Advanced Analysis of Algorithms Assignment
University of the Witwatersrand

UNIVERSITY OF THE
WITWATERSRAND,
JOHANNESBURG

Tevlen Naidoo (2429493)

April 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.[1]

[1]The second aim of this assignment will be fulfilled in Phase 2 of this assignment.

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.

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 on.

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 less 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 what we want, we randomly choose and remove a rectangle from the rectangle list until it contains the amount of rectangles we want.

This algorithm was inspired by the code provided by Dr Ian Sanders [5].

4.1.2 Pseudo-code for Generating the Input

```
Set numRectangles
Initialise an empty list of rectangles
Generate an initial rectangle r
Split r into 4 smaller rectangles
Add the 4 samller rectangles to the list

While length of rectangles list <= numRectangles do
    Choose a random rectangle from the list
    Split that rectangle into 4 smaller rectangles
    Remove the random rectangle from the list
    Add the 4 smaller rectangles to the list

While length of rectangle list not = numRectangles do
    Choose a random rectangle from the list
    Remove the random rectangle from the list

return list of rectangles
```

Figure 4.1: Pseudo-code for Generating the Input

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

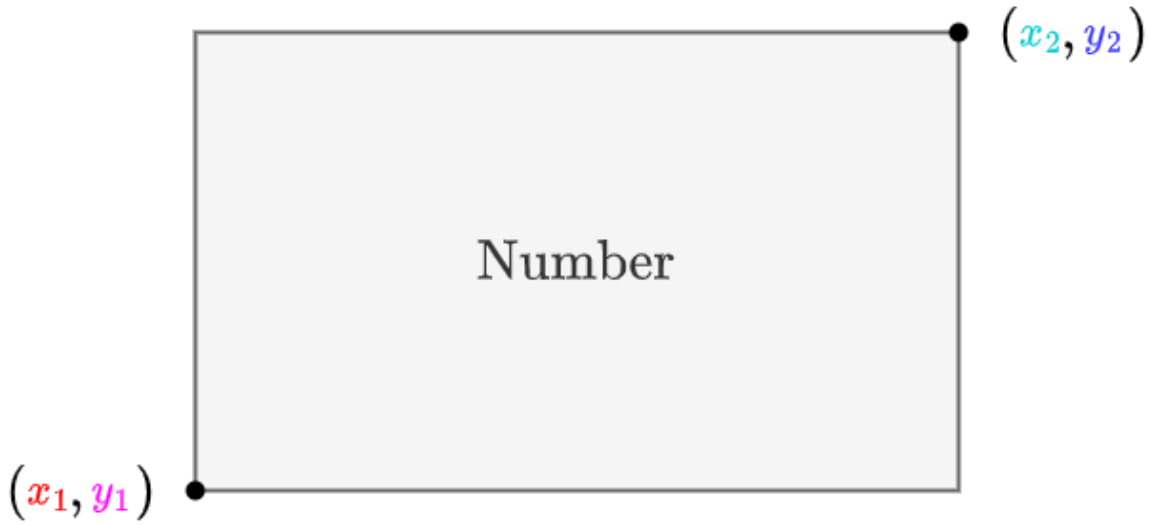


Figure 4.2: Visual Representation of the Input Structure

4.3 Visualising the Input

We will be using 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

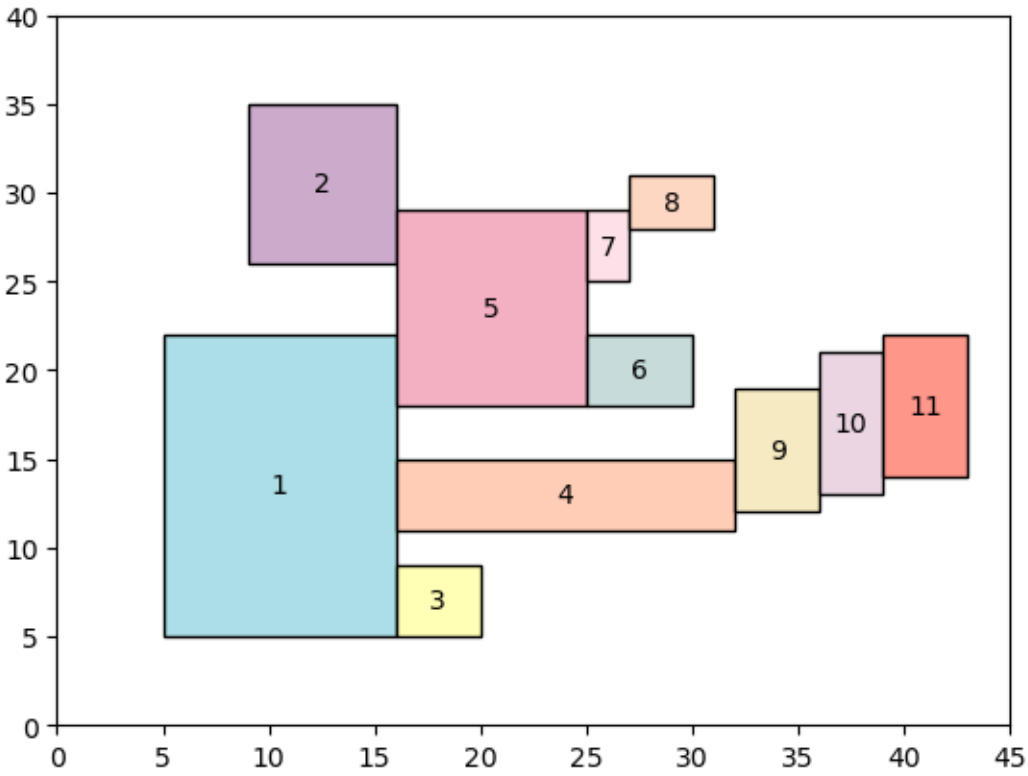


Figure 4.3: Visual Representation of the Data from *sample_input.csv*

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.

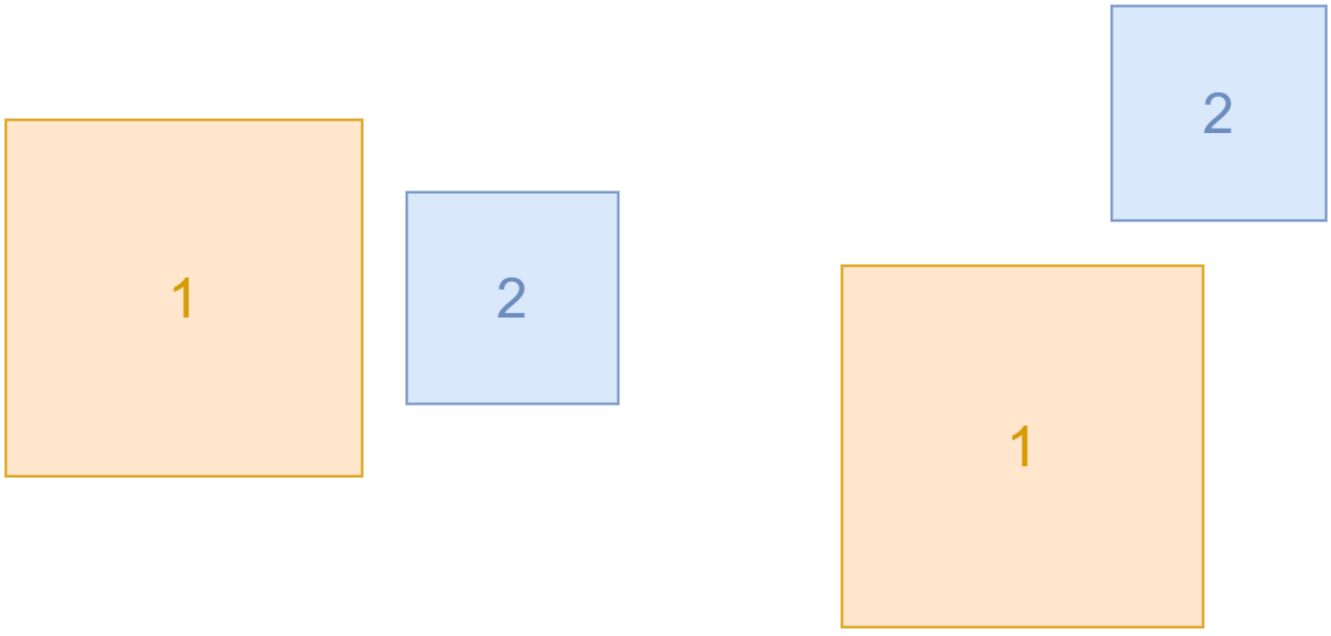


Figure 5.1: Rectangles that are NOT vertically adjacent

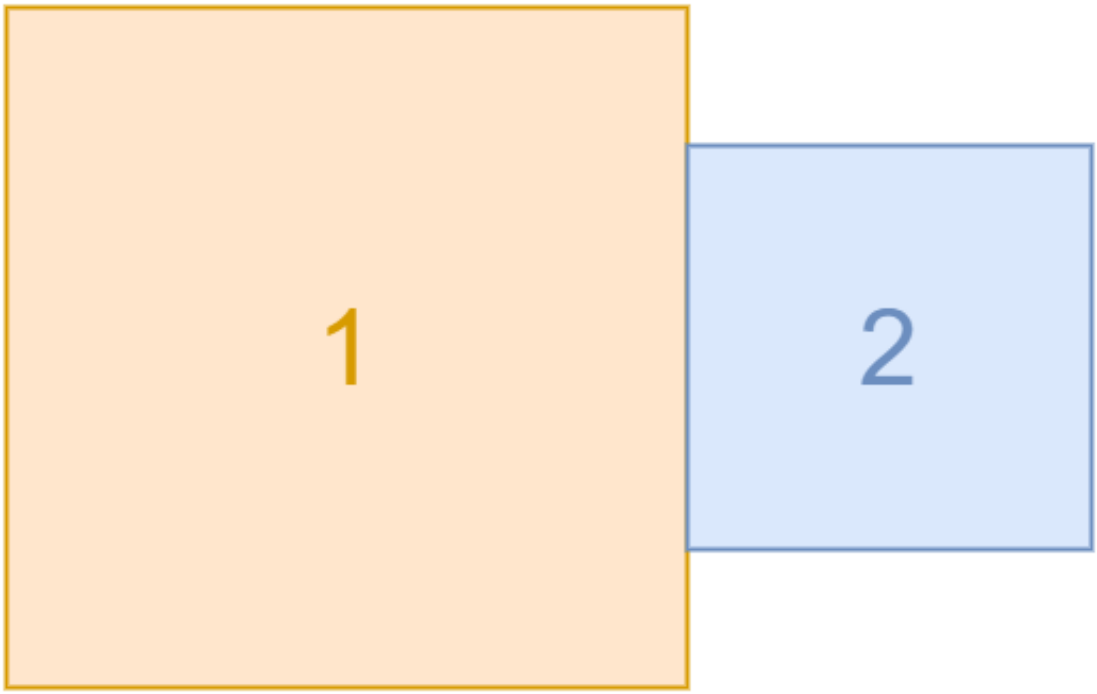


Figure 5.2: Rectangles that are vertically adjacent

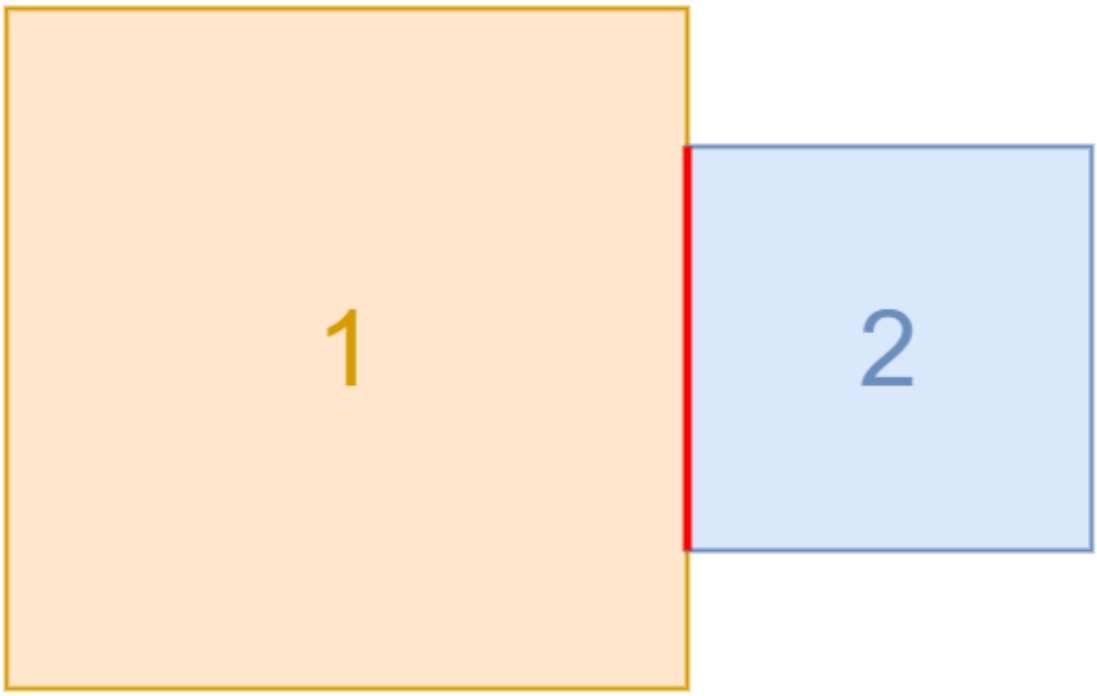


Figure 5.3: Indicating the region of vertical adjacency

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

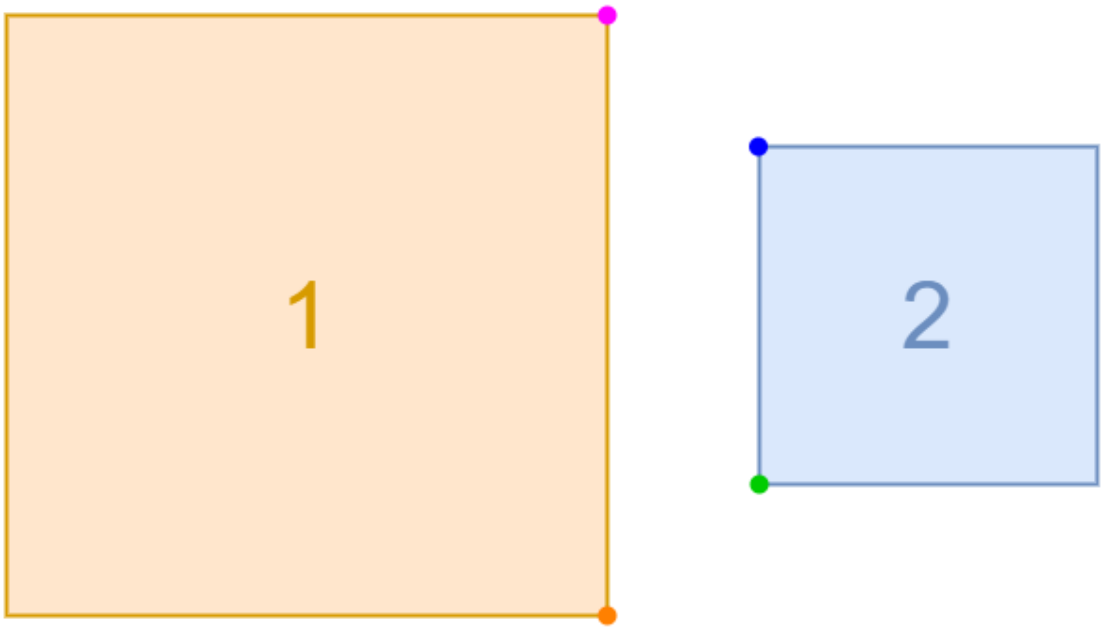


Figure 5.4: Two rectangles with the corners of importance highlighted

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

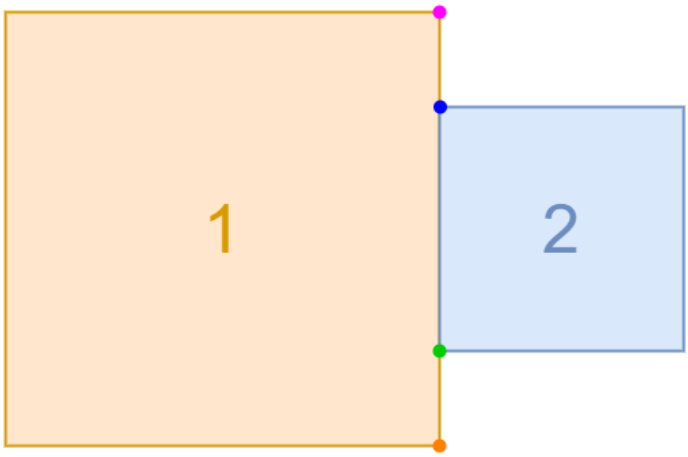
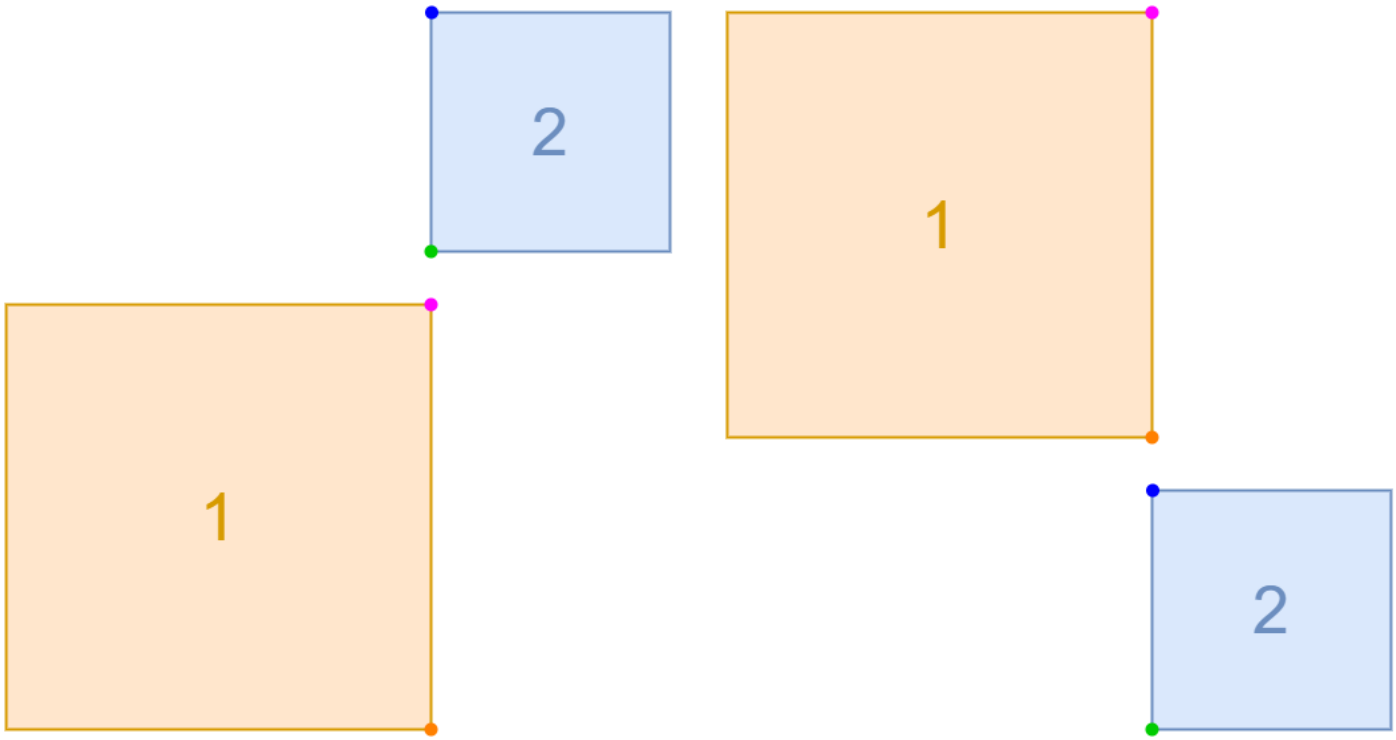


Figure 5.5: Example where the x-values of the top-right corner and the top-left corner are the same

5.2.2 Comparing y-values

Just by 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

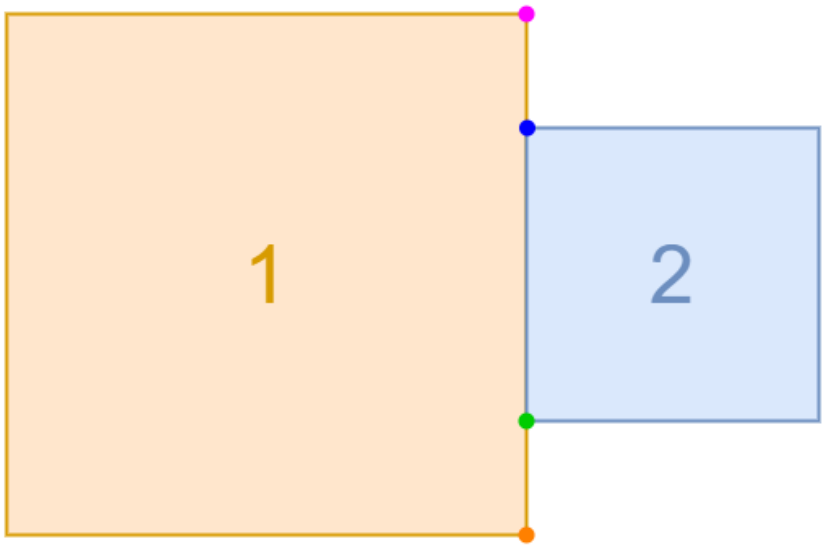
(b) Rectangle 2 is below Rectangle 1

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

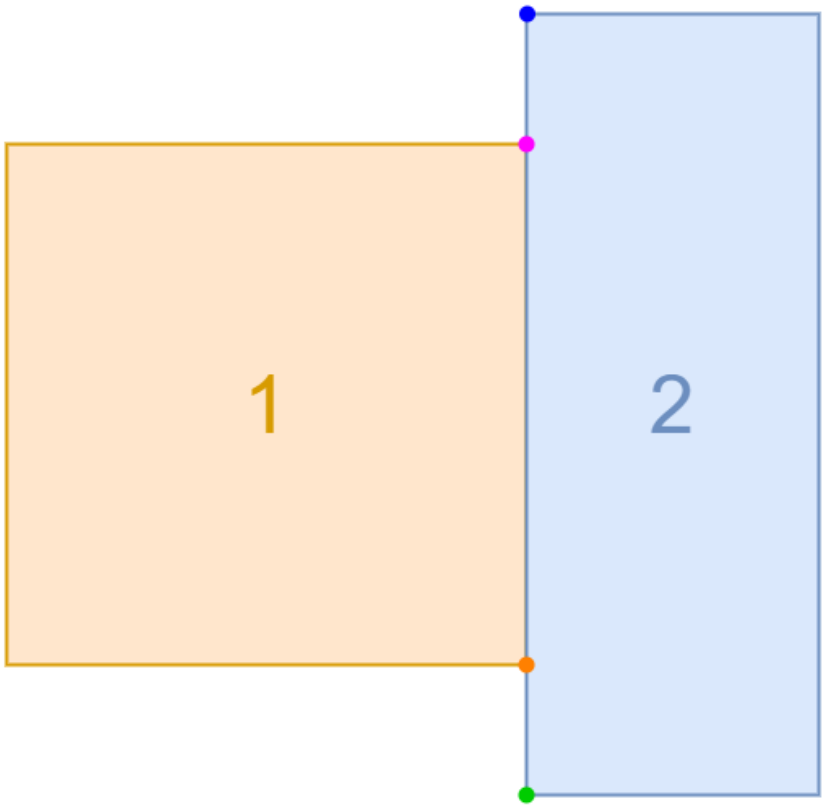


$$y_{BR} \leq y_{BL} < y_{TL} \leq y_{TR}$$

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

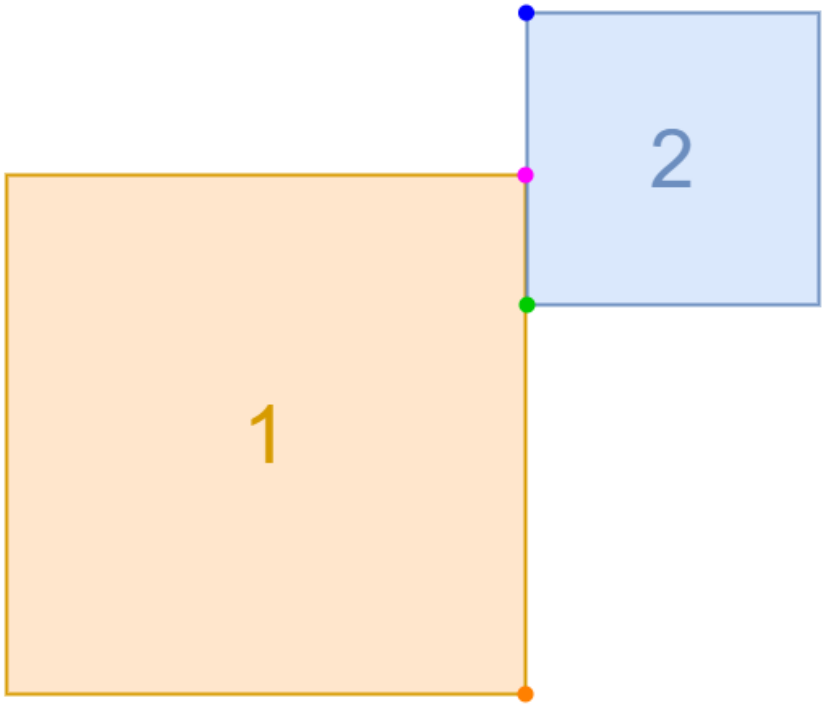


$$y_{BL} \leq y_{BR} < y_{TR} \leq y_{TL}$$

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 within the range of Rectangle 1's right corners

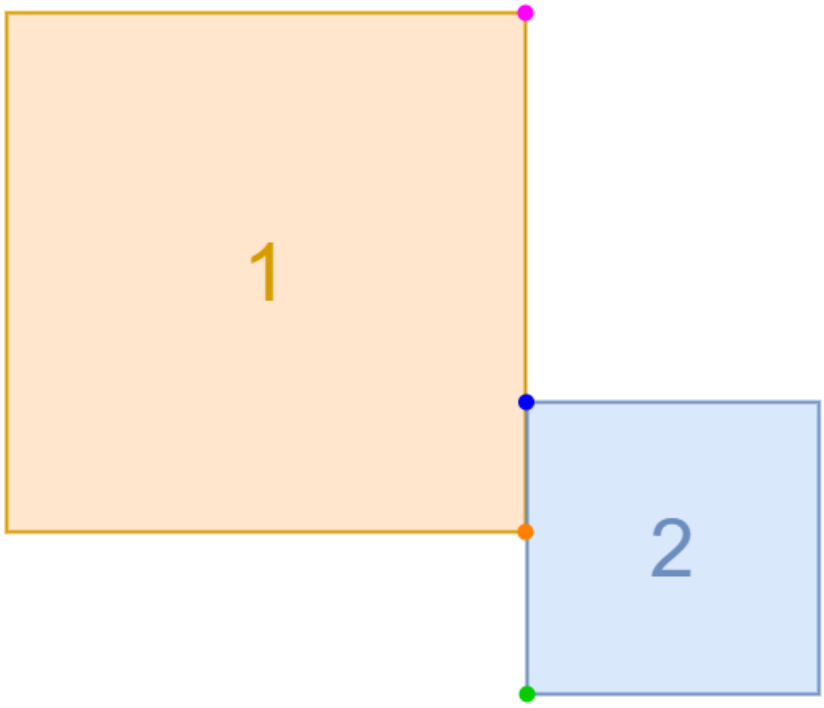


$$y_{BR} < y_{BL} < y_{TR}$$

or

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

Case 4: Rectangle 2's top-left corner lies within 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.

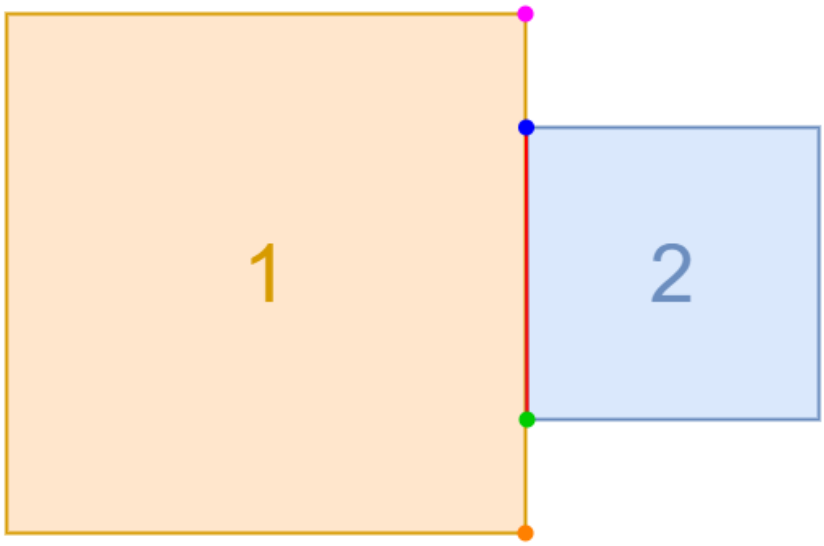
$$x = x_{TR}$$

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



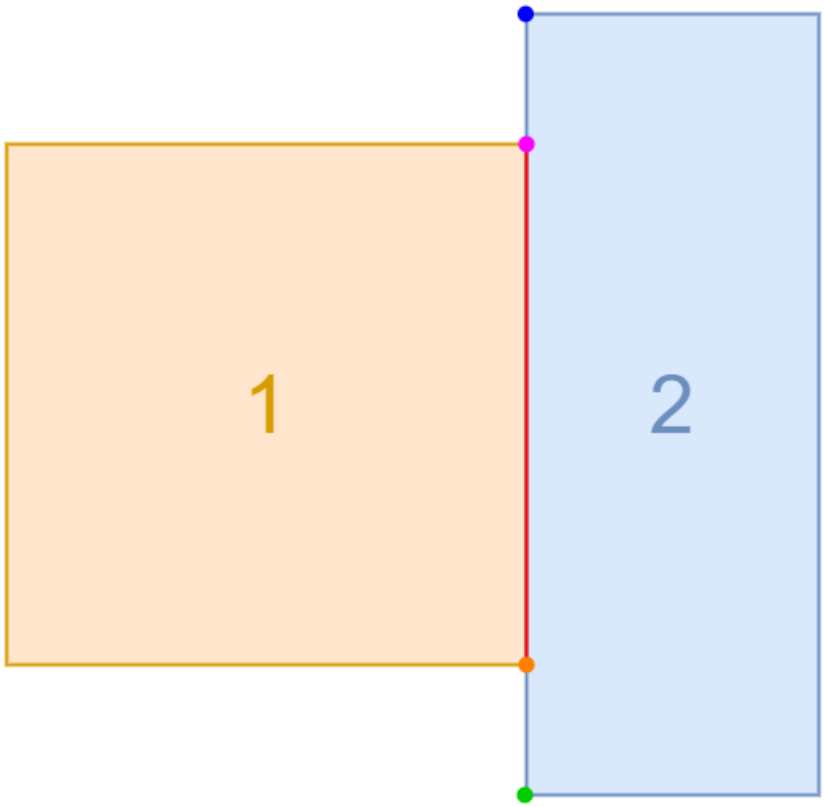
In this case, the range is the simply the range of Rectangle 2's left corners.

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

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



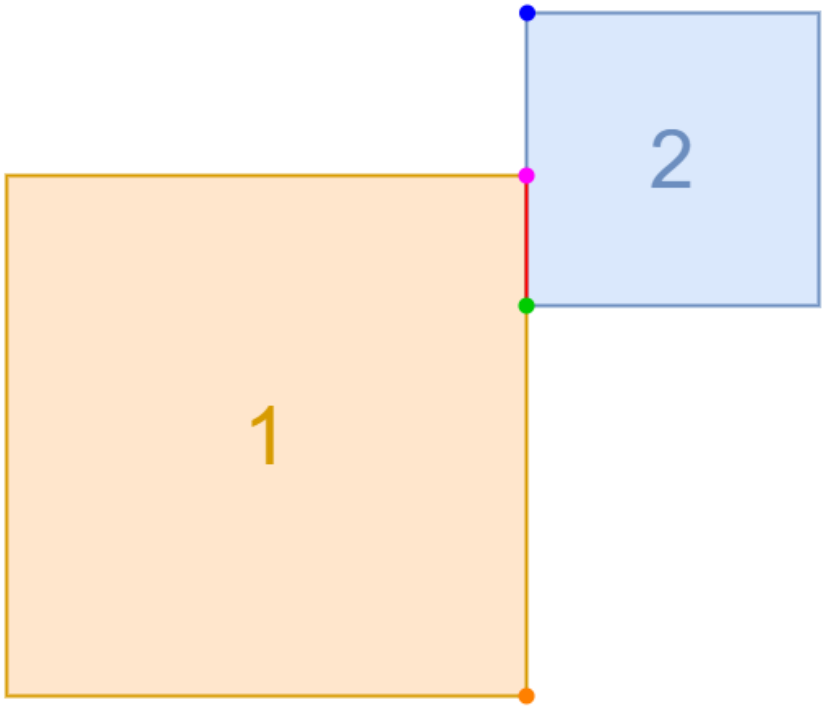
In this case, the range is the simply the range of Rectangle 1's right corners.

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

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 within the range of Rectangle 1's right corners



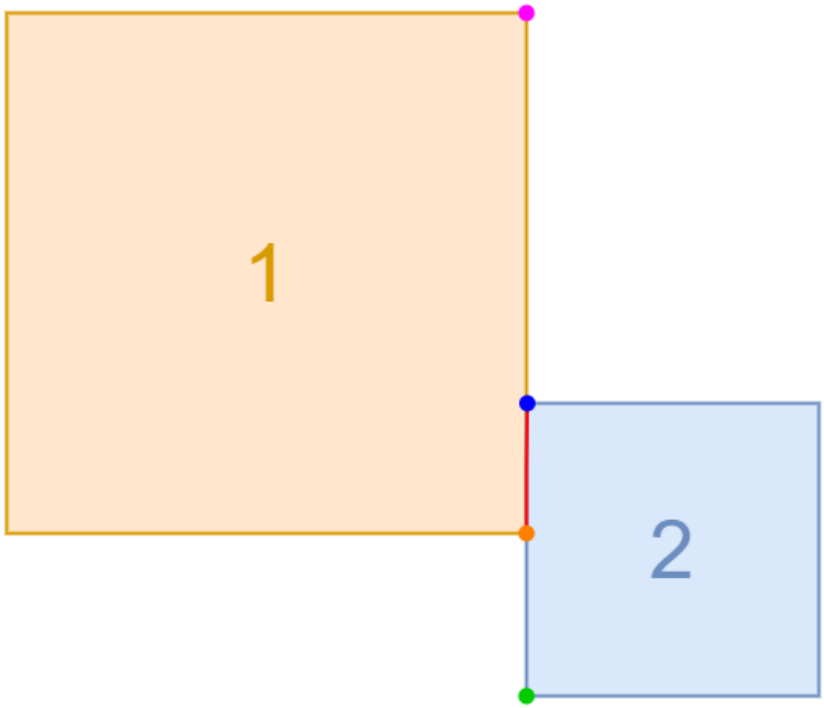
In this case, the range is the range between the top-right corner of Rectangle 1 and the bottom-left corner of Rectangle 2.

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

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 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 [3]. 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 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

function get_Adjacencies(Input: list of rectangles)
    for each rectangle r in the rectangle list:
        for every other rectangle in the rectangle list:
            if r.TR.x = other.TL.x then:
                if r.BR.y <= other.BL.y < other.TL.y <= r.TR.y then
                    other is adjacent to r
                    yt = other.TL.y
                    yb = other.BL.y
                else if other.BL.y <= r.BR.y < r.TR.y <= other.TL.y then
                    other is adjacent to r
                    yt = r.TR.y
                    yb = r.BR.y
                else if r.BR <= other.BL.y <= r.TR.y then
                    other is adjacent to r
                    yt = r.TR.y
                    yb = other.BL.y
                else if r.BR.y <= other.TL.y <= r.TR.y then
                    other is adjacent to r
                    yt = other.TL.y
                    yb = r.BR.y

                if other is adjacent to r then add to adjacency list for r
        add the adjacency list for r to the list of adjacency lists

    return the 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

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.

Chapter 7

The Output

The output that our Brute Force algorithm produces 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*

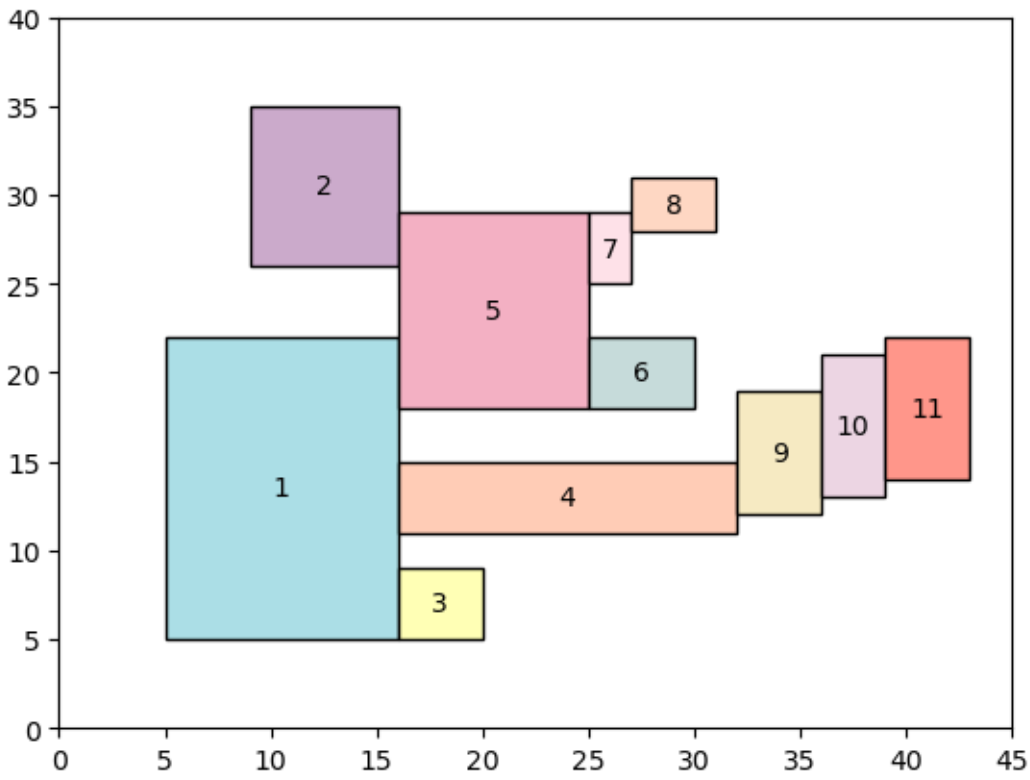


Figure 7.1: Visual Representation of the Data from *sample_input.csv*

Our Brute Force program 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

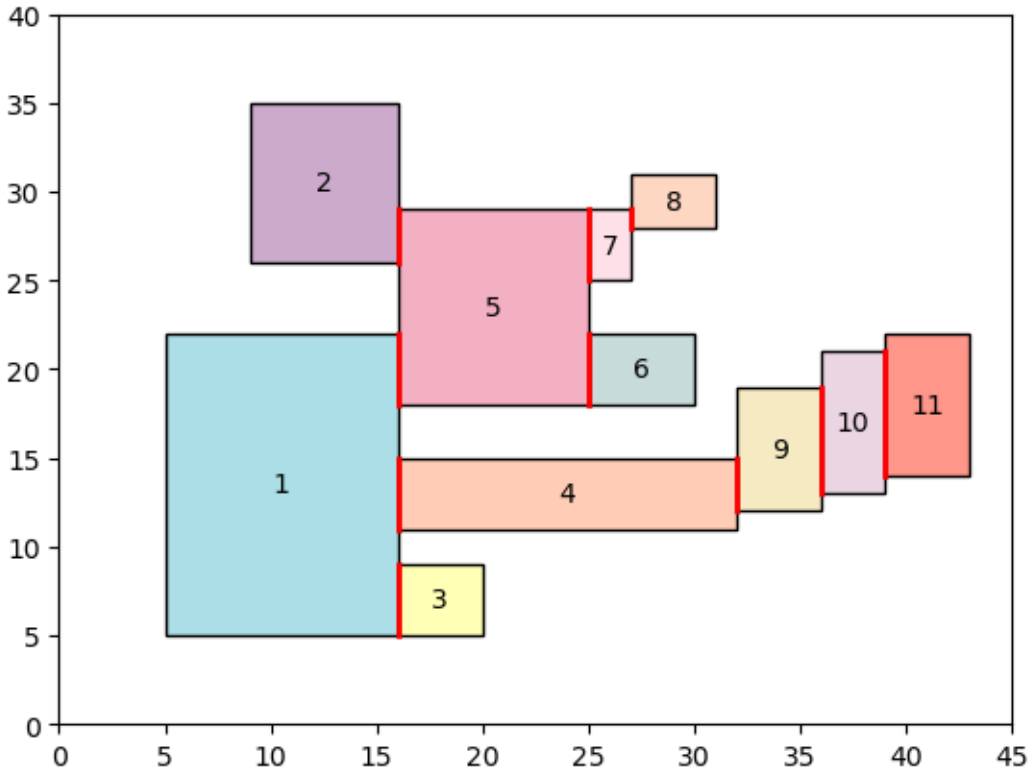


Figure 7.2: Visual Interpretation of the Data from *sample_output.csv*

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

Chapter 8

Methodology

8.1 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 amount 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`, 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.

8.2 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

8.3 Software

The following is 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 8.1: Software Specifications

8.4 Programming Language

For this experiment, both the program for generating the input files, and the Brute Force program was 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.

8.5 Data Structures

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

For both programs, we used Vectors [4] and custom Rectangle classes.

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 you 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 like 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++.

8.5.1 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 [4] 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.

8.5.2 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 [4] 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 was to be the output for this program.

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

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

8.6 Timing

Since this Brute Force Program was coded in `C++`, we make use of the `Chrono` library for `C++` [2].
To use the `Chrono` library, in the headers section of our `C++` program, we included the heading **#include** <chrono>. 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.
To add these execution times to our analysis file, 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 that measuring in milliseconds directly.

8.7 Data

8.7.1 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.

8.7.2 Output

Our Brute Force program produces two kinds of output:

- Brute Force output.
- Output for empirical analysis.

Brute Force Output

This is the output as seen in Chapter 7.

Having been given 18000 input files, our program produced 18000 output files.

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 Brute Force program to produce an output (measured in milliseconds).

It was this file that was used to produce Figure 9.1.

Chapter 9

Evaluation

9.1 Results

These are the results of our Brute Force program.

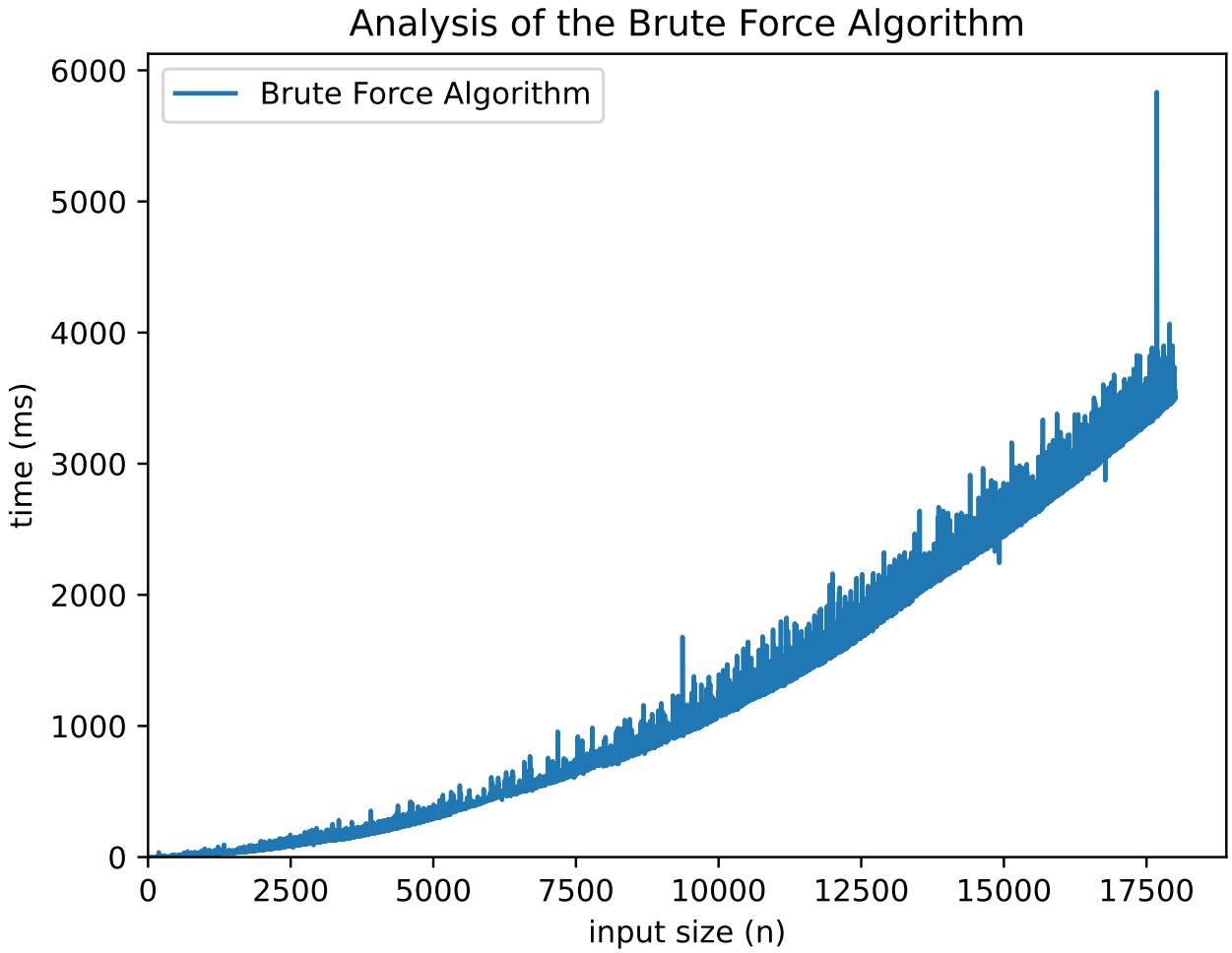


Figure 9.1: Input Size vs. Time Graph of the Brute Force Program

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.

9.2 Analysis

Looking at Figure 9.1, 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).

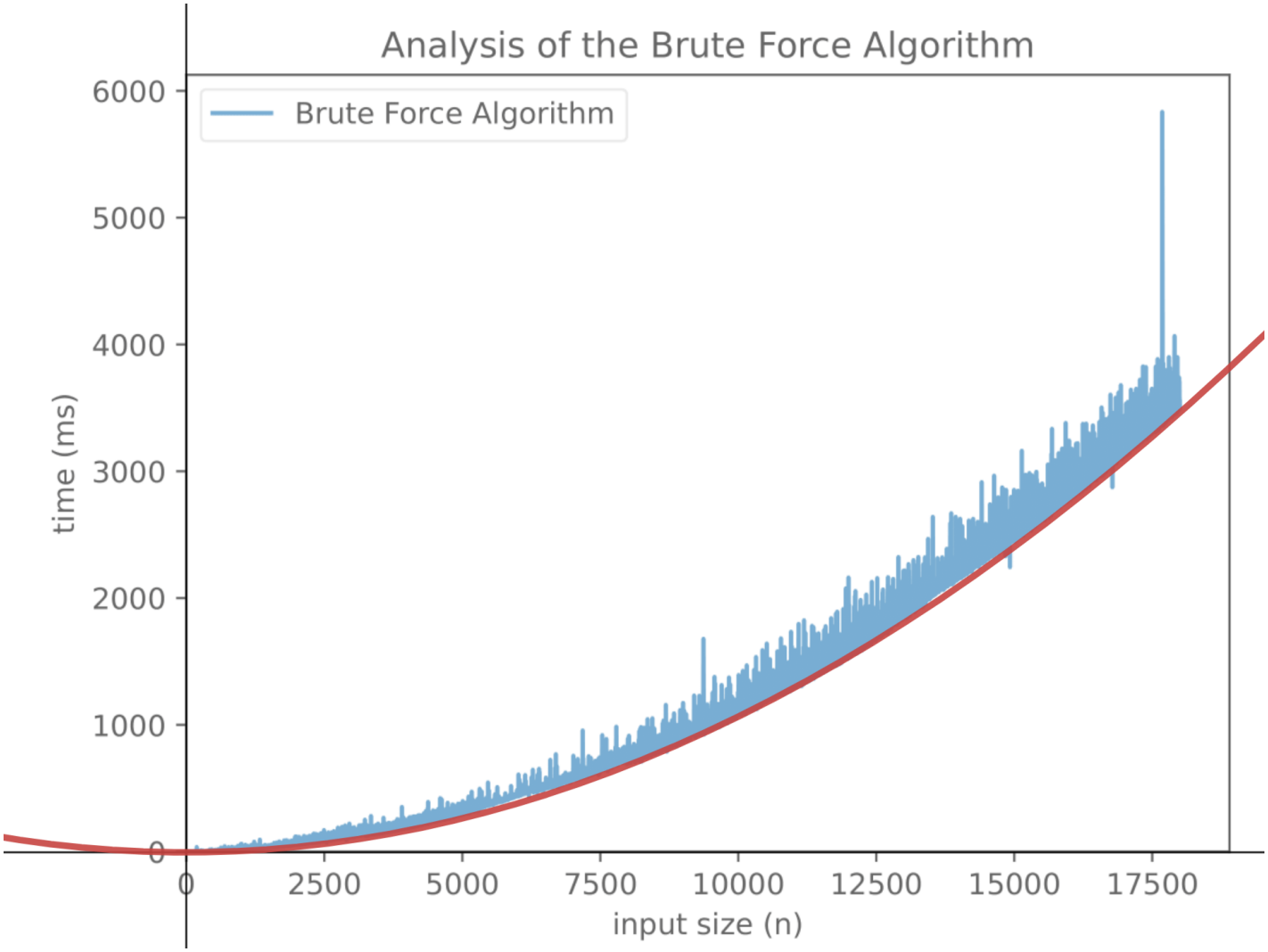


Figure 9.2: Visual Demonstration of Our Results Following a Quadratic Path

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.

Chapter 10

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.

In Phase 2 of this assignment, we will introduce 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 provided a foundation for 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#:~:text=A%20linear%20search%20is%20the%20simplest%20approach%20employed%20to%20search%20for%20an%20element%20in%20a%20data%20set.%20It%20examines%20each%20element%20until%20it%20finds%20a%20match%2C%20starting%20at%20the%20beginning%20of%20the%20data%20set%2C%20until%20the%20end.%20The%20search%20is%20finished%20and%20terminated%20once%20the%20target%20element%20is%20located.. [Accessed 13 Apr. 2023].

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

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

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

[5] 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 in not my own unaided work or that I have failed to acknowledge the source of the ideas or words in my writing.

Signature:  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);
    }
};
```