COMS3005A - Advanced Analysis of Algorithms

BSc Computer Science

Optimising vertical adjacencies between orthogonal rectangles

---

Author:

Luca von Mayer (2427051) 2427051@students.wits.ac.za

Note on plagiarism policy:

I declare that I am aware of, and understand the university's plagiarism policy. I promise to acknowledge and cite all sources that I have used in the preparation of this assignment.

May/June 2023 – 1st Semester

Contents

1 Aim/Intent:

The objective of this experiment is to develop an optimized algorithm for efficiently identifying vertical adjacencies between orthogonal rectangles, hence developing a theoretical analysis for this algorithm. The optimized algorithm aims to surpass the limitations of the brute force approach by reducing computational time complexity and eliminating unnecessary computations.

Once developed, the primary objective is to analyze its correctness and performance, relating our previous theoretical analysis to an empirical analysis. Following this, it will be compared with the previously developed the brute force algorithm. This analysis will involve evaluating the time complexity of the optimized algorithm across different sample sizes and parameter configurations.

The findings of this research will enhance our understanding of the scalability and efficiency of the optimized algorithm, paving the way for further advancements in this field.

2 Summary of Theory

2.1 Describing the problem

As mentioned in the previous phase, the problem of identifying adjacencies between orthogonal rectangles has wide-ranging applications in various fields, including computer science, facial recognition, and architecture [6, 10].

To establish the problem at hand, we define vertical adjacency as the sharing of a vertical edge. The left rectangle's right edge aligns with the right rectangle's left edge, indicating the same x-coordinate for these corresponding edges and overlapping y-coordinates. Shown by the various cases below.

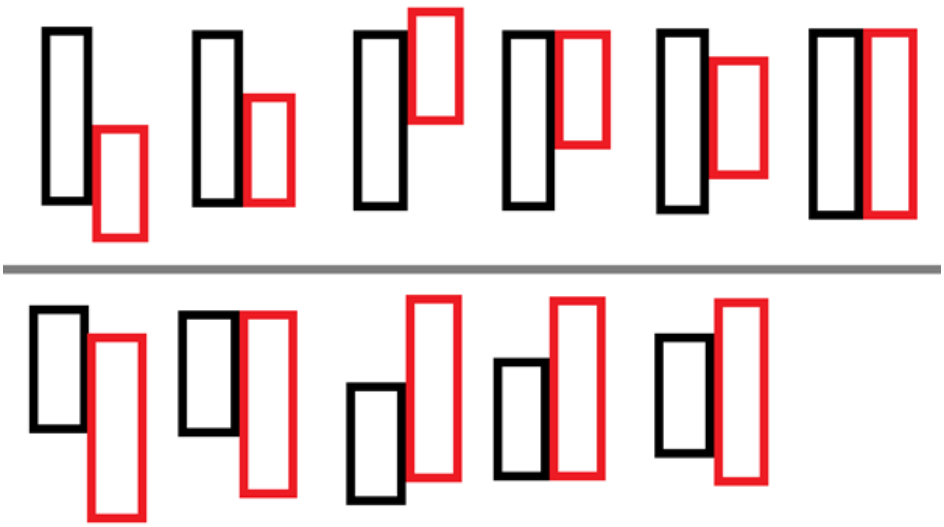

Figure 1: Cases for adjacencies

2.2 Design and analysis of brute force algorithm

To address this problem, the experimentation strategy for the optimized algorithm involves a systematic approach to identify and address the shortcomings of the original brute force algorithm, determining areas for improvement.

```
for r1 in Rectangles:
    for r2 in Rectangles:
        if r1 is not the same rectangle as r2:
            if (r1's right edge x value equals r2's left edge x value) AND
            ((both r1's y values are equal to both r2's values) OR
            (r1 overlaps r2 from above: r1's bottom y-coordinate is greater than r2's
            bottom y-coordinate AND less than r2's top y-coordinate) OR
            (r1 overlaps r2 from below: r1's top y-coordinate is greater than r2's
            bottom y-coordinate AND less than r2's top y-coordinate) OR
            (r1 completely encompasses r2: r1's bottom y-coordinate is less than or
            equal to r2's bottom y-coordinate AND r1's top y-coordinate is greater
            than or equal to r2's top y-coordinate)):
                Then r1 and r2 are marked as adjacent
```

Figure 2: Pseudocode for the brute force checkAdjacencies algorithm

As a reminder of the brute force functionality: Each case for an adjacency is considered as a condition within the algorithm, ensuring that the left and right edges of corresponding rectangles share the same x-coordinate and that their y-coordinates overlap. Following this the algorithm simply compares every rectangle with every other rectangle.

The brute force method's nested loop has constant time O(1) for the conditions, and iterates through all n rectangles with a nested loop. Since the loop executes all conditions, regardless of the input, the time complexity is expected to be the same across all cases. Thus, the algorithm exhibits a complexity of $\Theta(n^2)$.

2.3 Design and analysis of the improved algorithm

Looking at the algorithm, it becomes apparent that certain aspects of the original approach are unnecessary. One notable observation is that not all rectangles need to be compared with one another. It is evident that if two rectangles are located far apart from each other, then they cannot possibly have a vertical adjacency. Additionally, considering that the primary condition for an adjacency is the sharing of an x-coordinate on relevant edges, a straightforward strategy emerges: sorting the rectangles based on this x coordinate and comparing only the relevant pairs of rectangles.

A commonly employed algorithm in the realm of computational geometry for solving problems involving line intersections is the line sweep algorithm. This algorithm operates by traversing through the given space or interval in a sweeping manner, only attending to lines (or defined objects) as encountered by the sweep, most commonly this is used for finding intersections, but can be extended to other things as well (in our case adjacencies). Hence, the line sweep algorithm avoids unnecessary comparisons by excluding lines that are a significant distant from each other, thus focusing only on the relevant regions for computations [1].

Below is some psuedocode for a generalised line sweep for finding intersections between lines [9].

```
00  Algorithm sweepHVIntersection
    Input:  h_1, h_2, ..., h_n, n ≥ 1 (a set of n horizontal line segments in the plane) and
    v_1, v_2, ..., v_m, m ≥ 1 (a set of m vertical line segments in the plane)
    Output:    The line segments which intersect.
01      Create set eventPoint which is made up of the endpoints of horizontal line
         segments and the position of vertical line segments
02      Sort eventPoint based on x coordinate
        {break ties by putting starts of horizontal line segments before vertical line
         segments before ends of horizontal line segments and break
         further ties based on lower y value}
03      Set candidates to be empty
04      For each event in eventPoint
05          If event is the start of a horizontal line segment h
06              Then
07                  Insert h into the correct position in candidates based on y value
08          If event is the end of a horizontal line segment h
09              Then
10                  Remove the line segment h from candidates
11          If event is a vertical line segment v
12              Then
13                  Do a binary search on candidates to find the position of y_b
14                  While y value of line segment l in candidates is ≤ y_t
15                      If l intersects with v
16                          Then
17                              Output l and v
```

Figure 3: Psuedocode for a line sweep to identify line intersections

In summary, the optimized algorithm based on the line sweep technique provides a more efficient solution to identifying adjacencies between orthogonal rectangles, addressing the limitations of the brute force approach and demonstrating improved time complexity.

However, this above code is for line intersections, therefor it needs to be converted as such for the rectangle adjacencies implementations.

```
00 Algorithm checkAdjacenciesOptimized
Input: Rectangles (a set of rectangles)
Output: Updated set of rectangles with adjacencies

01 Create an empty list events
02 For each rectangle r in Rectangles:
03     Append (y value of r's bottom edge, "start", r) to events
04     Append (y value of r's top edge, "end", r) to events
05 Sort events based on the x-coordinate and event type

06 Create an empty list active
07 For each event in events:
08     If event is a "start" event:
09         For each r in active:
10             If r's right edge x value == event's left edge x value:
11                 Append event.rectangle to r.adjacencies
12             Else if event's right edge x value == r's left edge x value:
13                 Append r to event.rectangle.adjacencies
14         Append event.rectangle to active
15     Else if event is an "end" event:
16         Remove event.rectangle from active

17 Return the updated set of rectangles
```

Figure 4: Psuedocode for the optimised CheckAdjacencies algorithm

An overview on this psuedocode:

As before, the pseudocode describes an algorithm that takes a set of rectangles as input and updates the set by recording relevant adjacencies between the rectangles.

Initially, the algorithm iterates through each rectangle in the set and records the bottom and top edges of each rectangle. These edges are added to a list, indicating their position in the sweep. The sweep direction is trivial, in this case it is bottom to top. Each edge is associated with the rectangle it belongs to and is labeled as either a "start" or "end" edge, representing the bottom and top edges respectively. The rectangle itself is stored alongside these values in the tuple. Since we are iterating through the whole set of n rectangles we can theorise that this step would take O(n) time.

The list of edges is then sorted based on their values, ensuring that the sweep starts from the bottom of the space. Within each vertical position, the edges are further sorted based on their type, giving priority to the "start" edges. This sorting order allows the algorithm to first consider the bottom edges of the rectangles during the sweep. We can theorise that this step would take O(nlogn) time, as this is a commonly agreed on standard for a sorting complexity [8].

Following the sorting step, an empty list called "active" is initialized. This list keeps track of the rectangles that are currently under consideration for potential adjacencies. When encountering a "start" edge in the sweep, the algorithm checks if the rectangle associated with that edge shares an x-coordinate with any rectangle in the "active" list. If an adjacency is found, the relevant rectangles are updated accordingly. The current rectangle is then added to the "active" list since its top edge has not been encountered yet, therefore it is still a candidate for adjacencies with

other rectangles in the "active" list. Iterating through the events list and then the active list would likely take $O(n^2)$ time in the worst case..

Lastly, when an "end" edge (representing the top edge of a rectangle) is encountered, it signifies that the rectangle has been checked for all possible adjacencies with its neighboring rectangles in the y-plane. At this point, the rectangle is removed from the "active" list. This would involve inefficiently searching through the list for the removal, likely a O(n) time.

An additional enhancement can be made to improve the algorithm's efficiency. The current choice of data structure (list), has inefficient time complexity for lookup, insertion, and removal operations. To address this limitation, a different data structure that offers better performance for such operations, should be used. One option is to utilize a tree structure like a red-black tree [1, 9].

However, in the case of this algorithm, I have opted to use a Python library called "sortedcontainers" and their data structure known as a "SortedList". This particular data structure has demonstrated superior performance compared to many other structures [4] and offers a straightforward implementation and import process. By using this optimized data structure, the algorithm can achieve improved efficiency in handling the events and maintaining the active list of rectangles during the line sweep process. Creating the active SortedList takes O(n) time [4]. The SortedList uses a binary search tree internally to maintain the sorted order of elements.

Overall, the time complexity of the algorithm can be expressed as O(n) + $O(nlogn)$+ O(n) + $O(n^2)$ + any constants for assignments or updates, which simplifies to $O(n^2)$ in the worst case.

As you for the best and worst cases:

Worst case: The algorithm discards rectangles for comparison when the sweep reaches the top edge of a rectangle. Meaning that if all rectangles have the same values for their bottom and top edges all the rectangles in the space will be placed in the active list at the same time. This will result in a lot of comparisons. As our improved function is theorised to be in $O(n^2)$ we can assume that the worst case will be very close to this upperbound.

Best case: The best case is the opposite configuration of the worst case, with the rectangles sharing no bottom and top edges. This means that the active list will never consist of more than one rectangle. Despite having few comparisons compared to the worst case, the complexity would still be in $O(nlogn)$ due to the sorting.

Average case case: The average case can be represented in two ways, either we can simple have random configurations of input and use these to get an idea of the average case, or we can combine both the worst case and best case configurations (a grid). It is expected that this case would be in $O(nlogn)$ as well again due to the sorting.

3 Experimental Methodology

3.1 Hardware and Software

The experiment can be conducted using standard computer hardware and software. The hardware requirements are minimal, as the program is not computationally intensive, sample sizes are such that meaningful results can be acquired without being unnecessarily large. A computer with 16 GB of RAM and an i5 Intel processor with multiple cores is sufficient for the experiment. The operating system used during the experiment was Windows 11.

Python 3 was chosen as the programming language due to its simplicity, readability, and extensive support and libraries available online. The experiment was implemented using the Visual Studio Code integrated development environment (IDE) for its user-friendly interface, debugging capabilities, and seamless integration with Python 3. To visualize the results and generated input, the Matplotlib library was utilized. Additionally, the Time library was used to measure the execution time of the algorithm accurately. Furthermore, a non-standard library is used to allow for the use of efficient data structures, this library may need to be installed with the command: **pip install sortedcontainers** or **sudo pip install sortedcontainers** depending on OS.

3.2 Overall strategy

The experimental strategy employed in this report aims to test the adjacency algorithms as follows:

Generation of Input Sizes: A list of different values representing the number of rectangles to be generated, denoted as 'n', is created.

Execution of Algorithms: The generated 'n' values are iterated, and for each value, both the original brute force method and the optimized method are executed. The execution time for each algorithm is recorded for further analysis.

Data Analysis and Visualization: The recorded execution times for both algorithms are passed to a plotting method, which generates graphs to visualize the performance of the algorithms. Statistical data is also computed and included in the analysis.

Comprehensive Testing: The experimental process is repeated multiple times, considering various factors such as different input sizes, 'n' values, parameter configurations for the generating methods, and testing the average, best, and worst cases. This comprehensive testing approach allows for a thorough evaluation of the algorithms' efficiency and scalability.

3.3 Data structures and input sizes

3.3.1 Data structures

We need to define suitable data structures and input format for the experiment. The input will consist of a list of rectangles, where each rectangle is represented by the coordinates of its bottom-left and top-right corners. To facilitate the implementation, we introduce a Rectangle class as the main data structure, which includes the necessary attributes and methods.

The Rectangle class stores the rectangle's adjacency information in a list and provides methods such as equals() and toString() for comparison and printing purposes.

As mentioned before an important new data structure structure that is introduced is that of the sorted list, this data structure allows for the contents to not only remain sorted but for very efficient operations. Full documentation can be found here: [5].

```
class Rectangle:

    def __init__(self, num, x1, y1, x2, y2):
        self.rectNum = num
        self.bottom_left = (x1, y1)
        self.top_right = (x2, y2)
        self.adjacencies = []

    def equals(self, other):
        if (self.bottom_left == other.bottom_left) and (self.top_right == other.top_right):
            return True
        else:
            return False

    def toString(self):
        out = str(self.rectNum) + ', ' + str(len(self.adjacencies)) + ', '
        for r in self.adjacencies:
            out += str(r.rectNum) + ', ' + str(r.bottom_left[0]) + ', ' + str(
                r.bottom_left[1]) + ', ' + str(r.top_right[1]) + ', '
        print(out)
```

Figure 5: The final checkAdjacencies algorithm

3.3.2 Input sizes

In the experiment, we examined the performance of the adjacency identification algorithm using various input sizes.The experiment was conducted using three different input sizes: small, medium, and large.

For the small input size, 50 n values are tested which range from 10 to 400. This size allowed us to assess the algorithm's performance on a relatively small scale.

For the medium input size, 50 n values are tested which range from 10 to 400. This size aims to evaluate the algorithm's efficiency when dealing with a larger number of rectangles.

For the large input size, 50 n values are tested which range from 10 to 2000. This size pushed the algorithm to its limits, examining its performance under significant computational loads.

By employing these distinct input sizes, we aimed to capture the algorithm's behavior across varying problem complexities and cases, allowing us to effectively observe any trends or patterns of the algorithm.

3.4 Data generation for various cases

3.4.1 Random configuration (average case)

The method takes in three parameters, the number of rectangles you want to generate, the gird size that you want to generate them in, and finally the maximum area of any rectangle generated. The reason behind this last parameter is to ensure that no very large rectangles are generated that needlessly take up all the space. The method generates random coordinates (with Python's Random library). Following this it ensures the top right coordinate and bottom left coordinates are correctly orientated, then checks if the rectangle meets the area requirement. And finally before this rectangle is allowed, we need to see if it overlaps with any of the existing rectangles, to do this we check if each of the edges of the potential rectangle lie inside of any of the existing rectangle's boundaries.

```
def checkOverlap(bottom_left, top_right, Rectangles):
    for r in Rectangles:
        if (bottom_left[0] < r.top_right[0] and top_right[0] > r.bottom_left[0]
        and bottom_left[1] < r.top_right[1] and top_right[1] > r.bottom_left[1]):
            return True
    return False


def genRects(numrect, gridSize, maxArea):
    Rectangles = []
    for i in range(numrect):
        Area = maxArea + 1
        overlap = True

        while ((Area > maxArea) or (overlap == True)):
            bottom_left = (random.randint(0, gridSize), random.randint(0, gridSize))
            top_right = (random.randint(0, gridSize), random.randint(0, gridSize))

            while ((top_right[0] <= bottom_left[0]) or (top_right[1] <= bottom_left[1])):
                bottom_left = (random.randint(0, gridSize), random.randint(0, gridSize))
                top_right = (random.randint(0, gridSize), random.randint(0, gridSize))

            width = top_right[0] - bottom_left[0]
            height = top_right[1] - bottom_left[1]
            Area = width * height

            if (Area < maxArea):
                overlap = checkOverlap(bottom_left, top_right, Rectangles)
            else:
                overlap = True

        Rectangles.append(Rectangle(i, bottom_left[0], bottom_left[1], top_right[0], top_right[1]))

    return Rectangles
```

Figure 6: The algorithm to randomly generate rectangles

The following is a visualisation using the matplotlib library with parameters being: 500 rectangles on a 50 x 50 grid with a max area of 50. The white blocks with blue edges represent the

rectangles and the grey represents empty space.

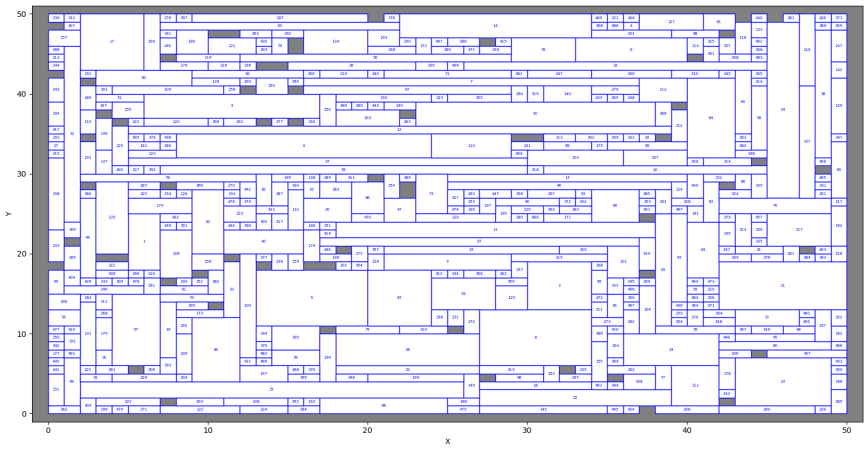

Figure 7: Example of randomly generated rectangles

3.4.2 Worst case

To generate this configuration, the code is very basic simple dividing up the space equally almost rectangles that span the whole height of the space.

```
def genWorstCaseRects(numrects, gridSize, maxArea):
    Rectangles = []
    width = gridSize / numrects
    height = maxArea / width

    for i in range(numrects):
        bottom_left = (i * width, 0)
        top_right = ((i + 1) * width, height)
        Rectangles.append(Rectangle(i, bottom_left[0], bottom_left[1], top_right[0], top_right[1]))

    return Rectangles
```

Figure 8: The algorithm to generate rectangles for the worst case

The following is a visualisation using the matplotlib library with parameters being: 100 rectangles on a 50 x 50 grid with a max area of 50. The white blocks with blue edges represent the rectangles and the grey represents empty space.

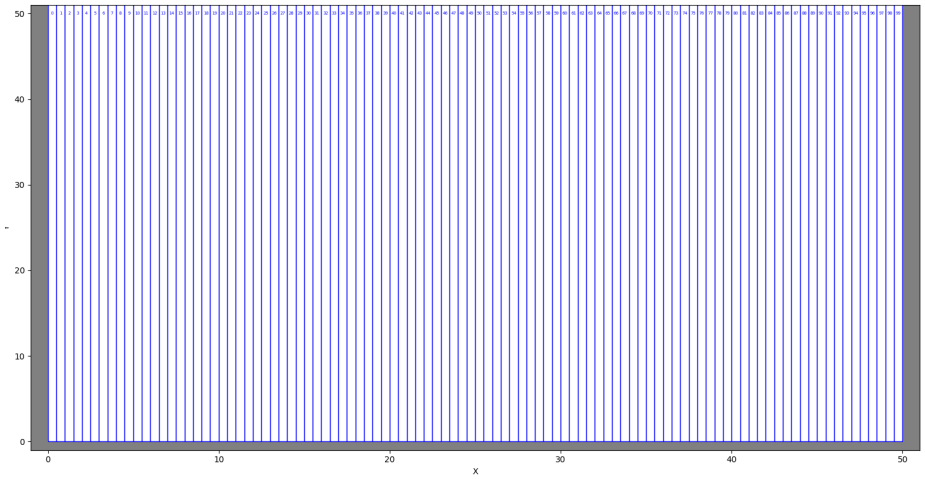

Figure 9: Example of worst-case generated rectangles

3.4.3 Best case

The best case is the opposite configuration of the worst case algorithm with the rectangles spanning the whole width of the space.

```
def genBestCaseRects(numrects, gridSize, maxArea):
    Rectangles = []
    height = gridSize / numrects
    width = maxArea / height

    for i in range(numrects):
        bottom_left = (0, i * height)
        top_right = (width, (i + 1) * height)
        Rectangles.append(Rectangle(i, bottom_left[0], bottom_left[1], top_right[0], top_right[1]))

    return Rectangles
```

Figure 10: The algorithm to generate rectangles for the best case

The following is a visualisation using the matplotlib library with parameters being: 100 rectangles on a 50 x 50 grid with a max area of 50. The white blocks with blue edges represent the rectangles and the grey represents empty space.

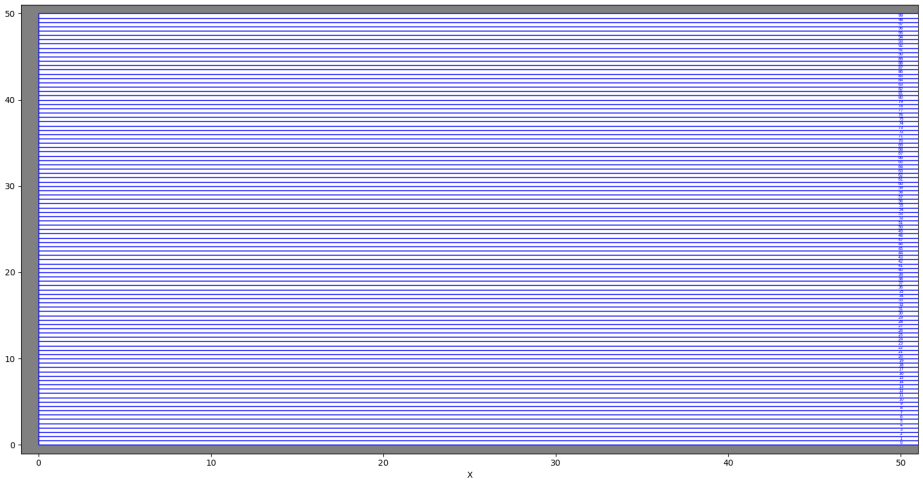

Figure 11: Example of best-case generated rectangles

3.4.4 Average case

This method is a combination of the best and worst cases, rater than spanning either the whole width or length it is a uniform grid.

```
def genAvgCaseRects(numrect, gridSize, maxArea):
    Rectangles = []
    side_length = math.sqrt(maxArea)
    width = side_length
    height = side_length
    x = 0
    y = 0

    for i in range(numrect):
        bottom_left = (x, y)
        top_right = (x + width, y + height)
        Rectangles.append(Rectangle(i, bottom_left[0], bottom_left[1], top_right[0], top_right[1]))

        x += width  # Move to the right for the next rectangle
        if x + width > gridSize:  # If the next rectangle exceeds the grid's width
            x = 0  # Move to the leftmost position in the next row
            y += height  # Move to the next row

    return Rectangles
```

Figure 12: The algorithm to randomly generate rectangles

The following is a visualisation using the matplotlib library with parameters being: 500 rectangles on a 50 x 50 grid with a max area of 50. The white blocks with blue edges represent the rectangles and the grey represents empty space.

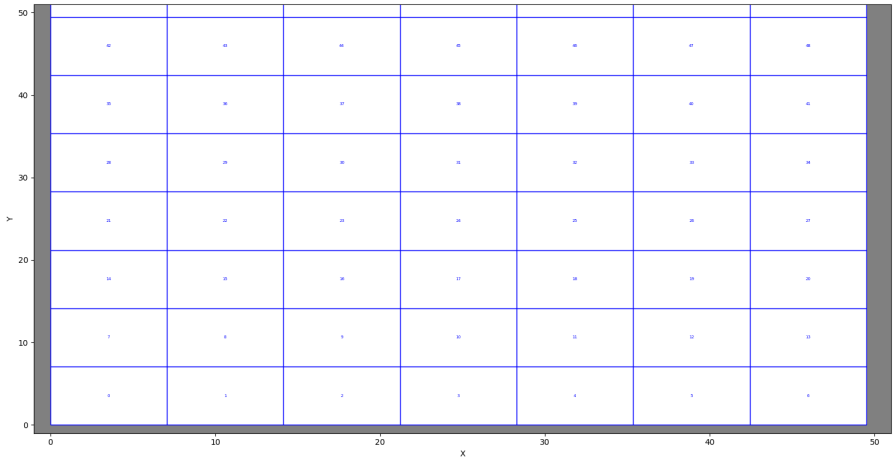

Figure 13: Example of randomly generated rectangles

3.5 Timing

As mentioned earlier the use of Pythons Time library, will be used to record the results. The following is the corresponding code:

```
# Making an array to test over different sample sizes
n_list = []
for i in range(50):
    value = int(10 + (i / 99) * (2000 - 10))
    n_list.append(value)

bf_times = []
o_times= []

for n in n_list:
    Rectangles = genBestCaseRects(100, 50, 50)
    bf_Rectangles  = copy.deepcopy(Rectangles)
    o_Rectangles  = copy.deepcopy(Rectangles)

    bf_start_time = time.time()
    bf_Rectangles = checkAdjacenciesBruteForce(bf_Rectangles)
    bf_end_time = time.time()
    bf_elapsed_time = (bf_end_time - bf_start_time) * 10**3
    bf_times.append(bf_elapsed_time)

    o_start_time = time.time()
    o_Rectangles = checkAdjacenciesOptimized(o_Rectangles)
    o_end_time = time.time()
    o_elapsed_time = (o_end_time - o_start_time) * 10**3
    o_times.append(o_elapsed_time)
```

Figure 14: Code for timing of the algorithm

4 Results

4.1 Statistical results generation and plotting

In order to interpret the results in a meaningful way code was added which not only plots the data for both algorithms and compares them on the same graphs, but also calculates the second degree polynomial via regression (best fit line), the R^2 value, the p-values and the confidence intervals for each plot.

4.2 Visualisation of results

The following are plots tested at increasing n values. A curve fitting has been added on the plot of degree 2. Using the scipy stats library some statistical information has been added to each of the plots in the top left corner for both algorithms (may need to zoom in to see the relevant statistics).

For the first test there will be 50 n values tested which range from 10 to 500. This will be done on a grid of 200 with a maximum area of 50.

The second test we will add more data and move up to 50 n values tested which range from 10 to 2000, with the same parameters.

Finally the third test we will add more data and move up to 100 n values tested which range from 10 to 4000, with the same parameters.

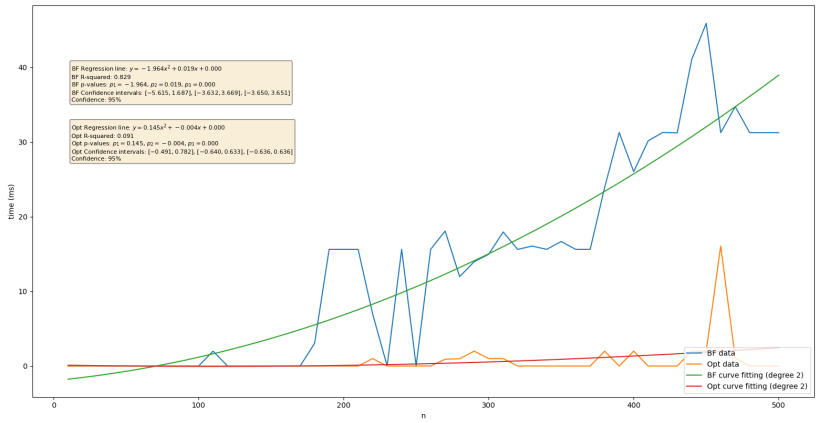

Figure 15.1: Random case for small n

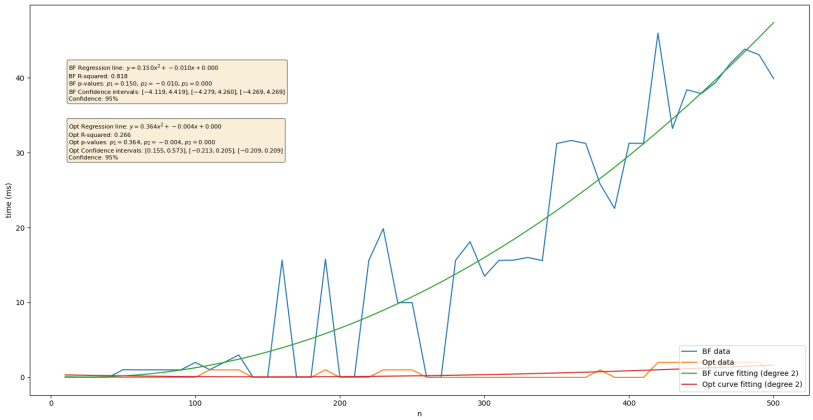

Figure 15.2: Average case for small n

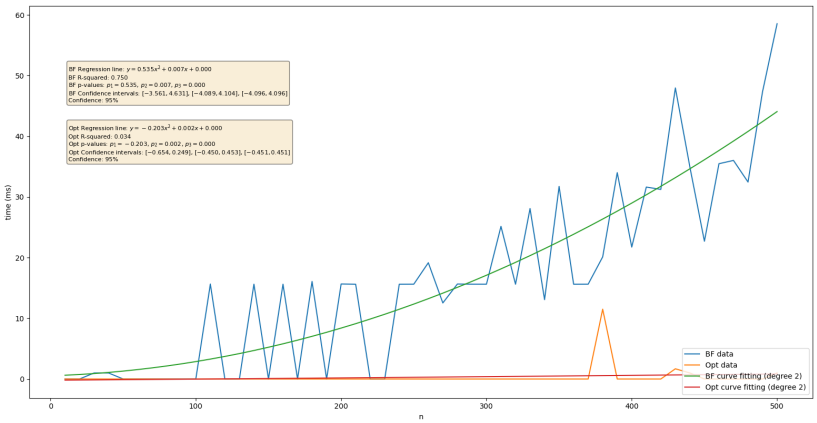

Figure 15.3: Best case for small n

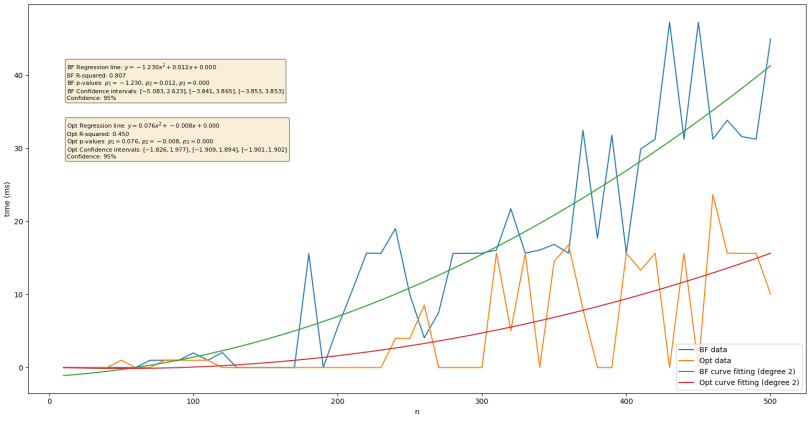

Figure 15.4: Worst case for small n

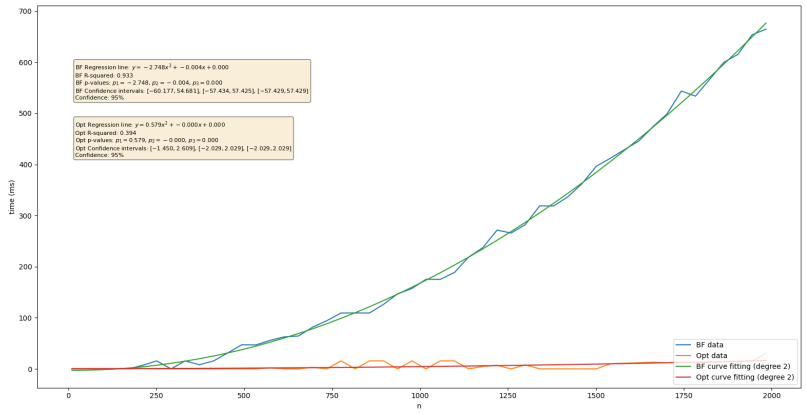

Figure 16.1: Random case for medium n

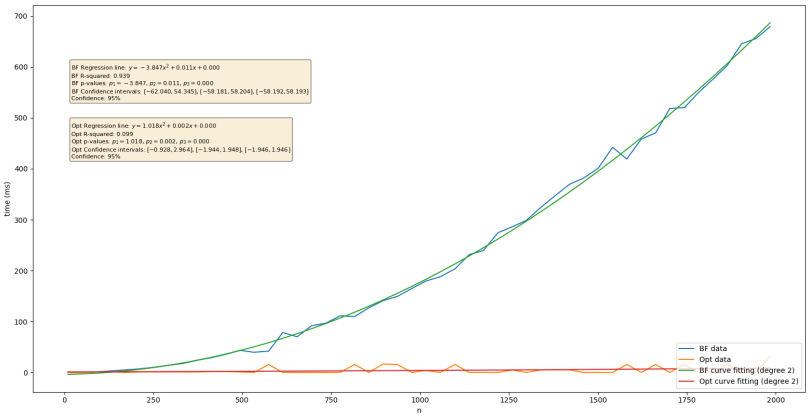

Figure 16.2: Average case for medium n

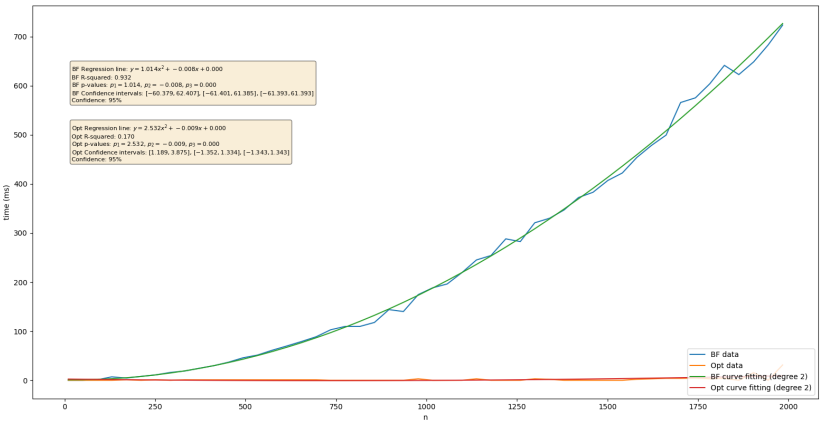

Figure 16.3: Best case for medium n

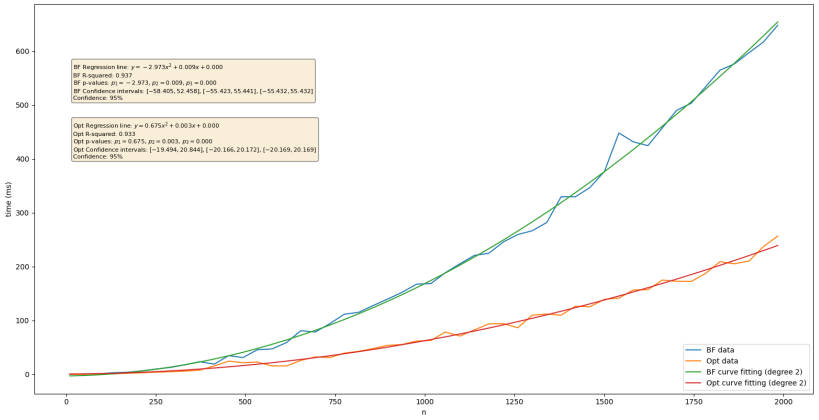

Figure 16.4: Worst case for medium n

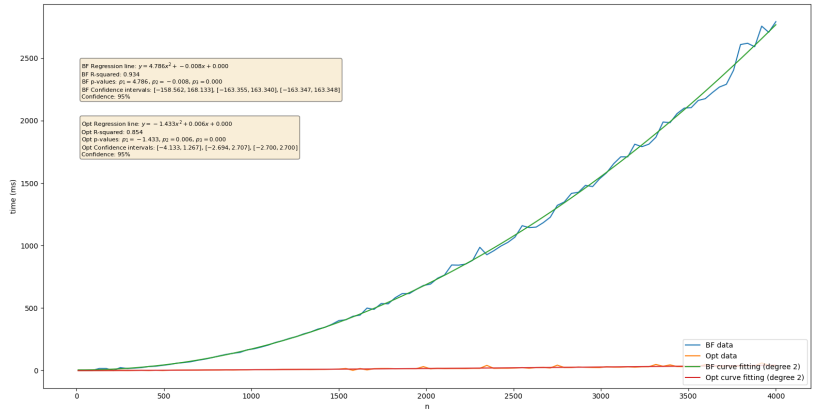

Figure 17.1: Random case for large n

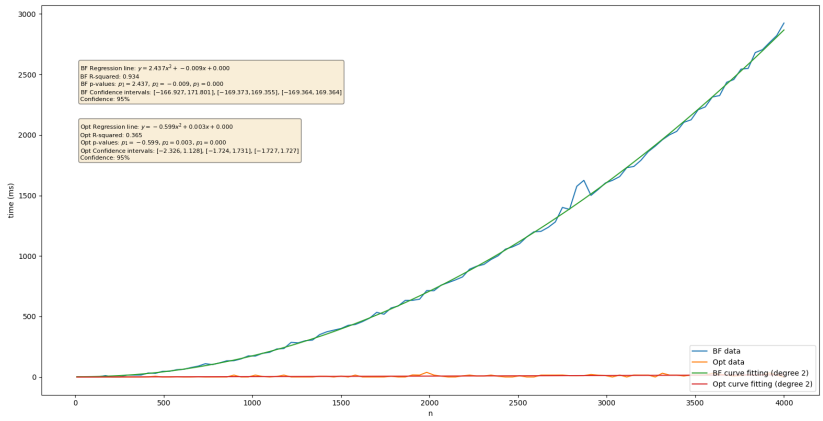

Figure 17.2: Average case for large n

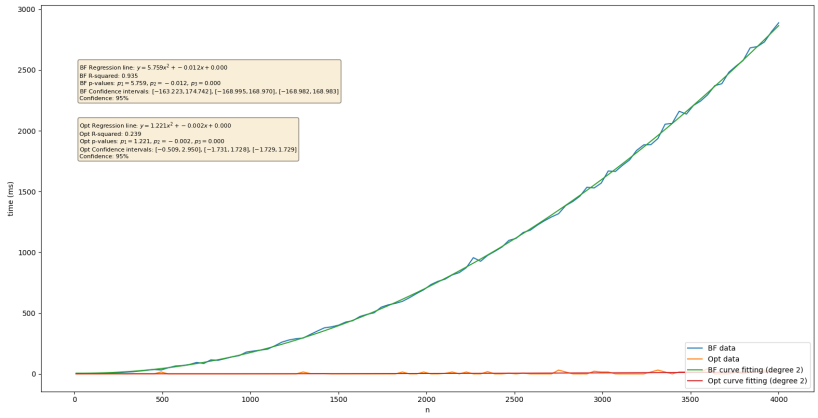

Figure 17.3: Best case for large n

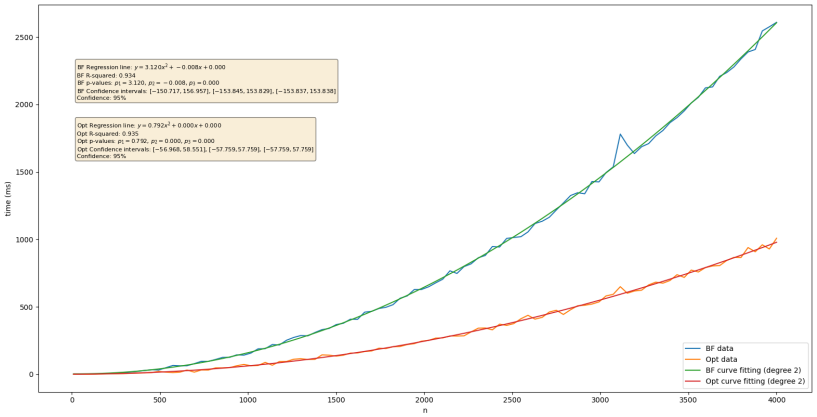

Figure 17.4: Worst case for large n

The following are three testes that change the area parameter of the generation to test if they achieve new results:

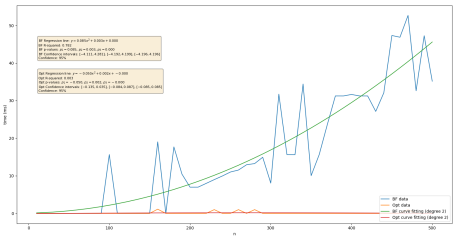

Figure 18.1: Area - 30

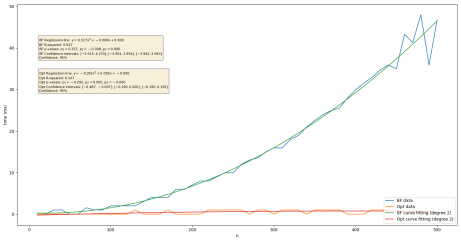

Figure 18.2: Area - 60

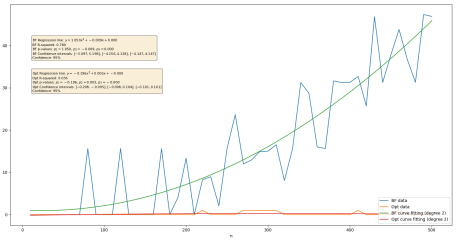

Figure 18.3: Area - 90

The final result is placing all 4 cases on one plot to easily compare there values and shapes. The plot uses the same values as the second test.

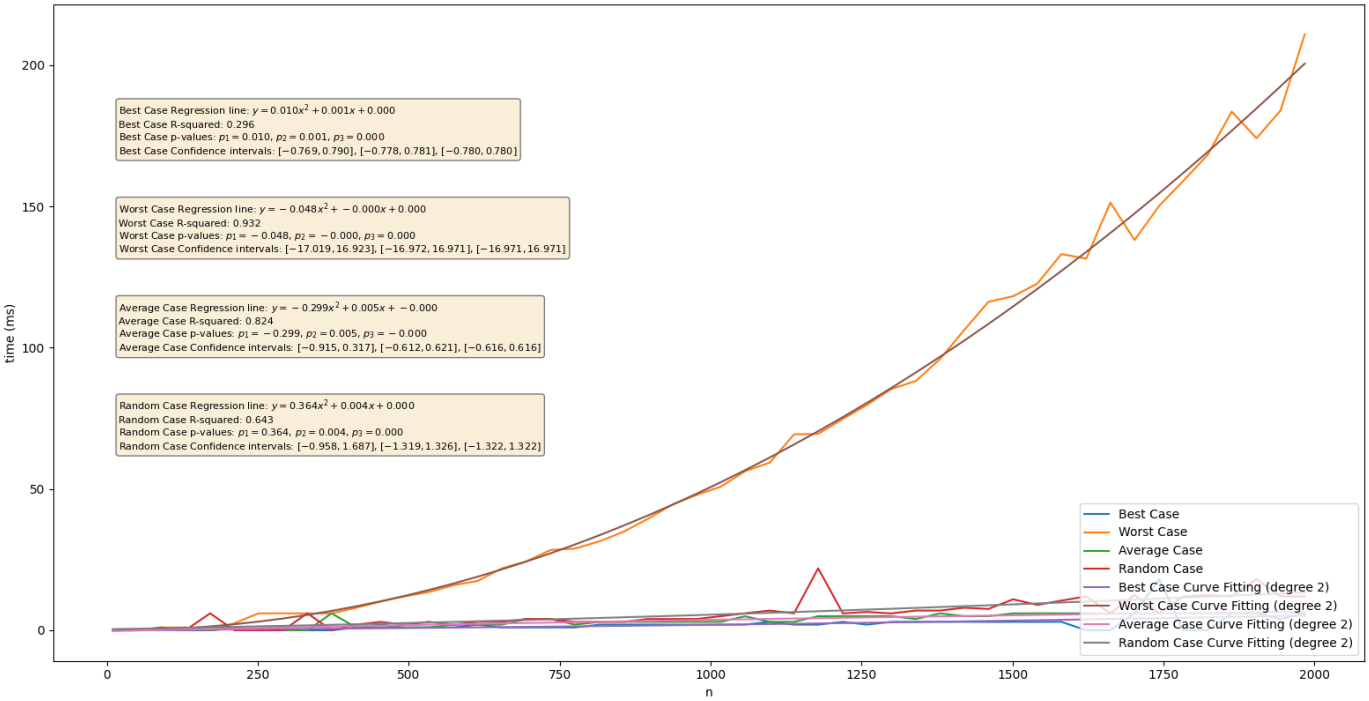

Figure 19.1: Graph showing all 4 cases

Case	Average Time (ms)
Best Case	2.16935e+00
Worst Case	6.77005e+01
Average Case	3.50677e+00
Random Case	5.98468e+00

Figure 19.2: Table showing the times of all 4 cases

4.3 Evaluation and interpretation

Upon examining the graphs, it is clear that they have highlighted the significant performance improvement achieved by the optimized algorithm over the original brute force approach across all tested scenarios and cases.

The graphs depicting the worst case scenario closely align with their expected behavior in that they resemble the graph of n^2 (but, with a much lower coefficient (better performance) than the brute force approach). However, the graphs for other cases consistently demonstrate much lower execution times, indicating that the algorithm's performance remains bounded by the theoretical complexity of $O(n^2)$ due to that worst case, with these better cases resembling the function of $O(nlogn)$. Additionally, these graphs often exhibit a consistently low profile, occasionally punctuated by sporadic spikes which are likely attributed to unfavorable sorting instances.

The coefficients in the regression line formula are statistically significant as they provide insight into how each coefficient contributes to the overall shape and structure of the best fit line. In all results, the x^2 coefficient has the highest magnitude, providing evidence that the algorithm's time complexity is indeed quadratic [3, 11]. However, in the optimised algorithm the x^2 coefficient has a much lower magnitude compared to that of the brute force, indicating that it is growing slower.

The curve fittings added to each of the brute force graphs have high R^2 values, indicating that the curves fit the data well, and there is not much variation in the data. This makes sense as the brute force approach performs the same amount of comparison regardless of rectangle configuration. The low R^2 values for the optimised algorithm's graphs show that there is variation in the data (its performance varies). This makes sense as the configuration and placement of rectangles was theorised to have a big impact on the performance of the algorithm from run to run. This shows that the algorithms adhere to their respective functions quite reliably. The R^2 value is an indicator of how consistently the algorithm performs for the given input data, and signifies that the algorithm's time complexity is a good fit for the input data [2].

The p-values provide an indication of how statistically significant the coefficients are. The x^2 coefficient has a high (and increasing) p-value, whereas the other coefficients have extremely low p-values. Therefore, the low values for the non-x^2 coefficients should be considered statistically relevant [7].

The evaluation of the experimental results provides valuable support for the theoretical analysis conducted earlier. The observed graph shapes, complemented by statistical data, confirm the expected performance characteristics of the algorithm.

In particular, the worst-case scenario serves as a significant validation point, as it pushes the algorithm to its limits and closely resembles the projected upper bound. The graph corresponding to the worst case exhibits a shape reminiscent of the n^2 curve (with a low coefficient), while the graphs of the other cases are shaped like that of a $nlogn$ curve, as predicted. This finding reinforces the notion that the optimized algorithm significantly outperforms the brute force approach, showcasing a substantial reduction in execution time.

5 Conclusion

The experiment conducted aimed to develop an optimized algorithm for efficiently identifying vertical adjacencies between orthogonal rectangles, and then providing a theoretical analysis of this algorithm. The objective was to surpass the limitations of the previous brute force approach by reducing computational time complexity and eliminating unnecessary computations.

Through the analysis of the experimental results, it has been demonstrated that the developed optimized algorithm significantly outperforms the brute force algorithm in terms of execution time. The graphs depicting the algorithm's performance clearly exhibit the expected shapes, confirming the algorithm's improved performance.

The experiment successfully achieved its objectives by developing and evaluating the optimized algorithm, comparing it with the brute force approach. The theoretical analysis conducted prior to the experiment aligns with the empirical findings, providing a comprehensive understanding of the algorithm's performance characteristics. The results obtained from this research provide valuable insights into the scalability and efficiency of the optimized algorithm, emphasising its practical significance and potential for broader applications.

References

[1] Abhishek Dey. Sweep line. https://www.thealgorists.com/Algo/SweepLine, 2023.

[2] Jason Fernando. R-squared: Definition, calculation formula, uses, and limitations. https://www.investopedia.com/terms/r/r-squared.asp, 2023.

[3] Jim Frost. Least squares regression: Definition, formulas example. https://statisticsbyjim.com/regression/least-squares-regression-line/, 2023.

[4] Grant Jenks. Performance comparison. https://grantjenks.com/docs/sortedcontainers/performance.html, 2019.

[5] Grant Jenks. Python sorted containers. https://grantjenks.com/docs/sortedcontainers/?ref=morioh.com&utm_source=morioh.com#api-documentation, 2019.

[6] Wasiq Khan, Abir Hussain, Kaya Kuru, and Haya Al-askar. Pupil localisation and eye centre estimation using machine learning and computer vision. http://dx.doi.org/10.3390/s20133785, 2020.

[7] Saul Mcleod. P-value and statistical significance: What it is why it matters. https://www.simplypsychology.org/p-value.html, 2023.

[8] Jaakko Moisio. Why sorting cannot be done faster than o(n log n)? https://www.jmoisio.eu/en/blog/2020/09/12/why-sorting-cannot-be-done-faster-n-log-n/, 2020.

[9] Ian Sanders. Intersections of horizontal and vertical line segments, 2023.

[10] Ian Sanders and Leigh-Ann Kenny. Heuristics for placing non-orthogonal axial lines to cross the adjacencies between orthogonal rectangles. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=352919c15bf0cf860da53824a1351f71f6006366, 2001.

[11] Jake VanderPlas. Statistics solutions. https://jakevdp.github.io/PythonDataScienceHandbook/05.03-hyperparameters-and-model-validation.html, 2016.