# CUSTOM PROJECT REPORT

# COS10009

DANG KHANH TOAN NGUYEN 103797499

# A COMPARISON BETWEEN PATHFINDING ALGORITHMS

**Abstract:** This paper focuses on three different types of pathfinding algorithms. Along with the O-notation, time complexity and space complexity are also examined. This report article briefly provides a study based on BFS, DFS and Dijkstra, as well as test them using randomly generated sets of nodes in different environment in order to make a comparision between them.

## TABLE OF CONTENTS

## I. INTRODUCTION

These days, with the great development of science and technology, especially Computer Science is one of the fields with the most application in life. Besides, Artificial intelligence (AI) in Computer Science is a science that studies methods for computers to have thinking like humans. Furthermore, AI also researches methods that can make computer do things that at that time could not be done by humans and it has been rapidly grown in the last decade. One of the AI researches is pathfinding, it is performed using one of several algorithms that returns a path from the source point to the destination point. This project analyzes the three most common path-finding, gathers data and compares the speed and efficiency of each algorithm.

## II. PATHFINDING ALGORITHMS

### 1. BREAST FIRST SEARCH

Breadth-first search (BFS) is a search algorithm that uses **FIFO** queue (First in First out) in a graph in which the search consists of only two operations: (a) given a vertex of the graph; (b) add vertices with the given vertex to the next possible list. The breadth-first search algorithm can be used for two goals: finding the path from a given origin to a vertex target, and finding the path from the vertex to all other vertices. [1]

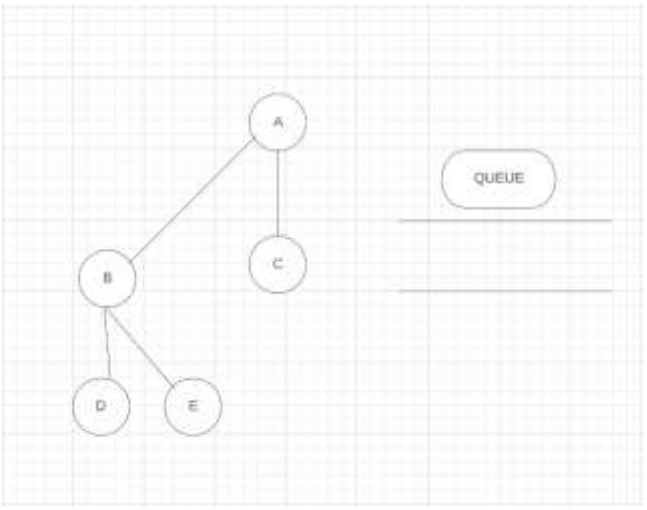
The basic idea of the algorithms is as follows:[2]

Step 1: Consider adjacent vertices that have not been traversed. Mark the vertex approved. Display that vertex and push it to the queue.

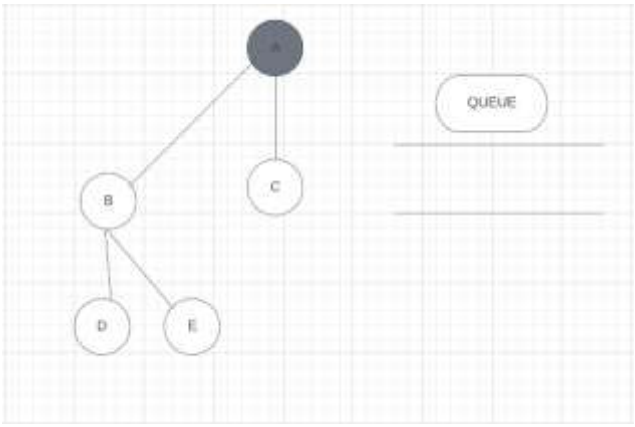
Step 2: If no adjacent vertex is found, delete the first vertex in the queue.

Step 3: Repeat step 1 and 2 until the queue is empty.

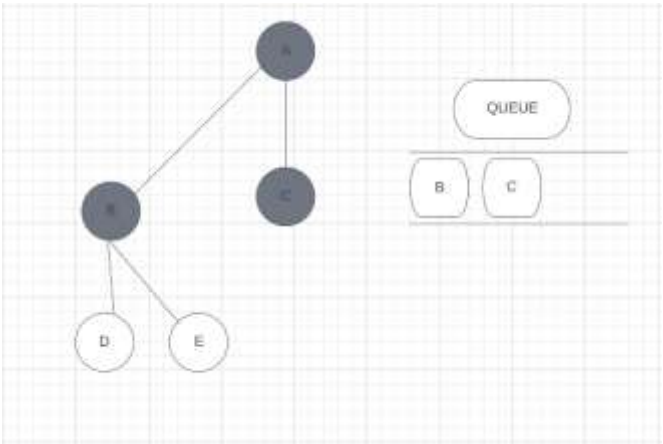
***Example:***



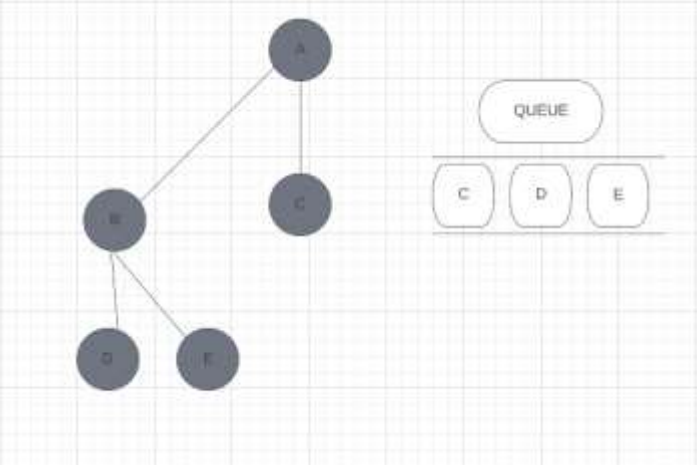

Starting from vertex A:



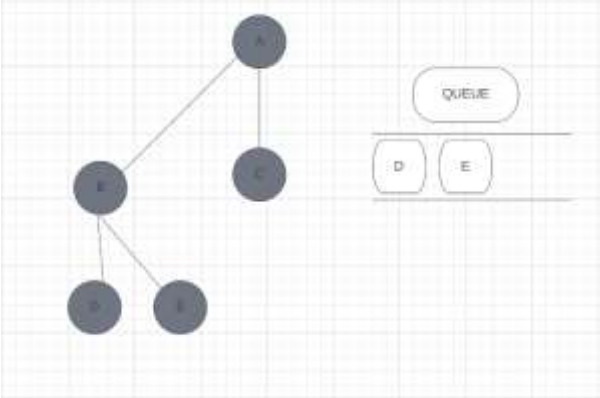

Then we find the vertex adjacent to vertex A that has not been traversed. In this example, we have 2 vertices B, and C, and put vertex B, and C in the queue.

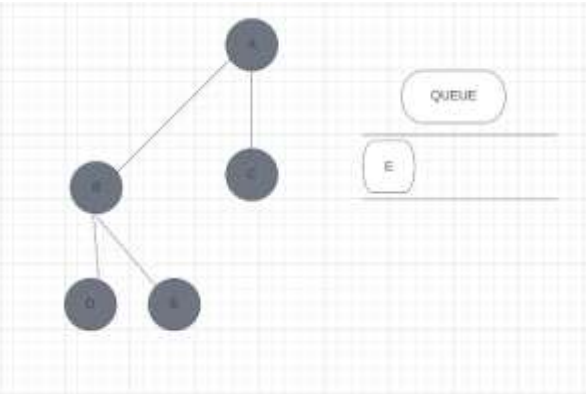


Now vertex A has no adjacent vertices. Enqueuing B from the queue. From B we have 2 adjacent verticles D, E. Queuing D, E into the queue.




Vertex B has no unvisited vertices now. Enqueuing C from the queue. The algorithm will continue until the queue is empty.




***Pseudocode:***

Enqueue the first vertex.

While Queue is not empty:

Check its adjacent vertex (not visited) and queue

Enqueue its adjacent vertices

***Time complexity:*** [2]

$V$ stands for vertices

$E$ stands for edges

$O(V + E)$ when Adjacency List is used.

$O(V^2)$ when Adjacency Matrix is used.

***Space complexity:*** [2] $O(b^d)$ where b is the branching factor and d is the length of the optimal path.

## 2. DEPTH FIRST SEARCH

Depth-first search (DFS) is an algorithm for traversing or searching a tree or a graph. Using the queue follows **LIFO** (Last in First out) rule, the algorithm starts at the root (or chooses a certain vertex as the root) and grows as far as possible on each branch.

Typically, DFS is a form of incomplete information search in which the search is developed. to the first child vertex of the node being searched until it reaches a vertex or to a node that has no children. Then the algorithm goes back to the vertex searched in the previous step. In the non-recursive form, all vertices waiting to be grown are added to a stack.[1]

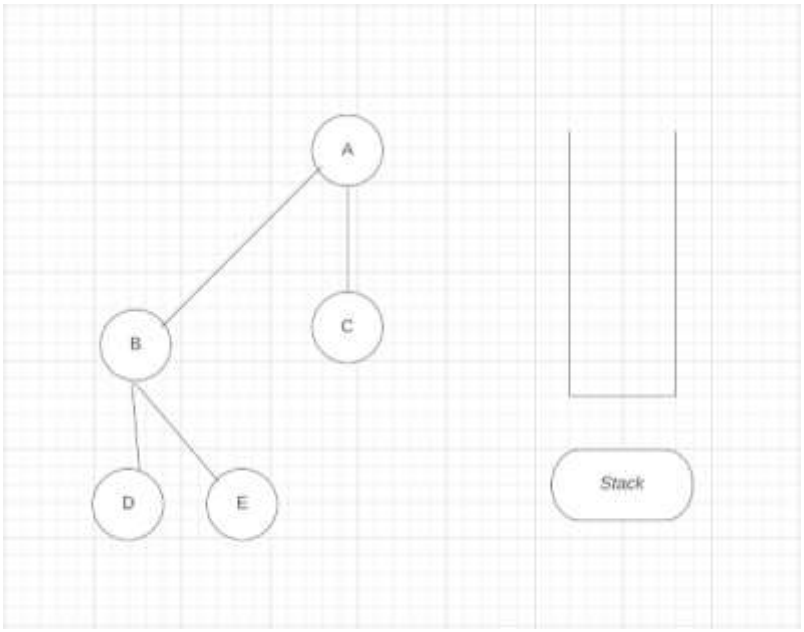
The basic idea of algorithms is as follows:[3]

Step 1: Consider to the adjacent vertex that has not yet been traversed. Mark the vertex that has been considered and push it into a stack.

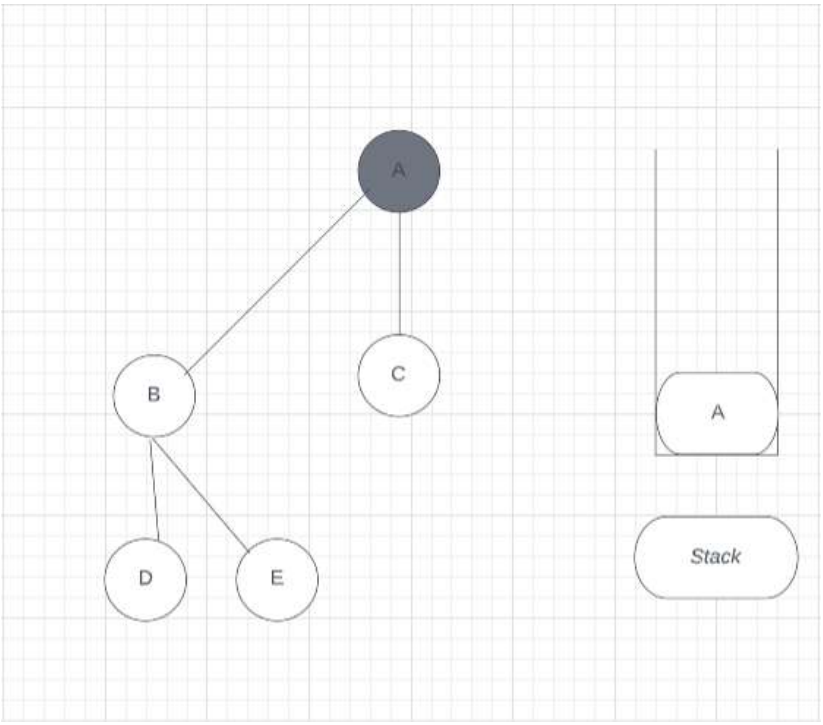
Step 2: If no adjacent vertex is found, get a vertex from the stack. Retrieve all vertices from the stack that have no adjacent vertices.

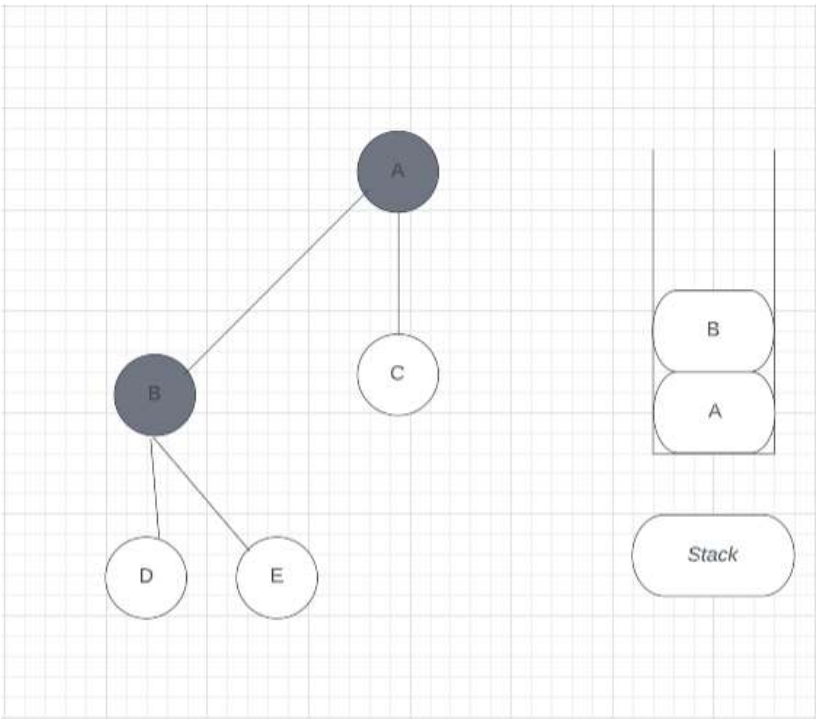
Step 3: Repeat step 1 and 2 until the queue is empty.

***Example:***

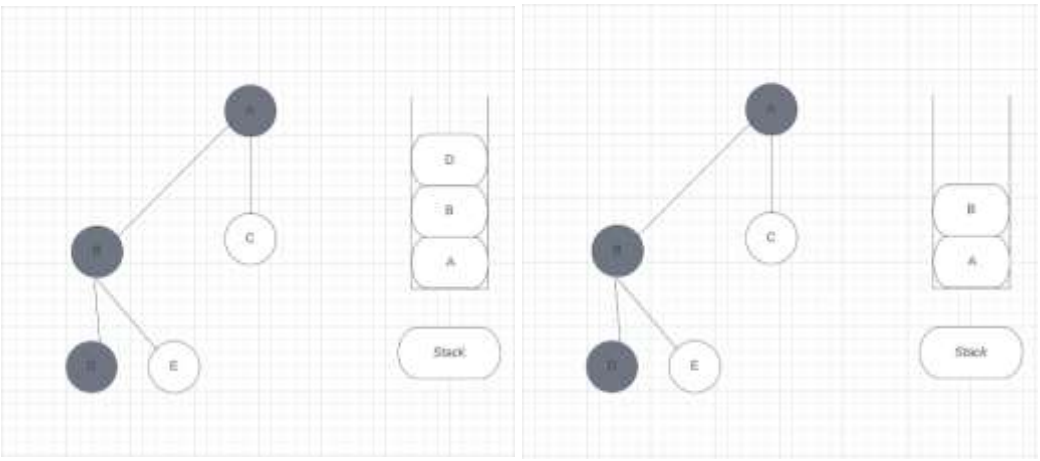



First, mark the vertex A as traversed and put it into the stack. Look at any adjacent vertices that have not been traversed from vertex A. In this example, we have vertices B and C. Take vertex B in alphabetical order and push it into the stack.

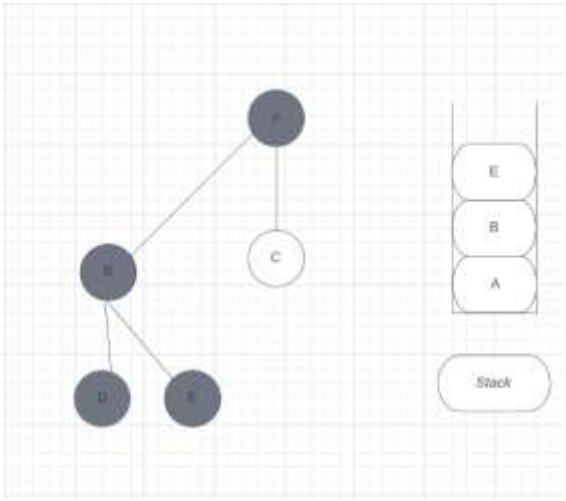


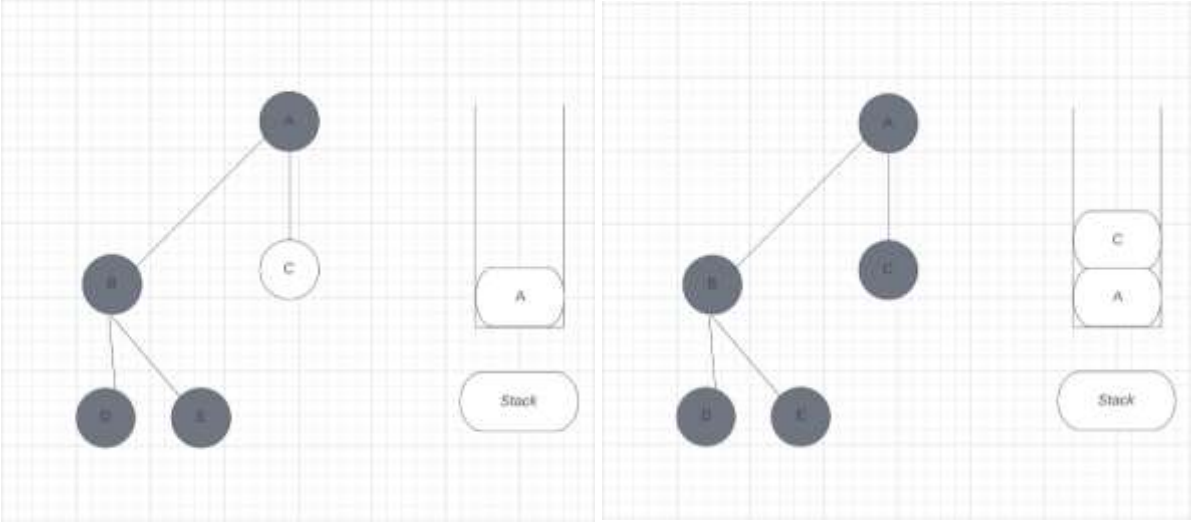

From vertex B, there are 2 adjacent vertices, D and E. Take vertex D in alphabetical order and push it into the stack. As now vertex D has no adjacent vertices, pop vertex D out of the stack.



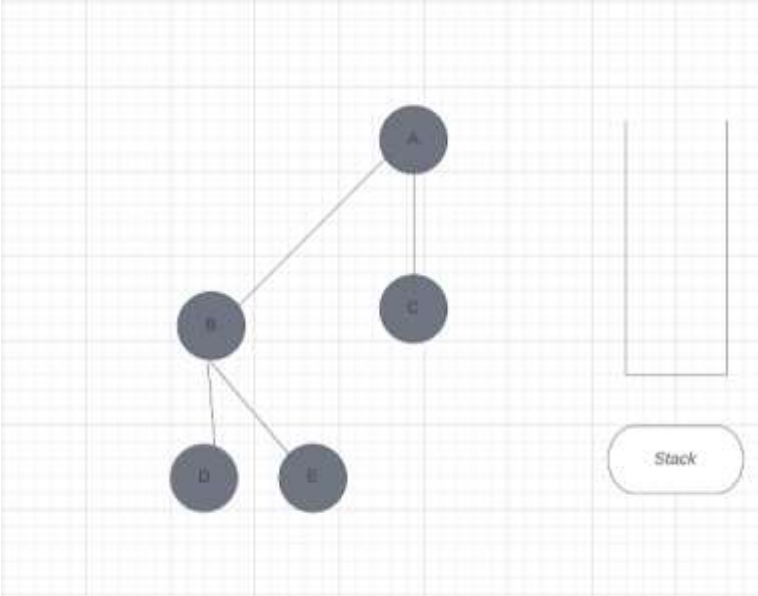

Check the top element of the stack to the previously traversed node and check if this vertex has any adjacent vertices that have not been traversed. In the example, vertex B is at the top of the stack. There is only 1 adjacent vertex of vertex B that has not been traversed, vertex E. Mark the vertex E as traversed and push it to the top of the stack.




As vertex E has no adjacent vertex that has not been traversed. Poping vertex E out of the stack. Return to vertex B, it now has no adjacent vertex that has not been traversed. Popping vertex B out of the stack. Return to vertex A, there is vertex C that is adjacent to vertex A and has not been traversed. Mark vertex C as traversed and push it on the top of the stack.




The algorithm will continue until the stack is empty.




***Pseudocode:***

Push the first vertex into the stack

While Stack is not empty:

current=top()

Check and put only one adjacent vertex in the stack

If all adjacent vertices are visited: pop(current)

***Time complexity:*** [2]

$V$ stands for vertices.

$E$ stands for edges.

$O(V + E)$ when Adjacency List is used.

$O(V^2)$ when Adjacency Matrix is used.

***Space complexity:*** [2]

m stands for the length of the longest path. For m nodes down the path, there are b nodes extra are stored for each of the m nodes. Therefore, the space complexity of the algorithm is $O(bm)$.

## 3. DIJKSTRA'S ALGORITHMS

Dijkstra's algorithm, named after Dutch computer scientist Edsger Dijkstra, is used to solve the **single-source shortest path problem** (SSPP) with the non-negative weight graph. In this paper, Dijkstra's algorithm is implemented with Priority Queue. [3]

Given a graph G(V, E) with edge weight and a source node u belongs to V. Find the shortest path from u to all other nodes on graph G.

The basic idea of the algorithms is as follows:[3]

Step 1: Initialize distances of all vertices as infinite.

Step 2: Create an empty priority queue. Each item in the priority queue is the edge from vertex u to v with its weight.

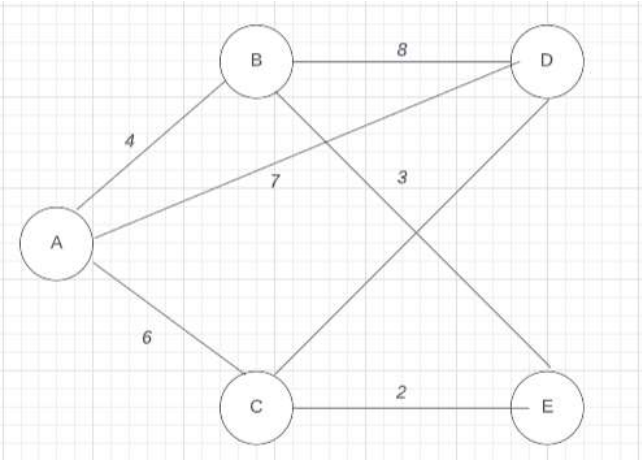
Step 3: Push the distance from the source vertex u and its weight to other adjacent vertices v into the priority queue.

Step 4: Pop the item with the minimum cost from the priority queue. Consider the adjacent vertex v of vertex u in the item which has the minimum cost. Push the distance from u to v

into the priority queue. If the distance from vertex u to v is less than the current distance being recorded, update the distance of vertex v

Step 5: Repeat step 4 again until the priority queue is empty.

***Example:***




| vertex | dist | pred |
|---|---|---|
| A | 0 | - |
| B | ∞ | - |
| C | ∞ | - |
| D | ∞ | - |
| E | ∞ | - |

Priority Queue

| From | |
|---|---|
| To | |
| Cost | |

In this example, we choose A as the source vertex. From vertex A, there are 3 adjacent vertices, B, C and D. Update the distance into the priority queue.

| From | A | A | A |
|---|---|---|---|
| To | B | C | D |
| Cost | 4 | 6 | 7 |

| vertex | dist | pred |
|---|---|---|
| A | 0 | - |
| B | 4 | A |
| C | 6 | A |
| D | 7 | A |
| E | ∞ | - |

The minimum item in the priority queue now is the cost from A to B. Pop the item from the priority queue. Start the process again from vertex B, there are 2 adjacent vertices, D and E. Update the distance into the priority queue.

| From | A | A | B | B |
|---|---|---|---|---|
| To | C | D | E | D |
| Cost | 6 | 7 | 7 | 12 |

| vertex | dist | pred |
|---|---|---|
| A | 0 | - |
| B | 4 | A |
| C | 6 | A |
| D | 7 | A |
| E | 7 | B |

The minimum item in the priority queue now is the cost from A to C. Pop the item from the priority queue. Start the process again from vertex C, there are 2 adjacent vertices, D and E. Update the distance into the priority queue.

| From | A | B | C | C | B |
|---|---|---|---|---|---|
| To | D | E | E | D | D |
| Cost | 7 | 7 | 8 | 9 | 12 |

| vertex | dist | pred |
|---|---|---|
| A | 0 | - |
| B | 4 | A |
| C | 6 | A |
| D | 7 | A |
| E | 7 | B |

The minimum item in the priority queue now is the cost from A to D. Pop the item from the priority queue. Start the process again from vertex D, but all path to vertex D has already updated. Skip and move to the next item.

| From | B | C | C | B |
|---|---|---|---|---|
| To | E | E | D | D |
| Cost | 7 | 8 | 9 | 12 |

| vertex | dist | pred |
|---|---|---|
| A | 0 | - |
| B | 4 | A |
| C | 6 | A |
| D | 7 | A |
| E | 7 | B |

The process with vertex E is the same as the previous evaluation with vertex D. The algorithm continue until the priority is empty.

| From |
|---|
| To |
| Cost |

| vertex | dist | pred |
|---|---|---|
| A | 0 | - |
| B | 4 | A |
| C | 6 | A |
| D | 7 | A |
| E | 7 | B |

The path with lowest cost from A to B is 4 (A→B)

The path with lowest cost from A to C is 6 (A→C)

The path with lowest cost from A to D is 7 (A→D)

The path with lowest cost from A to E is 7 (A→B→E)

***Pseudocode:***

Push the weight into the priority queue pq

While pq not empty:

Pop the edge from to u with minimum cost

For each adjacent vertex v of u:

If dist[v] > dist[u] + weight(u, v)

dist[v] = dist[u] + weight(u, v)

***Time complexity:*** [3] $O(E \log V)$

***Space complexity:*** [3]

Space complexity of Dijkstra's algorithm is $O(2V)$ where V denotes the number of vertices (or nodes) in the graph.

## III. RESEARCH METHODOLOGY

The research methodology used in this research is a comparative test in a simulation program with different types of labyrinths for each algorithm. The processing time, the length of the path and the number of expanded blocks are the case measured in this test. Several different labyrinths will be used. Upon completing the test, the result will be discussed.

## IV. RESULT AND DISCUSSION

### 1. Experiment

This is a screen of the simulation program using Python. The red cell with the number signifies the source and destination of the search process. The black cell represents the wall created by the researcher. The blue and green cells signify the expanded and search nodes. The pink cell signifies the path between the source and destination node. In this case, pink cells will appear if the path between the source node and the destination node is found.

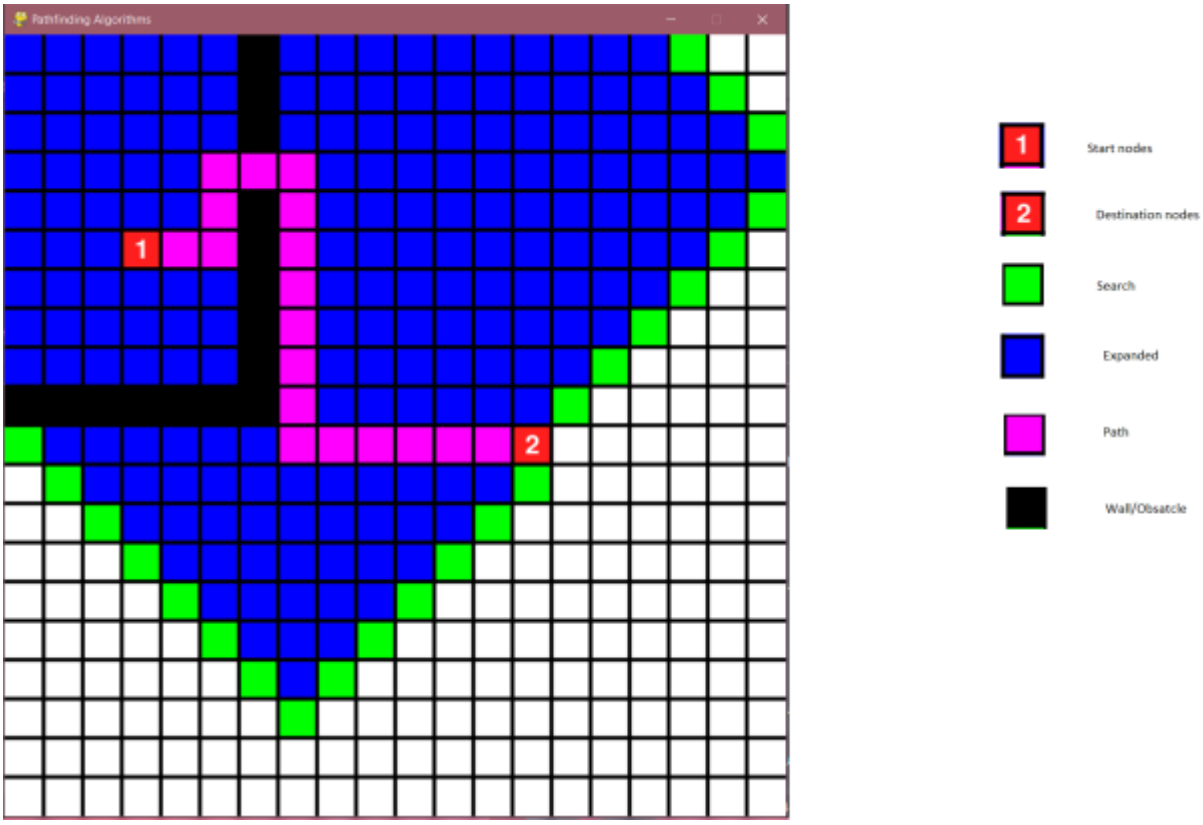



**Figure 1. Program Interface**

In this experiment, researcher is able to create arbitrary maze. When the searching proccessing starts, the source nodes (Red cell 1) will search for the path to the destination node (Red cell 2) with the applied algorithm. While there are still cells to be searched, the NPC will pick a searched cell and continue expanding and adding new searched cells until the expanded node is the destination node.

### 1.1. Environment 1

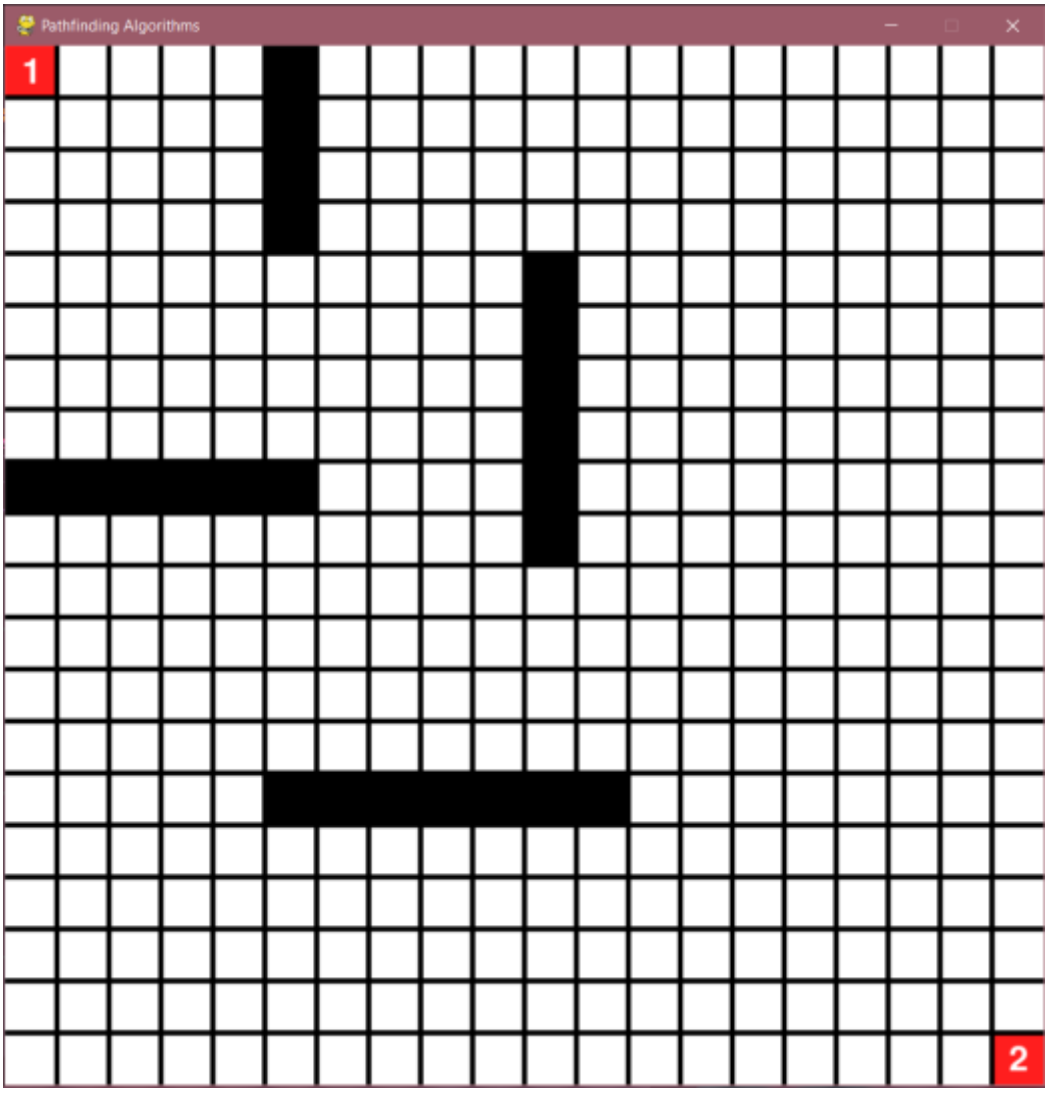
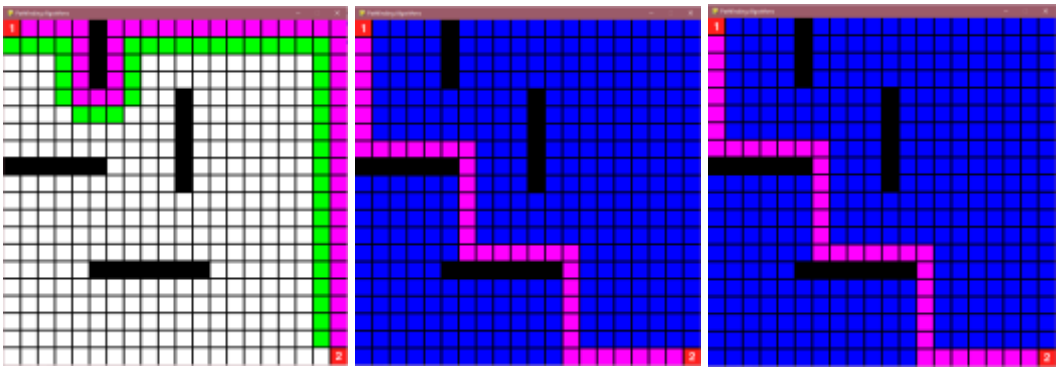



**Figure 1.1. Simulation in environment 1**



**(1)** **(2)** **(3)**

**Figure 1.1.1. Pathfinding in environment 1 using DFS (1), BFS (2), Dijkstra (3)**

The figure 1.1.1 above is the result of the searching process performed by each algorithm. Figure (1) shows the path found using DFS while figure (2) shows the usage of BFS, and figure (3) of using Dijkstra's algorithm. The calculation results of three algorithms are listed in the table below.

| | Time | Distance(Length) | Expanded nodes |
|---|---|---|---|
| DFS | 7.21s | 45 | 45 |
| BFS | 56.29s | 37 | 377 |
| Dijkstra | 55.53s | 37 | 377 |

**Comparison table in environment 1**

From the table above, the shortest path length is 38 cells were obtained by using BFS and Dijkstra's algorithm. The computational time required by Dijkstra's algorithm is slightly faster than BFS. Although DFS is the fastest algorithm in this case, it creates a longer path, which is 45 cells compare to BFS and Dijkstra.

### 1.2. Environment 2

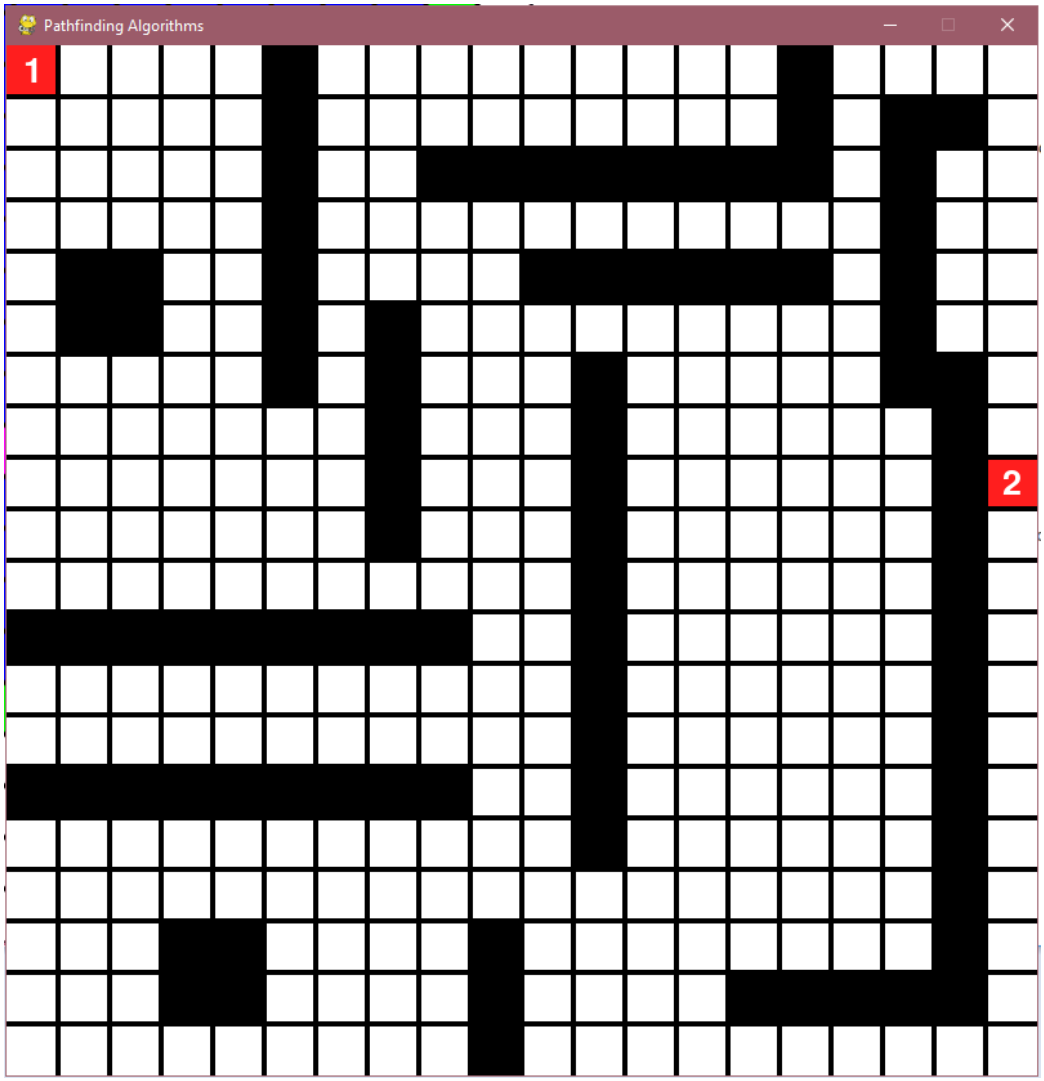
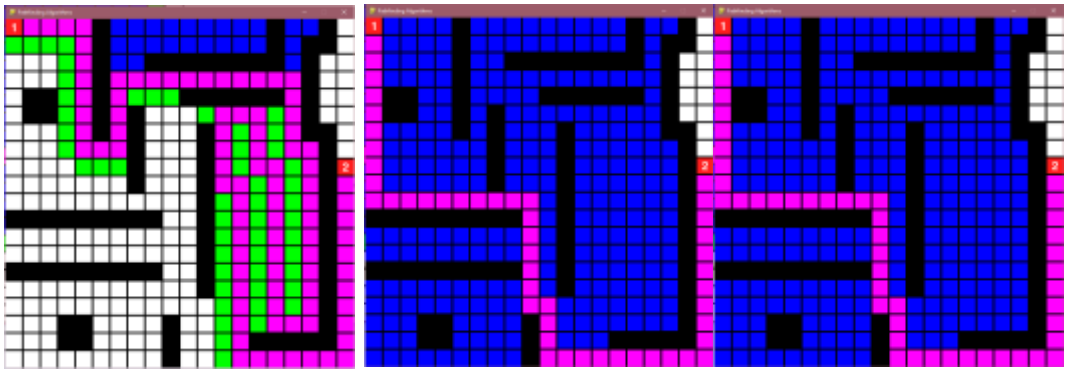



**Figure 1.2. Environment 2**



**(1)** **(2)** **(3)**

**Figure 1.2.1. Pathfinding in environment 2 using DFS (1), BFS (2), Dijkstra (3)**

| | Time | Distance(Length) | Expanded nodes |
|---|---|---|---|
| DFS | 17.25s | 90 | 114 |
| BFS | 44.35s | 49 | 311 |
| Dijkstra | 44.21s | 49 | 311 |

**Comparison table in environment 2**

From the table above, BFS has the shortest path similar to the Dijkstra algorithm with the length of 49 cells. The Dijkstra's algorithm is again slightly faster than BFS and has the number of expanded nodes equal to BFS algorithm with 311 cells. DFS has the fastest computation time than the other algorithms but has the longest path found with 90 cells. The number of nodes expanded by DFS is only one-third compared to BFS and Dijkstra's algorithms.

### 1.3. Environment 3

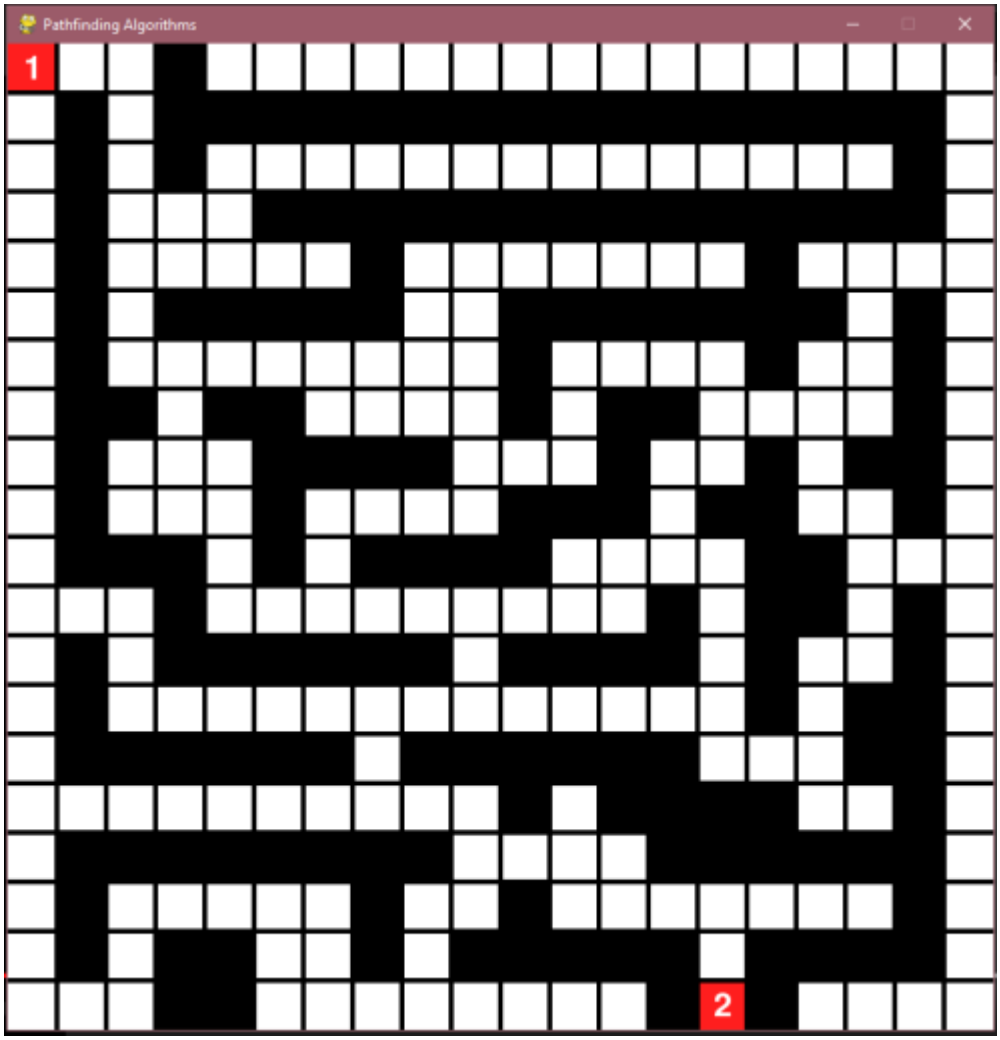
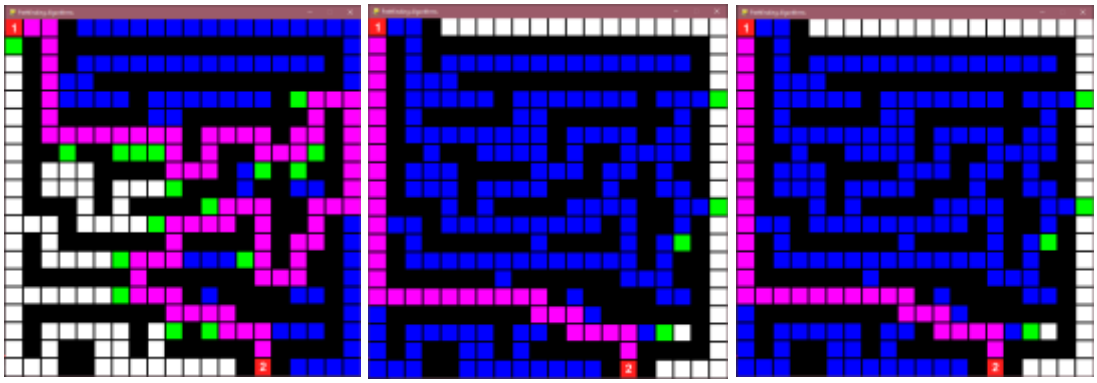



**Figure 1.3. Environment 3**



**(1)** **(2)** **(3)**

**Figure 1.3.1. Pathfinding in environment 3 using DFS (1), BFS (2), Dijkstra (3)**

| | Time | Distance(Length) | Expanded nodes |
|---|---|---|---|
| DFS | 20.11s | 74 | |
| BFS | 24.65s | 32 | 168 |
| Dijkstra | 24.15s | 32 | 168 |

**Comparison table in environment 3**

The number of cells required to reach the destination node from the source node is 32 for both BFS and Dijkstra. In terms of computation, Dijkstra is still faster than 0.5 seconds compared to BFS. However, DFS still has the fastest computation time. Dijkstra and BFS algorithms extend 168 cells to search for the shortest path, 33 cells more than DFS.

## 2. Data and analysis

The collecting data from 30 environments, using each of three algorithms – a total of 90 tests, which is shown in the table below. The table contains the following data:

1. The average time (seconds) spent searching a path from the source node to the destination node.
2. The average length of the path found.
3. The average number of expanded nodes.

| | Avg.Time(seconds) | Avg.Distance(cells) | Avg.Expanded nodes(cells) |
|---|---|---|---|
| DFS | 31.962 | 96 | 245 |
| BFS | 38.642 | 53 | 344 |
| Dijkstra | 37.985 | 53 | 344 |

Total number of successful path calculations: **28/30 (93.33%)**

Total number of times all algorithms found the same path: **3/30 (10.00%)**

From the table, the average path length and the average number of expanded nodes using BFS is relatively the same as using the Dijkstra's algorithms. The average time to find a path of Dijkstra is slightly faster than the BFS algorithm. Using the DFS algorithm returns a path in the fastest time. However, the path found by DFS algorithms may not be the optimal path. DFS also consumes less memory for its computation process than any other algorithm. Based on the search process, the DFS expands the most recently added node, which is the top most node in this case if possible. When it is blocked by an edge or obstacle, it continues to extend to the right node, then the bottom node, and the left node. The BFS algorithm expands the least recently added node, which means it expands as a wide net that covers every single node that is closest in order. Dijkstra's algorithms tend to search similar to BFS, except that each cell is assigned to a value. This value will be the distance from their parent to the current cell. Overall, we can conclude that the simulation to find the shortest path using Dijkstra's algorithm is slightly more effective than using the BFS algorithm.

## 3. Conclusion

Based on the result of this research, it can be determined that:

1. Dijkstra is the best pathfinding algorithm out of three algorithms. It is more efficient with a shorter search time compared to BFS and an optimal path compared to DFS.

2. DFS is better for finding the path in a short amount of time, but the path found may not be the shortest path. The DFS algorithm visits fewer nodes but often finds a longer path than BFS and Dijkstra.

**V. REFERENCES**

[1] Mark Needham and Amy E. Hodler, (2019) “Graph Algorithms: Practical Examples in Apache Spark and Neo4j”.

Retrieved from https://learning.oreilly.com/library/view/graph-algorithms/9781492047674/ch04.html#idm46681372905896

[2] Peter Norvig and Stuart J. Russell, (1995) “Artificial Intelligence: A Modern Approach”

Retrieved from https://zoo.cs.yale.edu/classes/cs470/materials/aima2010.pdf

[3] Bradley N. Miller (2011) “Problem Solving with Algorithms and Data Structures Using Python SECOND EDITION”

Retrieved from https://runestone.academy/ns/books/published/pythonds3/Graphs/DijkstrasAlgorithm.html

Program Code: James Charles Robinson, “Path Finding Visualisation with Pygame”

Source: https://github.com/James-Charles-Robinson/Path-Finding-Visualisation-with-Pygame/blob/master/README.md