files, we implement the following code.

```
auto duration =
    std::chrono::duration_cast<std::chrono::microseconds>(stop - start);
analysisFile << numRectangles << ',' << duration.count() / 1000.0 << '\n';
```

Note: we divide the duration count by 1000 as the duration is measured in microseconds and we want our units of time to be in milliseconds. It was found that measuring in microseconds was more accurate than measuring in milliseconds directly.

9.7 Data

9.7.1 Input

For this experiment, we generated 18000 input *.csv* files. They are each titled after the number of rectangles in the rectangle list there are.

For example, input file *in1000.csv* would be a file containing 1000 rectangles.

So we have input files that span from 1 rectangle, in the list, to 18000 rectangles, increasing by 1 incremental rectangle. We did this to have a wide variety of data. 18000 input files should be more than enough to ascertain the time complexities of the two algorithms.

9.7.2 Output

Our programs, `BruteForce.cpp` and `ImprovedMethod.cpp`, produce two kinds of output:

- Adjacency output.
- Output for empirical analysis.

Adjacency Output

This is the output as seen in Chapter 8.

Having been given 18000 input files, our programs produced 18000 output files each.

Output for Empirical Analysis

This output was a single *.csv* file that tracked two pieces of information:

1. The input size (the number of rectangles in the rectangle list).
2. The time taken for our program to produce an output (measured in milliseconds).

The empirical analysis files for our Brute Force and Improved Method programs were used to produce Figure 10.1 and Figure 10.3 respectively.

Chapter 10

Evaluation

10.1 Brute Force

10.1.1 Results

These are the results of our Brute Force program.

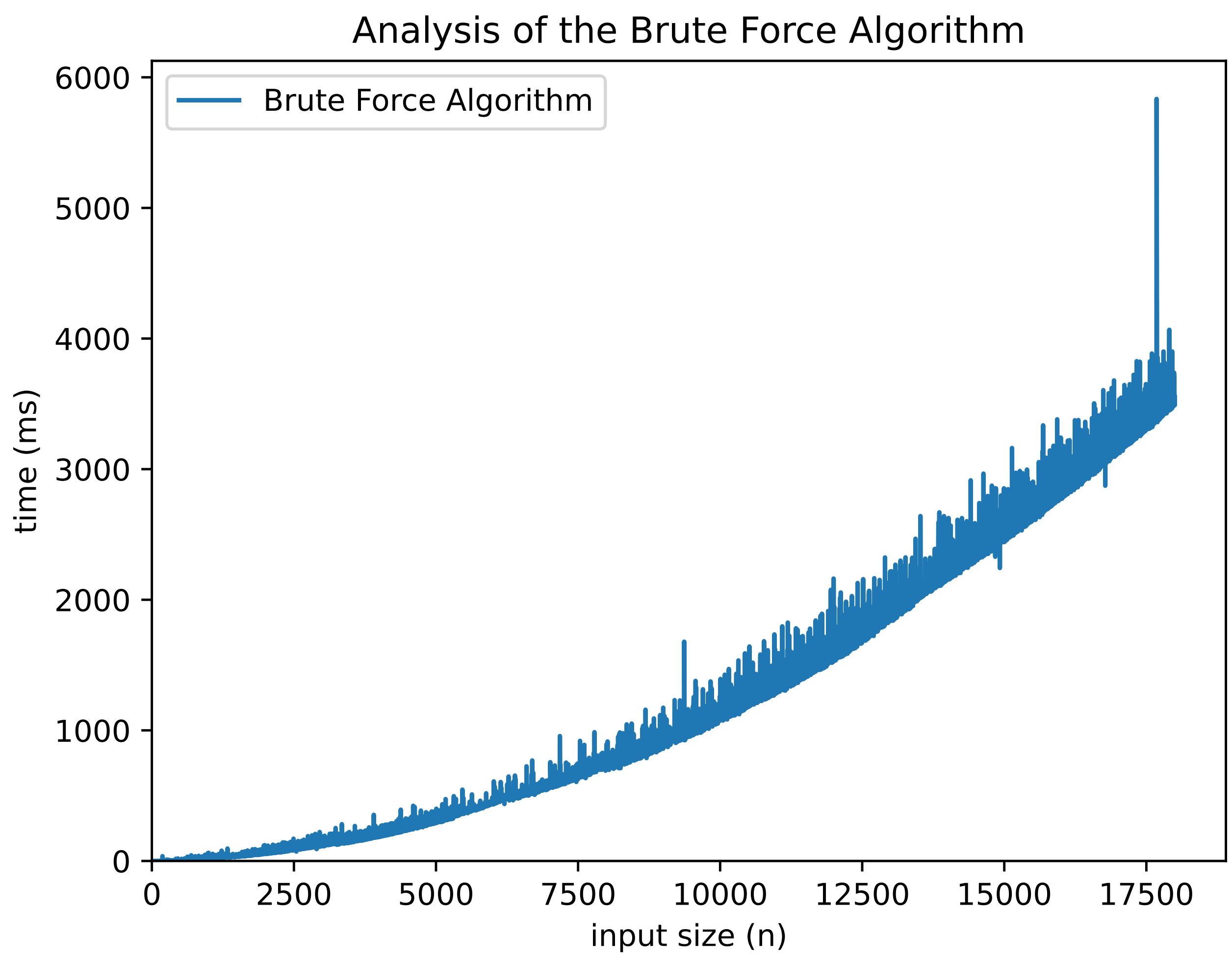


Figure 10.1: Input Size vs. Time Graph for the Brute Force Program

The Brute Force Program was the only program running. No other applications were open. Thus, no other resources could have been using the machine's computing power.

10.1.2 Analysis

Looking at Figure 10.1, we can see that the time complexity of our Brute Force algorithm is $O(n^2)$ as the graph is following a quadratic path.
These results corroborate our theoretical analysis of the Brute Force algorithm (See Section 6.3).

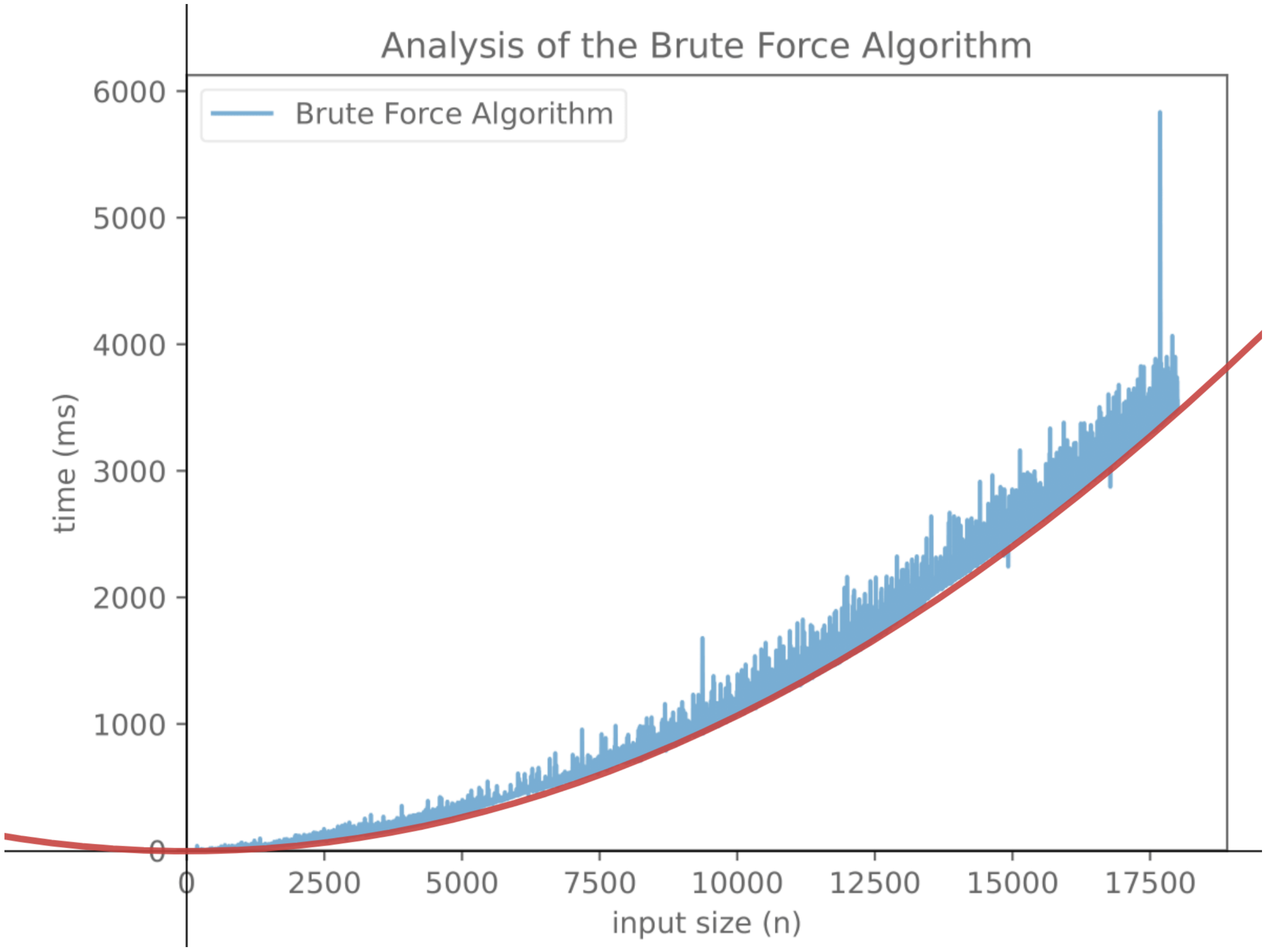


Figure 10.2: Visual Demonstration of Our Results Following a Quadratic Path

Note: The red line 🟥 is a function given by $y = 0.06x^2$.
This is not the actual function that describes the time complexity of our algorithm, it is merely used to demonstrate that our results do, in fact, follow a quadratic path.

10.2 Improved Method

10.2.1 Results

These are the results of our Improved Method program.

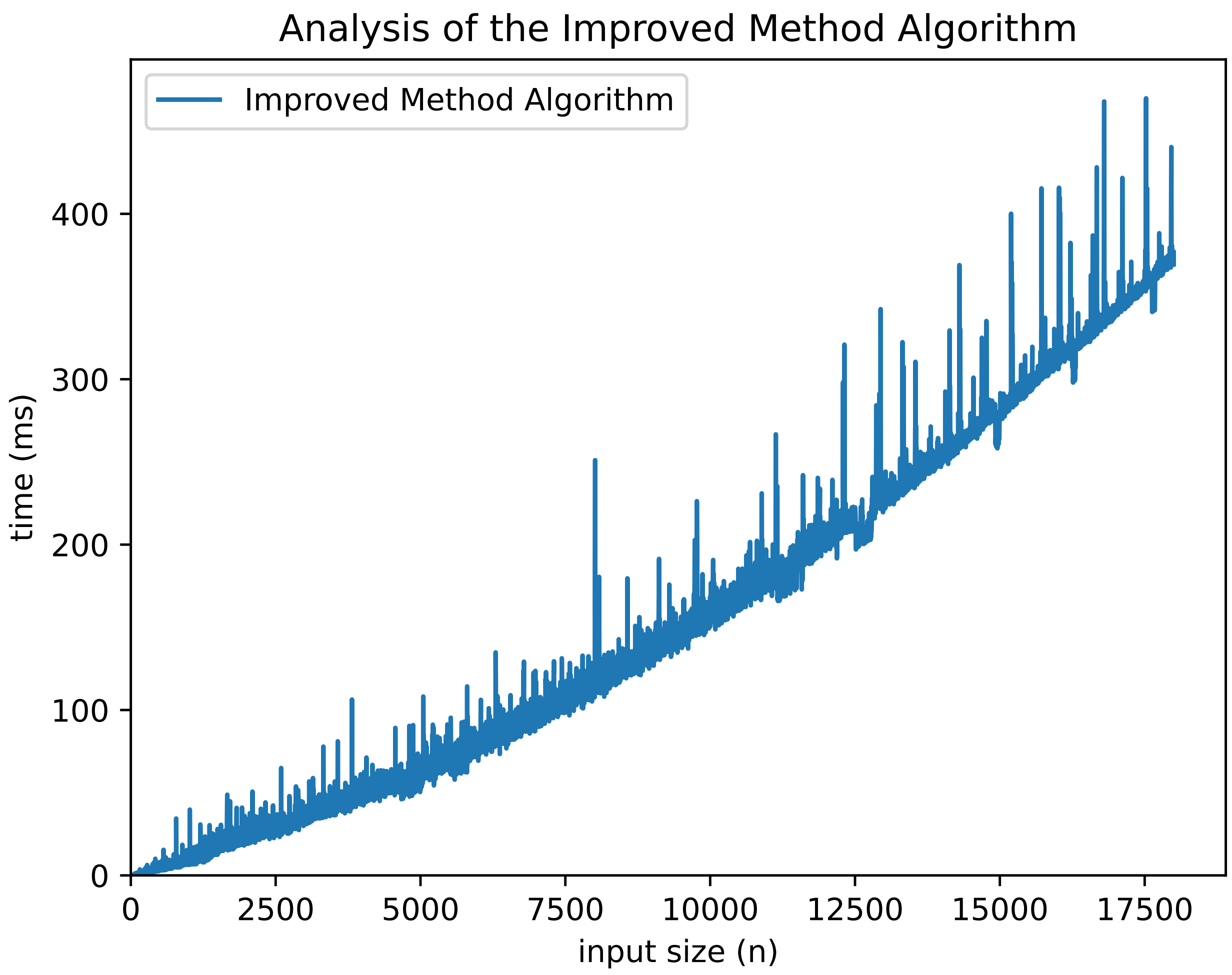


Figure 10.3: Input Size vs. Time Graph for the Improved Method Program

> The Improved Method Program was the only program running. No other applications were open. Thus, no other resources could have been using the machine's computing power.

10.2.2 Analysis

At first glance, it seems as though these results are worse due to the steeper slope of the graph compared to the Brute Force results. But, we must consider the maximum time taken for both graphs and look at both graphs together in order to pass judgement (which we will do in Section 10.3).

Looking at Figure 10.3, we can see that the time complexity of our Improved Method algorithm is $O(n \log n)$ as the graph is following a path that has a slight curve. In other words, a linearithmic path.

These results corroborate our theoretical analysis of the Improved Method algorithm (See Section 7.3).

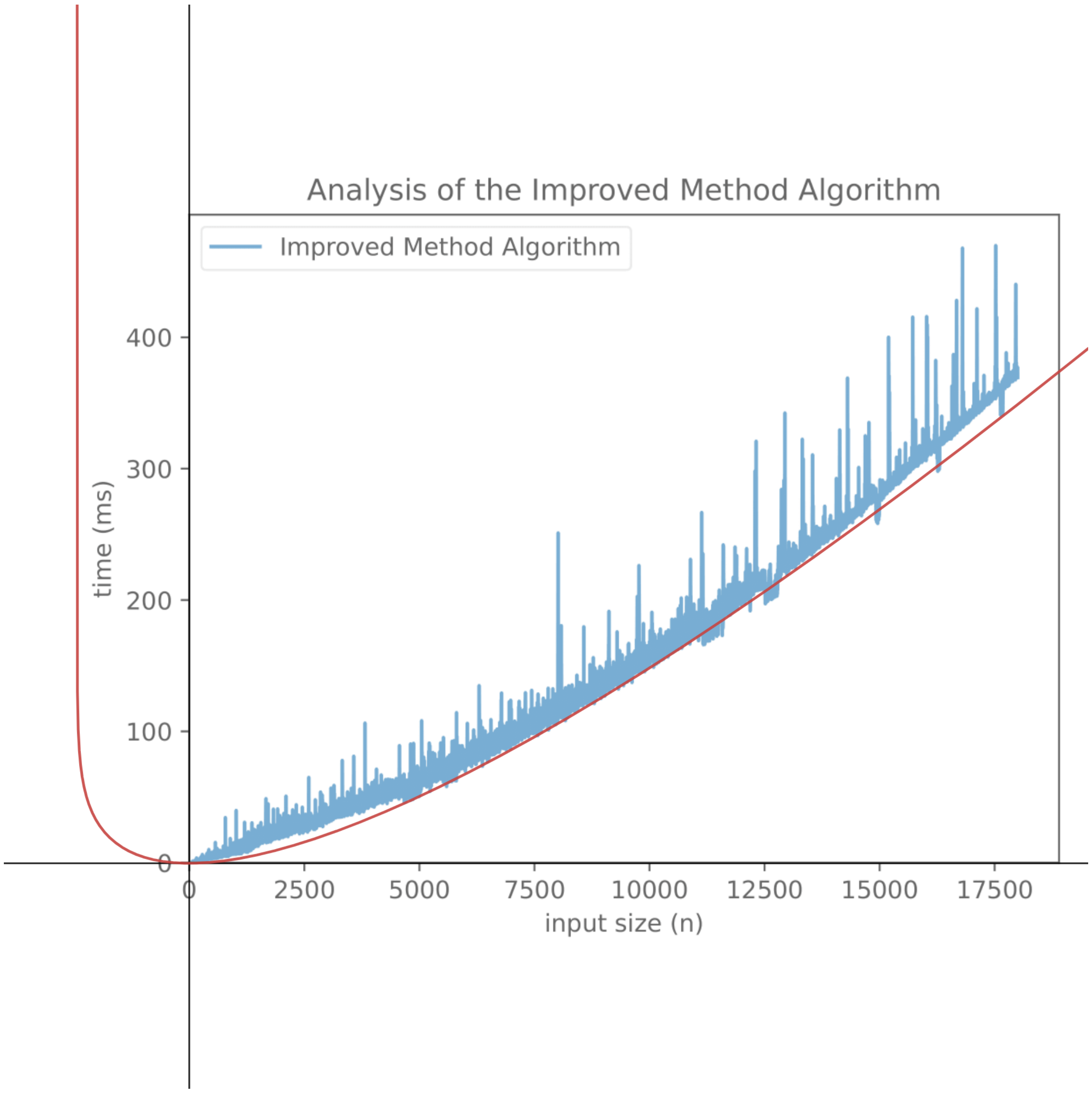


Figure 10.4: Visual Demonstration of Our Results Following a Linearithmic Path

Note: The red line ▬ is a function given by $y = (0.6x)\log(x+1)$.
This is not the actual function that describes the time complexity of our algorithm, it is merely used to demonstrate that our results do, in fact, follow a linearithmic path.

10.3 Comparing the Two Algorithms

According to our theoretical analysis of both algorithms, the Brute Force algorithm had a time complexity of $O(n^2)$, and our Improved Method algorithm had a time complexity of $O(n \log n)$. Therefore, we would expect the Improved Method to perform significantly better than the Brute Force algorithm.

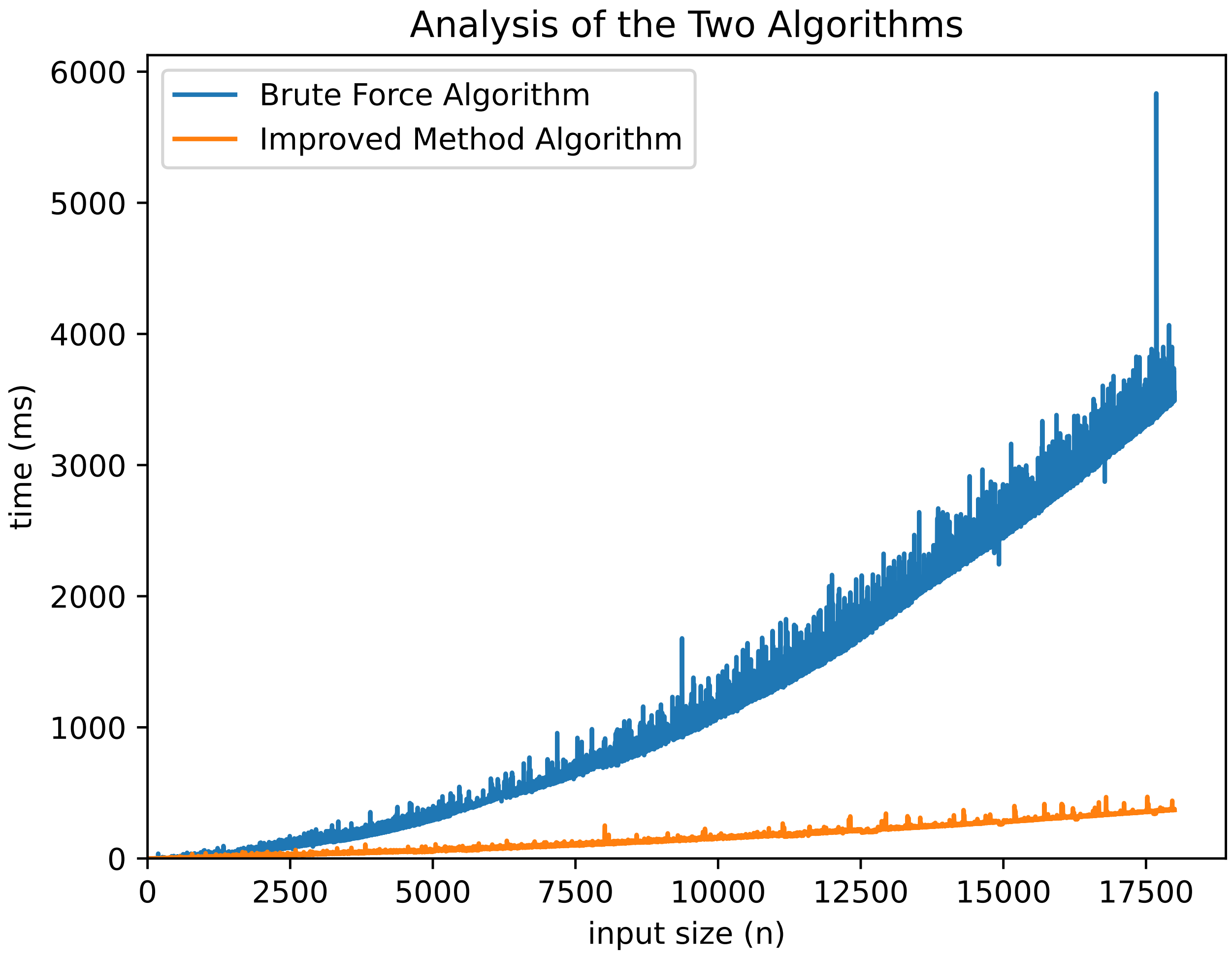


Figure 10.5: Input Size vs. Time Graph for Both Programs

Note: Both programs were tested on the same input data set.

Looking at Figure 10.5, we can clearly ascertain that, when given the same input, the Improved Method algorithm computes in a much faster time. As a result, we can conclude that the Improved Method algorithm performs far better than the Brute Force algorithm.

We can clearly see that the Brute Force algorithm is following a quadratic path, corroborating our $O(n^2)$ time complexity assessment. Although not so obvious, we can see a slight curve in the path of our Improved Method algorithm. This slight curve confirms that our Improved Method algorithm is following a linearithmic path and not a linear path which one may conclude at first glance. Again, our $O(n \log n)$ time complexity assessment has been proven. Now we can decisively conclude the time complexities of our two algorithms.

These results corroborate that we have successfully satisfied the second aim of this assignment.

Chapter 11

Conclusion

In conclusion, the development of an algorithm that can accurately and efficiently determine the vertical adjacencies of orthogonal rectangles is essential for various applications in computer graphics, architecture, and engineering.

We have introduced a new technique for solving problems more efficiently than the obvious brute force approach, by developing and testing an algorithm that uses geometric analysis and data structures to identify the vertical edges of each rectangle and compare them to those of adjacent rectangles.

The results of this experiment have yielded the development of a more efficient and accurate algorithm for identifying adjacencies of orthogonal rectangles. This assignment also demonstrated the experimental nature of Computer Science, where different algorithms that solve the same problem can be measured and evaluated based on their performance and theoretical analysis.

Overall, this assignment has successfully addressed the problem of determining vertical adjacencies between orthogonal rectangles and provided a solution that can be used in various applications.

Bibliography

[1] R. A S. *Linear Search Algorithm: Overview, Complexity, Implementation.* 2023. URL: `https://www.simplilearn.com/tutorials/data-structure-tutorial/linear-search-algorithm/`. [Accessed 13 Apr. 2023].

[2] Altenmann. *Sweep line algorithm, Wikipedia.* 2023. URL: `https://en.wikipedia.org/wiki/Sweep_line_algorithm`. [Accessed 13 May. 2023].

[3] GeeksforGeeks. *Chrono in C++.* 2017. URL: `https://www.geeksforgeeks.org/chrono-in-c/`. [Accessed 13 Apr. 2023].

[4] GeeksforGeeks. *Linear Search Algorithm.* 2019. URL: `https://www.geeksforgeeks.org/linear-search/`. [Accessed 13 Apr. 2023].

[5] GeeksforGeeks. *Set in C++ Standard Template Library (STL).* 2023. URL: `https://www.geeksforgeeks.org/set-in-cpp-stl/`. [Accessed 13 May. 2023].

[6] GeeksforGeeks. *Vector in C++ STL.* 2015. URL: `https://www.geeksforgeeks.org/vector-in-cpp-stl/`. [Accessed 14 Apr. 2023].

[7] S. Gupta. *Line sweep technique Tutorials.* 2022. URL: `https://www.hackerearth.com/practice/math/geometry/line-sweep-technique/tutorial/`. [Accessed 13 May. 2023].

[8] Dr Ian Sanders. *COMS3005A – Advanced Analysis of Algorithms – Assignment.* 2023. URL: `https://courses.ms.wits.ac.za/moodle/mod/resource/view.php?id=18935`. [Accessed 14 April. 2023].

[9] Dr Ian Sanders. *Empirical analysis of algorithms.* 2023. URL: `https://courses.ms.wits.ac.za/moodle/mod/resource/view.php?id=18936`. [Accessed 14 April. 2023].

[10] Dr Ian Sanders. *Python code to generate non-overlapping, adjacent, orthogonal rectangles.* 2023. URL: `https://courses.ms.wits.ac.za/moodle/mod/resource/view.php?id=19095`. [Accessed 14 Apr. 2023].

Appendix A

Plagiarism Declaration

University of the Witwatersrand, Johannesburg
School of Computer Science and Applied Mathematics
SENATE PLAGIARISM POLICY
Declaration by Students

I Tevlen Naidoo (Student number: 2429493) am a student registered for BSc Computer Science in the year 2023. I hereby declare the following:

- I am aware that plagiarism (the use of someone else's work without their permission and/or without acknowledging the original source) is wrong.
- I confirm that ALL the work submitted for assessment for the above course is my own unaided work except where I have explicitly indicated otherwise.
- I have followed the required conventions in referencing the thoughts and ideas of others.
- I understand that the University of the Witwatersrand may take disciplinary action against me if there is a belief that this is not my own unaided work or that I have failed to acknowledge the source of the ideas or words in my writing.

Signature: [illegible] Date: 10/04/2023

Appendix B

Sample Data

```
Number,x1,y1,x2,y2
1,5,5,16,22
2,9,26,16,35
3,16,5,20,9
4,16,11,32,15
5,16,18,25,29
6,25,18,30,22
7,25,25,27,29
8,27,28,31,31
9,32,12,36,19
10,36,13,39,21
11,39,14,43,22
```

Figure B.1: Data Contents of the *sample_input.csv* File

```
1,3,3,16,5,9,4,16,11,15,5,16,18,22
2,1,5,16,26,29
3,0
4,1,9,32,12,15
5,2,6,25,18,22,7,25,25,29
6,0
7,1,8,27,28,29
8,0
9,1,10,36,13,19
10,1,11,39,14,21
11,0
```

Figure B.2: Data Contents of the *sample_output.csv* File

Appendix C

Implementations of the `Rectangle` Classes

C.1 Implementation for the `Rectangle` Class in `GenerateRectangles.cpp`

```
class Rectangle
{
public:
    int x1; // x-value for the bottom-left corner of the rectangle
    int y1; // y-value for the bottom-left corner of the rectangle
    int x2; // x-value for the top-right corner of the rectangle
    int y2; // y-value for the top-right corner of the rectangle
    Rectangle(int x1, int y1, int x2, int y2) :
    x1(x1),
    y1(y1),
    x2(x2),
    y2(y2)
    {
    }
    std::string toString()
    {
        return "Rectangle with bottom-left corner at ("
        + std::to_string(x1) + ", " + std::to_string(y1)
        + " and top-right corner at (" + std::to_string(x2)
        + ", " + std::to_string(y2) + ")";
    }
};
```

C.2 Implementation for the `Rectangle` Class in `BruteForce.cpp`

```
class Rectangle
{
public:
    /*
    Custom Point class.
```

```
Stores the x and y coordinate of a single point.
*/
class Point
{
public:
    int x; // The x coordinate of the point.
    int y; // The y coordinate of the point.

    Point() {}

    Point(int x, int y)
        : x(x), y(y)
    {
    }
};

// The number of the rectangle.
int number;

// The x-coordinate of the bottom-left corner of the rectangle.
int x1;

// The y-coordinate of the bottom-left corner of the rectangle.
int y1;

// The x-coordinate of the top-right corner of the rectangle.
int x2;

// The y-coordinate of the top-right corner of the rectangle.
int y2;

// The width of the rectangle.
int width;

// The height of the rectangle.
int height;

// The bottom-left point (coordinates) of the rectangle.
Point bottom_left;

// The bottom-right point (coordinates) of the rectangle.
Point bottom_right;

// The top-left point (coordinates) of the rectangle.
Point top_left;

// The top-right point (coordinates) of the rectangle.
Point top_right;
```

```
    Rectangle(int number, int x1, int y1, int x2, int y2)
        : number(number),
          x1(x1),
          y1(y1),
          x2(x2),
          y2(y2),
          width(x2 - x1),
          height(y2 - y1),
          bottom_left(x1, y1),
          bottom_right(x2, y1),
          top_left(x1, y2),
          top_right(x2, y2)
    {
    }

    /*
    Returns a string with all the details of the rectangle.
    */
    std::string toString()
    {
        return "Rectangle " + std::to_string(number)
        + " with anchor point at ("
        + std::to_string(bottom_left.x)
        + ", " + std::to_string(bottom_left.y) + ") with width: "
        + std::to_string(width) + " and height: "
        + std::to_string(height);
    }
};
```

C.3 Implementation for the `Rectangle` Class in `ImprovedMethod.cpp`

```
/*
    Custom Rectangle class.
    Stored all the necessary details of the rectangle.
*/
class Rectangle
{
public:
    /*
        Custom Point class.
        Stores the x and y coordinate of a single point.
    */
    class Point
    {
    public:
        int x; // The x coordinate of the point.
        int y; // The y coordinate of the point.
    Point() {}

    Point(int x, int y)
        : x(x), y(y)
    {
    }

    std::string toString()
    {
        return '(' + std::to_string(this->x) + ','
                + std::to_string(this->y) + ')';
    }
};

/*
    Custom Details class
    Stores the details of the adjacency
*/
class Details
{
public:
    // The rectangle number of the adjacent number
    int rectangleNumber;

    // The x-coordinate where the rectangles meet
    int x;

    // The upper y-coordinate of the adjacency region
    int yt;

    // The lower y-coordinate of the adjacency region
    int yb;

    Details(int rectangleNumber, int x, int yt, int yb)
        : rectangleNumber(rectangleNumber),
          x(x),
          yt(yt),
          yb(yb)
    {
    }

    bool operator<(const Details &d) const
    {
        return this->rectangleNumber < d.rectangleNumber;
    }

    std::string toString()
    {
        return ',' + std::to_string(this->rectangleNumber) + ','
```

```
                + std::to_string(this->x) + ','
                + std::to_string(this->yb) + ','
                + std::to_string(this->yt);
    }
};

// The number of the rectangle.
int number;

// The x-coordinate of the bottom-left corner of the rectangle.
int x1;

// The y-coordinate of the bottom-left corner of the rectangle.
int y1;

// The x-coordinate of the top-right corner of the rectangle.
int x2;

// The y-coordinate of the top-right corner of the rectangle.
int y2;

// The width of the rectangle.
int width;

// The height of the rectangle.
int height;

// The bottom-left point (coordinates) of the rectangle.
Point bottom_left;

// The bottom-right point (coordinates) of the rectangle.
Point bottom_right;

// The top-left point (coordinates) of the rectangle.
Point top_left;

// The top-right point (coordinates) of the rectangle.
Point top_right;

/*
    Vector that stores the details of the adjacency
    of the adjacent rectangles
*/
std::set<Details> adjacencies;

Rectangle() {}

Rectangle(int number, int x1, int y1, int x2, int y2)
    : number(number),
        x1(x1),
        y1(y1),
        x2(x2),
        y2(y2),
        width(x2 - x1),
        height(y2 - y1),
        bottom_left(x1, y1),
        bottom_right(x2, y1),
        top_left(x1, y2),
        top_right(x2, y2)
    {
    }

    bool operator<(const Rectangle &b) const
    {
        return this->number < b.number;
    }

    /*
        Returns a string with all the details of the rectangle.
    */
    std::string toString()
    {
        return "Rectangle " + std::to_string(number)
                + " with anchor point at ("
                + std::to_string(bottom_left.x) + ", "
                + std::to_string(bottom_left.y) + ") with width: "
                + std::to_string(width) + " and height: "
                + std::to_string(height);
    }
};
```

Appendix D

Implementation of the Event Class

```
/*
    Custom Event class
    Stores the details of the Event
*/
class Event
{
public:
    Rectangle r;          // The rectangle the event is referencing
    Rectangle::Point p;   // The point the event is referencing
    int type;             // The type of event
    Event(Rectangle &r, Rectangle::Point p, int type)
    {
        this->r = r;
        this->p = p;
        this->type = type;
    }
    bool operator<(const Event &e) const
    {
        if (this->p.x != e.p.x)
        {
            return this->p.x < e.p.x;
        }
        else
        {
            if (this->p.y != e.p.y)
            {
                return this->p.y < e.p.y;
            }
            else
            {
                return this->type < e.type;
            }
        }
    }
};
```