Comparison of A* and Dijkstra's algorithms

How does the A* algorithm compare to Dijkstra's algorithm in terms of efficiency and accuracy for pathfinding problems?

Computer science (HL)

Extended Essay

Word count: 3732

Table of content

Table of content **1**
Introduction **3**
Background **4**
What is a pathfinding algorithm? 4
Dijkstra's Algorithm **6**
History 6
Working Principle 7
Key characteristics 7
Variations and Optimizations 8
A* Algorithm **8**
History 8
Working Principles 9
What is a Heuristic 9
1. Euclidean distance 9
2. Manhattan distance 10
Variations and Optimizations 10
Methodology **10**
Hypotheses 10
Objective 11
Environment 11
Simulation Environment 12
Algorithm implementation 15
Data Collection 18
Criteria for Evaluation 18
Comparative analysis **19**
Test 1: Small-sized map (10*10) 19
Test 1: Overview and Analysis 21
Test 1: Implications 21
Test 2: Medium-sized map (50*50) 21
Test 2: Overview and Analysis 24
Test 2: Implications 24
Test 3: Large-size map (100*100) 25
Test 3: Overview and Analysis 26
Test 3: Implications 27
Limitations and Recommendations 27
Conclusions **28**
Summary of Findings and Concluding Thoughts 28
Bibliography **29**

Appendix..... 30
Java Code.....30

Introduction

In today's world, how would people get around town if not for GPS or Google Maps? How would our robots know the path they need to walk? All of these problems are solved with the implementation of path-finding algorithms. Path-finding algorithms are the backbone of many modern-day technologies we rely on for navigation and precision. From the vast majority of path-finding algorithms in today's computer science realm this essay will be comparing the leaders of the path-finding algorithms which are Dijkstra's algorithm and the A* algorithm. Dijkstra's algorithm was the leader of path-finding algorithms for many years and still is to this day. With its simplicity of implementation, speed and accuracy of operations. The second algorithm which this essay will be comparing is the A* algorithm which is relatively new compared to Dijkstra's algorithm but has proven to be revolutionary with its efficient use of heuristic properties and speed of operations.

However, the complexities of these two algorithms bring forth an interesting question: How do they compare in the real of efficiency and accuracy when finding the shortest path? This essay will answer this long-awaited question and provide a comprehensive analysis of both algorithms. The comparison will be delving into each algorithm's working principles. Figuring out how they both work, what is the accuracy of each algorithm, how much computing power they take up when facing exponentially increasing problems, how easy it is to implement each algorithm and the efficiency of these algorithms when dealing with issues found in the modern day world.

The significance of this study is to analyze and show which path-finding algorithm is best in which scenario. With this information computer scientists, engineers and

enthusiasts will not be having difficulties when faced with the question of which algorithm to choose in which scenario. It has the potential to help people make informed decisions when choosing the most suitable algorithm for their specific path-finding needs. By understanding the advantages and disadvantages of the A* and Dijksta's algorithm researchers and engineers can optimize the performance of their applications and be sure of faster and more accurate results.

This essay is structured to first provide insight into each path-finding algorithm, followed by an in-depth comparative analysis of the A* algorithm and Dijkstra's algorithm and finally, a conclusion summarizing the finding and answering the research question of: "How does the A* algorithm compare to Dijkstra's algorithm in terms of efficiency and accuracy for pathfinding problems?".

Background

What is a pathfinding algorithm?

Before going into the topic of comparing both pathfinding algorithms the question: "What is a path-finding algorithm, where are they used and what specific purpose do they serve in a practical sense as well as in the computer science realm?" should be answered. To begin the following question of "What is a path-finding algorithm?" should be answered.

A path-finding algorithm is a computational procedure to find the best path between two points in a given space. The "best" path can be defined in many ways. It can be the fastest path, the shortest distance, or the least cost. It all depends on the problem and

the criteria to solve that problem. Graphs are commonly used together with path-finding algorithms where the locations are represented as nodes (or vertices) and the paths between them as edges. The edges can then be modified to have weights, representing distances, cost, or any other metric relevant to the problem. Then when the algorithm has its space and weights it can start to use different search techniques in order to fulfill its goal. Pathfinding algorithms have a rich history dating back to the 1950s-1960s when the first pathfinding algorithm was created by Edsger W. Dijkstra in 1956 the algorithm is known today as Dijkstra's algorithm. Later on, the A* algorithm was founded in 1968 by Peter Hart, Nils Nilsson and Bertram Raphael. Later on, other algorithms appeared but they were variants of either algorithm with specific uses in the industry.

The following industries are where these algorithms could be used.

1. Navigation Systems:
    a. The most popular and everyday usage of these algorithms is the modern-day navigation system or GPS. The GPS has only one goal in mind. To go from point A to point B in the fastest way possible. This is where these kinds of algorithms shine since their whole purpose is to find the fastest route.
2. Video Games:
    a. Video games also use path-finding algorithms in order to make the game either look more realistic by having characters drive in a specific way or to make the game more challenging. For example, in racing games the use of these algorithms could help increase the difficulty by having your opponents drive the slowest or the fastest route possible.

3. Robotics:
    a. It would not be right if robots were not mentioned in this list. Robots in particular the ones that have to move around in the environment have to have some pathfinding algorithm so that they move around efficiently and avoid any obstacles in their path.
4. Network Routing
    a. Even in situations where a pathfinding algorithm would appear to be illogical, these algorithms are still used. In network routing, these algorithms serve the purpose of determining the most efficient data transmission routes.

Dijkstra’s Algorithm

History

Dijkstra's algorithm was first created in 1956 by Edsger W. Dijkstra. It has now become the foundation of all other pathfinding algorithms. Famous for its use in solving shortest-path problems on a graph (Dijkstra, 1959). The algorithm works on a weighted graph with each edge having some kind of value on it representing the weight then the algorithm moves on it which will be later explained. The algorithm is relevant until now because of its wide use in the roles of optimizing network flow, resource allocation and route planning across different applications (Tanenbaum & Wetherall, 2010)

Working Principle

As mentioned above Dijkstra's algorithm works on a weighted graph with nodes. It first initializes the starting values of all nodes making the starting node zero and all other nodes infinity. Then it starts to go through the graph updating the values of the nodes. It updates the values by going through all the paths from the start point and checking whether the weights of the lines or edges connecting the nodes add up to more or less than the value on the node. If the value is less than the node value is updated and the path is chosen to be the shortest, if it is the opposite the value of the node stays the same and the path is never touched again. A priority queue is often implemented to help more effectively select the next node on the path and also ensure that the algorithm is indeed moving along the shortest path. This process continues until the endpoint is reached and the shortest path is then found (Cormen et al., 2009).

Key characteristics

The algorithm has an impressive computational efficiency with the time complexity of O(V^2) for a graph of V vertices. But that's not it for the time complexity of this algorithm could be reduced even further with the use of a priority queue making the time complexity O(E + V log V) (Cormen et al., 2009). However, it must be noted that the algorithm works only with non-negative weights, if by any chance negative weights were to be used in a graph and this algorithm was to be used, it could give incorrect results.

Variations and Optimizations

From 1956 up until now Dikstra's algorithm has undergone many changes and modifications from different scientists to increase the algorithm's performance. One of the most popular adaptations is the Bidirectional Dijkstra's algorithm, which simultaneously explores paths from the start and the target nodes until they meet reducing the computational time (Goldberg & Harrelson, 2005). Another optimization has adopted the Fibonacci heaps, which decreases the algorithm's complexity to O(E + V log V) making the algorithm more efficient in sparse graphs (Fredman & Tarjan, 1987). These variations and optimizations of Dijkstra's algorithm highlight the versatility of this algorithm and its relevance in the field of computer science.

A* Algorithm

History

The A* algorithm was developed in 1968 by Hart, Nilsson and Raphael and since then it has made huge advancements in pathfinding and graph traversal algorithms (Hart, Nilsson, & Raphael, 1968). Made originally as an advancement of Dijkstra's algorithm it introduces heuristics to estimate the cost of reaching the end goal. Resulting in faster and more efficient pathfinding in many scenarios. The algorithm has now found widespread application in various fields, including computer gaming, robotics, and artificial intelligence.

Working Principles

The A* algorithm operates by maintaining a tree of paths originating at the start node and expanding to other nodes one edge at a time till it reaches the end. It employs the “best-first” search strategy which involves the usage of a priority queue to perform a repeated selection of minimum (estimated) cost nodes to expand. The algorithm uses both the actual cost from the start node to a given node and a heuristic estimate of the cost from that given node to the goal. This combination ensures that the A* algorithm moves using the fastest route often resulting in a much faster result time in scenarios with complex and vast search spaces (Russell & Norvig, 2009).

What is a Heuristic

In the context of the A* algorithm, the heuristic is a function that estimates the cost from the current point to the end goal. The exact formula for the heuristic depends on the type of problem faced. However, a common choice of pathfinding problems in a two-dimensional space is the Euclidean distance, used when agents can only move freely in any direction, or the Manhattan distance, used when agents can only move along grid lines, between the current node and the goal.

1. Euclidean distance
    a. The Euclidean distance between two points (x_1, y_1) and (x_2, y_2) in a two-dimensional space is calculated using the formula:

$$h(n) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$ (Suryadibrata et al., 2019)

2. Manhattan distance

    a. The Manhattan distance, also known as L1 norm or taxicab distance, between two points (x_1, y_1) and (x_2, y_2) is calculated via the formula:

$$h(n) = |x_2 - x_1| + |y_2 - y_1|$$ (Craw, 2017)

In these formulas, h(n) represents the heuristic value of a node n, and (x_1, y_1) and (x_2, y_2) represent the coordinates of that node and the goal node, respectively.

Variations and Optimizations

As time advanced the algorithm has seen many variants and optimisations to fill the gaps of specific problem domains and requirements. Techniques such as pattern databases, jump point search, and hierarchical pathfinding have been introduced to enhance the algorithm's performance, mostly recognised in scenarios with large databases and complex search spaces (Sturtevant, 2012).

Methodology

Hypotheses

The following hypothesis was created for this experiment. Dijkstra's algorithm will be slower in every test because it does not have the heuristic functions in it. It is also not made specifically for these problems making it not optimized to its full potential. The only test where Dijkstra's algorithm will be more or less successful will be the small map test or in other words the test that is least complex and does not require many

operations to be made. It will be successful there not because it will be better but because the difference will be so minimal that the hassle of implementing A* would not be worth it in the end results. As for the A* algorithm, it will be better in all tests in terms of speed and accuracy. The A* algorithm uses heuristic functions as well as a priority queue which makes it a lot faster than Dijkstra's algorithm, especially in tasks with increasing difficulty and complexity. Because Dijkstra's algorithm will be trying to search more paths while the A* algorithm will have a more or less clear path to the end.

Objective

The objective is to make a program that would be able to give both of these algorithms some kind of test. In this case, the program conducts a head-to-head comparison between Dijkstra's algorithm and the A* algorithm in a maze-solving context, measuring their performance in terms of speed efficiency, and accuracy. It was chosen to be a maze-solving-context because a maze is the closest example of streets. Each street could be thought as a cell in a maze. The only difference is that the streets in a maze are limited to four different entry and exit points. But besides that, it is the perfect example where it could be shown how the algorithm travels in an environment that is limited by some factors like walls and dead-ends. It would perfectly mimic a street with the walls being sides of the street and dead-ends being the streets with dead-ends.

Environment

The program was run on a desktop computer with hardware configurations of:

- GeForce RTX 3070 graphics card

- AMD Ryzen 7 5800x CPU
- 32 GB of RAM

The choice of programming language was Java since it has a large community of developers with resources and support, the huge list of libraries that were used in this project, performance wise Java Just-In-Time (JIT) compiler is relatively fast compared to other languages, and the use of GUI libraries that were very helpful in this project.

Simulation Environment

The program simulates a grid of two arrays that is equal on all edges. The program then creates a maze using the Depth-First Search (DFT) algorithm which is an algorithm which visits all the nodes in a stack which for our environment would be the maze cells. It does that by going from a cell to all of its neighbours. It was the most simple and efficient way to make a randomly generated maze every time. DFT also ensures that the maze always has an end since the algorithm visits every cell and ensures connectivity. The starting and end points can be chosen by the user but in this project, they were always set to the top left and bottom right corners. To ensure that there is no subjectivity and that the test is objective and fact-based.

Code of maze generation:

```java
private void generateMaze() {
    int gridSize = 100;  // Increased grid size
    grid = new Cell[gridSize][gridSize];

    // Initialize all cells
    for (int i = 0; i < gridSize; i++) {
        for (int j = 0; j < gridSize; j++) {
            grid[i][j] = new Cell(i, j, isWall: false);
        }
    }

    // Set start and end points
    start = grid[0][0];
    end = grid[gridSize - 1][gridSize - 1];

    // Ensure the starting and ending cells are not walls
    start.isWall = false;
    end.isWall = false;

    // Start maze generation from the first cell
    start.visited = true;
    generateMaze(start);

    panel.repaint();  // Refresh the panel to display the new maze
}
```

Figure 1: Code of maze generation. Made by author in IntelliJ IDEA

```
void generateMaze(Cell start) {
    List<int[]> directions = Arrays.asList(
            new int[]{-1, 0}, // up
            new int[]{1, 0},  // down
            new int[]{0, -1}, // left
            new int[]{0, 1}   // right
    );

    Collections.shuffle(directions); // randomize the directions

    for (int[] direction : directions) {
        int newX = start.x + direction[0];
        int newY = start.y + direction[1];

        if (newX >= 0 && newX < grid.length && newY >= 0 && newY < grid[0].length) {
            Cell nextCell = grid[newX][newY];

            if (!nextCell.visited) {
                nextCell.visited = true;

                // Break wall between cells
                if (direction[0] == -1) {
                    start.topWall = false;
                    nextCell.bottomWall = false;
                } else if (direction[0] == 1) {
                    start.bottomWall = false;
                    nextCell.topWall = false;
                } else if (direction[1] == -1) {
                    start.leftWall = false;
                    nextCell.rightWall = false;
                } else if (direction[1] == 1) {
                    start.rightWall = false;
                    nextCell.leftWall = false;
                }

                generateMaze(nextCell);
            }
        }
    }
}
```

Figure 2: Code of maze generation. Made by author in IntelliJ IDEA

Algorithm implementation

Both Dijkstra's algorithm as well as A* have been implemented in the program with standard configurations meaning that both of the algorithms are not optimized nor are they changed in any way to make them faster in a maze. The only difference is that the A* algorithm is using Manhattan heuristic function considering the grid-based constraints.

A* algorithm implementation:

```
private void runAstar(JLabel timeLabel, JLabel blocksLabel) {
    new Thread(() -> {
        // Reset state of all cells
        for (Cell[] row : grid) {
            for (Cell cell : row) {
                cell.gCost = Integer.MAX_VALUE;
                cell.hCost = 0;
                cell.previous = null;
                cell.state = Cell.State.OPEN;
            }
        }

        // Start the timer and explored blocks counter
        final int[] exploredBlocks = {0};
        final long[] startTime = {System.currentTimeMillis()};

        PriorityQueue<Cell> openSet = new PriorityQueue<>(Comparator.comparingInt(this::getFCost));
        start.gCost = 0;
        openSet.add(start);
        while (!openSet.isEmpty()) {
            Cell current = openSet.poll();
            current.state = Cell.State.CLOSED;
            SwingUtilities.invokeLater(panel::repaint);
            try { Thread.sleep( millis: 10); } catch (InterruptedException ignored) {}

            exploredBlocks[0]++;  // Increment explored blocks counter

            if (current.equals(end)) {
                reconstructPath(current);
                SwingUtilities.invokeLater(panel::repaint);

                long endTime = System.currentTimeMillis();  // End the timer

                System.out.println("Path found");
                System.out.println("A* Explored blocks: " + exploredBlocks[0]);
                System.out.println("A* Time taken: " + (endTime - startTime[0]) + "ms");

                SwingUtilities.invokeLater(() -> {
                    blocksLabel.setText("A* Explored Blocks: " + exploredBlocks[0]);
                    timeLabel.setText("A* Time Taken: " + (endTime - startTime[0]) + "ms");
                });

                return;
            }

            for (Cell neighbor : getNeighbors(current)) {
                if (neighbor.isWall)
                    continue;

                int tentativeGCost = current.gCost + distanceBetween(current, neighbor);
                if (tentativeGCost < neighbor.gCost) {
                    neighbor.previous = current;
                    neighbor.gCost = tentativeGCost;
                    neighbor.hCost = heuristic(neighbor, end);
                    if (!openSet.contains(neighbor)) {
                        neighbor.state = Cell.State.OPEN;
                        openSet.add(neighbor);
                    }
                }
            }
        }

        System.out.println("No path found");
    }).start();
}
```

Figure 3: A* implementation. Made by author in IntelliJ IDEA

Dijkstra’s algorithm implementation:

```java
private void runDijkstra(JLabel timeLabel, JLabel blocksLabel) {
    new Thread(() -> {
        // Reset state of all cells
        for (Cell[] row : grid) {
            for (Cell cell : row) {
                cell.gCost = Integer.MAX_VALUE;
                cell.hCost = 0;
                cell.previous = null;
                cell.state = Cell.State.OPEN;
            }
        }

        // Start the timer and explored blocks counter
        final int[] exploredBlocks = {0};
        final long[] startTime = {System.currentTimeMillis()};

        PriorityQueue<Cell> openSet = new PriorityQueue<>(Comparator.comparingInt(this::getGCost));
        start.gCost = 0;
        openSet.add(start);
        while (!openSet.isEmpty()) {
            Cell current = openSet.poll();
            current.state = Cell.State.CLOSED;
            SwingUtilities.invokeLater(panel::repaint);
            try { Thread.sleep( millis: 10); } catch (InterruptedException ignored) {}

            exploredBlocks[0]++;  // Increment explored blocks counter

            if (current.equals(end)) {
                reconstructPath(current);
                SwingUtilities.invokeLater(panel::repaint);

                long endTime = System.currentTimeMillis();  // End the timer

                System.out.println("Path found");
                System.out.println("Dijkstra's Explored blocks: " + exploredBlocks[0]);
                System.out.println("Dijkstra's Time taken: " + (endTime - startTime[0]) + "ms");

                SwingUtilities.invokeLater(() -> {
                    blocksLabel.setText("Dijkstra's Explored Blocks: " + exploredBlocks[0]);
                    timeLabel.setText("Dijkstra's Time Taken: " + (endTime - startTime[0]) + "ms");
                });

                return;
            }

            for (Cell neighbor : getNeighbors(current)) {
                if (neighbor.isWall)
                    continue;

                int tentativeGCost = current.gCost + distanceBetween(current, neighbor);
                if (tentativeGCost < neighbor.gCost) {
                    neighbor.previous = current;
                    neighbor.gCost = tentativeGCost;
                    if (!openSet.contains(neighbor)) {
                        neighbor.state = Cell.State.OPEN;
                        openSet.add(neighbor);
                    }
                }
            }
        }

        System.out.println("No path found");
    }).start();
}
```

Figure 3: Dijkstra's implementation. Made by author in IntelliJ IDEA

Data Collection

The program measures the number of cells explored, the shortest path length and the amount it took for each program to reach that time. The time however could be quicker but I have placed a short timer in order to have visuals of how the algorithms explore otherwise it would be nearly instant. The experiment will be conducted 10 times with small mazes that would be 10*10, 10 times with medium mazes that would be 50*50, and 5 times with large mazes that would be 100*100. Both of the algorithms will be tested on the same maze but the maze will change each time so there will be 10 maze changes in each maze size. The results will be placed on a table and analyzed. The graph will have 3 main parameters which will be time, cells explored and shortest path. The time parameter will determine the amount of time taken for the algorithm to reach the end. The cells explored parameter will determine how many cells the algorithm explored thus showing the algorithm's efficiency, and the shortest path parameter will show if both algorithms have found the shortest path.

Criteria for Evaluation

After exploring the basis of pathfinding algorithms and stating the implementation and challenges of these algorithms some criteria should be established to have a clear comparative analysis of the chosen algorithms. This essay will focus on the performance, time complexity of each algorithm, implementation difficulty and run-time. These criteria will help have a comprehensive and objective comparison of both of these algorithms. It will also help to establish the strengths and weaknesses of each algorithm in various scenarios.

Comparative analysis

Test 1: Small-sized map (10*10)

	Dijkstra's algorithm			A* algorithm		
Attempt	Time (ms)	Cells explored	Shortest path	Time (ms)	Cells explored	Shortest path
1	835	54	Found	764	50	Found
2	918	60	Found	741	48	Found
3	684	44	Found	671	43	Found
4	1484	97	Found	1375	89	Found
5	1120	73	Found	1002	66	Found
6	1204	78	Found	1178	76	Found
7	1277	83	Found	1185	77	Found
8	1112	72	Found	1046	68	Found
9	1397	91	Found	1142	74	Found
10	906	59	Found	756	49	Found

Table 1: 10*10 map results

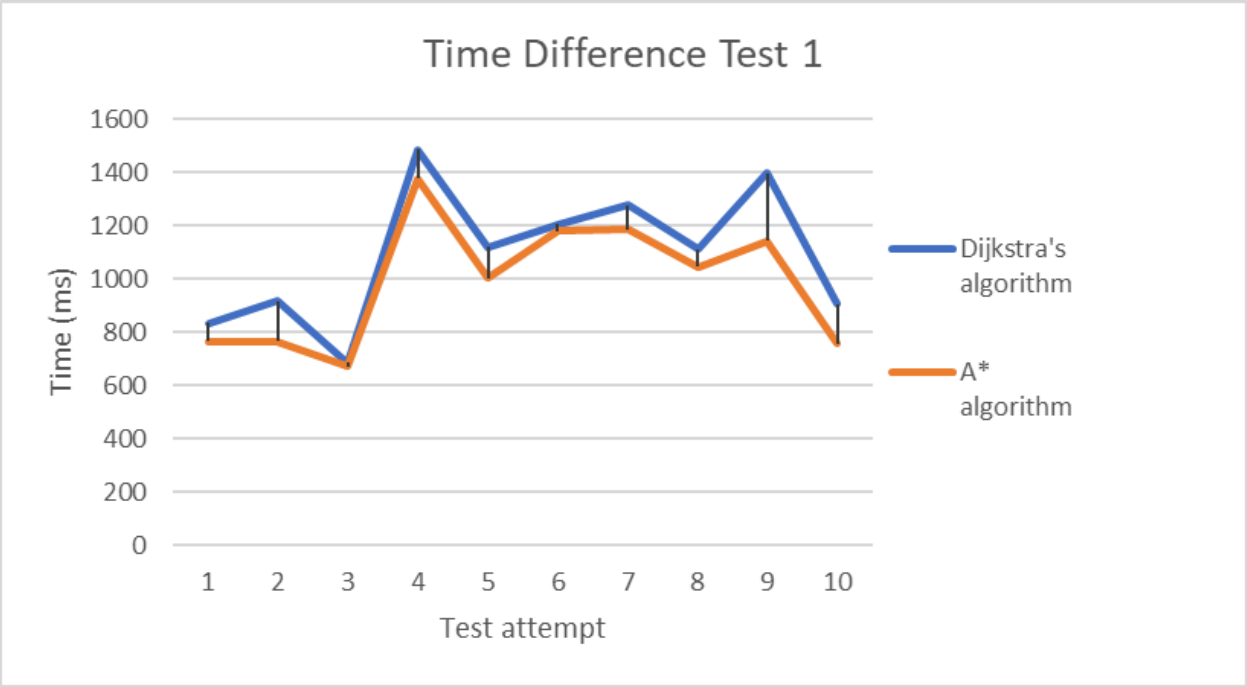

Chart 1: Run-Time difference from A* to Dijkstra's algorithms during Test 1

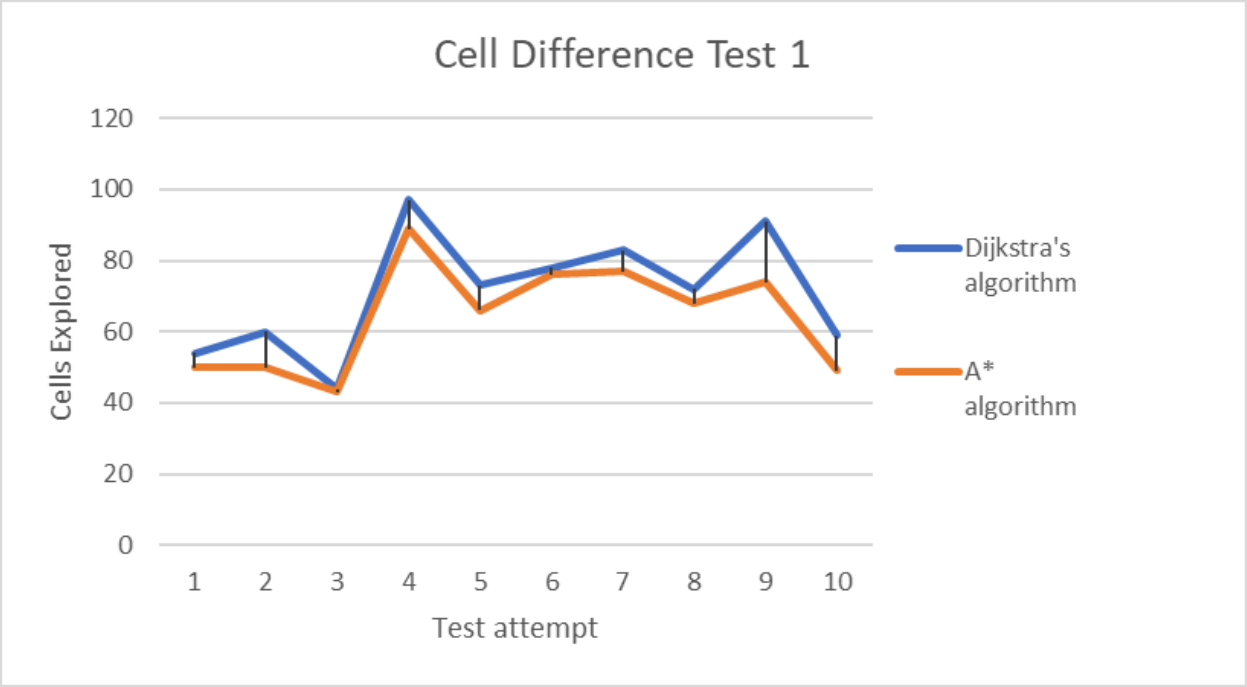

Chart 2: Amount of searched cells with A* and Dijkstra's algorithms during Test 1

Test 1: Overview and Analysis

As we can see the time it took for Dijkstra's algorithm to solve the maze ranges from 684 ms to 1484 ms with an average time of 10093.7 ms and cells searched in that time ranges from 44 to 97 with an average of 71.1. The results for A* are not so different time taken ranges from 671 ms to 1375 ms with an average of 988.3 ms and the cells searched ranges from 43 to 89 with an average of 64.2. The results show that the time efficiency of the A* algorithm is 9.63%. It is also 9.7% more accurate with fewer cells explored. This can be attributed to the heuristic function of the A* algorithm making the searches more accurate and efficient. Though the difference is not that significant indicating that although on small maps the efficiency of A* is present it is not noticeable enough to make a big difference.

Test 1: Implications

These results show us that on small maps, the A* algorithm does have the advantage but only slightly. For some programs, the slight benefit might not always justify the time and complexity of implementing the algorithm into the program. This is crucial considering for some applications simplicity and ease of implementation are essential.

Test 2: Medium-sized map (50*50)

	Dijkstra's algorithm			A* algorithm		
Attempt	Time (ms)	Cells	Shortest	Time (ms)	Cells	Shortest

		explored	path		explored	path
1	12444	1107	Found	10447	950	Found
2	15489	1416	Found	11548	1124	Found
3	21989	2024	Found	18749	1796	Found
4	7770	709	Found	5282	598	Found
5	12098	1117	Found	9800	978	Found
6	26054	2396	Found	23574	2046	Found
7	26366	2443	Found	23467	2234	Found
8	8056	744	Found	6541	567	Found
9	16203	1509	Found	14874	1369	Found
10	13406	1235	Found	12775	1097	Found

Table 2: 50*50 map results

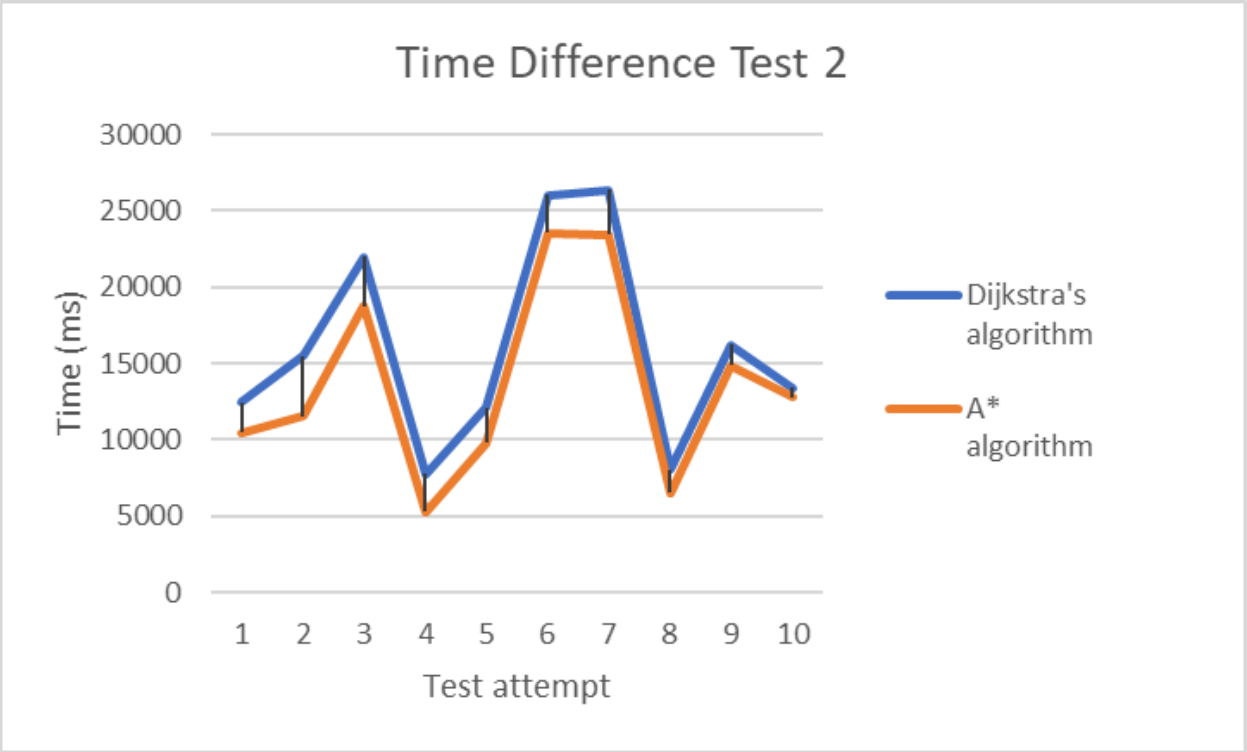

Chart 3: Run-Time difference from A* to Dijkstra's algorithms during Test 2

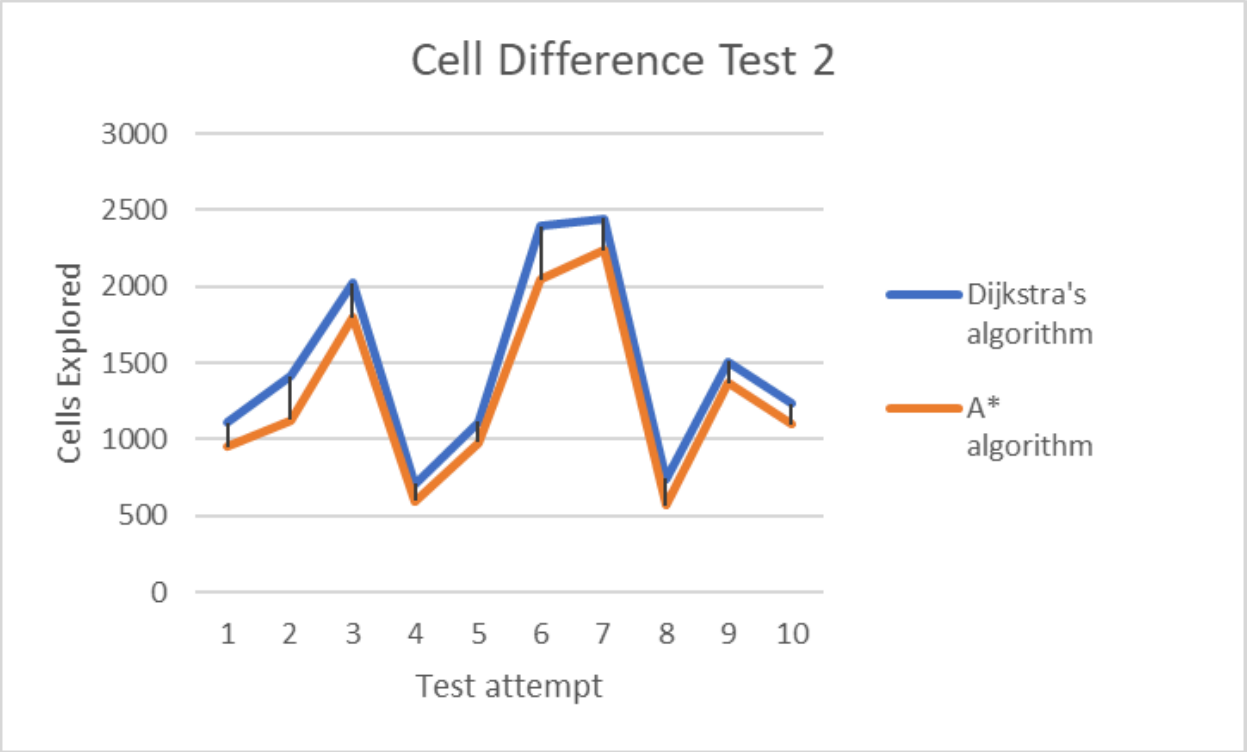

Chart 4: Amount of searched cells with A* and Dijkstra's algorithms during Test 2

Test 2: Overview and Analysis

Results of the second test reveal that both algorithms succeeded in their task of finding the end of the maze. The Dijkstra's algorithm had an average time of 15987.5 ms and explored an average of 1470 cells while the A* had an average time of 13705.7 ms and explored an average of 1275.9 cells. This time the results show that the A* algorithm had a 14.27% increase in time efficiency and an accuracy increase of 13.2% over Dijkstra's algorithm. A* showed a clear edge over Dijkstra's algorithm outperforming it 10 times out of 10. This performance increase can again be attributed to the heuristic function coupled with the use of a priority queue in the A* algorithm.

Test 2: Implications

These results prove that with more complex and larger environments the A* algorithm takes its place as the best algorithm out of the two. Having a clear increase in the time efficiency which jumped from 9.63% to 14.27%. This increase could also have happened because in the previous test, the algorithm didn't have the space nor the time to show its capabilities and now that it does it can unleash its true potential. There has also been an increase in the accuracy jumping from 9.7% to 13.2% once again proving that with smaller environments the A* algorithm can not show its true potential since there isn't much to explore in the environment but with larger and more complex environments it dominates.

Test 3: Large-size map (100*100)

	Dijkstra's algorithm			A* algorithm		
Attempt	Time (ms)	Cells explored	Shortest path	Time (ms)	Cells explored	Shortest path
1	54571	4979	Found	43152	3630	Found
2	48256	4419	Found	32329	3765	Found
3	64467	5987	Found	48457	3752	Found
4	112884	9585	Found	88584	8275	Found
5	61398	5663	Found	52732	3409	Found

Table 3: 100*100 map results

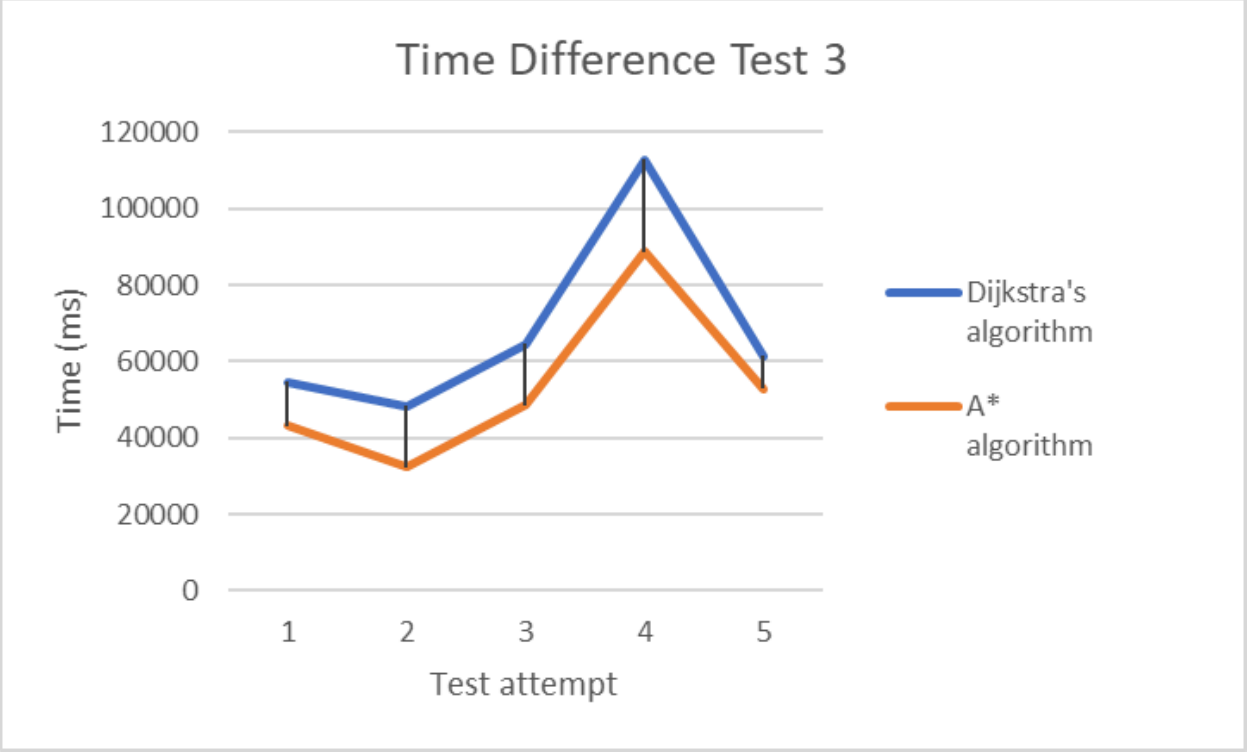

Chart 5: Run-Time difference from A* to Dijkstra's algorithms during Test 3

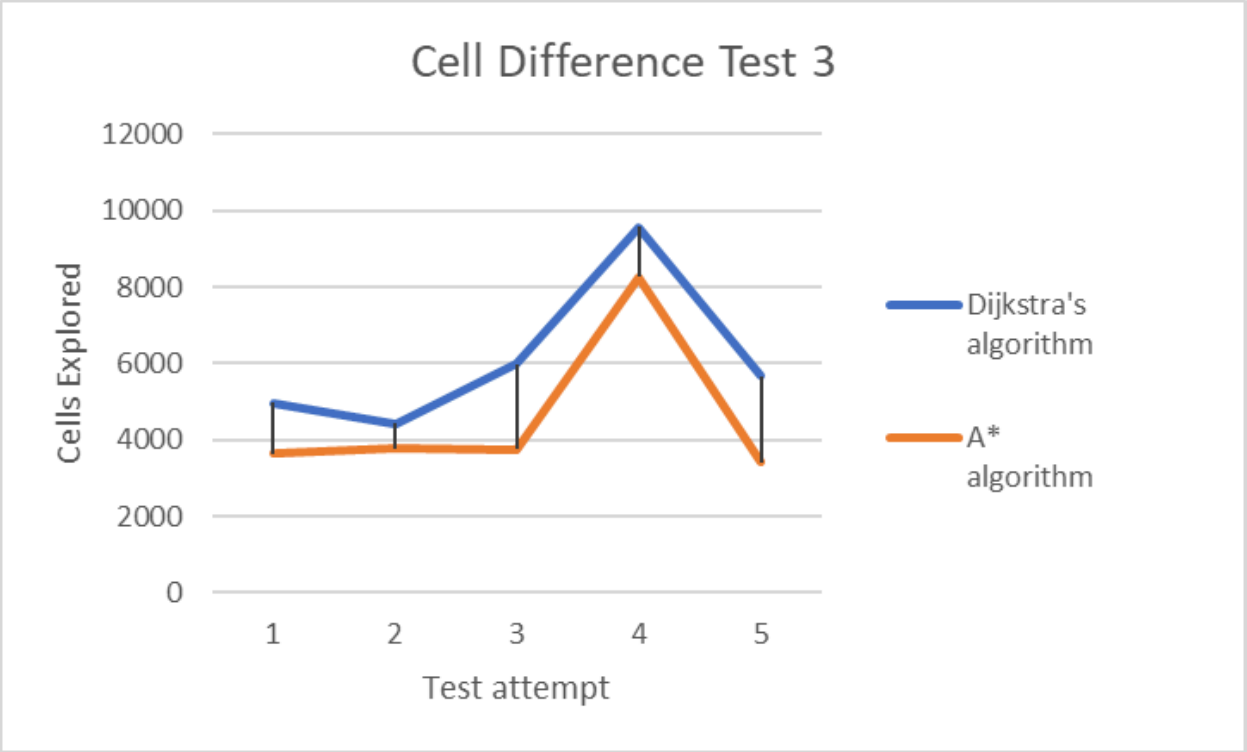

Chart 6: Amount of searched cells with A* and Dijkstra's algorithms during Test 3

Test 3: Overview and Analysis

The test results reveal that both algorithms succeed in their designed tasks and found the end in all five attempts. Dijkstra's algorithm had an average time of 68315.2 ms and explored an average of 6126.6 cells. While once again the A* algorithm had an advantage in all criteria with an average time of 53050.8 ms and explored an average of 4566.2 cells. This time the A* algorithm has shown a significant advantage when it comes to time efficiency and accuracy. It has approximately a 22.34% increase in time efficiency and a 25.47% increase in cell exploration compared to Dijkstra's algorithm. The performance gap is once again caused by the heuristic functions of the A*

algorithm making it evident that on larger data sets the A* algorithm beats Dijkstra's algorithm by a long shot.

Test 3: Implications

These findings prove that the A* algorithm is the superior algorithm in terms of pathfinding problem-solving. Although Dijkstra's algorithm was lost in all tests it would still have uses in smaller and less complex environments since it is very easy to implement and the difference between the two algorithms on smaller datasets is not significant. But when it comes to larger and more complex environments it would be the better choice to choose the A* algorithm because of its heuristic functions and priority queue implementation. Its ability to use these functions results in substantial gains in terms of making it preferable for time-sensitive applications in intricate settings.

Limitations and Recommendations

While the study provides valuable insight it is not without limitations. The tests were only done on a maze with one path to the end. It would have been interesting to see whether the results would change if more paths were added to the end. Another limitation was that the results are only from the maze even though these algorithms have so many other uses than navigating a maze. The third and last limitation of this study is since the maze is randomly generated recreating these test results would be almost impossible.

Future researchers could test both of these algorithms in more than just maze navigation. They could also do more tests that would ensure unbiased results. Another recommendation for future researchers is to have settings pre-made so that they could

be recreated and tested by others and the real impact of each algorithm could be seen by other researchers.

Conclusions

Summary of Findings and Concluding Thoughts

In this comparative analysis between Dijkstra's and A* algorithms some important insights have been made regarding the performance of both algorithms in settings with different intensities.

- Small-sized Map: In simple and least intense settings the A* algorithm managed to win in terms of efficiency and accuracy. Although it was only by a short margin.
- Medium-sized Map: In a more intense setting the A* algorithm managed to get a considerable advantage compared to the small-sized map. Showing performance increases of up to 14.27% in time efficiency and 13.2% in accuracy.
- Large-sized Map: In the most intense setting the A* algorithm had made enormous increases in time efficiency and accuracy. Outperforming Dijkstra's algorithm by 22.34% in time efficiency and 25.47% in accuracy.

After testing both of the algorithms a conclusion can be made that in more complex and larger environments the A* is the better choice with a significant increase in time to solve the maze and also a significant decrease in cells searched while trying to reach

the end indicating that it explores less of the environment and will use much less memory. But when it comes to less complex and small environments it would be more beneficial to use Dijkstra's algorithm for its ease of use and results not far off from A*. It would take much less time to implement Dijkstra's algorithm compared to A* but in situations where there are larger settings it is worth it. Proving the hypothesis to be true.

Bibliography

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms, third edition*. http://portal.acm.org/citation.cfm?id=1614191

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, *1*(1), 269–271. https://doi.org/10.1007/bf01386390

Fredman, M. L., & Tarjan, R. E. (1987). Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, *34*(3), 596–615. https://doi.org/10.1145/28869.28874

Goldberg, A. V., & Harrelson, C. (2005). Computing the shortest path: A search meets graph theory. *Symposium on Discrete Algorithms*, 156–165. https://doi.org/10.5555/1070432.1070455

Hart, P. D., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, *4*(2), 100–107. https://doi.org/10.1109/tssc.1968.300136

Russell, S. J., & Norvig, P. (2010). *Artificial intelligence a modern approach*. London.

Sturtevant, N. R. (2012). Benchmarks for Grid-Based Pathfinding. *IEEE Transactions on Computational Intelligence and AI in Games*, *4*(2), 144–148. https://doi.org/10.1109/tciaig.2012.2197681

Tanenbaum, A. S., & Wetherall, D. (1981). *COMPUTER NETWORKS, 5TH EDITION*. http://repository.fue.edu.eg/xmlui/handle/123456789/5593

Craw, S. (2017). Manhattan distance. In *Springer eBooks* (pp. 790–791).

https://doi.org/10.1007/978-1-4899-7687-1_511

Suryadibrata, A., Young, J. C., & Luhulima, R. (2019). Review of Various A* Pathfinding Implementations in Game Autonomous Agent. International Journal of New Media Technology, 6(1), 43–49. https://doi.org/10.31937/ijnmt.v6i1.1075

Appendix

This is the Java code of the program that was made to compare both of these algorithms.

Javja Code

```
package ExtendedEssay;

import java.util.*;
import javax.swing.*;
import java.awt.*;
import java.util.List;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;

public class MazeApp {

   private static final Color COLOR_WALL = Color.BLACK;
   private static final Color COLOR_START = Color.BLUE;
   private static final Color COLOR_END = Color.RED;
   private static final Color COLOR_OPEN = Color.WHITE;
   private static final Color COLOR_CLOSED = Color.GRAY;
   private static final Color COLOR_PATH = Color.GREEN;

   private Cell[][] grid;
   private Cell start, end;
   private JFrame frame;
   private MazePanel panel;

   public MazeApp() {
       initGUI();
   }

   private void initGUI() {
       frame = new JFrame("Pathfinding Visualizer");
       panel = new MazePanel();

       // Create a JPanel for the controls and stats
```

```
        JPanel controlPanel = new JPanel(new GridLayout(3, 2));

        // Create buttons
        JButton generateMazeButton = new JButton("Generate Maze");
        JButton runAstarButton = new JButton("Run A*");
        JButton runDijkstraButton = new JButton("Run Dijkstra");
        JButton clearButton = new JButton("Clear Start/End");

        // Create labels for the stats
        JLabel aStarTimeLabel = new JLabel("A* Time: ");
        JLabel aStarBlocksLabel = new JLabel("A* Blocks: ");
        JLabel dijkstraTimeLabel = new JLabel("Dijkstra Time: ");
        JLabel dijkstraBlocksLabel = new JLabel("Dijkstra Blocks: ");

        // Add action listeners
        generateMazeButton.addActionListener(e -> {
            generateMaze();
            aStarTimeLabel.setText("A* Time: ");
            aStarBlocksLabel.setText("A* Blocks: ");
            dijkstraTimeLabel.setText("Dijkstra Time: ");
            dijkstraBlocksLabel.setText("Dijkstra Blocks: ");
        });

        runAstarButton.addActionListener(e -> {
              runAstar(aStarTimeLabel, aStarBlocksLabel);  // Pass the labels to
the method
        });

        runDijkstraButton.addActionListener(e -> {
               runDijkstra(dijkstraTimeLabel, dijkstraBlocksLabel);  // Pass the
labels to the method
        });

        clearButton.addActionListener(e -> {
            start = null;
            end = null;
            for (int i = 0; i < grid.length; i++) {
                for (int j = 0; j < grid[0].length; j++) {
                    Cell cell = grid[i][j];
                    cell.visited = false;
                    cell.gCost = Integer.MAX_VALUE;
                    cell.hCost = 0;
                    cell.previous = null;
                }
            }
               panel.repaint();  // Repaint the panel to remove the start/end
points
        });
```

```
        // Add buttons and labels to the control panel
        controlPanel.add(generateMazeButton);
        controlPanel.add(runAstarButton);
        controlPanel.add(runDijkstraButton);
        controlPanel.add(clearButton);
        controlPanel.add(aStarTimeLabel);
        controlPanel.add(aStarBlocksLabel);
        controlPanel.add(dijkstraTimeLabel);
        controlPanel.add(dijkstraBlocksLabel);

        frame.setLayout(new BorderLayout());  // Set the layout manager
        frame.add(panel, BorderLayout.CENTER);  // Add the panel to the center
         frame.add(controlPanel, BorderLayout.NORTH);  // Add the control panel
to the top
        frame.setSize(800, 800);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }

    class Cell {
        int x;
        int y;
        boolean visited;
        boolean topWall = true;
        boolean bottomWall = true;
        boolean leftWall = true;
        boolean rightWall = true;
        boolean isWall;
        int gCost;
        int hCost;
        Cell previous;
        enum State {
            OPEN, CLOSED, PATH
        }

        State state;


        Cell(int x, int y, boolean isWall) {
            this.x = x;
            this.y = y;
            this.isWall = isWall;
            this.gCost = Integer.MAX_VALUE;
            this.hCost = 0;
            this.previous = null;
            this.state = State.OPEN;
        }
    }
```

```
    int getFCost(Cell cell) {
        return cell.gCost + cell.hCost;
    }

    int getGCost(Cell cell) {
        return cell.gCost;
    }

    void reconstructPath(Cell endCell) {
        Cell current = endCell;
        while (current != null) {
            current.state = Cell.State.PATH;
            current = current.previous;
        }
    }

    List<Cell> getNeighbors(Cell cell) {
        List<Cell> neighbors = new ArrayList<>();
         int[][] directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};  // Up, down,
left, right

        for (int[] direction : directions) {
            int newX = cell.x + direction[0];
            int newY = cell.y + direction[1];

              // Check if the coordinates are in bounds and the cell is not a
wall
                 if (newX >= 0 && newX < grid.length && newY >= 0 && newY <
grid[0].length && !grid[newX][newY].isWall) {
                   // Check if there's a wall between the current cell and the
neighbor
                if ((direction[0] == -1 && !cell.topWall) ||  // Up
                        (direction[0] == 1 && !cell.bottomWall) ||  // Down
                        (direction[1] == -1 && !cell.leftWall) ||  // Left
                        (direction[1] == 1 && !cell.rightWall)) {  // Right
                    neighbors.add(grid[newX][newY]);
                }
            }
        }

        return neighbors;
    }



    int distanceBetween(Cell cell1, Cell cell2) {
        return 1;
    }
```

```
    int heuristic(Cell cell, Cell end) {
        return Math.abs(cell.x - end.x) + Math.abs(cell.y - end.y);
    }

    private void generateMaze() {
        int gridSize = 100;  // Increased grid size
        grid = new Cell[gridSize][gridSize];

        // Initialize all cells
        for (int i = 0; i < gridSize; i++) {
            for (int j = 0; j < gridSize; j++) {
                grid[i][j] = new Cell(i, j, false);
            }
        }

        // Set start and end points
        start = grid[0][0];
        end = grid[gridSize - 1][gridSize - 1];

        // Ensure the starting and ending cells are not walls
        start.isWall = false;
        end.isWall = false;

        // Start maze generation from the first cell
        start.visited = true;
        generateMaze(start);

        panel.repaint();  // Refresh the panel to display the new maze
    }

    void generateMaze(Cell start) {
        List<int[]> directions = Arrays.asList(
                new int[]{-1, 0}, // up
                 new int[]{1, 0},  // down
                 new int[]{0, -1}, // left
                 new int[]{0, 1}   // right
        );

        Collections.shuffle(directions); // randomize the directions

        for (int[] direction : directions) {
            int newX = start.x + direction[0];
            int newY = start.y + direction[1];

                  if (newX >= 0 && newX < grid.length && newY >= 0 && newY <
grid[0].length) {
                Cell nextCell = grid[newX][newY];

                if (!nextCell.visited) {
```

```
                    nextCell.visited = true;

                    // Break wall between cells
                    if (direction[0] == -1) {
                        start.topWall = false;
                        nextCell.bottomWall = false;
                    } else if (direction[0] == 1) {
                        start.bottomWall = false;
                        nextCell.topWall = false;
                    } else if (direction[1] == -1) {
                        start.leftWall = false;
                        nextCell.rightWall = false;
                    } else if (direction[1] == 1) {
                        start.rightWall = false;
                        nextCell.leftWall = false;
                    }

                    generateMaze(nextCell);
                }
            }
        }
    }


    private void setStartEndPoints() {
        panel.repaint();
    }

    private void runAstar(JLabel timeLabel, JLabel blocksLabel) {
        new Thread(() -> {
            // Reset state of all cells
            for (Cell[] row : grid) {
                for (Cell cell : row) {
                    cell.gCost = Integer.MAX_VALUE;
                    cell.hCost = 0;
                    cell.previous = null;
                    cell.state = Cell.State.OPEN;
                }
            }

            // Start the timer and explored blocks counter
            final int[] exploredBlocks = {0};
            final long[] startTime = {System.currentTimeMillis()};

                                        PriorityQueue<Cell>    openSet    =    new
PriorityQueue<>(Comparator.comparingInt(this::getFCost));
            start.gCost = 0;
            openSet.add(start);
            while (!openSet.isEmpty()) {
```

```
                Cell current = openSet.poll();
                current.state = Cell.State.CLOSED;
                SwingUtilities.invokeLater(panel::repaint);
                  try { Thread.sleep(10); } catch (InterruptedException ignored)
{}

                exploredBlocks[0]++;  // Increment explored blocks counter

                if (current.equals(end)) {
                    reconstructPath(current);
                    SwingUtilities.invokeLater(panel::repaint);

                    long endTime = System.currentTimeMillis();  // End the timer

                     System.out.println("Path found");
                                 System.out.println("A* Explored blocks: " +
exploredBlocks[0]);
                            System.out.println("A* Time taken: " + (endTime -
startTime[0]) + "ms");

                    SwingUtilities.invokeLater(() -> {
                                 blocksLabel.setText("A* Explored Blocks: " +
exploredBlocks[0]);
                              timeLabel.setText("A* Time Taken: " + (endTime -
startTime[0]) + "ms");
                    });

                    return;
                }

                for (Cell neighbor : getNeighbors(current)) {
                    if (neighbor.isWall)
                        continue;

                                        int tentativeGCost = current.gCost +
distanceBetween(current, neighbor);
                    if (tentativeGCost < neighbor.gCost) {
                        neighbor.previous = current;
                        neighbor.gCost = tentativeGCost;
                        neighbor.hCost = heuristic(neighbor, end);
                        if (!openSet.contains(neighbor)) {
                            neighbor.state = Cell.State.OPEN;
                            openSet.add(neighbor);
                        }
                    }
                }
            }

            System.out.println("No path found");
```

```
        }).start();
    }

    private void runDijkstra(JLabel timeLabel, JLabel blocksLabel) {
        new Thread(() -> {
            // Reset state of all cells
            for (Cell[] row : grid) {
                for (Cell cell : row) {
                    cell.gCost = Integer.MAX_VALUE;
                    cell.hCost = 0;
                    cell.previous = null;
                    cell.state = Cell.State.OPEN;
                }
            }

            // Start the timer and explored blocks counter
            final int[] exploredBlocks = {0};
            final long[] startTime = {System.currentTimeMillis()};

                                        PriorityQueue<Cell>    openSet    =    new
PriorityQueue<>(Comparator.comparingInt(this::getGCost));
            start.gCost = 0;
            openSet.add(start);
            while (!openSet.isEmpty()) {
                Cell current = openSet.poll();
                current.state = Cell.State.CLOSED;
                SwingUtilities.invokeLater(panel::repaint);
                  try { Thread.sleep(10); } catch (InterruptedException ignored)
{}

                exploredBlocks[0]++;  // Increment explored blocks counter

                if (current.equals(end)) {
                    reconstructPath(current);
                    SwingUtilities.invokeLater(panel::repaint);

                    long endTime = System.currentTimeMillis();  // End the timer

                     System.out.println("Path found");
                            System.out.println("Dijkstra's Explored blocks: " +
exploredBlocks[0]);
                       System.out.println("Dijkstra's Time taken: " + (endTime -
startTime[0]) + "ms");

                    SwingUtilities.invokeLater(() -> {
                            blocksLabel.setText("Dijkstra's Explored Blocks: " +
exploredBlocks[0]);
                         timeLabel.setText("Dijkstra's Time Taken: " + (endTime -
startTime[0]) + "ms");
```

```
                    });

                    return;
                }

                for (Cell neighbor : getNeighbors(current)) {
                    if (neighbor.isWall)
                        continue;

                                            int  tentativeGCost  =  current.gCost  +
distanceBetween(current, neighbor);
                    if (tentativeGCost < neighbor.gCost) {
                        neighbor.previous = current;
                        neighbor.gCost = tentativeGCost;
                        if (!openSet.contains(neighbor)) {
                            neighbor.state = Cell.State.OPEN;
                            openSet.add(neighbor);
                        }
                    }
                }
            }

            System.out.println("No path found");
        }).start();
    }

    class MazePanel extends JPanel {

        int cellSize = 20; // Size of each cell

        MazePanel() {
            addMouseListener(new MouseAdapter() {
                @Override
                public void mouseClicked(MouseEvent e) {
                    int x = e.getX() / cellSize;
                    int y = e.getY() / cellSize;

                    if (start == null) {
                        if (!grid[y][x].isWall) {
                            start = grid[y][x];
                            start.gCost = 0;  // Add this line
                             repaint();
                        }
                    } else if (end == null) {
                        if (!grid[y][x].isWall && !grid[y][x].equals(start)) {
                            end = grid[y][x];
                            repaint();
                        }
                    }
```

```
                }
            });
        }

        @Override
        protected void paintComponent(Graphics g) {
            super.paintComponent(g);

            if (grid != null) {
                for (int i = 0; i < grid.length; i++) {
                    for (int j = 0; j < grid[0].length; j++) {
                        Cell cell = grid[i][j];

                        // Set the color based on the cell's state
                        if (cell.isWall) {
                            g.setColor(COLOR_WALL);
                        } else if (cell.equals(start)) {
                            g.setColor(COLOR_START);
                        } else if (cell.equals(end)) {
                            g.setColor(COLOR_END);
                        } else {
                            switch (cell.state) {
                                case OPEN:
                                    g.setColor(COLOR_OPEN);
                                    break;
                                case CLOSED:
                                    g.setColor(COLOR_CLOSED);
                                    break;
                                case PATH:
                                    g.setColor(COLOR_PATH);
                                    break;
                            }
                        }

                        // Draw the cell
                                g.fillRect(j * cellSize, i * cellSize, cellSize,
cellSize);

                        // Draw the cell's walls
                        g.setColor(COLOR_WALL);
                        if (cell.topWall) g.drawLine(j * cellSize, i * cellSize,
(j + 1) * cellSize, i * cellSize);
                          if (cell.rightWall) g.drawLine((j + 1) * cellSize, i *
cellSize, (j + 1) * cellSize, (i + 1) * cellSize);
                           if (cell.bottomWall) g.drawLine(j * cellSize, (i + 1) *
cellSize, (j + 1) * cellSize, (i + 1) * cellSize);
                            if (cell.leftWall) g.drawLine(j * cellSize, i *
cellSize, j * cellSize, (i + 1) * cellSize);
                    }
```

```
                }
            }
        }
    }

    public static void main(String[] args) {
        new MazeApp();
    }
}
```