Warsaw University of Technology

FACULTY OF
ELECTRONICS AND INFORMATION TECHNOLOGY

Institute of Computer Science

Bachelor's diploma thesis

in the field of study Computer Science
and specialisation Computer System Networks

A Comparison of Efficiencies of Different Pathfinding Algorithms in Video Games using Grid-Based Maps

Abhiraj Singh
student record book number 300765

thesis supervisor
Roman Podraza

WARSAW 2023

A Comparison of Efficiencies of Different Pathfinding Algorithms in Video Games using Grid-Based Maps

Abstract. This thesis is focused on comparing the efficiencies of different pathfinding algorithms in video games using grid-based maps. Considering the amount of pathfinding algorithms available to be used, this project aims to distinguish between a few of them to find which algorithm is the most efficient. To achieve this goal, the algorithms were tested in various environments which mimic video game grid maps, data was recorded and analyzed to draw conclusions on the efficiencies of the algorithms. Another goal of this thesis is to take the best performing pathfinding algorithms and implementing them into a video game to see their real-time execution as well as to be able to find out the impact of the algorithms on the computational time of calculating the path from the starting node to the ending node. Once the algorithms are implemented in the video game, they will be thoroughly tested and conclusions will be drawn on the real-life impact of using different pathfinding algorithms.

Keywords: Pathfinding algorithms, Dijkstra' algorithm, Breadth-First-Search algorithm, Best-First-Search algorithm, A* algorithm, grid map, Python, Pygame

A Comparison of Efficiencies of Different Pathfinding Algorithms in Video Games using Grid-Based Maps

Streszczenie. Praca dyplomowa porównania wydajności różnych algorytmów odnajdywania ścieżek w grach wideo przy użyciu map opartych na siatce. Biorąc pod uwagę liczbę dostępnych algorytmów odnajdywania ścieżek, projekt ten ma na celu znalezienie najbardziej wydajnego algorytmu. Aby osiągnąć ten cel, algorytmy zostały przetestowane w różnych środowiskach, które naśladują mapy siatek gier wideo. Wyniki zostały zarejestrowane i przeanalizowane w celu wyciągnięcia wniosków na temat wydajności algorytmów. Kolejnym celem tej pracy jest wykorzystanie najlepiej działających algorytmów wyszukiwania ścieżek i zaimplementowanie ich w grze wideo, aby zobaczyć ich wykonanie w czasie rzeczywistym, a także móc poznać wpływ algorytmów na czas obliczeniowy obliczania ścieżki z od węzła początkowego do węzła końcowego. Po zaimplementowaniu algorytmów w grze wideo zostaną one dokładnie przetestowane i zostaną wyciągnięte wnioski na temat rzeczywistego wpływu różnych algorytmów odnajdywania ścieżek.

Słowa kluczowe: algorytmy wyszukiwania najkrótszej ścieżki, algorytm Dijkstry, algorytm Breadth-First-Search, algorytm Best-First-Search, algorytm A*, mapa siatki, Python, Pygame

załącznik nr 10 do zarządzenia
nr 46 /2016 Rektora PW

Politechnika Warszawska
Warsaw University of Technology

…….....
miejscowość i data
place and date

……..
imię i nazwisko studenta
name and surname of the student
……..
numer albumu
student record book number
…….…….
kierunek studiów
field of study

OŚWIADCZENIE

DECLARATION

Świadomy/-a odpowiedzialności karnej za składanie fałszywych zeznań oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie, pod opieką kierującego pracą dyplomową.
Under the penalty of perjury, I hereby certify that I wrote my diploma thesis on my own, under the guidance of the thesis supervisor.

Jednocześnie oświadczam, że:
I also declare that:

- niniejsza praca dyplomowa nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 roku o prawie autorskim i prawach pokrewnych (Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.) oraz dóbr osobistych chronionych prawem cywilnym,
- *this diploma thesis does not constitute infringement of copyright following the act of 4 February 1994 on copyright and related rights (Journal of Acts of 2006 no. 90, item 631 with further amendments) or personal rights protected under the civil law,*

- niniejsza praca dyplomowa nie zawiera danych i informacji, które uzyskałem/-am w sposób niedozwolony,
- *the diploma thesis does not contain data or information acquired in an illegal way,*

- niniejsza praca dyplomowa nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadawaniem dyplomów lub tytułów zawodowych,
- *the diploma thesis has never been the basis of any other official proceedings leading to the award of diplomas or professional degrees,*

- wszystkie informacje umieszczone w niniejszej pracy, uzyskane ze źródeł pisanych i elektronicznych, zostały udokumentowane w wykazie literatury odpowiednimi odnośnikami,
- *all information included in the diploma thesis, derived from printed and electronic sources, has been documented with relevant references in the literature section,*

- znam regulacje prawne Politechniki Warszawskiej w sprawie zarządzania prawami autorskimi i prawami pokrewnymi, prawami własności przemysłowej oraz zasadami komercjalizacji.
- *I am aware of the regulations at Warsaw University of Technology on management of copyright and related rights, industrial property rights and commercialisation.*

Oświadczam, że treść pracy dyplomowej w wersji drukowanej, treść pracy dyplomowej zawartej na nośniku elektronicznym (płycie kompaktowej) oraz treść pracy dyplomowej w module APD systemu USOS są identyczne.
I certify that the content of the printed version of the diploma thesis, the content of the electronic version of the diploma thesis (on a CD) and the content of the diploma thesis in the Archive of Diploma Theses (APD module) of the USOS system are identical.

.....
czytelny podpis studenta
legible signature of the student

Contents

1. Introduction

The research topic is 'A Comparison of Efficiencies of Different Pathfinding Algorithms in Video Games using Grid-Based Maps.' This study is significant due to pathfinding algorithms evolving and being utilized in different upcoming technologies where understanding the most efficient pathfinding algorithm is essential.

1.1. Pathfinding Algorithms

Pathfinding is the plotting, by a computer application, of the best route between a starting and ending point[1]. Pathfinding algorithms are used in Global Positioning Systems (GPSs) to find the most efficient route between the client and their destination. They are used in video games as well, to compute the shortest route between two points or between the player and a Non-Player Character (NPC) and between the start and end of a maze. A few examples of pathfinding algorithms used today are A* algorithm, Breadth-First-Search(BFS) algorithm and Dijkstra's Algorithm. The efficiency of pathfinding algorithms depend on two things: how optimal the calculated route is, and how efficient the algorithm was in finding that shortest route. Which, in other words, means the lower the number of operations to calculate the shortest route, the more efficient the algorithm is.

1.1.1. Pathfinding Applications

There were a few applications of pathfinding algorithms stated previously, adding on to that as our technology develops, pathfinding algorithms are vital in the robotics industry. There are two kinds of robots which utilize pathfinding algorithms: exploration and industrial robots. Firstly exploration robots use pathfinding algorithms to reach dangerous locations where humans cannot. An example of that is with the exploration of different planets utilizing robots which have the capability to safely explore planets without wandering off or getting wrecked due to the use of pathfinding algorithms. Industrial robots are used to perform basic actions such as carrying items from one place to another, for example in a factory. Another application of pathfinding algorithms are for automated vehicles which are still under development however the algorithms are a vital part of the upcoming projects for these vehicles[2].

Pathfinding algorithms are implemented in computer games as well, where the system should coordinate recreated NPCs around the virtual map of the game, this could be the foremost complex utilization of pathfinding algorithms. The computer needs to evaluate and decide the path which is the most optimal; in terms of time taken to reach the endpoint as well as the safety of that path, as in some games certain paths can deal damage to the NPC due to dynamic objects, for example fire(dynamic object) dealing damage to an enemy(NPC). These dynamic objects can have their own unique behaviour which could

be controlled by the computer which then requires the computer to consider how the dynamic objects need to operate in order to obtain the most optimum route[2].

In addition, a major application of pathfinding algorithms, is their use in GPS systems, or any type of satellite navigation of real life locations. Companies such as Garmin and Google use pathfinding algorithms in their satnav and Google's 'Google Maps' software. Dijkstra's algorithm is the basic algorithm which can be used in GPS systems, as this study progresses and an overview of the different pathfinding algorithms are explained, it will give a better understanding as to how a pathfinding algorithm can be applied to geographic locations and routing between two points. GPS systems today are required to produce results faster and more accurately, hence certain changes or optimizations are constantly being made to certain pathfinding algorithms, such as Dijkstra's algorithm in order to create the most efficient software[3].

1.2. History of Pathfinding Algorithms

The origin of pathfinding algorithms comes from a man named Edsger Dijkstra, a Dutch computer scientist, who came up with the first shortest route algorithm and published it in 1959. A quote from Dijkstra himself is, "What is the shortest way to travel from Rotterdam to Groningen, in general: from given city to given city. It is the algorithm for the shortest path, which I designed in about twenty minutes. One morning I was shopping in Amsterdam with my young fiancée, and tired, we sat down on the café terrace to drink a cup of coffee and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path. As I said, it was a twenty-minute invention. In fact, it was published in '59, three years later. The publication is still readable, it is, in fact, quite nice. One of the reasons that it is so nice was that I designed it without pencil and paper. I learned later that one of the advantages of designing without pencil and paper is that you are almost forced to avoid all avoidable complexities. Eventually, that algorithm became to my great amazement, one of the cornerstones of my fame." This became the dawn of the evolution of pathfinding algorithms, with different kinds of solutions being implemented and Dijkstra's algorithm being tweaked to seek improvement[4].

1.3. Thesis goals

The goals of this work can be broken down into points as follow:

- Establish the pathfinding algorithms and their parameters
- Create different grid-based environments for the algorithms to be experimented on
- Perform all simulations, gather data and calculated the efficiencies of each algorithm in each environment
- Draw conclusions on which algorithm is most efficient
- Provide a real life demonstration which implements the same algorithms in a 2D grid-based video game

- Experiment with real-time demonstrations of the performances of the algorithms
- Compare results of theoretical and real life experiment

2. Pathfinding Algorithms

Firstly, it is essential to understand how the different pathfinding algorithms work in order to assess their efficiency and then compare them to one another. Starting with an introduction to Dijkstra's algorithm, followed by the Best-First-Search algorithm, Breadth-First-Search algorithm and finally the A* algorithm. Each algorithm will be explained with the aid of Python examples of the algorithms.

2.1. Dijkstra's Algorithm

Dijkstra's Algorithm calculates the shortest route, for a weighted graph, from a single-source node (a point) to all other points on the weighted graph; not only from one point to another. Due to this application, the algorithm is sometimes referred to as single-source shortest path algorithm[5]. This algorithm can only be used on a graph which has non-negative weight on every edge however the graph can be directed (arrows pointing in the direction of connection between nodes) or undirected[6].

```python
def dijkstra_algorithm(graph, start_node):
    unvisited_nodes = list(graph.get_nodes())

    # Save the cost of visiting each node and update it as we move along the graph
    shortest_path = {}

    # We'll use this dict to save the shortest known path to a node found so far
    previous_nodes = {}

    # We'll use max_value to initialize the "infinity" value of the unvisited nodes
    max_value = sys.maxsize
    for node in unvisited_nodes:
        shortest_path[node] = max_value
    # However, we initialize the starting node's value with 0
    shortest_path[start_node] = 0

    # The algorithm executes until we visit all nodes
    while unvisited_nodes:
        # The code block below finds the node with the lowest score
        current_min_node = None
        for node in unvisited_nodes: # Iterate over the nodes
            if current_min_node == None:
                current_min_node = node
            elif shortest_path[node] < shortest_path[current_min_node]:
                current_min_node = node

        # The code block below retrieves the current node's neighbors and updates their distances
        neighbors = graph.get_outgoing_edges(current_min_node)
        for neighbor in neighbors:
            tentative_value = shortest_path[current_min_node]
                              + graph.value(current_min_node, neighbor)
            if tentative_value < shortest_path[neighbor]:
                shortest_path[neighbor] = tentative_value
                # We also update the best path to the current node
                previous_nodes[neighbor] = current_min_node

        # After visiting its neighbors, we mark the node as "visited"
        unvisited_nodes.remove(current_min_node)

    return previous_nodes, shortest_path
```

Listing 1. Python example of Dijkstra's algorithm[7]

The code above shows us the logic behind Dijkstra's algorithm. It has the starting source node with distance set to 0. The distance of all the other nodes on the weighted graph are set to infinity because their weights are still unknown, however each node does

have its own value for the weight. Then, there is a list created of the nodes which have not been visited and visited nodes are taken out of the list. For better understanding, below there is an example of an undirected, weighted graph.

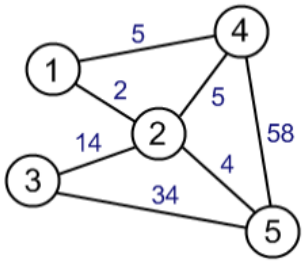


Figure 2.1. Example of undirected, weighted graph[8]

Here let us use node 1 as the source node and node 5 as the destination node. The algorithm starts at the source node and checks for its neighboring nodes, in the example graph, that would be nodes 2 and 4. The program then checks the weights of the edges leading to the neighboring nodes, it adds the distance to the current node plus the weight of the edge, compares them and takes the smallest one and stores the calculated value as new weight; in the example graph, node 2 will be taken. After this, node 2 becomes the new source node and the process of checking its neighboring nodes is repeated. After acquiring the new source node, which in the example graph will be our destination node 5, the previous source node (node 2) is taken out of the list of unvisited nodes. The process is finished once the destination node has been removed from the unvisited list and the list of visited nodes will be the shortest path from the starting node to the destination node.

This algorithm can be regarded as a brute-force search type of greedy algorithm since it makes the most effective decision on local information. However due to this, all nodes and edges on the weighted graph may not be analyzed in order for the algorithm to find the shortest route between the starting and destination node[9].

2.2. Best-First-Search Algorithm

The Best-First-Search pathfinding algorithm is a search algorithm which makes use of a heuristic function, $h(n)$, to order the nodes by how far they are from the destination node[10]. A heuristic is an approximation of how close you are to the target[11], which in this case would be how far the current node is from the destination node. The difference between Dijkstra's algorithm and Best-First-Search algorithm is that Dijkstra's algorithm selects the node which is closest to the current node whilst BFS algorithm selects the node which is closest to the destination node.

```
def best_first(graph, start_vertex, target):
    # Create the priority queue for open vertices.
    vertices_pq = PriorityQueue()

    # Adds the start vertex to the priority queue.
    vertices_pq.put(start_vertex)

    # The starting vertex is visited first and has no leading edges.
    # If we did not put it into 'visited' in the first iteration,
    # it would end up in 'visited' during the second iteration, pointed to
    # by one of its children vertices as a previously unvisited vertex.
    visited[start_vertex] = None

    # Loops until the priority list gets empty.
    while not vertices_pq.empty():
        # Gets the vertex with the lowest cost.
        vertex = vertices_pq.get()
        print(f'Exploring vertex {vertex.entity()}')
        if vertex.entity() == target:
            return vertex
        # Examine each non-visited adjoining edge/vertex.
        for edge in graph.adjacent_edges(vertex):
            # Gets the second endpoint.
            v_2nd_endpoint = edge.opposite(vertex)

            if v_2nd_endpoint not in visited:
                # Adds the second endpoint to 'visited' and maps
                # the leading edge for the search path reconstruction.
                visited[v_2nd_endpoint] = edge

                vertices_pq.put(v_2nd_endpoint)
    return None
```

Listing 2. Python example of Best-First-Search algorithm[12]

This is a python example of the Best-First-Search algorithm, the first thing which is declared are two lists, one closed list and one open list. The starting node is placed inside the open list, its neighboring nodes are checked to find the heuristic best node which is the node estimated to be the best to reach the destination node. If that heuristic best node is the destination node then the route from the initial node to destination node is output. If it isn't the destination node, all neighboring nodes are placed into the open list and the heuristic best node is removed from the open list and added to the closed list. Now the heuristic best node becomes the current node, same as the initial node was, and the algorithm is looped and same steps are repeated until the heuristic best node is the destination node.

Best-First-Search algorithm is used due to its efficiency and the minimal number of operations it takes to find the optimal route. However, as the BFS algorithm has a greedy approach and always takes the node which is closest to the destination node, there are times when the Best-First-Search algorithm will not output the shortest route. For example if one wanted to plot a course on a map to get from one place to another, using the Best-First-Search algorithm could give results involving routes which could be very hard to traverse or potentially totally impossible to use, hence even though the program output a short route very quickly, it might not be the shortest due to external factors which aren't considered such as the cost to get from one node to another[10].

2.3. Breadth-First-Search Algorithm

Breadth-First-Search algorithm is a traversal algorithm where it starts from a selected source node, and it traverses the graph layer by layer meaning it explores each neighbouring node and then move towards the next level of nodes[13]. Applying this algorithm onto a grid-based map means the algorithm will traverse the grid around the starting node layer by layer until it reaches the target node.

```
visited = [] # List for visited nodes.
queue = []     #Initialize a queue

def bfs(visited, graph, node): #function for BFS
  visited.append(node)
  queue.append(node)

  while queue:          # Creating loop to visit each node
    m = queue.pop(0)
    print (m, end = " ")

    for neighbour in graph[m]:
      if neighbour not in visited:
        visited.append(neighbour)
        queue.append(neighbour)
```

Listing 3. Python example of BFS algorithm[14]

Above we have a python example of BFS algorithm where we assume we have a graph created, insert node into the queue until all the neighbouring vertices are marked. Once that vertex is visited it is removed from the queue. The process is repeated until the target node is reached.

2.4. A* Algorithm

A* algorithm is a combination of Best-First-Search algorithm and Dijkstra's algorithm. It uses the distance from the current node to the lowest costing neighboring node as well as the heuristic estimated distance from the lowest costing neighboring node to the destination node. What this means is that is that the A* algorithm prioritizes routes which seem to be leading closer to the destination node plus have the lowest code to get to that node; resulting in the A* algorithm only finding the shortest route to one final node rather than all the nodes like Dijkstra's algorithm[15]. In standard terminology, the A* algorithm works on the basis of:

$$f(n) = g(n) + h(n)$$

where:

- n is a node
- $g(n)$ is the cost from the current node to the next node n
- $h(n)$ is the heuristic estimated cost from the node n to the destination node
- $f(n)$ is the total path cost[15]

```
def a_star_algorithm(self, start, stop):
    # open_lst is list of nodes which have been visited
    open_lst = set([start])
    # closed_lst is a list of nodes which have been visited
    closed_lst = set([])
    # path has present distances from start to all nodes, default value is +inf
    path = {}
    path[start] = 0
    par = {} # par contains an adjacent mapping of all nodes
    par[start] = start
    while len(open_lst) > 0:
        n = None
        for v in open_lst: # it will find a node with the lowest value of f() -
            if n == None or path[v] + self.h(v) < path[n] + self.h(n):
                n = v;
        if n == None:
            return None
        if n == stop: # if current node is stop then start again from start
            reconst_path = []
            while par[n] != n:
                reconst_path.append(n)
                n = par[n]
            reconst_path.append(start)
            reconst_path.reverse()
            return reconst_pat
        # for all neighbors of current node
        for (m, weight) in self.get_neighbors(n):
        # if current node is not in either list add to open_lst, n is it's par
            if m not in open_lst and m not in closed_lst:
                open_lst.add(m)
                par[m] = n
                path[m] = path[n] + weight
            # check if it's quicker to first visit n, then m and if so, update par
            else:
                if path[m] > path[n] + weight:
                    path[m] = path[n] + weight
                    par[m] = n
                    if m in closed_lst:
                        closed_lst.remove(m)
                        open_lst.add(m)
        # remove n from open_lst, add to closed_lst since all neighbors checked
        open_lst.remove(n)
        closed_lst.add(n)
    return None
```

Listing 4. Python example of A* algorithm[16]

Above we can see an example of the A* algorithm written in python. Essentially there is a closed and open list in this pseudocode as well, just like the Best-First-Search algorithm, with the starting node is placed inside the open list. As long as the open list is not empty, the program finds the node with the lowest value for $f(n)$, n, and removes it from the open list and adds it to the closed list. For that node's neighboring nodes we add them to the open list if they aren't in any list, calculate the value of $f(n)$ of the neighboring node. If the neighboring node is on the open list, check if the value of $g(n)$ for that node is better than the previous path from the start node to the current node. If it is better, transfer the current node from closed list to open list and set the node n as the parent node. Node n is deleted from the open list and added to the closed list. Once node n is the destination node, the program outputs the shortest path.

2.5. Bi-Directional Pathfinding Algorithms

Each of the algorithms discussed previously can be implemented bi-directionally. What this means is that instead of the algorithms trying to find the shortest path from one starting node to another destination node, the destination node also acts as the starting node and the original starting node then becomes the destination node. The algorithm begins from both nodes and start pathing towards each other and once the common explored node is found the shortest path is then the result of the algorithm.

2.6. Heuristic Functions

Regarding the pathfinding algorithms which use a heuristic function which are Best-First-Search and A* in the cases discussed previously, there are few ways in which the heuristic function could be implemented. Manhattan distance, Euclidean distance, Chebyshev distance and Octile distance are all distance metrics which calculate a particular result based on two separate data points. We will briefly discuss these different functions.

2.6.1. Euclidean Distance

Linear distance is the shortest distance between any two points, also called Euclidean distance, if all angular motions are allowed instead of grid directions (horizontal, vertical, diagonal). This is an example of uniform cost search where the next target node is chosen based on the cost up to the current point, as a result the lowest cost node is chosen every time. Therefore it isn't the most efficient function since it takes time to explore all the nodes however it still is a complete and reliable function[17].

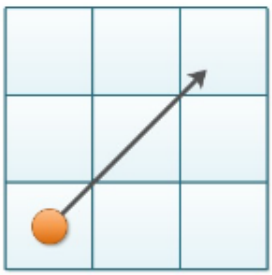


Figure 2.2. Euclidean Distance[18]

In a N dimensional space, a singular point can be represented as $(x1, x2, ..., xN)$. If we consider two points $P1$ and $P2$:

$$P1 : (X1, X2, ..., XN)$$

$$P2 : (Y1, Y2, ..., YN)$$

Then the Euclidean distance between $P1$ and $P2$ and the heuristic function is[18]:

$$h(x) = \sqrt{(x1 - y1)^2 + (x2 - y2)^2 + ... + (xN - yN)^2}$$

2.6.2. Chebyshev Distance

Chebyshev distance outputs the absolute magnitude of the differences between two points' coordinates.

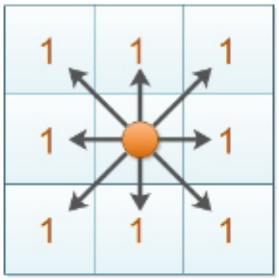


Figure 2.3. Chebyshev Distance[18]

In a N dimensional space, a singular point can be represented as $(x1, x2, ..., xN)$. If we consider two points $P1$ and $P2$:

$$P1 : (X1, X2, ..., XN)$$

$$P2 : (Y1, Y2, ..., YN)$$

Then the Chebyshev distance between $P1$ and $P2$ and the heuristic function is[18]:

$$h(x) = MAX(|xi - yi|) \quad \text{where } i \text{ is 1 to } N$$

2.6.3. Manhattan Distance

The Manhattan distance is taken as the standard heuristic for square grids, defined as the sum of the absolute differences between the Cartesian coordinates of the start and end positions. In pathfinding, the Manhattan distance is the distance between a start node and a destination node when movement is constrained to the vertical or horizontal axis of a square grid[17].

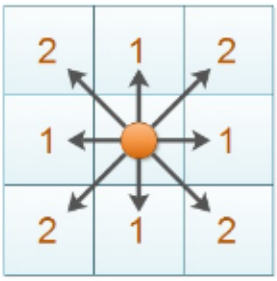


Figure 2.4. Manhattan Distance[18]

In a N dimensional space, a singular point can be represented as $(x1, x2, ..., xN)$. If we consider two points $P1$ and $P2$:

$$P1 : (X1, X2, ..., XN)$$

$$P2 : (Y1, Y2, ..., YN)$$

Then the Manhattan distance between $P1$ and $P2$ and the heuristic function is[18]:

$$h(x) = |x1 - y1| + |x2 - y2| + ... + |xN - yN|$$

This will be the function which will be used and implemented in further experimentation since we want movement constrained to the vertical and horizontal axis in our grid-based maps and this function performs that most efficiently.

2.6.4. Octile Distance

Octile distance is the distance between two diagonal points. It is possible to move vertically, horizontally and diagonally. The Manhattan distance of 2 up and then 2 to the right is 4 units, but only 2 units diagonally which would be the Octile distance[17].

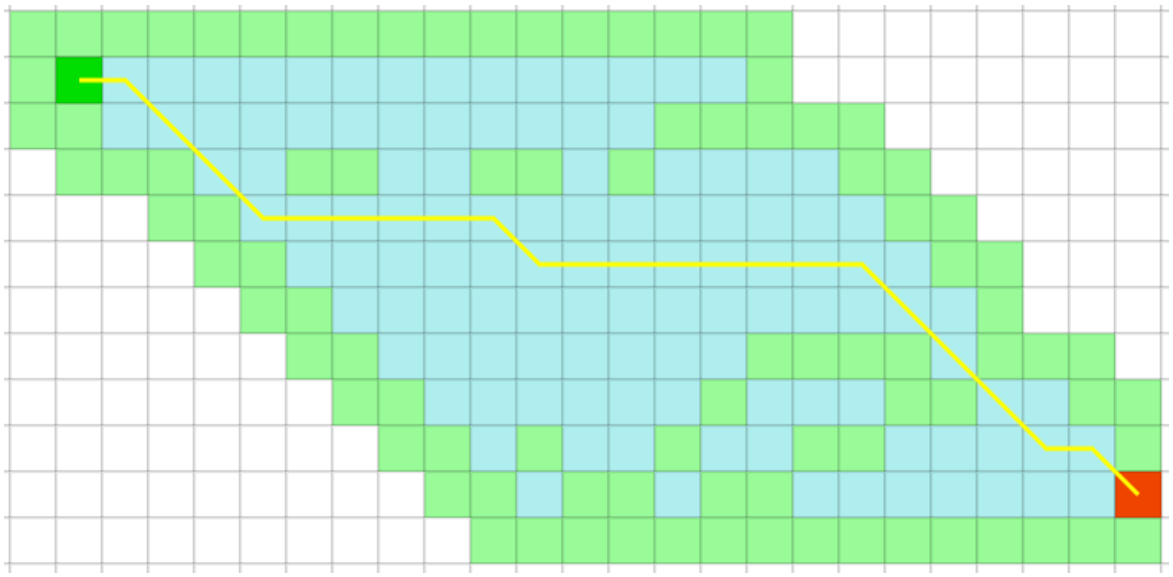


Figure 2.5. Octile Distance[19]

The function of the Octile distance is as follows[17]:

$$h(x) = MAX((x1 - x2), (y1 - y2) + (\sqrt{2} - 1)) * MIN((x1 - x2), (y1 - y2))$$

2.7. Grids

When this research paper is centered around pathfinding algorithms' efficiency on grid-based maps, we should define exactly what a grid is. There are few main types of grids used regarding pathfinding; triangular, hexagonal and square grids which are all regular grids. A regular grid is made up of points that are connected by edges which uniformly separate a map into a set of regular shapes which for example for Figure 2.5 are squares. The movement of a character in a grid like in Figure 2.5 can be either vertical and horizontal only, making it a 4-way, or allow diagonal movement as well making it 8-way. Below we can observe a hexagonal grid[17]:

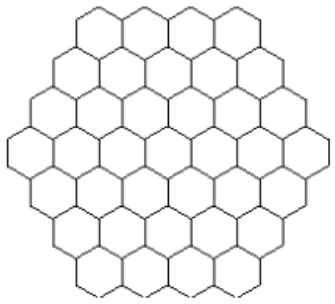


Figure 2.6. Hexagonal Grid

A reason to use hexagonal grid would be to reduce search time and memory usage[17]. Finally, we can see a triangular grid below:

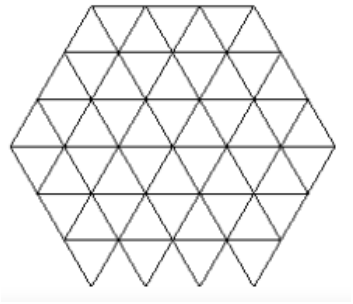


Figure 2.7. Triangular Grid

This is the least popular type of grid out of the ones mentioned, but is still used for some research purposes[17].

3. Simulation of Pathfinding Algorithms

The simulations will be conducted using a pathfinding simulation software made by Xueqiao Xu called Pathfinding.js[19]. I will be comparing the efficiencies of Dijkstra's algorithm, BFS algorithm, Best-First-Search algorithm, A* algorithm and their bi-directional variations on a 25x25 pixel grid and simulating different scenarios using different environments including simple obstacles, video game map replicas and mazes.

3.1. Environments

The following simulations of environments were made using the Pathfinding.js software[19]. On the 25x25 grids, each blank pixel (white pixel) on the grid will be a representation of a node and each black pixel aren't nodes (acting like walls/obstacles in a video game). The green pixel is the starting node and the red pixel is the destination node. When a path is found a yellow line connects the green and red pixel showing the route obtained. There are also additional light green and blue pixels which denote the nodes checked by the algorithm in trying to find the shortest path. These are the environments which will be used in this experiment:

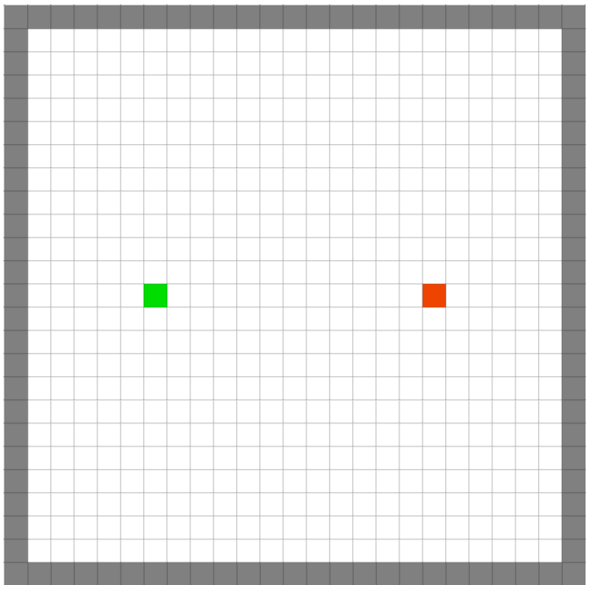


Figure 3.1. Environment 1

First environment is basic open setup with no interfering walls in between the two points. This will showcase how the standard version of the algorithm works.

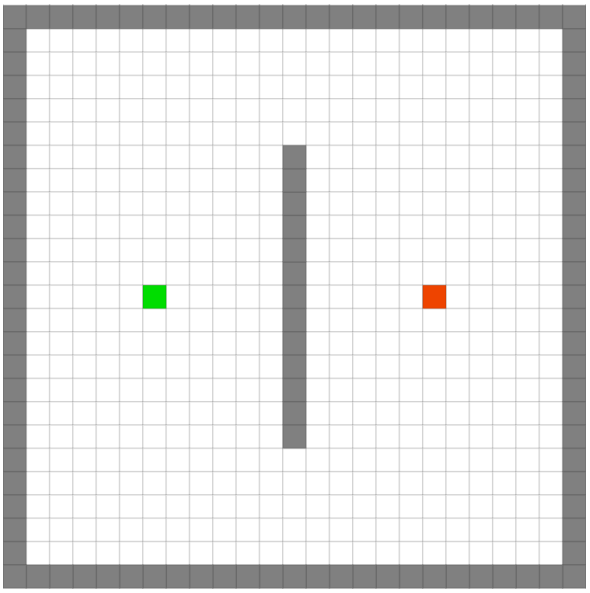


Figure 3.2. Environment 2

Second environment is also rather simple, one wall setup between the two points to further showcase how the different algorithms overcome the obstacle in-between their starting position and end goal.

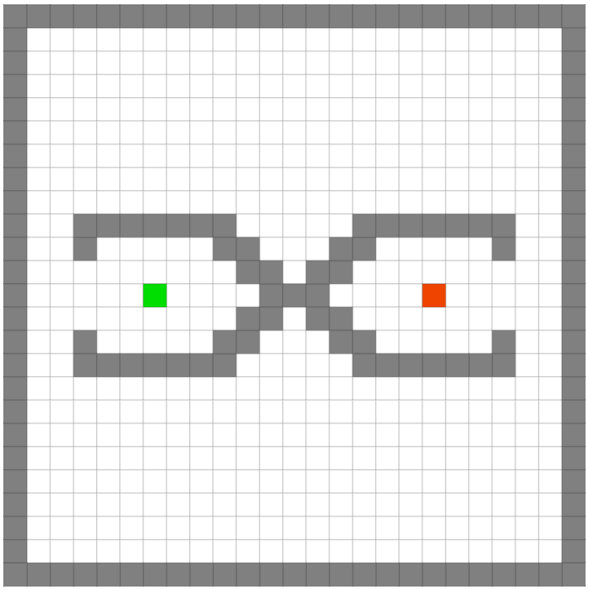


Figure 3.3. Environment 3

Third environment remains symmetrical but adds a slight complexity to the map. Should showcase how the greedy nature of some algorithms affect their ability to find the shortest route.

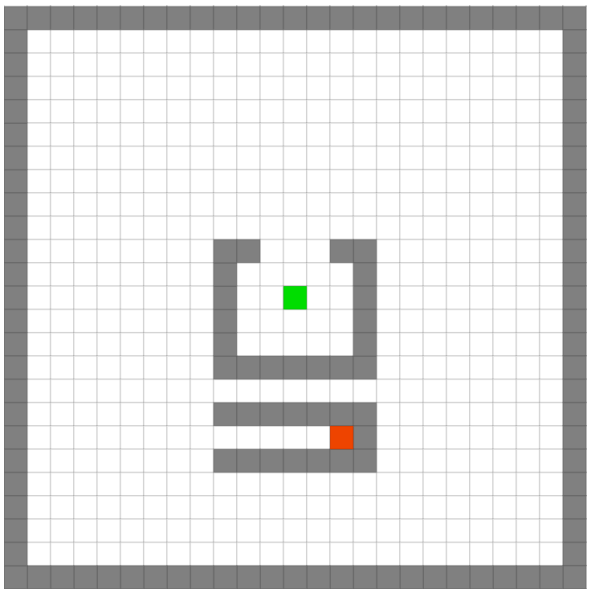


Figure 3.4. Environment 4

Fourth environment is unsymmetrical which should create a difference between Uni-Directional and Bi-Directional algorithms. This layout should yet again showcase the greedy nature of certain algorithms and we could possibly see those algorithms able to only find a short path and not the shortest path.

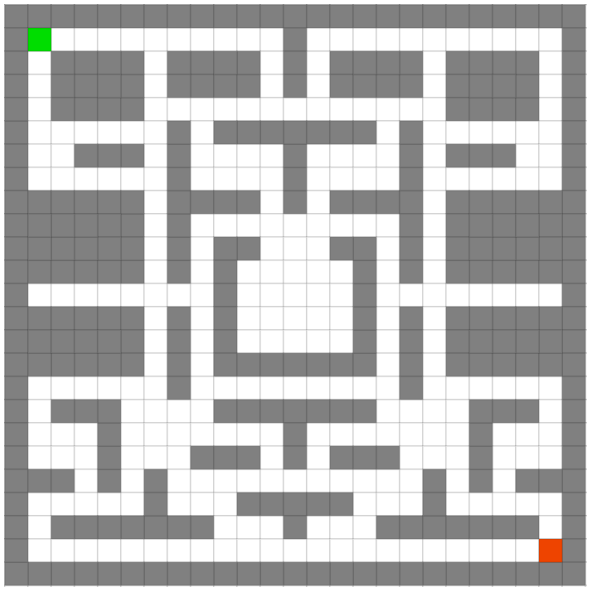


Figure 3.5. Environment 5

Fifth environment is meant to mimic a video game map and is inspired by the Pac-Man game. This should showcase how these algorithms would perform in real video game scenarios.

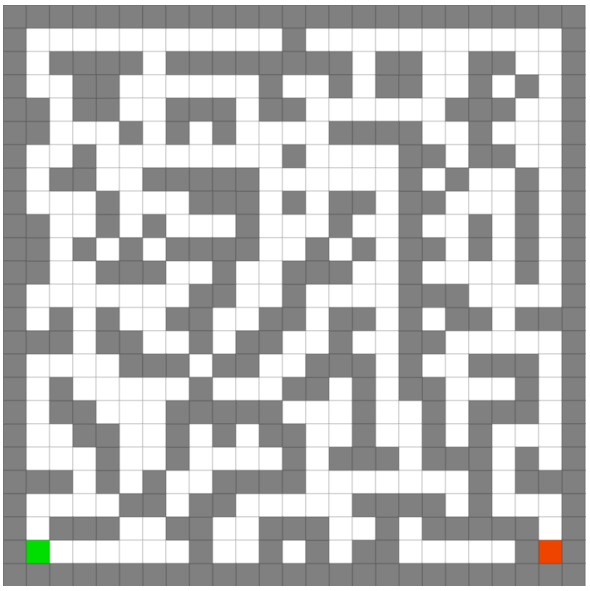


Figure 3.6. Environment 6

Sixth environment is meant to mimic a maze-like video game map. This should showcase how the algorithms traverse through a maze to find the shortest route.

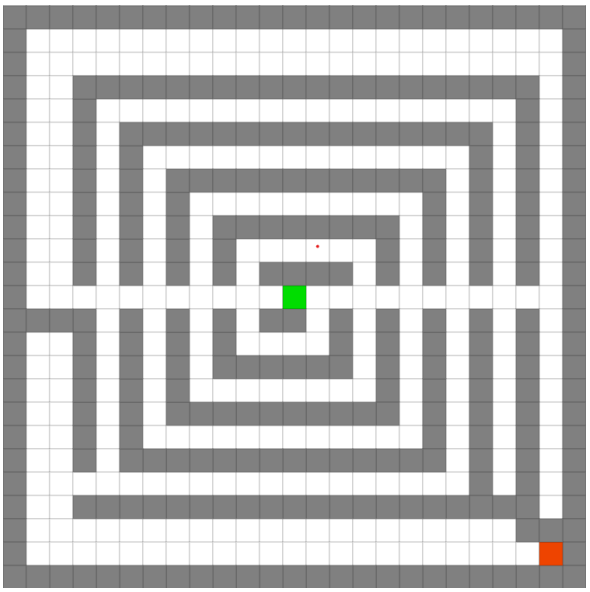


Figure 3.7. Environment 7

Seventh environment is meant to test a different layout for a potential video game map. There are obstacles placed in a specific manner to try test the efficiency of the algorithms.

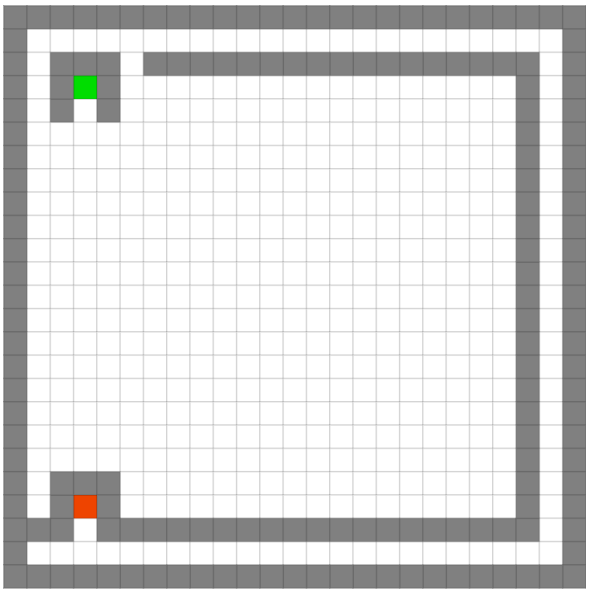


Figure 3.8. Environment 8

Eighth environment is meaning to test the efficiency of Bi-directional versions of the algorithm as well as the impact of greediness on the efficiency.

3.2. Procedure and Variable

The procedure of conducting this experiment which compares the efficiency of different pathfinding algorithms is as follows:

1. The environment is setup in the online software
2. An algorithm is selected and the simulation is initiated
3. Once a route is obtained from the starting node to the destination node, the length of the route and the total number of operations which were done by the algorithm in order to obtain that route are recorded in Table 3.1
4. Steps 2 and 3 are repeated for all other algorithms that are being analysed
5. Once all data is recorded for the first environment, all previous steps are repeated for the rest of the environments the algorithms are being tested on
6. Once all data is gathered on each environment and pathfinding algorithm, the efficiency is calculated as the percentage of the length of route obtained over the total number of operations

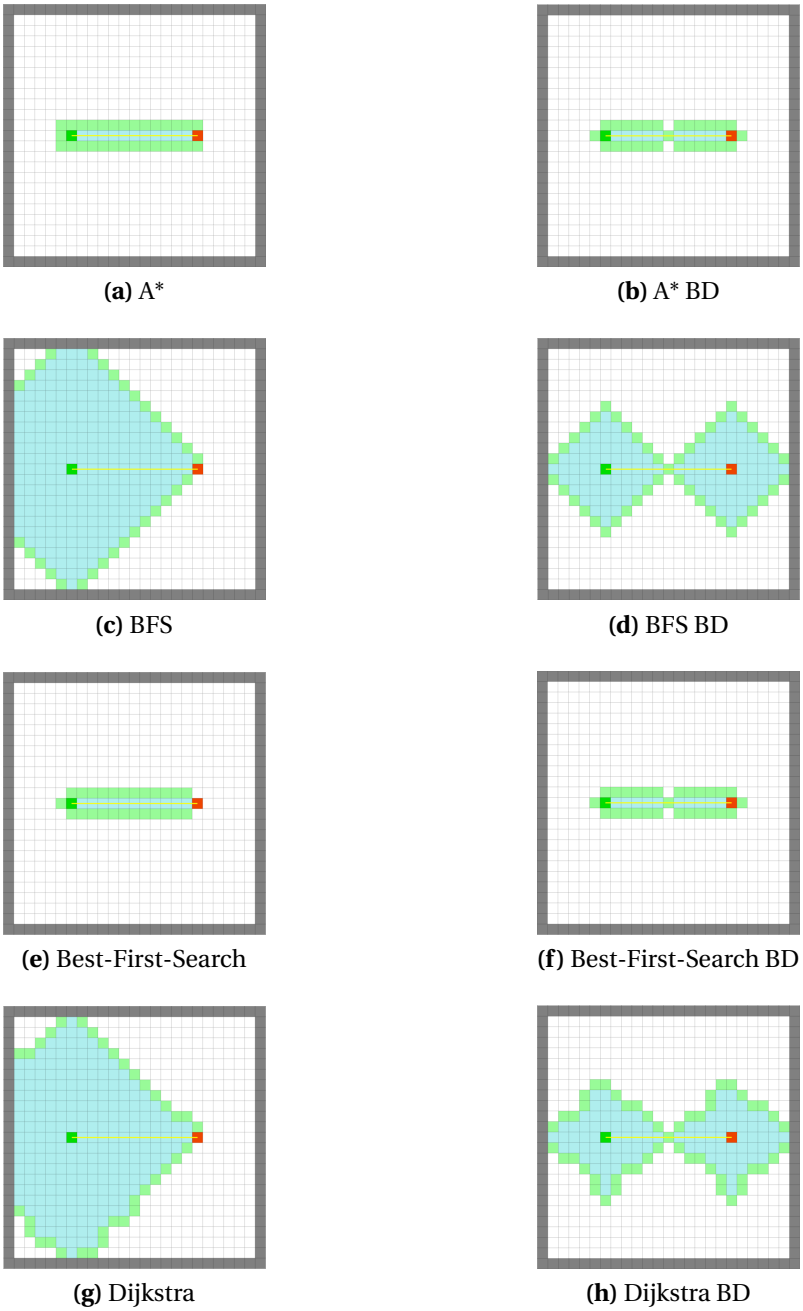
In order to make this experiment fair and unbiased, there have to be some variables which will be controlled. Firstly, all of the simulations and results recorded come from one software, if this wasn't the case then the consequence would be that different software could have different versions of the pathfinding algorithms which can influence the shortest route and number of operations, leading to unreliable results. Secondly the graph must be setup exactly as shown the in the 8 environments, all the black, white, green and red pixels need to be in the exact same positions throughout each of the simulations in

order to ensure the reliability of results obtained. Lastly there are 4 different distance heuristics which can be used for A* and Best-First-Search algorithms which are Manhattan, Euclidean, Octile and Chebyshev. Each simulation done using A* and Best-First-Search need to be conducted using one distance heuristic, Manhattan, due to the fact that changing heuristics can affect the number of operations and the shortest route.

3.3. Simulation

Each of the environments will be tested using Dijkstra's algorithm, Breadth-First-Search Algorithm, Best-First-Search Algorithm, A* Algorithm and all their bi-directional variations. The simulation results for each environment will be visually represented using the software chosen; the number of operations, the length of shortest route, and efficiency for each algorithm in each environment will be calculated and recorded in a table and further evaluated. The calculation of the efficiency will be done on the basis of the ratio of the length of the route to the number of operations. By multiplying the calculated ratio by 100, the percentage efficiency will be obtained.

3.3.1. Environment 1



(a) A*

(b) A* BD

(c) BFS

(d) BFS BD

(e) Best-First-Search

(f) Best-First-Search BD

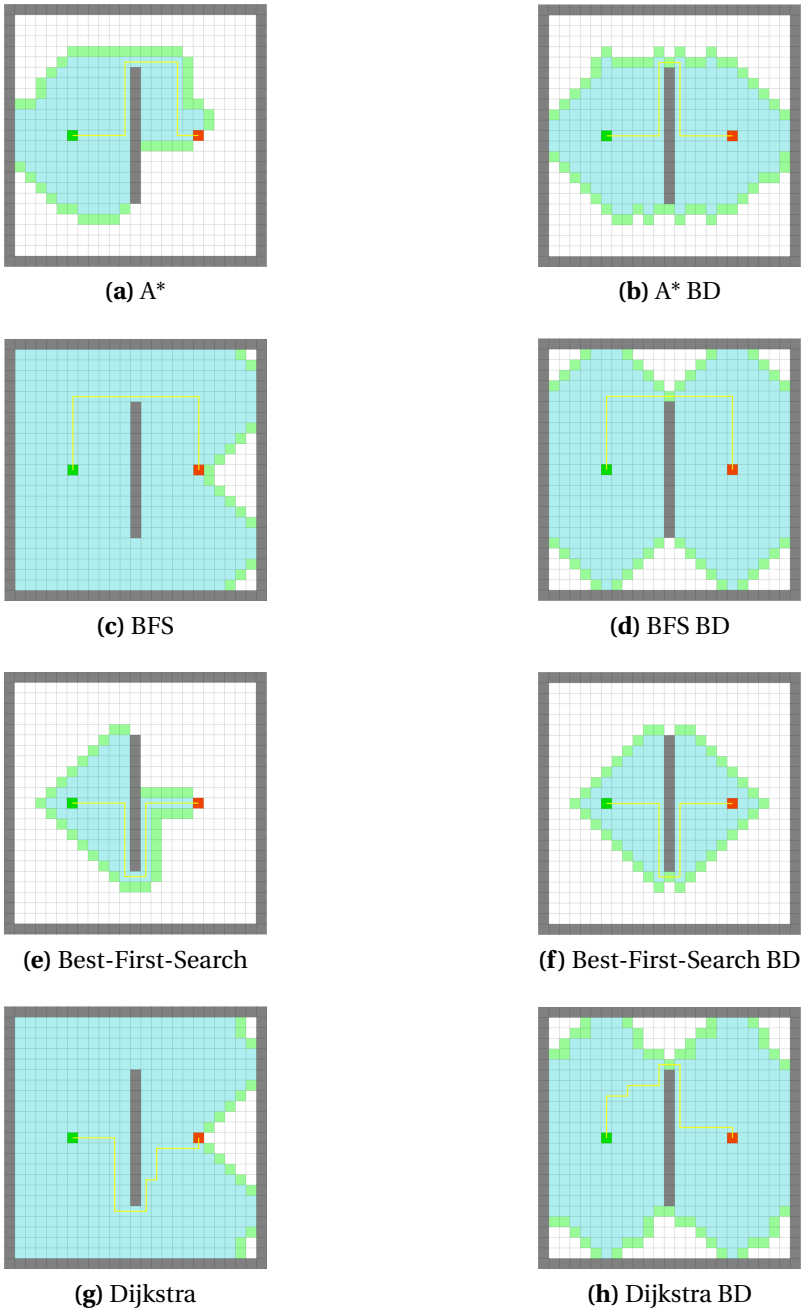
(g) Dijkstra

(h) Dijkstra BD

Figure 3.9. Experiment on Environment 1

From Figure 3.9 we can observe the standard behavior of the algorithms. A* and Best-First-Search reach the target node with least number of visible operations due to their greedy nature. BFS and Dijkstra traverse every node without knowing the direction in which the target node is, therefore number of operations will be inevitably higher. Bi-directional versions of the algorithms have highest reduction in operation count for BFS and Dijkstra.

3.3.2. Environment 2

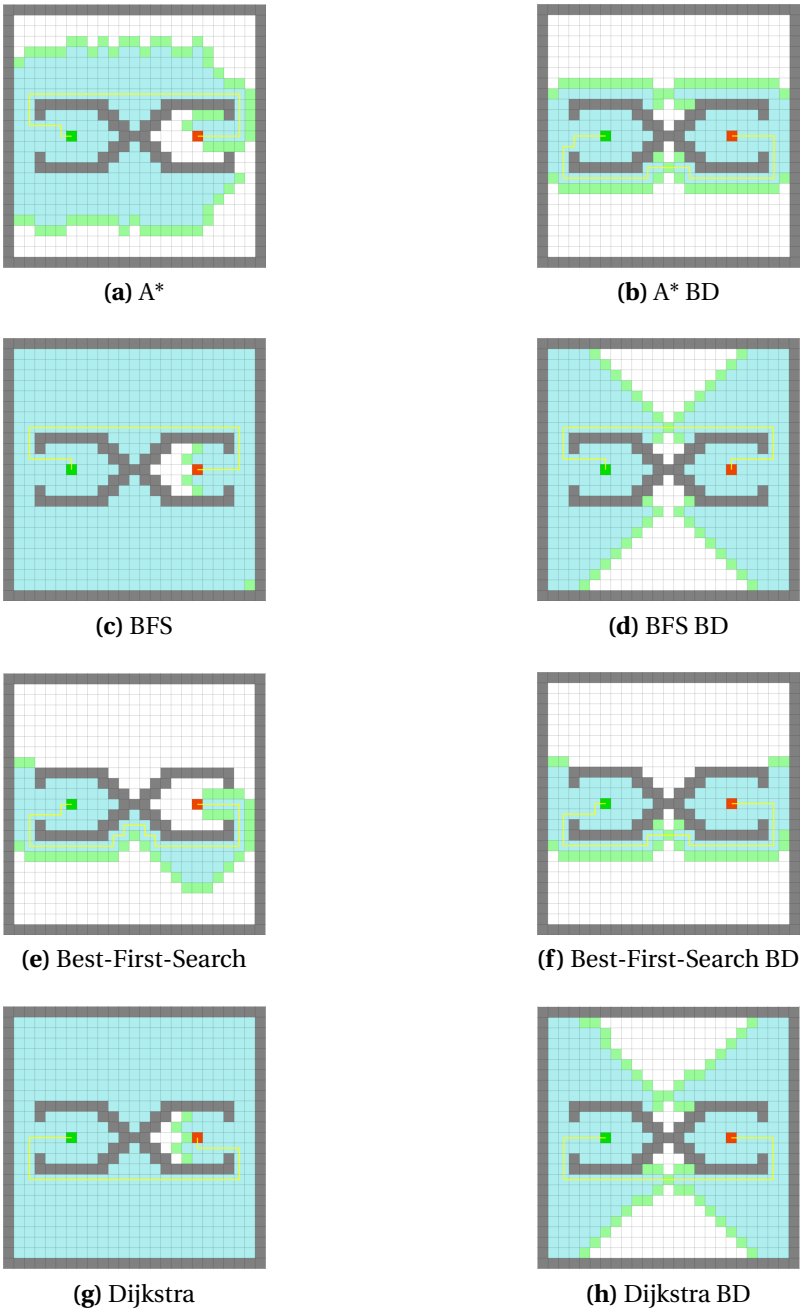


(a) A*
(b) A* BD
(c) BFS
(d) BFS BD
(e) Best-First-Search
(f) Best-First-Search BD
(g) Dijkstra
(h) Dijkstra BD

Figure 3.10. Experiment on Environment 2

From Figure 3.10 we can observe that the wall obstacle makes it so that each algorithm actually finds its own unique path to the destination node. Since the map is symmetrical the bi-directional variations create a symmetrical traversal of the nodes checked.

3.3.3. Environment 3



(a) A*

(b) A* BD

(c) BFS

(d) BFS BD

(e) Best-First-Search

(f) Best-First-Search BD

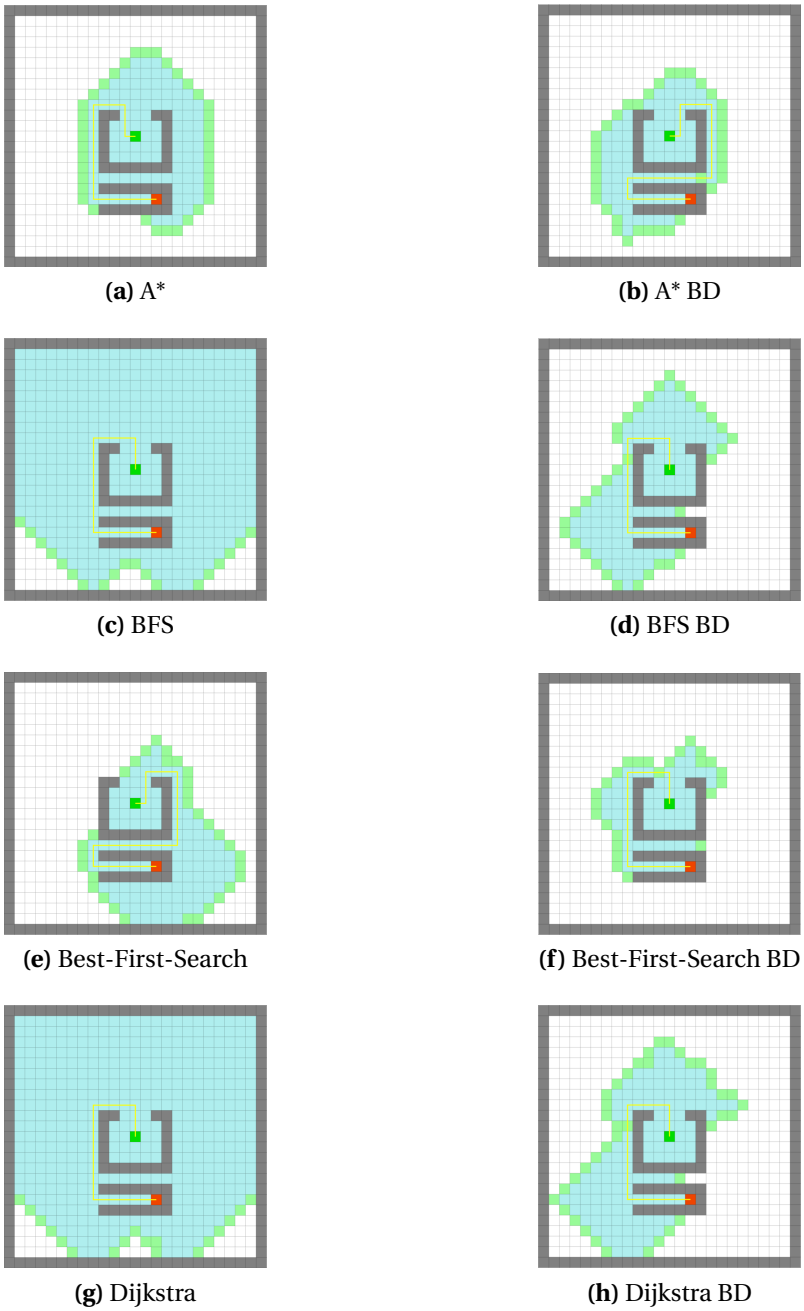
(g) Dijkstra

(h) Dijkstra BD

Figure 3.11. Experiment on Environment 3

From Figure 3.11 we can observe that the nodes checked by the algorithms have a similar pattern to the previous environment. However in A* BD and both Best-First-Search algorithms the greediness leads the algorithm to make a misstep in the direction of the destination node resulting in not finding the shortest path.

3.3.4. Environment 4



(a) A*

(b) A* BD

(c) BFS

(d) BFS BD

(e) Best-First-Search

(f) Best-First-Search BD

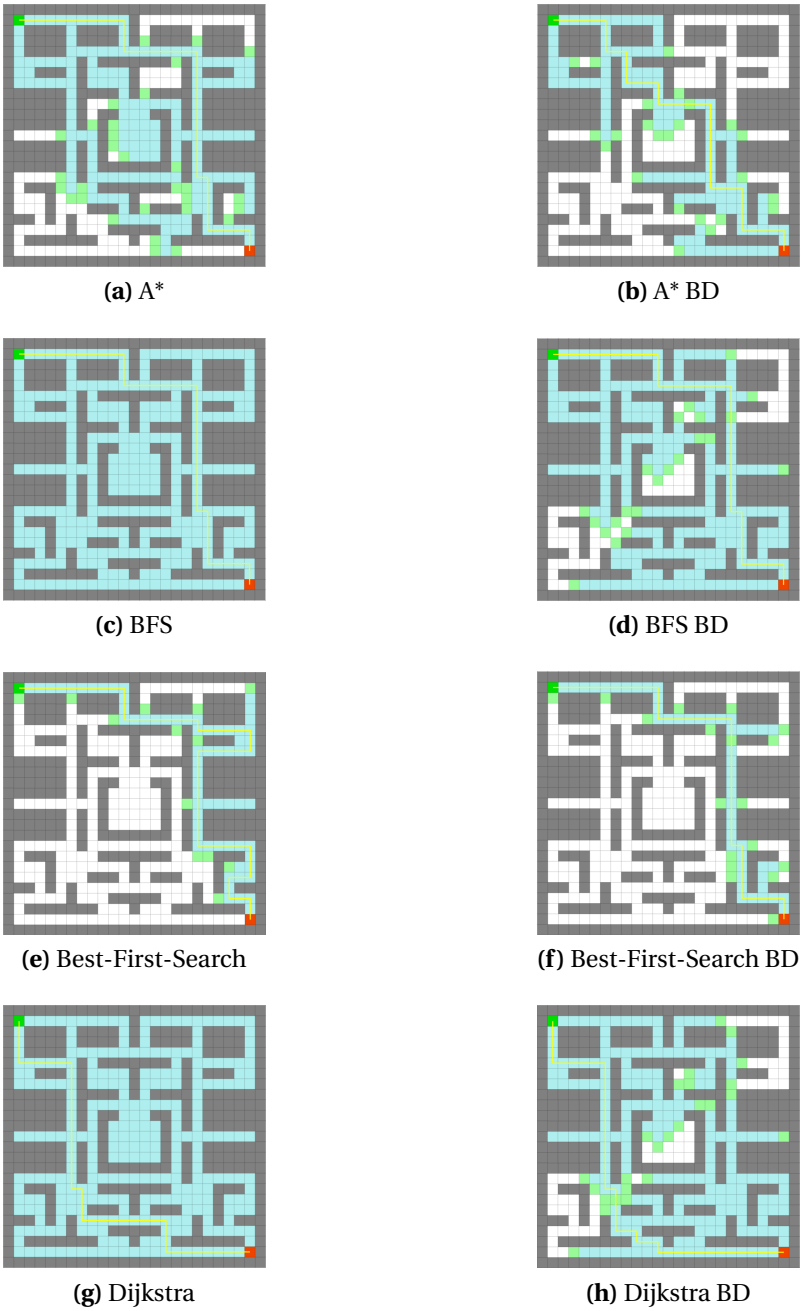
(g) Dijkstra

(h) Dijkstra BD

Figure 3.12. Experiment on Environment 4

From Figure 3.12 we can observe how the algorithms perform in an unsymmetrical environment. We can see that the greedy algorithms explore more nodes closer to the destination node even though the right side of that node is blocked, this showcases that A* approach compared to Best-First-Search could have an advantage. Bi-directional versions of BFS and Dijkstra significantly help the algorithms find the shortest path faster.

3.3.5. Environment 5



(a) A*

(b) A* BD

(c) BFS

(d) BFS BD

(e) Best-First-Search

(f) Best-First-Search BD

(g) Dijkstra

(h) Dijkstra BD

Figure 3.13. Experiment on Environment 5

From Figure 3.13 we can observe how the algorithms behave in a proper video game map layout. BFS and Dijkstra searched every node on the map to find the shortest path, this guarantees shortest however heavily impacts efficiency. Best-First-Search was the only algorithm not to find the shortest path, and the path it did find is not optimal for a video game scenario since each extra step in the route found is time wasted within the video game. However the bi-direction version was able to successfully find the shortest path.

3.3.6. Environment 6

(a) A*

(b) A* BD

(c) BFS

(d) BFS BD

(e) Best-First-Search

(f) Best-First-Search BD

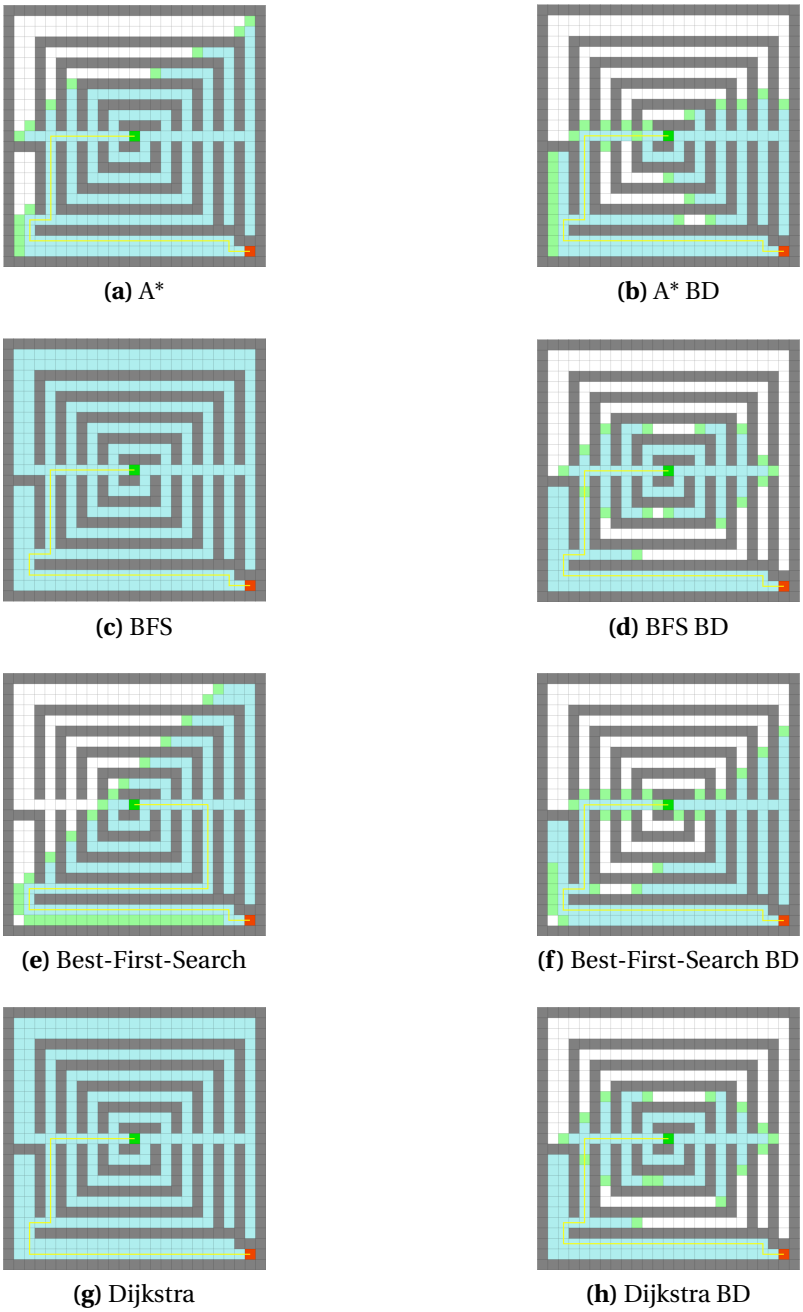
(g) Dijkstra

(h) Dijkstra BD

Figure 3.14. Experiment on Environment 6

From Figure 3.14 we can observe how the algorithms perform in a maze-like map. Similar to the previous environment, BFS and Dijkstra traversed every accessible node on the map to find the shortest path which lowers efficiency. Both variations of Best-First-Search failed to find the shortest path while still traversing a similar amount of the map like A* and bi-directional A*.

3.3.7. Environment 7



(a) A*

(b) A* BD

(c) BFS

(d) BFS BD

(e) Best-First-Search

(f) Best-First-Search BD

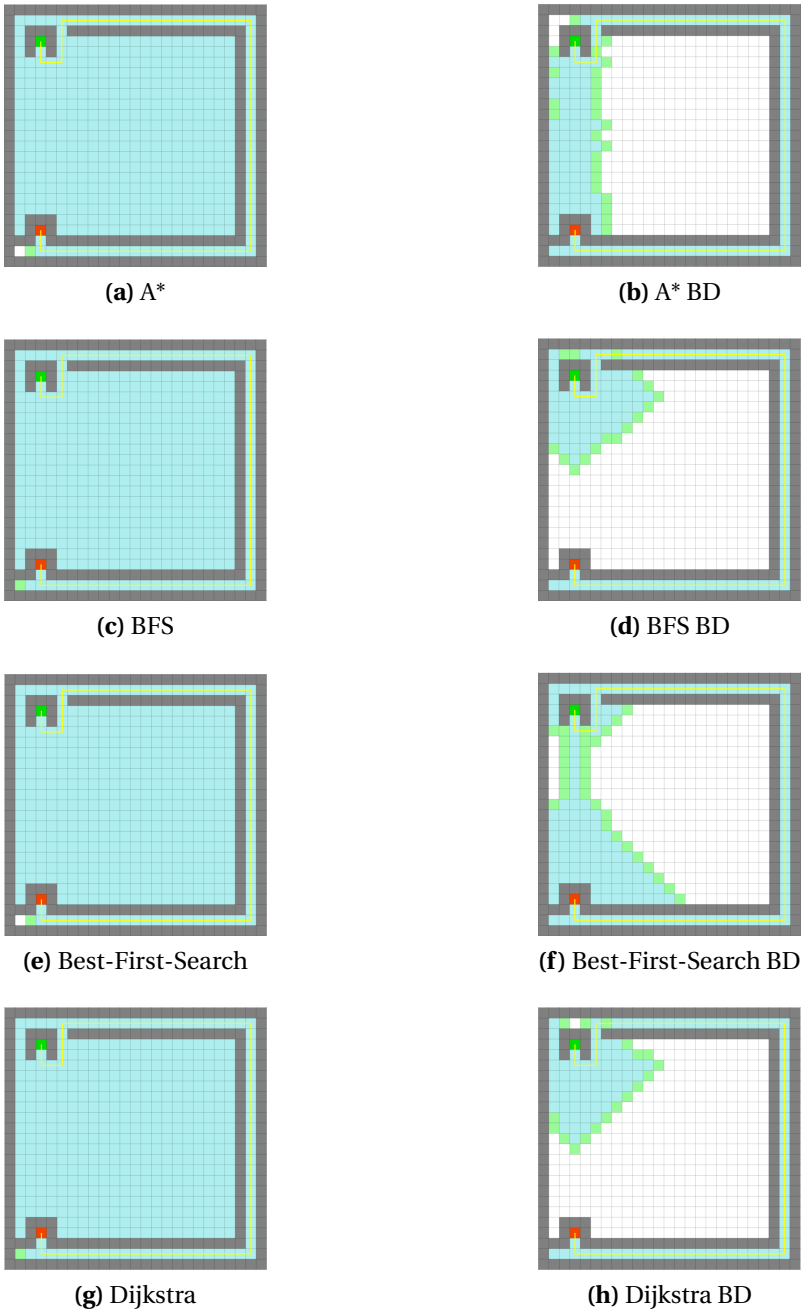
(g) Dijkstra

(h) Dijkstra BD

Figure 3.15. Experiment on Environment 7

From Figure 3.15 we can observe how poorly Best-First-Search performed compared to the shortest path it could obtain. Another interesting observation is bi-directional A* has the lowest efficiency, marginally but still lower.

3.3.8. Environment 8



(a) A*

(b) A* BD

(c) BFS

(d) BFS BD

(e) Best-First-Search

(f) Best-First-Search BD

(g) Dijkstra

(h) Dijkstra BD

Figure 3.16. Experiment on Environment 8

From Figure 3.16 we can observe that all algorithms successfully found the shortest route, however bi-directional A* yet again lacked in efficiency compared to BFS and Dijkstra due to its greedy element. Bi-directional Best-First-Search performed with even lower efficiency due to its purely greedy nature.

3.4. Data Acquired

In Table 3.1 we have all the data acquired through the experiments on all the environments using the intended algorithms. The table is sorted by each of the eight environments, within each environment eight algorithms with the final length of route obtained in pixels, number of operations (nodes visited by the algorithm) in order to find the path, whether or not the path obtained is in-fact the shortest route and finally the efficiency of obtaining the route.

Table 3.1. Data of Results

Environment	Algorithm	Length (pixels)	Num. of Op.	Shortest route	Efficiency (%)
1	A*	12	55	YES	21.82
	A* BD	12	51	YES	23.53
	BFS	12	523	YES	2.29
	BFS BD	12	289	YES	4.15
	Best-FS	12	51	YES	23.53
	Best-FS BD	12	51	YES	23.53
	Dijkstra	12	506	YES	2.37
	Dijkstra BD	12	255	YES	4.71
2	A*	26	381	YES	6.82
	A* BD	26	520	YES	5.00
	BFS	26	969	YES	2.68
	BFS BD	26	849	YES	3.06
	Best-FS	26	188	YES	13.83
	Best-FS BD	26	287	YES	9.06
	Dijkstra	26	958	YES	2.71
	Dijkstra BD	26	848	YES	3.07
3	A*	36	559	YES	6.44
	A* BD	38	319	NO	11.91
	BFS	36	927	YES	3.88
	BFS BD	36	671	YES	5.37
	Best-FS	40	226	NO	17.70
	Best-FS BD	38	256	NO	14.84
	Dijkstra	36	937	YES	3.84
	Dijkstra BD	36	656	YES	5.49
4	A*	22	262	YES	8.40
	A* BD	30	215	NO	13.95
	BFS	22	887	YES	2.48
	BFS BD	22	272	YES	8.09
	Best-FS	30	266	NO	11.28

	Best-FS BD	22	153	YES	14.38
	Dijkstra	22	904	YES	2.43
	Dijkstra BD	22	312	YES	7.05
5	A*	44	388	YES	11.34
	A* BD	44	260	YES	16.92
	BFS	44	596	YES	7.38
	BFS BD	44	460	YES	9.57
	Best-FS	58	151	NO	38.41
	Best-FS BD	44	122	YES	36.07
	Dijkstra	44	596	YES	7.38
	Dijkstra BD	44	459	YES	9.59
6	A*	84	547	YES	15.36
	A* BD	84	359	YES	23.40
	BFS	84	568	YES	14.79
	BFS BD	84	403	YES	20.84
	Best-FS	86	430	NO	20.00
	Best-FS BD	86	328	NO	26.22
	Dijkstra	84	568	YES	14.79
	Dijkstra BD	84	393	YES	21.37
7	A*	42	445	YES	9.44
	A* BD	42	314	YES	13.38
	BFS	42	652	YES	6.44
	BFS BD	42	300	YES	14.00
	Best-FS	56	370	NO	15.14
	Best-FS BD	42	300	YES	14.00
	Dijkstra	42	652	YES	6.44
	Dijkstra BD	42	300	YES	14.00
8	A*	70	917	YES	7.63
	A* BD	70	296	YES	23.65
	BFS	70	919	YES	7.62
	BFS BD	70	264	YES	26.52
	Best-FS	70	917	YES	7.63
	Best-FS BD	70	346	YES	20.23
	Dijkstra	70	919	YES	7.62
	Dijkstra BD	70	270	YES	25.93

3.5. Analysis of Data

Starting with the analysis of the number of times each algorithm failed at calculating the shortest route, we only have Best-First-Search and its bi-directional variation failing to find the shortest routes as well as bi-directional A* failing twice. The occurrence for Best-First-Search failing to find the shortest path was five out of eight times. The occurrence for bi-directional Best-First-Search failing to find the shortest path was two out of eight times which is a significant improvement with regards to this algorithm.

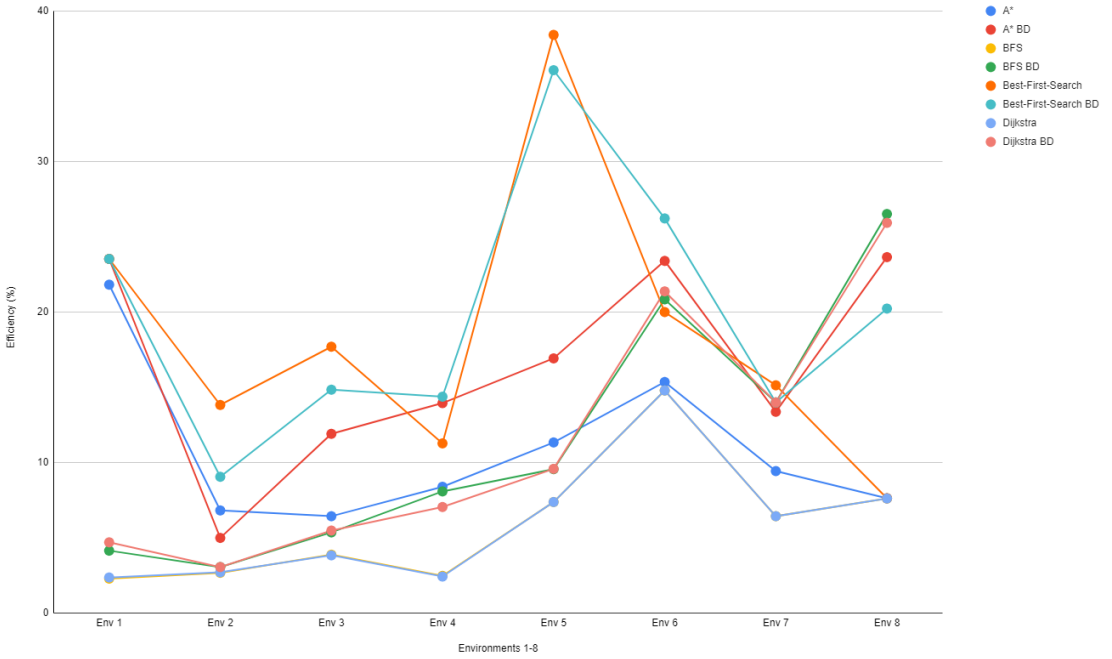


Figure 3.17. Efficiencies of all Pathfinding Algorithms for each Environment

For environment 1, the 3 most efficient algorithms are the ones which at least failed once to find the shortest path, next closest fully reliable algorithm would be A* with a significantly higher efficiency than the rest of the algorithms left.

For environment 2 Best-First-Search is the most efficient by a large margin, being over twice as efficient as A*. However in terms of the algorithms which always yield the shortest route A* is still coming out on top in this scenario.

For environment 3, the most efficient algorithm which found the shortest route was yet again A*, while here we notice the one and only time 3 algorithms not finding the shortest path due to their greedy nature.

In environment 4 bi-directional Best-First-Search was the most efficient by a significant margin however it isn't optimal in every situation hence A* is yet again the most efficient consistent algorithm.

In environment 5 is the exact same case as in environment 4, bi-directional Best-First-Search is most efficient by a large margin but A* remains the most reliable choice.

In environment 6 comes a twist where bi-directional A* is the most efficient however out of the list of optimal algorithms bi-directional Dijkstra comes out on top as the most efficient algorithm. A* comes in as third most efficient behind bi-directional BFS too.

In environment 7 is a similar case as environment 6 where bi-directional BFS and bi-directional Dijkstra are more efficient than A*.

Finally, in environment 8, for the first time bi-directional BFS is the most efficient by over three times that of A* and bi-directional Dijkstra coming in close second in efficiency.

The two environments (1 and 2) in which Best-First-Search algorithm was successful were more basic environments without any advanced obstacles however in those environments it was always the most efficient. BFS and Dijkstra were very inefficient in those very environments compared to A* and Best-First-Search algorithms, this aligns with the way the algorithms function and without the greedy aspect of the algorithm in simpler environments with little to no obstacles they take more time to find the shortest path. We see an uptrend in Figure 3.17 for bi-directional versions of BFS and Dijkstra being more efficient as progressing through the environments. They are the more efficient in the more complex environments 6,7 and being the most efficient in environment 8 with far more obstacles, while being algorithms that guarantee finding the shortest path.

3.6. Evaluation

We can firstly conclude that if one wants to implement Best-First-Search into a video game for example, it is more optimal to implement bi-directional variation of it. However we have seen that it isn't always successful in finding the shortest path and in situations where this is vital one should consider using a different algorithm.

Looking at the performance of BFS, it always finds the shortest path which makes it a reliable algorithm to implement in real-life solutions. It is a very similar case with Dijkstra's algorithm and both perform at the same or very similar efficiency as one another. However their bi-directional versions were the most efficient algorithms in three out of eight environments, which were also of higher complexity than all the other environments except the fourth.

A* algorithm was the most reliable algorithm whilst being the most efficient majority of the time, but as discussed earlier bi-directional Dijkstra and BFS perform more efficiently in higher complexity environments which are the kinds of scenarios which are most practically used in video games or any other real-life solutions. If one wanted to implement a uni-directional algorithm then A* is the most efficient always, it has proven to be a more efficient version of both BFS and Dijkstra's algorithms.

If we were to make a hierarchy of the most efficient algorithms to least while prioritizing the shortest path then A* would be on top, as it as proven its consistency and overall

efficiency in different scenarios. Bi-directional Dijkstra's algorithm would be a close second with bi-directional BFS being right behind. These three algorithms are the algorithms that could be used and implemented in any real-life applications and will yield promising results. Best-First-Search resulted in being the worst algorithms when prioritizing finding the shortest path, and in few cases resulting in paths which were no where close to the shortest.

If one doesn't prioritize the algorithm finding the shortest route but wants to utilize as less resources as possible and just find a short route then either bi-directional A* or bi-directional Best-First-Search would be the algorithms that could be implemented. These algorithms have a high success rate to find the shortest path in which case they find it with the highest efficiency as well, but in the off-chance they fail to do so, one still obtains a short path at high efficiency.

3.7. Conclusion

In conclusion, different pathfinding algorithms were proven to be the most efficient in different environments. Returning to the research topic of this thesis 'A Comparison of Efficiencies of Different Pathfinding Algorithms in Video Games using Grid-Based Maps' we can conclude that certain algorithms such as A* are always reliable and efficient to use in majority of cases, there are couple of alternative algorithms which can be implemented in video games using grid-based maps such as bi-directional Dijkstra or bi-directional BFS.

The individual efficiency of each algorithm would vary depending on the map of the particular video game and the start/end nodes. Even though the alternative algorithms proved to be more efficient in slightly more complex scenarios, the lack of efficiency in the simpler environments deem them less overall efficient compared to A*. Either way, these three algorithms would be fit for commercial use in video games with grid-based maps, and since not every environment possible was tested one would have to conduct their own line of testing for their video game for further data regarding their situation.

4. Project

In this section we will discuss the details related to the project implemented using some of the pathfinding algorithms which have been discussed in this paper.

4.1. Project Design

After extension research into the efficiencies of different pathfinding algorithms in a set environment, it was concluded that the algorithms should be put to the test in a real-life application. As the project is based around pathfinding efficiency in video games with grid-based maps, the project will be a video game utilizing a grid-based map which has pathfinding algorithms implemented.

The main design is a player controlled character summoned into a grid map where the player will be able to move vertically and horizontally on the grid map. There are NPCs which will target the player and chase him/her down while the goal of the player is to collect all the items in on the map without getting caught. It is a simple concept, Pac-Man would be the closest application which already exists, however in this project the NPCs or enemies of the player will be implemented using a different pathfinding algorithm.

We will have two enemies, each one of the enemies will pathfind to the player using a different algorithm; BFS and A*. These algorithms were chosen since they always yield the shortest path compared to Best-First-Search, and A* has proven to be an enhanced version of Dijkstra's algorithm therefore it was decided to compare the above two algorithms. By implementing this, the player will be able to see in real-time how the algorithms perform while the player and NPC position are dynamically changing every second. We will record computational time and number of operations to compare the two algorithm's efficiencies.

4.2. Project Implementation

This project will be implemented using Python, more specifically the Pygame library. Python is one of the most popular programming languages used worldwide today, it is often used to build all kinds of software (including games), make websites, analyse data, etc. This makes it a general-purpose language since it isn't confined to solve a particular task but can solve a variety of problems and create different programs.

Pygame is a Python wrapper for the SDL library, also known as the Simple DirectMedia Layer. The use of SDL is to enable cross-platform access to the system's components, particularly multimedia ones such as keyboard, mouse, video, etc. Pygame allows us to create our own game which we will be able to use for our project to demonstrate the purposes of it.

4.3. Project Outcome

Below we can observe a screenshot of the starting of the game:

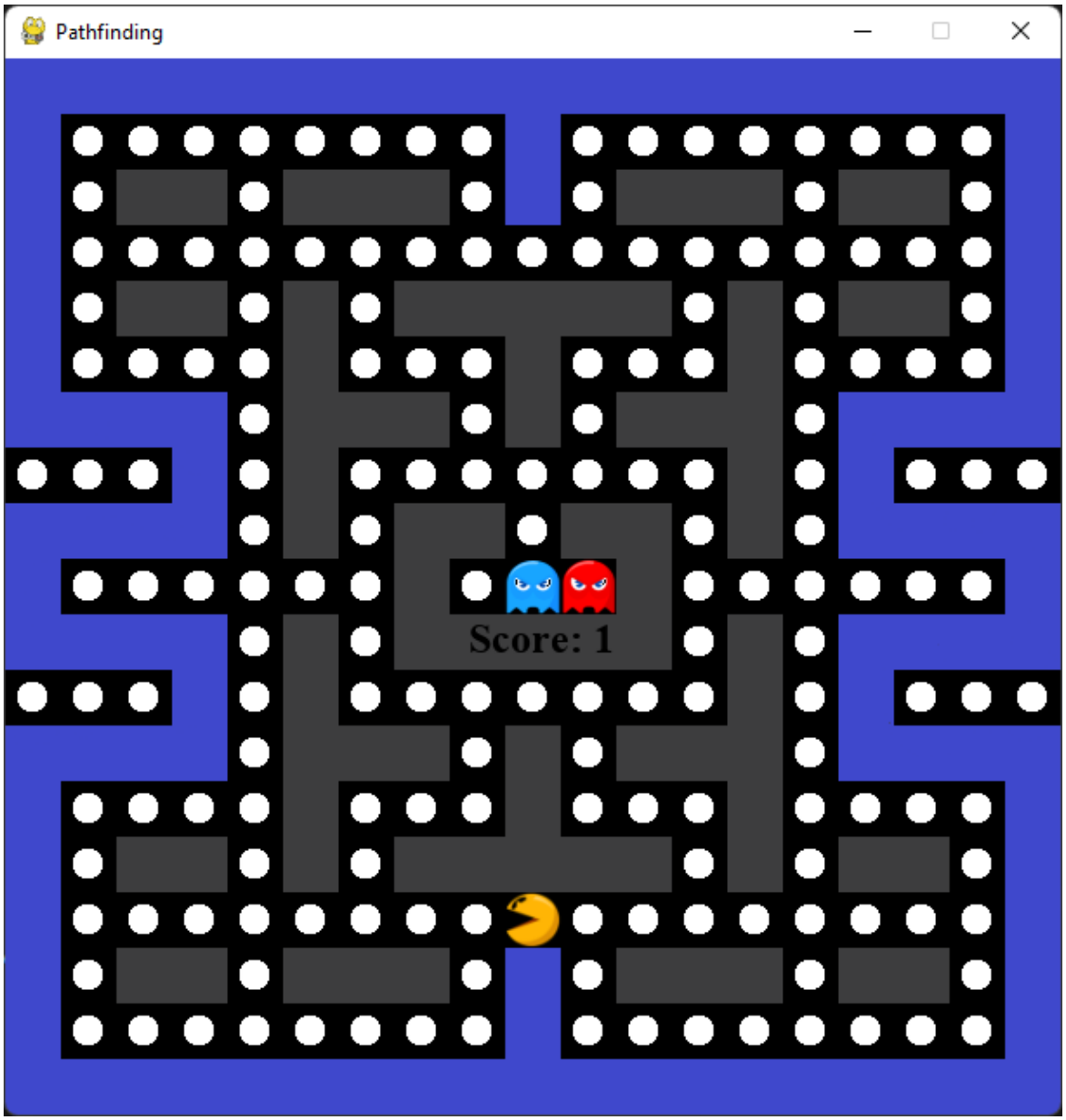


Figure 4.1. Starting Position of the Game

The orange character is the user as the player being controlled, the red and blue NPCs are the enemies which movements are implemented using BFS and A* algorithms respectively and are programmed to chase the player. The blue and gray lines are walls within the map, neither player nor NPC can pass through them, the white dots across the map are the items which the player must collect in order to beat the level.

4.4. Project Results

After implementing the pathfinding algorithms and conducting a significant amount of tests we can make a few observations.

4.4.1. Static Scenario

Firstly, we tested how the two algorithms performed when finding the path towards the player who is staying static in the starting position. Below we can observe the paths each enemy took, blue path being A* algorithm and red path being BFS algorithm.

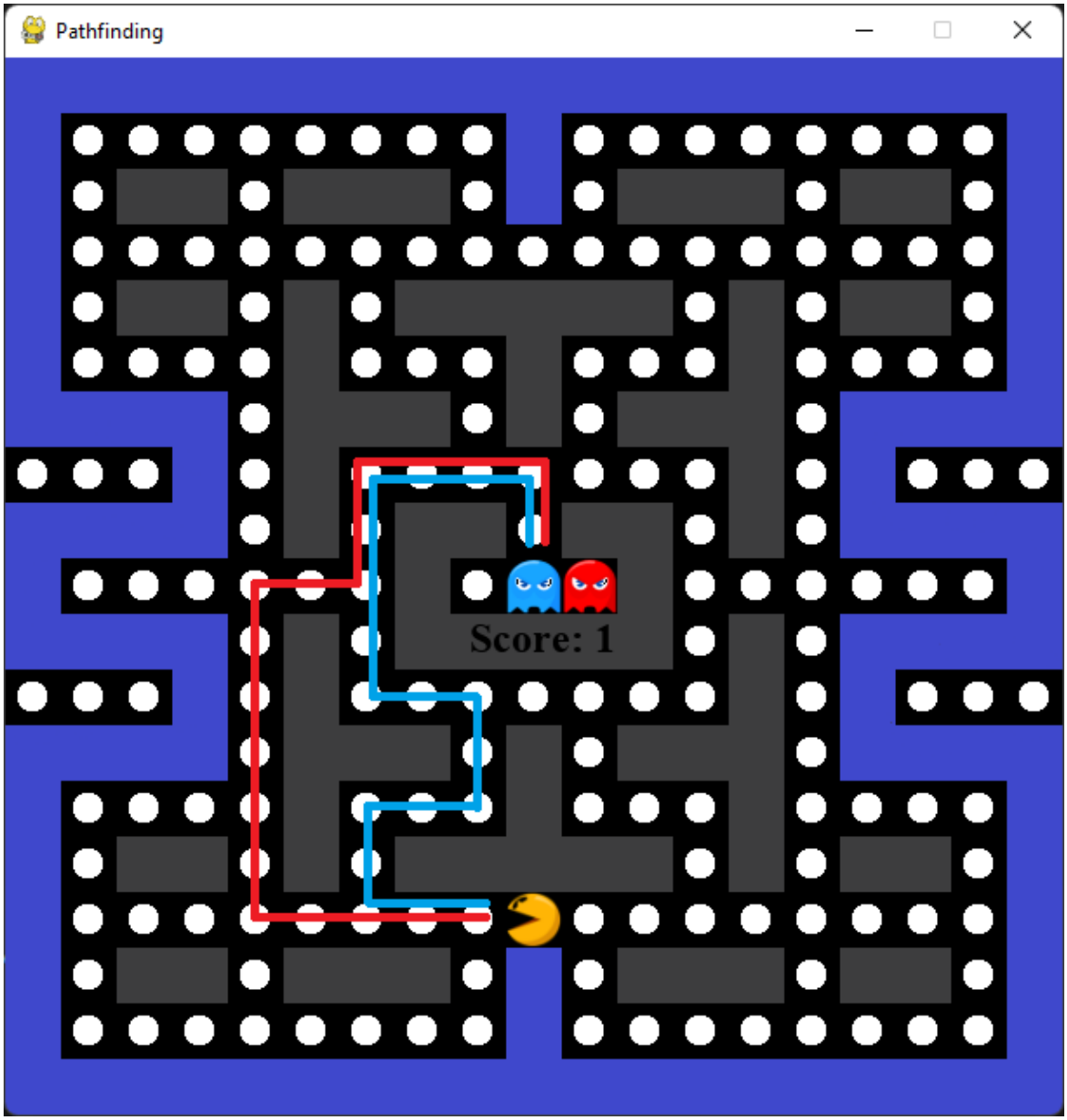


Figure 4.2. Routes of Algorithms when Player is Static

These two paths are equidistant, both algorithms found the shortest path to the player,

however the following table shows computational time and number of operations from the starting positions of the enemies to the target static player.

Table 4.1. Static Scenario Results.

Algorithm	Computational Time (s)	Number of Operations
BFS	0.0035	531
A*	0.0005	189

From Table 4.1 we can observe that even though both algorithms successfully found the shortest path to the player, A* algorithm's computational time was seven times faster than BFS algorithm's, with almost three times less operations needed to find the shortest path. This computational time difference might seem insignificant with respect to a small scale game like this, however noticing a multi-fold difference between both algorithms even at this scale shows that the impact on greater scenarios such a navigation systems would result in many resources saved if A* was the chosen algorithm.

4.4.2. Dynamic Scenario

The game was thoroughly tested, played through a significant amount of times to find whether or not there is actually a difference in how the enemies act when actually playing the game. When both enemies started together off of the same position, they would always remain in sync with one another, on a rare occasion disperse when there are two shortest paths to the player, but then meet again. However, throughout experimentation, on average A* remained at least seven times faster than BFS with regards to time complexity, while number of operations would vary but A* needing significantly less operations to find the path in each given scenario.

4.5. Project Conclusion

In conclusion, this project was significant to apply the knowledge gained from the experimentation conducted in this paper. Another variable of computation time was able to be compared between the algorithms to find out real-life impact of using the algorithms. Both BFS and A* performed as expected, A* needing less number of operations to find the shortest path compared to BFS while both always managing to find the shortest route. Computational time for BFS being higher than for A* is justified by the fact that we already concluded that BFS requires more operations, therefore more resources of the system being used which causes an increase in computation time. This brings us to the conclusion that for larger scale video games or projects, A* algorithm would be the most efficient and optimal algorithm to use in order to achieve finding the shortest path with the least number of operations and therefore resulting in lowest resources used and computational time.

5. Future Work

This research paper only dives into the analysis of a set number of pathfinding algorithms operating on a 2D square grid-based map. We could have analysed more complex environments which are closer to real-life applications to obtain more results. We could do further research into different shaped grids, such as triangular or hexagonal tiles. In our research we restrict movement to horizontal and vertical, allowing diagonal movement will completely change how the heuristic functions work and the impact on the shortest route found could be significant. We could also change the heuristic function used with certain algorithms, use Euclidean distance or Chebyshev or perhaps compare these functions to see which are the most efficient.

One can conduct further research into 3D maps, as well as other types of maps such as navigation meshes, since video games are become more and more advance by the years, 3D maps are becoming exceedingly common in a variety of scenarios. Researching 3D maps would also mean researching different, possibly more advanced and mature algorithms better suited for 3D environments meaning data set could potentially grow leading to better experimentation and analysis.

Regarding the project, if the further work would be based on 3D maps then the video game in the project could also be a 3D game, which usually gives a more immersive experience for the user. There could potentially be a possibility to combine the use of all pathfinding algorithms, where at each step of the game when needing to find the next best node, the code will analyse the results of all the pathfinding algorithms and use the most efficient one in the given situation and be able to dynamically change pathfinding algorithm to keep it at its peak efficiency.

6. Summary

The work in this thesis has covered four different pathfinding algorithms, Dijkstra, BFS, Best-First-Search and A*, which have all been discussed as well as other factors such as heuristic functions, types of grids and bi-directional algorithms.

The four algorithms with their bi-directional versions were experimented on eight different environments. Data such as number of operations, length of route and whether or not the algorithm found the shortest path from starting to ending point from which the efficiency for each algorithm in each environment was calculated and analysed. Conclusions were drawn on which algorithm is the most efficient as well as some algorithms being more efficient in particular scenarios than others.

Based on the conclusions, BFS and A* algorithms were selected and implemented into a video game with a grid map to demonstrate real-life application of the algorithms. The player playing the game is in control of a character that is in a maze, with the goal of collecting all the items on the map while getting chased by two enemies each chasing the player using BFS and A* respectively. The shortest path from the enemy to the player is calculated after every step the enemy makes, meaning the shortest path is dynamically being calculated to account for the player position change within the map.

The data drawn from the project introduced a new variable of computational time for each algorithm to find the shortest path from the enemy to the player. This may not be a significant factor in a small video game project, however the difference in computational times between BFS and A* were multi-fold which implies that on a bigger project where each millisecond and resource used plays an important role, the pathfinding algorithm selection will play a vital part in ensuring the project is as efficient as it could be.

Bibliography

[1] *Pathfinding*. [Online]. Available: https://www.yourdictionary.com/pathfinding.

[2] S. L. Lev Krayzman Nilay Kumar, *Pathfinding applications*. [Online]. Available: https://mbhs.edu/~lpiper/pathfinding/applications.php.

[3] ganga4518, *The algorithms behind the working of google maps*. [Online]. Available: https://blog.codechef.com/2021/08/30/the-algorithms-behind-the-working-of-google-maps-dijkstras-and-a-star-algorithm/.

[4] P. L. Frana and T. J. Misa, "An interview with edsger w. dijkstra", *Commun. ACM*, vol. 53, no. 8, pp. 41–47, Aug. 2010, ISSN: 0001-0782. DOI: 10.1145/1787234.1787249. [Online]. Available: https://doi.org/10.1145/1787234.1787249.

[5] J. Morris, *Dijkstra's algorithm*, 1998. [Online]. Available: https://www.cs.auckland.ac.nz/software/AlgAnim/dijkstra.html.

[6] C. W. Thaddeus Abiy Hannah Pang, *Dijkstra's shortest path algorithm*. [Online]. Available: https://brilliant.org/wiki/dijkstras-short-path-finder/.

[7] A. Klochay, *Implementing dijkstra's algorithm in python*, Oct. 2021. [Online]. Available: https://www.udacity.com/blog/2021/10/implementing-dijkstras-algorithm-in-python.html.

[8] *Mathematics*. [Online]. Available: https://math.stackexchange.com/questions/1136972/how-to-normalize-edges-weight-between-0-and-1.

[9] Computerphile, *Dijkstra's algorithm - computerphile*. [Online]. Available: https://www.youtube.com/watch?v=GazC3A4OQTE.

[10] M. Sirota, *Algorithms - best - first*. [Online]. Available: https://cs.stanford.edu/people/eroberts/courses/soco/projects/2003-04/intelligent-search/best.html.

[11] A. S. Steve Mussmann, *Graph search algorithms*. [Online]. Available: https://cs.stanford.edu/people/abisee/gs.pdf.

[12] M. Horvat, *The best-first search algorithm in python*. [Online]. Available: https://blog.finxter.com/the-best-first-search-algorithm-in-python/.

[13] P. Garg, *Breadth first search*. [Online]. Available: https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/.

[14] S. Bhadaniya, *Breadth first search in python (with code) | bfs algorithm*, Dec. 2020. [Online]. Available: https://favtutor.com/blogs/breadth-first-search-python.

[15] A. Patel, *Introduction to a**, Apr. 2022. [Online]. Available: http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html.

[16] *The insider's guide to a* algorithm in python*. [Online]. Available: https://www.pythonpool.com/a-star-algorithm-python/.

[17] H. K. Sidhu, "Performance evaluation of pathfinding algorithms", M.S. thesis, University of Windsor, 2020.

[18] *Euclidean vs manhattan vs chebyshev distance*. [Online]. Available: https://iq.opengenus.org/euclidean-vs-manhattan-vs-chebyshev-distance/.

[19] *Pathfinding.js*. [Online]. Available: https://qiao.github.io/PathFinding.js/visual/.

List of Figures

List of Tables

Listings

List of Appendices

Appendix 1. Installation

To successfully run the project, one must have the following dependencies installed:

- Python
- PIP
- Pygame
- Pillow

Instructions

If the user has the source code for the project then the user must do the following:

1. Download Python3
2. Install pip for python
3. Install pygame using pip
4. Install Pillow using pip
5. Navigate to the folder where Game.py is located
6. Execute command python3 Game.py
7. Start playing with the keyboard's arrow keys