Roskilde University

international bachelor in natural sciences

Comparing Path-Finding Algorithms

Comparing A*, Dijkstra, Breadth First Search and Depth First Search

Authors:
Christian Weibel
Dennis-Vlad Chirila
Kyle Johnson
Lukas Rasocha
Matthias Kaas-Mason
Robert Špralja

Supervisor:
Line Reinhardt

May 27, 2020

Abstract

Finding the shortest path between two places in a fast and efficient way is a problem that many are looking to solve. In our report we looked at this problem in a unique way by implementing and comparing some path-finding algorithms in the p5.js editor. We looked at 4 methods of path-finding each with a different approach to solving the problem. We tested these 4 algorithms on maps we created to find their performance in time, complexity, etc. We used these factors to compare all the algorithms and find out which perform best in different cases. In the end we found that the A* algorithm performed the best based on the tests that were done.

Contents

1 Introduction

Many problems of engineering which are of scientific importance can be related to the general problem of finding the shortest path through a graph. Shortest path algorithms are computer programs used to reveal the shortest way from position A to position B in a graph, where A is the starting position, and B the end or final position. There are many different algorithms that trade off speed and accuracy of the shortest path. Our research question is "How effective are different shortest path finding algorithms". To solve this we will first explain the graph type we chose to use in section 2.2. We also need some methods to compare the results of these algorithms which will be explained in section 2.8.

We will keep our focus on four important algorithms because they form the basis of most high end programs that are used today. These algorithms are Depth First Search, Breath First Search, Dijkstra, and A*. The aim of this project is to assemble the known theory, to measure the performances on different maps, and finally to sum up a detailed comparison regarding the efficiency of these algorithms.

The specific graph used here - being a visual representation - is commonly referred to as a 'map', because the aim is to show the positions of certain objects over an area. In this case, the certain objects are : the starting position (A), the end position (B), the obstacles and possible paths. The map we chose to use is a simple grid where each node is connected to at most 8 adjacent nodes, 4 being diagonals. Each node can be available or a blocked path. In our implementation we chose to allow the algorithm to pass diagonally through 2 blocked nodes.

A position is linked to specific 'coordinates', and the coordinates depend on the map that is being used. It comes down to the definition of the parameters that are chosen to be used. What units do we use as measures? What is the origin? What is defined as a dimension in our map? Therefore, the possible parameters that can be used to define a position of an object in space have great diversity.

In this project we are trying to see if we can implement efficient path-finding algorithms and test them on simple maps we created to find out which algorithm performs the best. We aim to restrict ourselves on the algorithms used to find the shortest path in length. The range of their applications is wide, while the purpose is generally to save time and/or energy. The algorithms are used in common GPS (global positioning system), on mobile phones apps such as Google maps, for professional software in many industrial fields from goods deliveries up to space travel, but also for routing wires on circuit boards.

Originally used with the aim to find a path out of a maze, the first serious algorithms were developed in the 19th Century, with the apparition of Depth First Search. Since then, the algorithms efficiency were increased in relation to the specific purposes of the field in which they are used.

2 General Theory

2.1 Programming Language and Libraries

Within web programming there are 3 key features that are present in every webpage. The first two are HTML and CSS, neither of which are the focus of this essay, so briefly: these control the look and structure of a given webpage.

JavaScript is the final key feature and it provides all the flexibility and control to dynamically update content, control multimedia and much more. So if you ever see a webpage do anything other than just sit there chances are its JavaScript. JavaScript runs in the virtual machine created by your browser to show webpages.

We used the p5.js library to control all of the graphical input and output. p5 is a JavaScript library (written in JavaScript) that makes the control and automation of many drawing and HTML canvas features a lot easier. p5 is essentially a version of Processing but for modern environments. So while we could write everything ourselves without too many changes, p5.js streamlined this aspect of development.

The consequence of this is that the code for our algorithms can be copy/pasted into another piece of JavaScript code that has no references to p5.js and it will run the exact same way (with no alterations required). Since, only the drawing is handled by p5.js. Making this a truly JavaScript only implementation of these algorithms.

An important distinction between p5.js and normal JavaScript is that p5 has a method that is called continuously throughout execution (namely the "draw()" function). Whereas normal JavaScript consists of a collection of "callback" functions that are called upon interactions with the web page. The consequence of this is while a normal algorithm will have a setup followed by a loop that calculates the result in its entirety, we have a setup function and a step function. The step function is called within the draw() function to emulate a loop calculating the result all at once. The positive of this added complexity is that is allows us to visualize our results and the operation of each algorithm step by step. It is also important to note that the JavaScript was run locally on a computer, therefore all the data needed to be collected on the specific computer to avoid misleading results.

2.2 Graph Theory

As defined by Bondy and Murty in 1976 [1] a graph G in its essence is an ordered triple, consisting of a non-empty set $V(G)$ representing the nodes, a non-empty set $E(G)$ representing the edges, and a function $\psi_G(e)$ which associates each edge e with an unordered pair of nodes, as seen in equation (1).

$$G = (V(G), E(G), \psi_G(e)) \tag{1}$$

Graphs got their name from the fact that they are often represented graphically. This is often done by representing nodes as points, and edges as lines, as seen in Figure 1 defined by equations (2), note that the relative position of the points and length of the lines generally bear no significance in graphical representations of graphs[1].

$$G = (V(G), E(G), \psi_G(e)) \tag{2a}$$

$$V(G) = \{A, B, C, D, E\} \tag{2b}$$

$$E(G) = \{e_1, e_2, e_3, e_4, e_5, e_6\} \tag{2c}$$

$$\psi_G(e_1) = (A, B) \tag{2d}$$

$$\psi_G(e_2) = (A, D) \tag{2e}$$

$$\psi_G(e_3) = (B, C) \tag{2f}$$

$$\psi_G(e_4) = (B, D) \tag{2g}$$

$$\psi_G(e_5) = (C, D) \tag{2h}$$

$$\psi_G(e_6) = (C, E) \tag{2i}$$

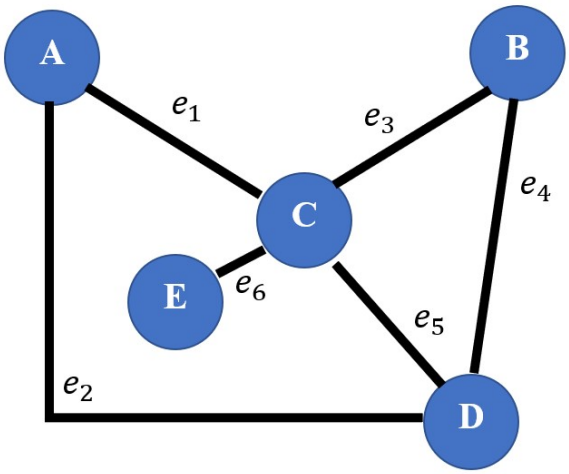


Figure 1: Graphical representation of graph G from equation (2)

[1]In some of our graphs, specifically grids, relative distances will be representative

If a node v is in the ordered pair associated by $\psi_G(e)$ with edge e, then the node v and the edge e are considered *incident*. A *path* is defined as a sequence of distinct and interchanging nodes and edges, where every edge succeeding or preceding some node and the node are incident. A path must also end and start with a node (i. e. between the *start* and *end* node). A path between A and B from equation (2) could be $path_1 = Ae_1Ce_3B$ or $path_2 = Ae_2De_5Ce_3B$. If we define the length of a path as the number of edges in its sequence (e.g. $length(path_1) = 2$, $length(path_2) = 3$) we can also define a shortest path between two nodes. A shortest path between two nodes is any path between the two nodes whose length is smaller or equal to all other paths between the two nodes [1]. If there is no path between 2 nodes, consequently there is no shortest path, and as such the shortest path problem has no solution.

If we associate a real-numbered *weight* with each edge, we construct a *weighted* graph. We can do this by expanding a graph G to a ordered quadruple, with a function $w_G(e)$ which associates each edge with its weight as its fourth entry. If all weights of the graph are equal to one[2], then this graph would be no different than a *non-weighted* graph. The notion of a path in a weighted graph stays the same, while for the notion of a shortest path we need to expand the notion of length. In a weighted graph the length of the path can be defined as the sum of the weights of the edges in the path sequence [1].

2.3 Terminology

When we say node we are referring to what is more formally known as a vertex in Graph Theory, and an edge is the path/line that connects two nodes. When we are discussing weights we are referring to the cost associated with travelling along a given edge. The terms "visited" and "exploring" refer to when an algorithm is operating on a given node. A node that is referred to as explored is therefore one that the algorithm has already considered.

2.4 Algorithms

To fully understand the theory of our algorithms it is first important to understand what a path-finding algorithm is. An algorithm is any computational procedure that takes a value, or set of values, as input and produces some value, or set of values, as output. Therefore an algorithm is a sequence of steps that transform the input into the output [2]. Informally an algorithm is usually introduced when we face a problem that can be solved by doing the same certain steps one at a time.

Algorithms are not only relevant to the computer science topic, in fact we face different algorithms on daily basis. For example a recipe to make a cake is an algorithm itself. It is a sequence of steps that take an input (i.e. ingredients) and create an output (a cake). With this definition of an algorithm it will be easy to explain how a path-finding algorithm behaves. It is an algorithm that evaluates nodes on a map to produce a possible connection between point A and point B.

[2]Note that if all weights were equal positive values, the graph would be equal to a non-weighted one in all but lengths

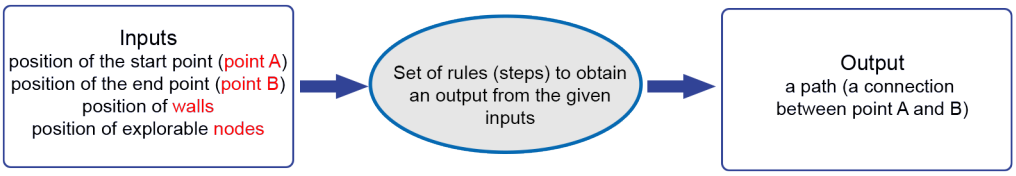


Figure 2: Path-finding algorithms diagram

An optimal path-finding algorithm must constantly make decisions about which node to explore next. If it explores nodes which cannot be on an optimal path, it is wasting effort. On the other hand we need to be careful not to ignore nodes that might be on an optimal path. Different path-finding algorithms use different methods and logic to explore nodes and in the upcoming sections we will focus on studying and comparing those methods.

2.5 Graph Data Structures

We used an adjacency list to store our graphs/maps in memory. This method relies on storing what nodes are adjacent to other nodes. This can be stored as such:

```
let adjacency_list = {
                      "A": ["C", "D"],
                      "B": ["C", "D"],
                      "C": ["A", "B", "D", "E"],
                      "D": ["A", "B", "C"],
                      "E": ["C"]
                      };
```

The adjacency list operates as a key value pair. Such that the key is a node and the value is an array of all the nodes adjacent to it. Using integer values for node names allows for this to be stored as a multi-dimensional array that implicitly lists the adjacent node numbers.

```
let adjacency_list = [[2, 3],
                      [2, 3],
                      [0, 1, 3, 4],
                      [0, 1, 2],
                      [2]];
```

We can use a similar data structure to store the weights to travel to each node. Using the same graph as before we get the corresponding weight list to our adjacency list:

```
let weight_list = [[e1, e2],
                   [e3, e4],
                   [e1, e3, e5, e6],
                   [e2, e4, e5],
                   [e6]];
```

The reason this is useful is because the index of the weight is the exact same as the index of the nodes it is connecting. For example on our graph "A"(0) is connected to "C"(2) and "D"(3). This is shown by:

```
adjacency_list[0] = [2, 3]
weight_list[0]    = [e1, e2]
```

So we know that:

```
adjacency_list[0][0]
weight_list[0][0]
```

Are referring to the same piece of the graph but have different information. This allows us to iterate through the adjacency list and access the correct edge as it is the same as what iteration we are on.

2.6 Graph Interpretation Data Structures

Now we will look at how our program looks at a 3x3 map and represents it. Graphically this is displayed:

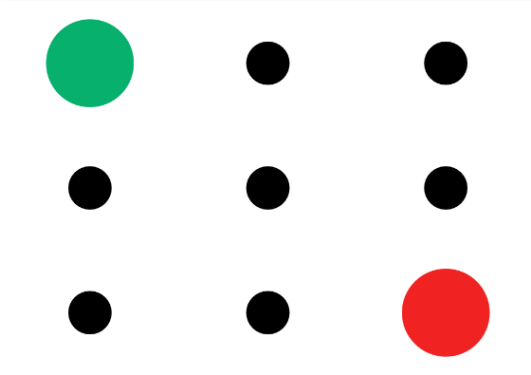


Figure 3: Simple 3x3 grid showing a start (green dot), an end (red dot), and possible places to move (black dots)

This is stored in memory as a multidimensional array:

```
map = [["S", 0,  0],
       [ 0,  0,  0],
       [ 0,  0, "E"]];
```

Here "S" represents the start, 0 is an open space that can be moved to, and "E" represents the end. This can be thought of as a grid (much like a chess board). The number 1, would represent a node that cannot be travelled to and as such it is a wall.

While this is useful we need a way to interact with our adjacency list so we generate a map with the node numbers (names):

```
interpreted_map = [[0, 1, 2],
                   [3, 4, 5],
                   [6, 7, 8]];
```

This is done by iterating over the map array and labelling every node that it is possible to move to (i.e. not a wall). This is useful because we can easily find what node we are looking at/for when iterating over the map as the node number is stored at the same index as the node itself.

If there is a wall this is always stored as -1 in the interpreted map. This is for error checking, and is useful as we know that the adjacency list (array) doesn't have an index for -1.

Iterating over the map array again (in the same order) then produces the adjacency list for each node in the array. This is done by looking at all the nodes in the map and then adding all the nodes directly adjacent to it to an array. This is also where the weights are calculated.

Another visualization of this is shown in figure 4 with weights of edges added it, and node numbers:

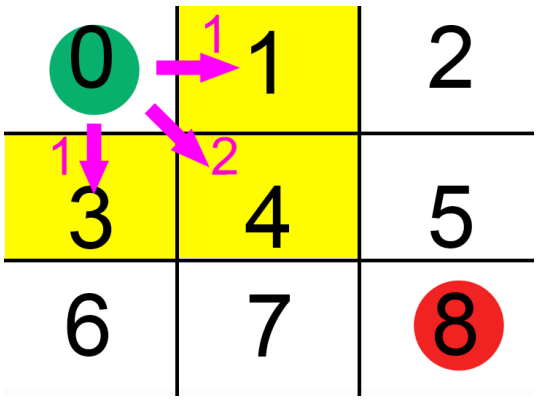


Figure 4: 3x3 map with node values and example of weights from the starting node to its neighbours (yellow)

This is an example of the same plain general 3x3 map structure (Figure 4), where now instead of showing the black dots we show the corresponding indexes. Yellow nodes are the nodes adjacent to the node 0 and the magenta colored arrows represent the weights of the yellow nodes.[3].

It is important to use the same data structure for all of the algorithms to make them comparable. Putting all the data structures we have so far discussed we generate a new

[3]Please note that although we hold only either a value 1 or 2 in our "weight" lists, in the algorithms we take a square root of those weights so 1 remains 1 but diagonals will be the square root of 2

data structure. This structure is an array of nodes, each node has properties for its index, adjacent nodes, the weight of each of the edges to these nodes, as well as the x, y coordinates within our map array. The example below is how we represent the above maps in memory:

```
0: node
    val: 0
    adj: Array[3]
        0: 1
        1: 3
        2: 4
    weights: Array[3]
        0: 1
        1: 1
        2: 2
    x: 0
    y: 0
1: node
    .
    .
    .
8: node
```

The first parameter "val" stands for a value and represents the index in the adjacency list. So the most left top node holds value 0 the node next to it holds value 1 etc. (Figure 4) Walls are not represented in the data structure so the algorithms do not consider them as nodes. The next parameter "adj" stands for adjacency and contains an array of nodes values adjacent to the node. Parameter "weights" has an array that is parallel to the "adj" array and contains the weights of the paths to the corresponding nodes, which in this case means that getting to the node 1 (Figure 4) from the node 0 has cost (weight) 1 (one step). Getting to the node 3 (Figure 4) has also weight 1 but getting to the node 4 has weight 2 ($\sqrt{2}$ when implemented to the algorithms), because we wanted to put bigger cost (weight) on a diagonal movement. Parameters "x" and "y" represent the xy coordinates of the node within the map.

2.7 Big O Notation

Big O notation is used in computer science and some branches of mathematics. If we have a function f which describes a certain aspect of an algorithm based on a number n, then the function g can be a big O notation of f if equation (3) is satisfied, where C is a finite constant and when n is substantially large. The function f could be execution time, or memory space, while n could be the length of an input string, or the number of elements of an input list. If equation (3) is satisfied then we say $f(n) = O(g(n))$, generally the function $g(n)$ should have the least possible value in the limiting case of n [3].

$$f(n) \leq Cg(n),\ \{n > n_0\} \text{ where } n_0 \text{ is some finite real number} \tag{3}$$

Here are some examples of big O notations, in table 1 you can see how functions of different big O notations have different growth.

- $O(1)$ - constant time, e.g. determining whether a number is even
- $O(logn)$ - logarithmic time, e.g. pushing a node into a binary heap
- $O(n)$ - linear time, e.g. finding an element in an unsorted array
- $O(nlogn)$ - asymptotic time, e.g. Theoretical complexity of Dijkstra's algorithm when applied on our sub-type of graph
- $O(n^2)$ - quadratic time, e.g. multiplying two n-digit numbers by hand
- $O(2^n)$ - exponential time, e.g. recursively calculating the n-th Fibonacci number
- $O(n!)$ - factorial time, e.g. generating all permutations of a string

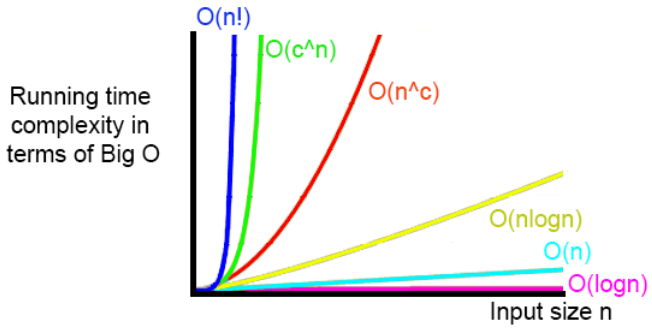


Figure 5: Different running time complexities over the input size n

$f(n)$	133	$25logn$	$55n+105$	$33nlogn$	$3n^2+55n$	2^n	$n!$
$n=1$	133	0	160	0	58	3	3
$n=10$	133	83	655	1096	850	2048	362880
$n=100$	133	166	5605	21925	35500	1.27×10^{30}	9.33×10^{157}
$g(n)$	1	$logn$	n	$nlogn$	n^2	2^n	$n!$
$O(g(n))$	$O(1)$	$O(logn)$	$O(n)$	$O(nlogn)$	$O(2^n)$	$O(2^n)$	$O(n!)$

Table 1: Examples of functions $f(n)$ and their corresponding functions $g(n)$ and $O(g(n))$

2.8 Comparison

To find out what path finding algorithm is performing the best we need to have a method of comparison. In this section we will explain what factors we chose to compare to see which algorithm performs the best. One very important factor is the length of the path found by the algorithm as we are looking for shortest path. This is done by measuring the length of the line that is created from the start to end point. These algorithms also need to be efficient so we need to measure the time it takes to find the path. This is measured in milliseconds and also in the number of times the algorithm is called. With this data we can compare the algorithms on maps of different sizes and see how they perform.

3 Depth First Search Algorithm (DFS)

3.1 Introduction to Depth First Search

Depth First Search (DFS) is an algorithm that operates on graph or tree data structures, it operates in a "natural" way to visit every node in a graph. As such, it is the simplest algorithm we will be using to search our graphs [4].

The name (Depth First Search) comes from the way that it operates on graphs. This is because it always tries to move "down" the graph, this is achieved by moving to the first child node of the node it is on (and then it's first child node) [4].

3.2 Data Structures Used in DFS

Depth First Search requires the use of a stack [4]. The stack can be thought of as a physical stack or pile of something (for example plates or cards), however, only the very top of the stack is accessible. Whether that is for adding an an element or removing one.

This basic implementation of a stack is considered LIFO (Last-in First-out) as a way to represent how it operates [4]. There are 3 key operations that can be performed on the stack:

1. Looking at the values in the stack (peek/top)
2. Adding a value to the top of the stack (push)
3. Removing a value from the top of the stack (pop)

However we only needed to use the push and pop operations in our implementation of DFS.

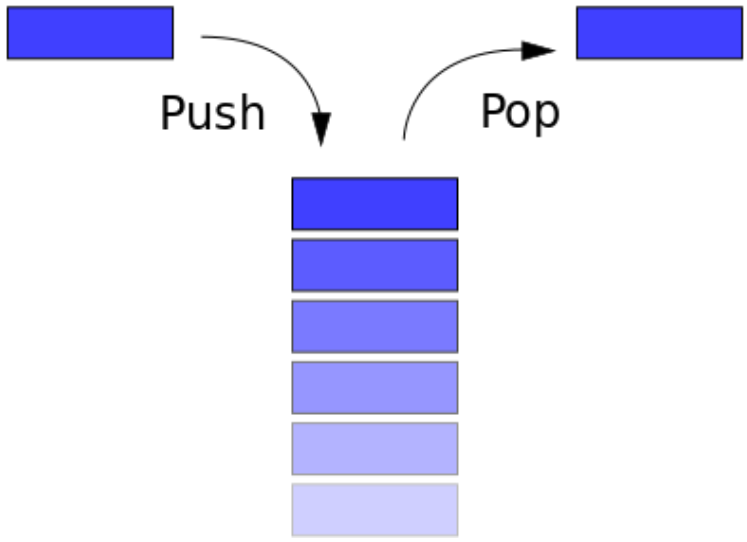


Figure 6: This is a representation of how the stack operates [5]

Depth First Search can be implemented in either a recursive or non-recursive method. While the non-recursive method uses an explicit definition of a stack, the recursive method uses the implicit call stack of the programming language it is implemented in.

In our case we used an explicit stack, created by defining an array in JavaScript:

```
let stack = [];
stack.push(element); // Adds an element to the top of the stack
let value = stack.pop(); // Removes the element at the top of the stack
```

As JavaScript does not natively support the use of a stack, we needed to define one ourselves. Luckily, JavaScript does have built in methods that act on arrays, namely push(), and pop() which work exactly as described above, making it extremely simple to implement a stack in JavaScript with no gain in complexity.

3.3 DFS Algorithm

Depth First Search traverses a graph by arriving on a node, then marking it as explored, then adding (pushing) all its unvisited neighboring nodes to the stack. It then pops the top node off the stack and then traverses to it. It then repeats this procedure on that node. If there are no nodes added to the stack it just pops the next value off the stack. This is equivalent to backtracking to a node that still has unvisited neighbors and traversing to it's unexplored neighbors.

This algorithm only terminates after the stack is empty (i.e. there is nothing left the algorithm can explore) or the node it has just arrived on is the node it was searching for.

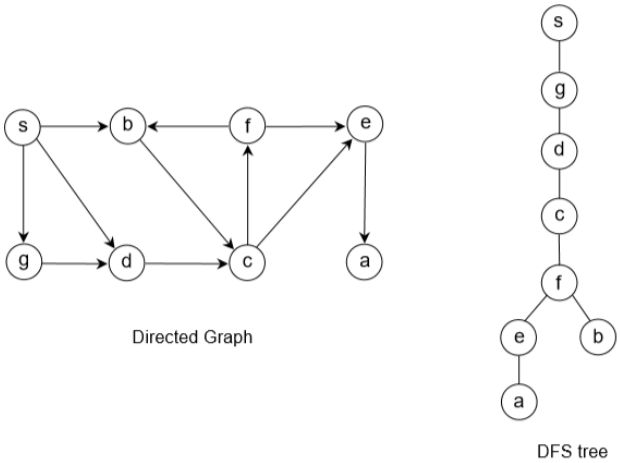


Figure 7: Example of DFS searching a graph [6]

To explain this we will run through the first few iterations, showing the stack, and explaining what is happening in Figure 7. We will take our start node as "s" and terminate after exploring the whole graph.

To start we push "s" to the stack. Our algorithm marks "s" as visited and then pushes its neighbors (in lexicographical order) to the stack:

```
stack = ["b", "d", "g"]
```

Then the last added element is popped from the stack and the algorithm operates on it. In this case it is "g". "g" only has one neighbor "d" which is already in the stack, so it is not added again. Our stack ends up looking like this:

```
stack = ["b", "d"]
```

We then move to the next element at the top of the stack "d". Its neighbor "c" is added to the stack:

```
stack = ["b", "c"]
```

The next step moves to "c" and adds "f" and "e"

```
stack = ["b", "f", "e"]
```

The next step moves to "e" and adds "a":

```
stack = ["b", "f", "a"]
```

The next step moves to "a" and since there are no neighbors nothing is added:

```
stack = ["b", "f"]
```

The next step moves to "f" and since "e" has already been explored it is not added to the stack and since "b" is already on the stack it is not added again:

```
stack = ["b"]
```

The next step moves to "b" and since nothing else can be added the stack is now empty:

```
stack = []
```

When the stack is empty we know the entire (available) graph has been explored and the algorithm is done searching.

DFS does not always find the shortest path to a selected node but will find all connected nodes [6]. The efficiency of the algorithm also highly depends on the placement of the start and end nodes as well as the order neighbors are added to the stack.

3.4 DFS Pseudocode

```
DFS(G, s, e) (G is the adjacency list, s is the start node, e is the end node)
    // Initialization Step
    stack := []
```

```
    for each node n in G, visited[n] := false;
    push s to stack

    // Search algorithm
    while stack is not empty do
        u := pop stack

        if u is e then return end if

        for each node n in G[u] do
            if not visited[n] then
                push n to stack
                visited[n] := true
            end if
        end for
    end while
END DFS()
```

3.5 DFS Distance

DFS is very clearly not strictly a "pathfinding" algorithm, as it rather focuses on traversing an entire graph. Because of this, we interpreted DFS's "recommended" path to be the order it traverses the nodes in (whether a path to the end node is found or not).

One way of representing this is by drawing a line that interconnects all of the points in the order they are traversed in.

The issue with this is that there are times where the line does not travel from one adjacent node to the next but moves in between several nodes to travel towards the next node that is explored by DFS. This is the case when there are no new nodes that are added to the stack (from the node it is on). So the algorithm "backtracks" to another node where there are still adjacent "unexplored" nodes.

The solution to this is to make sure that every line connects the parent node to its children. The parent node is the node the algorithm added the adjacent, unseen child nodes to the stack from. This results in a "spider's web" looking path, that doesn't seem particularly useful for anything, but it does visualize the algorithm accurately (This is shown in Figure 8).

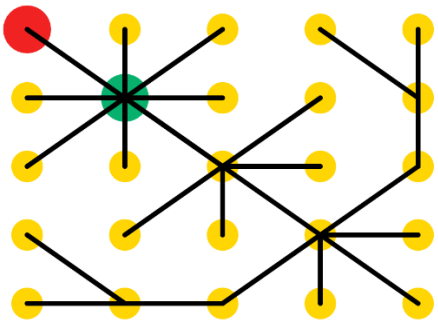


Figure 8: DFS's actual path shown by connecting the parent and children nodes (DFS tree)

Because this DFS tree representation did not seem like a particularly useful representation for a pathfinder, we decided to program the line so that if the next node in the path was not adjacent to the previous node it, the line would split, from the parent node to the 2 children nodes. This means that the path more closely represents the order that nodes are explored in but also splits the path by "backtracking" if it doesn't make sense to continue this line (this is shown in Figure 9). We used this representation to calculate the "distance" DFS travels. Realistically DFS wouldn't be used to find a path, and if it was it would no longer be a "pure" DFS algorithm and have some extra algorithms working with it, so we decided to just track the distance it travels in total because it would be a more interesting result and it visualized DFS's process best.

One way we could change the algorithm is by including a parent/child relationship between the nodes, where the parent is the node that added the child node to the stack. This could then be iterated over backwards to find the path (this is the exact same approach BFS takes to finding the path, and is covered in its respective section 4.3). The resulting path would be similar to Figure 8 except without any branching and just show the path between the start and end nodes, while this would be a better estimate for a path we decided to use a different method as we wanted to differentiate between DFS and BFS more, especially since this way of designating the path would still be more inaccurate than BFS's method.

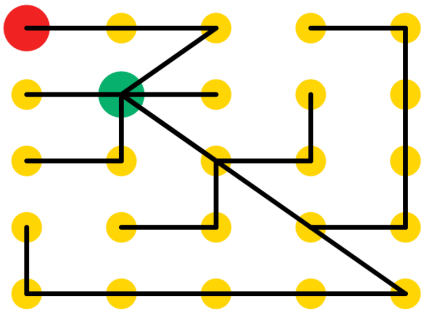


Figure 9: How we represented DFS' path

3.6 Analysis of DFS

The DFS algorithm depends heavily of the position of the start and end node. This can be seen clearly in Figure 10 where the left shows the worst case for DFS. The algorithm will favor searching nodes that are to the right and down and this can be seen in the image on the right as it finds the shortest possible path. Therefore, DFS can be really unreliable, as there are cases where it performs the fastest but that happens only on specific kinds of maps.

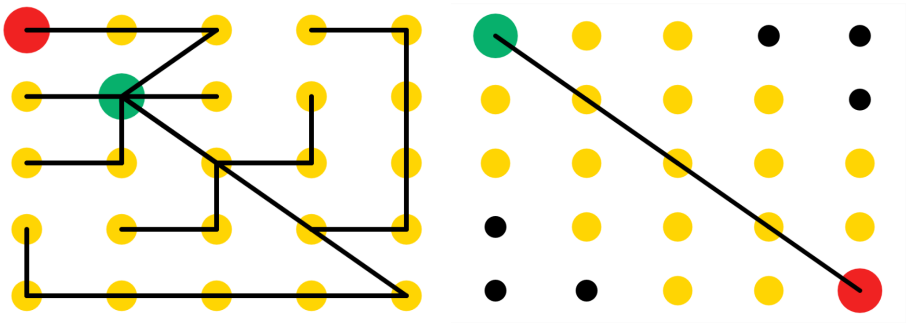


Figure 10: Showing how start/end position effects DFS' path

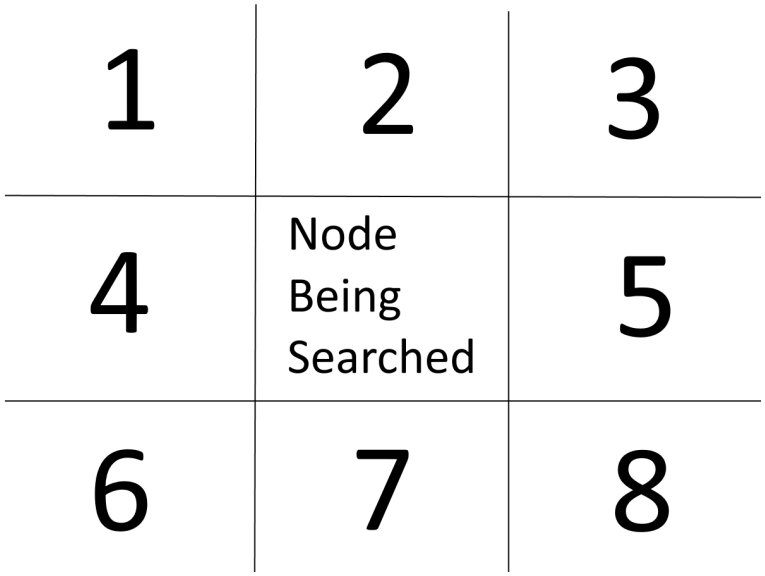


Figure 11: Showing the order adjacent nodes are added to the adjacency list

The reason for this behaviour is due to the order that the node's are placed in, in the adjacency list. This has to do with our algorithm for converting a displayable map to an adjacency list. Figure 11 shows the order that each node adjacent to the center node is added to the adjacency list. With 1 being added first and 8 being added last. This results in a list that looks like this:

```
list = [1, 2, 3, 4, 5, 6, 7, 8];
```

This coincidentally (roughly) matches the node numbers, this is to say lower value nodes are added to the adjacency list first. When the DFS algorithm looks at the adjacency list it adds all the neighbors (that haven't been added) to the stack in the same order. Hence our stack looks like this:

```
stack = [1, 2, 3, 4, 5, 6, 7, 8];
```

However due to the stack's LIFO order, these numbers are removed from the stack in reverse order. Meaning that our DFS algorithm "prefers" to move to the highest node number that is available out of the nodes it is adjacent to. The consequence of this behaviour is that DFS travels down and to the right preferentially (then down, right, left then finally up).

This was something we only noticed after actually programming the algorithm and experimenting with it. In practise, this behaviour could likely be programmed to act in whatever order the programmer wants, however, this would have little to no effect on the overall efficiency.

3.7 Big O notation for DFS

The worst case for DFS is that it searches the entire graph before finding the end node, as shown in Figure 10 on the left. Because we used an adjacency list, the lookup time for any adjacent nodes is linear: $O(1)$[7]. Therefore the time complexity is at least $O(V)$ where V is the number of nodes in the graph as we would have to traverse through each node[7].

Furthermore, the worst case for doing this would be if the algorithm traverses along every edge between the nodes. Therefore, the complexity of going through all E edges between the nodes is $O(2E) = O(E)$, because we would traverse on any edge between 2 nodes twice. The sum of these two complexities and the complexity of the algorithm is therefore $O(V) + O(E) = O(V + E)$[7]. Because our implementation of the graph is a 2 dimensional grid we can guarantee that the maximum number of edges leaving any one node is 8, and therefore, the maximum possible amount of edges is $8V$. So our complexity can be rewritten as $O(V + 8V) = O(V)$.

4 Breadth First Search Algorithm (BFS)

4.1 Introduction to Breadth First Search

Breadth First Search (BFS) is a simple search algorithm that is very similar to DFS (explained in the chapter above). It works by searching through all the adjacent nodes and then searching through all the nodes adjacent to those (this makes it look like the algorithm's search area is expanding around the start node until it reaches the end node). If the graph were in a tree structure, this would look like searching level by level (shown in Figure: 14). Searching all the child nodes in a level before moving on to the next level.

This algorithm gets its name (Breadth First Search) from how it searches the graphs. It works by searching the "width" of the graph before going down to the next layer of nodes. This is what makes it different from DFS as explained in Section 3 because it stays closer to the start position before searching out. Still the algorithm will exhibit the same worst case complexity as DFS.

4.2 Data Structures Used in BFS

Bredth First Search requires the use of a queue. The queue can be thought of as a queue of people waiting in line. This is because the first person in the line is the first to do whatever, and the people behind follow in order.

The basic implementation of a queue is considered FIFO (First-in First-out) as a way to represent how it operates [4]. There are 3 key operations that can be performed on the queue:

1. Looking at the front value in the queue (peek/front)
2. Adding a value to the back/top of the queue (push/enqueue)
3. Removing a value from the front/bottom of the queue (shift/dequeue)

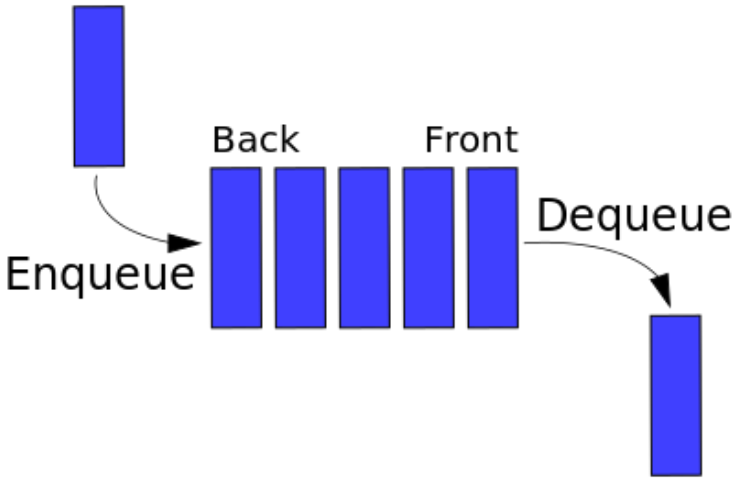


Figure 12: This is a representation of how the queue operates [8]

However we only need to use the shift and pop operations in our implementation of BFS.

```
let queue = [];
queue.push(element); // Adds an element to the back of the queue (Enqueue)
let value = queue.shift(); // Removes the element at the front of the queue (Dequeue)
```

Much like with the stack, JavaScript does not support the use of an actual queue but it can be simulated with no gain to complexity with the above code. Which are built in functions that operate on arrays.

4.3 BFS Algorithm

The Breadth First Search algorithm will first search all the neighbour nodes and then work through the next set of neighbours [4]. This is done by using a queue and adding the neighbours of a node to the back of this queue (enqueue), and then repeat this on each node in the queue (dequeue). The BFS algorithm will work its way through the queue adding new neighbours to the end of the queue and searching the next in the front of the queue.

The BFS algorithm also keeps track of the parent nodes in order to find a path. A parent node is the first (and only) node that adds its adjacent (children) nodes to the queue; If another node were to later be explored and have adjacent nodes that have already been added to the queue, they are neither added to the queue nor is it marked as the parent of those adjacent nodes. Once the algorithm finds the end node it will calculate the path it took by starting with the end node: adding it to the path: looking at what its parent is, then adding the parent to the path, then looking at what the parent's parent is and so on.

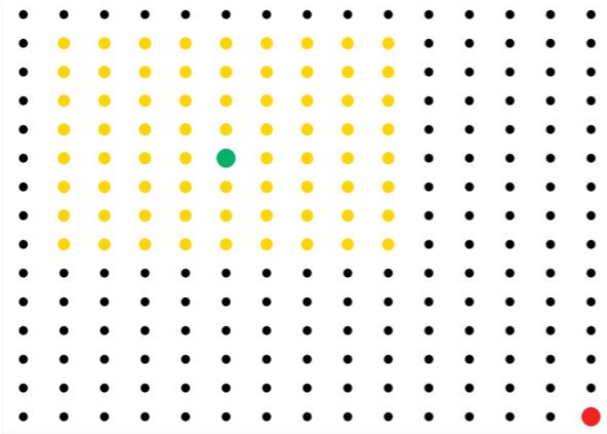


Figure 13: How BFS searches the graph

This is more simply explained by Figure 14, where there are less cases to consider. In Figure 14, white coloring means that the node has not been interacted with yet, grey means it in in the queue, and black means it has been considered by the algorithm. Each sub image is a step in the BFS algorithm and underneath each sub image next to the label the queue is shown.

BFS works great on maps where all lengths are equal (unweighted) and will find the shortest path in that case [4]. For our graph diagonals are longer than moving directly and BFS does not take this into account. Its behaviour can be seen in Figure 13: it creates a box that increases in size around the start node (these are the nodes that have been added to the queue). Other algorithms have been made that build on BFS such as Dijkstra that takes weights into account when looking for the shortest path.

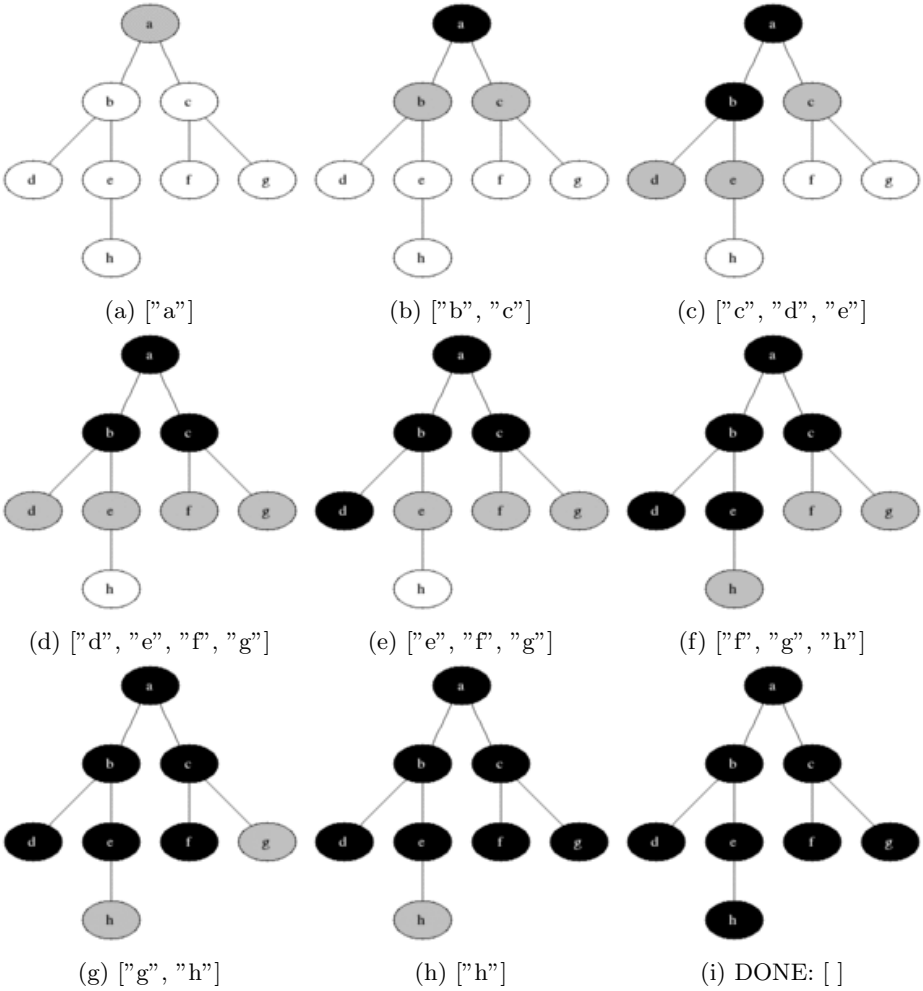


Figure 14: BFS traversing a tree, with the queue shown at each step.
It is a modified version of [9]

In Figure 14(a) the algorithm is initialized with "a" as the start node, so it is automatically added to the queue. Next in Figure 14(b) "a" is dequeued from the queue so the algorithm can be performed on it. Hence, the nodes "b" and "c" are added to the queue and the parent of "b" and "c" is labelled "a" since "a" was the first and only node that added "b" and "c" to the queue. Next in (c) "b" is dequeued and its adjacent nodes ("a", "d", "e") that have not already been considered ("d", "e") are enqueued. Leaving a queue of ["c", "d", "e"]. The process then continues dequeing the front of the queue and enqueueing new nodes to the back of the queue.

In Figure 14(i) the parent of "b" and "c" is "a", "d" and "e" is "b", "f" and "g" is "c", and the parent of "h" is "e". For example, if we were searching how to get from "a" to "h", we would then look at "h"'s parent "e", "e"'s parent "b", and "b"'s parent "a". We would then terminate and return the path ["h", "e", "b", "a"]

4.4 BFS Pseudocode

```
BFS(G, s, e)
    queue := []
    for each node n in G, visited[n] := false
    for each node n in G, parent[n] := -1
    enqueue s to queue

    while queue is not empty do
        u := dequeue queue

        // If we are done return the "optimal" path
        if u is e then return PATH(s, e, parent, []) end if

        for each node n in G[u] do
            if not visited[n] then
                enqueue n to queue
                visited[n] := true
                parent[n] = u
            end if
        end for
    end while
end BFS()

// Recursive function that returns the path (an array of every node in the path in order)
PATH(start, cur, parent, path)
(start is the start node, cur is a node,
parent is the array of parent nodes, path is the return variable)
    path.push(cur) // Add to return variable
    if cur is not start then
        PATH(start, parent[cur], parent, path) // Traverse
    else
        return path
    end if
end PATH()
```

4.5 Big O for BFS

To calculate big O for BFS we must consider the lookup time for the adjacency list which is $O(1)$ [7]. In the worst case we will have to search each node, this gives us a minimum complexity of $O(V)$ where V is the number of nodes. We also know that in the worst case we would also have to traverse every edge twice, once for each node that is connected to it. Giving us a complexity of $O(2E) = O(E)$, where E is the number of edges. Adding these 2 operations and their complexities we get $O(V) + O(E) = O(V + E)$, because we need to visit every node and traverse every edge [7]. Since we can garantee that the maximum

number of edges is $8V$ (in the worst case every node can be connected to 8 other nodes). Taking this into account we get $O(V + 8V) = O(V)$ as our final complexity for BFS. This makes intuitive sense because the BFS algorithm is essentially DFS but with a different data structure that has the same lookup time.

4.6 Analysis of BFS

The BFS algorithm as expected does not work well with diagonals. In Figure 15 BFS can be seen of the left while Dijkstra is on the right. Dijkstra finds the shortest path because it takes diagonal weight into account. BFS does not do this and can be seen taking unnecessary diagonals when going straight would be shorter.

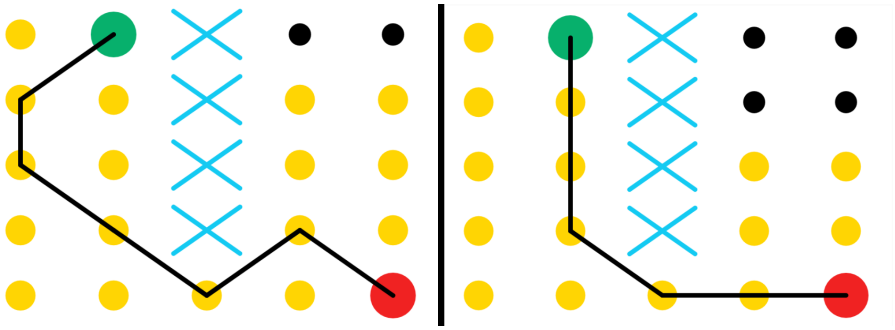


Figure 15: Example of BFS (left) and Dijkstra (right)

In Figure 15 using euclidean distances, where the distance between every node that is directly, up, down, left, or right is 1, and the distance for diagonals is $\sqrt{2}$. In this case the distance BFS decides is 8.071, and Dijkstra's distance is 6.414. If this were an unweighted graph or the all the distances had an equal value, in this case 1, BFS's route would have a distance of 6 and Dijkstra's route would have a distance of 6.

In the case of an unweighted graph, BFS is just as good at finding the shortest path as Dijkstra, and therefore always finds the shortest path. This is because when all the weights are equal, Dijkstra's priority queue degenerates into a FIFO queue.

5 Dijkstra Algorithm

5.1 Introduction to Dijkstra

Created in 1959 by Edsger Wybe Dikstra [10], this algorithm finds the length of the shortest path from some node to all nodes in a weighted graph with non-negative weights. The algorithm can be easily modified to allow to stop executing after it found the length of the shortest path to a specific node. The algorithm stops executing after it popped the terminus node from the heap not when it is added to the heap. When we are referring to Dijkstra we are referring to this modified algorithm. The time complexity of the original Dijkstra's algorithm depends on the number of edges and the number of nodes as seen in equation (4), as described in *Introduction to Algorithms* [2].

$$O((V+E)logV), \text{ where } V \text{ is the number of nodes and } E \text{ is the number of edges} \quad (4)$$

In the case of our graph represented as a grid the complexity can be rewritten as equation (5), because we know that the number of edges is at most $8n$.

$$O(nlog(n)), \text{where } n \text{ is the number of nodes} \quad (5)$$

5.2 Data Structures used in our implementation of Dijkstra

To achieve the theoretical complexity of the Dijkstra's algorithm ($O(nlogn)$), we have to use a smart data structure to find the least distant node in the open set, as described in *Introduction to Algorithms* [2]. We will attempt to achieve this by using a priority queue based on a binary heap.

A binary heap is a type of binary tree where all rows with the exception of the last row should be filled. All parent nodes of a heap should also satisfy some comparative function when compared with their children. In less general terms there are two main types of heaps: max-heaps, for which all parent nodes must be greater than or equal to their children; and min-heaps, for which all parent nodes must be smaller than or equal to their children, as described in *Introduction to Algorithms* [2]. We will mostly be concerned by min-heaps like the one seen in Figure 16.

There are two main functions governing the behaviour of heaps: a function which adds a node to the binary heap, which we will be calling the push method; and a function which removes the root node off the binary heap. When we execute both of these functions we must make sure that the rules of the heap are still satisfied, as described in *Introduction to Algorithms* [2].

An implementation of the push method for a min-heap is given in Algorithm 1. We will show a step by step example of pushing v (node (1)) into the heap Q (Figure 16). First we add the node v (node (1)) to the last position in the heap, as seen in Figure 17a. p is the parent node of v (node (2)), because p is greater than v we swap them, as seen in Figure

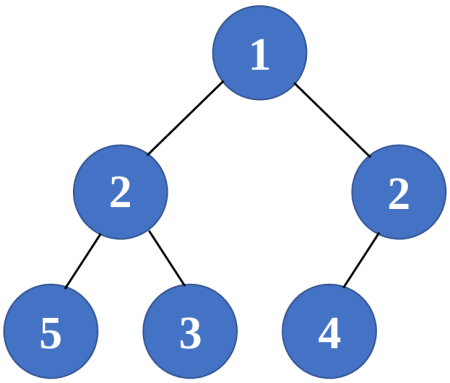


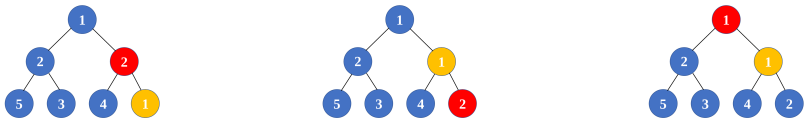
Figure 16: Example of binary min-heap. This binary min-heap was procured by inserting 6 numbers in the following order: $2, 5, 4, 3, 1, 2$

17b. p is the parent node of v (node (1)), it is not greater than v so the algorithm stops executing.

Algorithm 1 min-heap push method

```
Q ← the binary min-heap
v ← the to be added node
v is added to the last position of Q
do
    if v is the top most node in Q then
        break
    end if
    p ← the parent node of v
    swap v and p in Q
while p > v
return
```

An implementation of the pop method for a min-heap is given in Algorithm 2. We will show a step by step example of popping Q (Figure 17c). v is the last node in the heap node (2), and TOP is the root node (node (1)), as seen in Figure 18a. We swap v and TOP as seen in Figure 18b. Then we remove TOP as seen in Figure 18c. $leftChild$ is node (2), and $rightChild$ is node (1), as seen in Figure 18d. Some of v's children are smaller or equal than v, and $rightChild$ is smaller than $leftChild$ so we swapped $rightChild$ and v as seen in Figure 18e. Finally v has only one child and this child is greater than v so no swapping takes place as seen in Figure 18f.

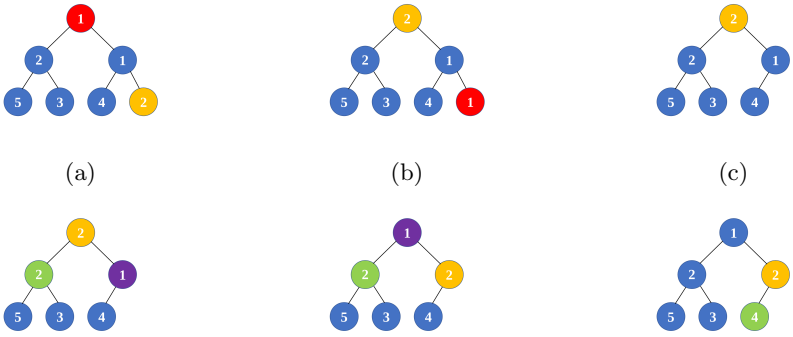


(a) (b) (c)

Figure 17: Here are shown 3 steps of pushing node (1) into the heap (Figure 16)
.

Algorithm 2 min-heap pop method

$Q \leftarrow$ the binary min-heap
$v \leftarrow$ the last node in Q
$TOP \leftarrow$ the root node of Q
swap v and TOP
remove TOP from Q
do
  if v has no children **then**
    break
  end if
  $leftChild \leftarrow$ the left child of v
  $rightChild \leftarrow$ the right child of v
  if $leftChild \leq rightChild$ **then**
    swap v and $leftChild$
  else
    swap v and $rightChild$
  end if
while $leftChild \leq v$ OR $rightChild \leq v$
return TOP



(a) (b) (c)

(d) (e) (f)

Figure 18: Here are shown 6 steps of removing the root node from a heap (Figure 16)

When we will be adding nodes to our min-heap in our implementation of Dijkstra's algorithm, we will be representing them as ordered pairs, where the first entry is the id of the added node, and the second entry is their weight. We wish to order the min-heap by weight (the second entry). A binary heap can conveniently be stored in an array. If the root node is at index 1, we can easily calculate the index of the parent, left child and right child of

any node if we know its index. $parentIndex(i) = floor(\frac{i}{2})$, $leftChildIndex(i) = 2i$, and $rightChildIndex(i) = 2i + 1$. Also whenever we add or remove elements to the heap we increment a size variable, so we know that whenever adding a node to the last position this is at index $size + 1$, as described in *Introduction to Algorithms* [2].

5.3 Dijkstra - the modified Dijkstra's algorithm

Dijkstra as implemented by us is described in Algorithm 3

Algorithm 3 Our implementation of Dijkstra's algorithm

```
1:  G ← the graph
2:  Q ← the binary min-heap
3:  A ← origin node
4:  Ω ← terminus node
5:  visited[for each v in G] ← false
6:  distance[for each v in G] ← inf
7:  parent[for each v in G] ← undefined
8:  push (A, 0) into Q
9:  distance[A] ← 0
10: while Q is not empty do
11:     (X, L) ← pop Q
12:     while visited[X] do
13:         if Q is empty then
14:             return No Solution!
15:         end if
16:         (X, L) ← pop Q
17:     end while
18:     visited[X] ← true
19:     if X = Ω then
20:         Y ← parent[Ω]
21:         path ← empty string
22:         path ← concatenate(Y, path)
23:         while Y ≠ A do
24:             Y ← parent[Y]
25:             path ← concatenate(Y, path)
26:         end while
27:         return path, distance[Ω]
28:     end if
29:     for each v adjacent to X do
30:         if distance[v] ≤ L + weight(X, v) then
31:             continue
32:         end if
33:         parent[v] ← X
34:         distance[v] ← L + weight(X, v)
35:         push (v, L + weight(X, v)) into Q
36:     end for
37: end while
```

5.4 Analysis of Dijkstra

Figure 19 shows the execution of Algorithm 3 in 11 steps.

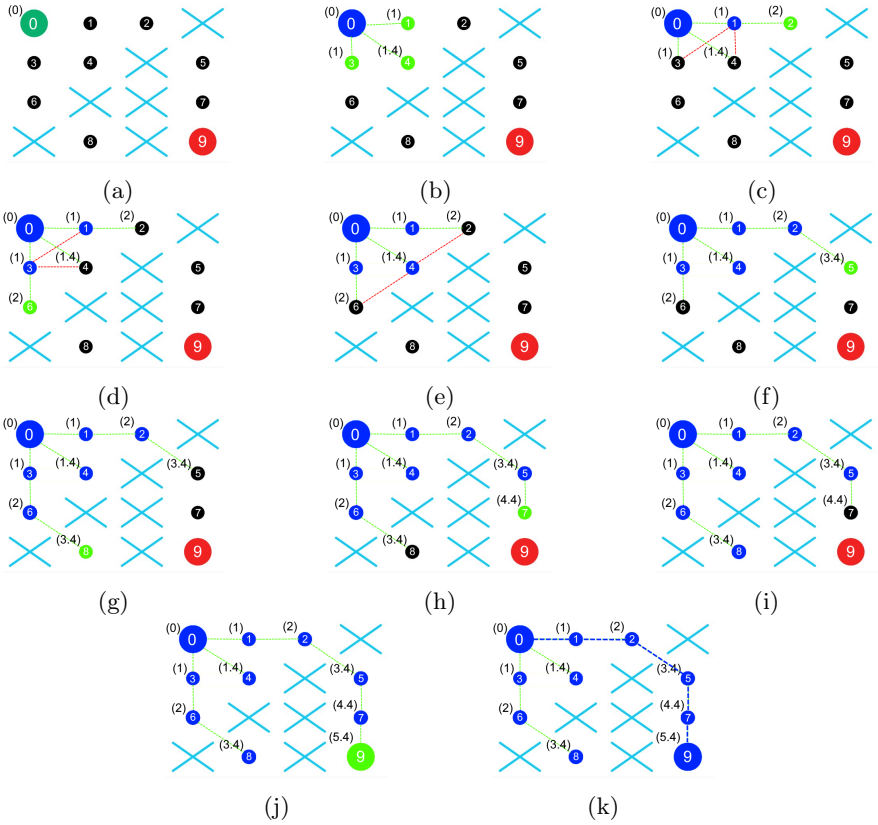


Figure 19: In these figures steps of our implementation of Dijkstra's algorithm are shown. v_0 is the origin node, and v_9 is the terminus node. For each step all nodes added to the heap are coloured green, whereas all *visited* nodes are coloured blue, a green line denotes the connection between a child and its parent, whereas a red line denotes a longer path which is discarded because a shorter path. The current *distance* to 1 decimal place is shown in parentheses to the top-left of each node. On the last figure the shortest path is denoted by a blue line.

6 The A* Algorithm

6.1 Introduction to A*

A* algorithm originates from the Dijkstra algorithm but improves the efficiency of it by adding an estimated distance for every node reached to the target node (end node) in order to skip the nodes that seems to deviate from the shortest path [11]. Considering the additional calculations have a time complexity of $O(1)$ A* algorithm would have the same worst case time complexity as Dijkstra (eq. 5).

6.2 A* Algorithm

A mathematical equation used to define the A* algorithm is equation (6) [11]:

$$F(n) = G(n) + H(n) \tag{6}$$

In equation (6) n represents the ID (index) for the selected node being checked in the grid. F cost is defined to be the sum of G cost and H cost, where G cost is the number of steps that needed to be done to get from the starting node to the current node n and H cost is the estimated distance from the node n to the end point. A* algorithm implies the checking of each n node from the grid (starting off with the neighbour cells of the start point) and picking the neighbour cells that have the smallest $F(n)$ value to continue with.

The estimation function $H(n)$ is selected in order to fit the graph and problem to be solved. The Euclidean distance (eq. 7) is one of the common functions selected for $H(n)$

$$H(n) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{7}$$

The distance to the end node $H(n)$ calculated using the Euclidean distance between two nodes (7) where (x_1, y_1) are the current point coordinates and (x_2, y_2) represent the end point coordinates that we want to reach [11].

In equation (7) $H(n)$ stands for heuristic. The heuristic is what makes A* the fastest performing path-finding algorithm as it uses the heuristic to "predict" which nodes should be explored next based on their distance to the end point. As mentioned in the section 6.1 A* originates from Dijkstra algorithm, the principle of those algorithms is therefore the same, but we can say that the heuristic in Dijkstra is set to 0, whereas in A* the heuristic plays a huge role and that is why A* performs much faster which will be more proven in the section 7.

Heuristics are important as they often lead quickly to solutions that we would otherwise reach much more expensively by trying to explore all the nodes in the map. Mainly because it is an approach that does not guarantee the right answer but speeds up the whole process by finding a satisfactory path since finding the optimal solution (the shortest path) might

be often impractical. Furthermore we can say that a heuristic can work as a shortcut that makes it easier for the algorithm to make decisions on what nodes to explore [12].

Theoretically since A* uses heuristic it does not always guarantee to find the shortest path (like Dijkstra does). However in our test runs that will be more discussed in the section 7 we were not able to find such a map that would make A* deviate from the shortest path. This might be because we were doing our test runs on rather simple maps (max 50x50) with random obstacles which was not enough for the algorithm to prioritise the time over the "correctness". In the section 6.6 we will try different heuristic methods as well as trying to put bigger impact on the heuristic from which will be visible that the algorithm does not find the shortest path but finds a path faster by not exploring certain nodes.

Even though we decided to use the Euclidean distance as our heuristic we will try other methods as well to see whether the output of the algorithm changes.

$$H(n) = |x_1 - x_2| + |y_1 - y_2| \tag{8}$$

Equation 8 is called the Manhattan distance and is defined to be the distance between two points being the node (x_1, y_1) and the endpoint (x_2, y_2) measured along axes at right angles.

6.3 Data Structures used in A*

Our implementation also uses a binary min-heap, as seen in Section 5.2, but we input ordered triples, with the first entry begin the id of the node, the second entry the g cost, and the third entry the h cost. The elements are sorted by the f cost.

6.4 A* pseudocode

```
function A*(start,goal)
  closedset := the empty set
  openset := {start}
  came_from := the empty map

  g_score[start] := 0
  f_score[start] := g_score[start] + heuristic(start, goal)

  while openset is not empty
      current := the node in openset having the lowest f_score[] value
      if current = goal
          return pathFind(came_from, goal)

      remove current from openset
      add current to closedset
      for each neighbor in neighbor_nodes(current)
```

```
            if neighbor in closedset
                continue
            tentative_g_score := g_score[current] + dist_between(current,neighbor)

            if neighbor not in openset or tentative_g_score < g_score[neighbor]
                came_from[neighbor] := current
                g_score[neighbor] := tentative_g_score
                f_score[neighbor] := g_score[neighbor] + heuristic(neighbor, goal)
                if neighbor not in openset
                    add neighbor to openset

    return failure
```

6.5 Big O notation for A*

The worst case scenario for A* is if it needs to traverse the entire graph. Because of the use of a min-heap priority queue the lookup time for any adjacent nodes is $O(logV)$ where V is the number of nodes. Similar to both DFS, BFS, and Dijkstra the worst case is when the entire graph is explored (All V nodes through all E edges), and the complexity of this is $O(V + E)$. Because after traversing through any edge to any vertex the next node is dequeued from the priority queue, we multiply these complexities. Leaving us with $O(V + E) \cdot O(logV) = O((V + E) \cdot logV)$. Next the heuristic is also calculated for every node that is moved to, the big O for a square root (necessary for the euclidean distance). Because not much information is given about how JavaScript calculates square roots we will call its complexity $O(S(d))$ where S is the complexity of the square root for a number that is d, digits long. This leaves the complexity $O((V + E)(logV)(S(d)))$. Because the maximum number of edges would be 8V we can further assume the complexity to be $O(V(logV)S(d))$.

6.6 Analysis of A*

As mentioned earlier A* performs the fastest because of its heavy usage of heuristic. First it starts by exploring all the neighbours of the starting point and chooses the one with the lowest F cost. (Figure 20)

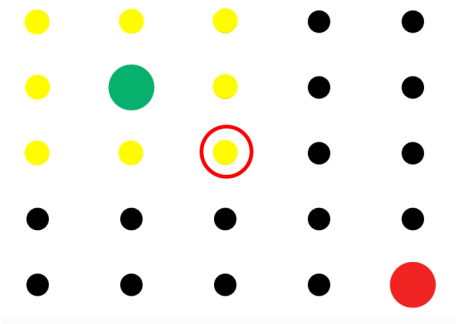


Figure 20: Step 1: Exploring neighbours of the starting point (green dot)

In the next step the algorithms explores the neighbours of the node previously chosen (the node in a red circle). In a case where diagonals have weights 1 we also showed G cost and H cost of the two nodes (Figure 21). As you can see those two nodes share some neighbours, it is therefore important not to overwrite the G cost with a bigger value then was it original and update the G cost with a smaller value when we found a cheaper way of getting there.

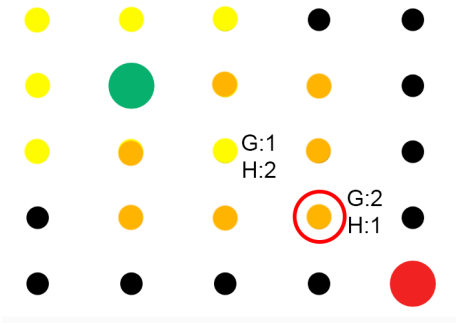


Figure 21: Step 2: Exploring neighbours (orange dots) of the node that was chosen next

Finally in Figure 22 we again explored the neighbours of the previously chosen node (the one with the lowest F cost) and we explored its neighbours, since one of those neighbours was the end point itself, the algorithms finished and output the path of the nodes that have been once previously chosen.

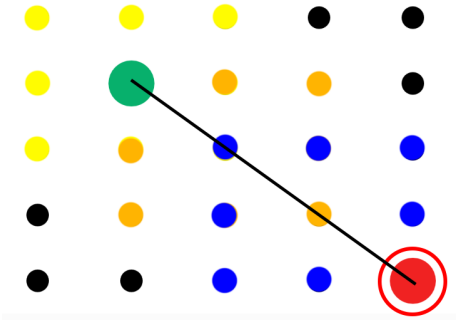


Figure 22: Step 3: Exploring neighbours (blue dots) of the node that was chosen next and finding the shortest path (black line)

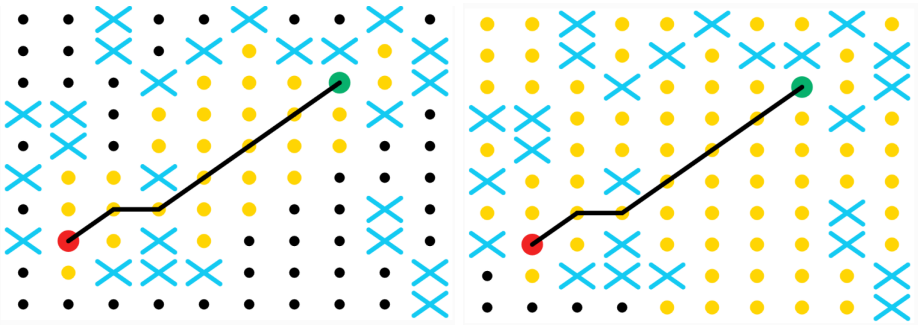


Figure 23: Example of A* (left) and Dijkstra (right)

In Figure 23 we can clearly see the "magic" of heuristic in action. As you can see Dijkstra had to explore all the nodes except for 5, whereas A* could skip the nodes that were not relevant and could not been on the shortest path. In this specific case Dijkstra needed to be called 63 times to find the shortest path and A* 11 times.

There are plenty of different heuristic to choose from. For the sake of this project we are going to look at the Manhattan distance (eq. 8) and the Euclidean distance (eq. 7). The Manhattan distance is usually used on a square grid with 4 allowed directions or on a hexagon grid with 6 directions. The Euclidean distance on the other hand is advised to be used on a square grid that allows any direction of movement.

Since Euclidean distance is shorter than Manhattan distance A* will take longer to run but it will be more likely to find the shortest path.

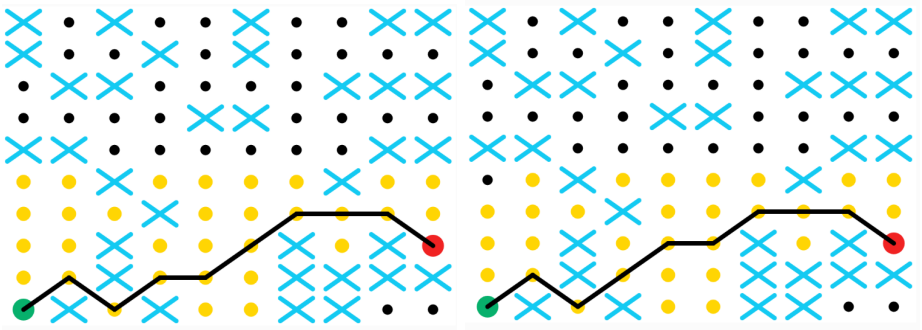


Figure 24: Example of A* with Euclidean distance (left) and Manhattan distance (right)

In Figure 24 we can actually see an example where Manhattan heuristic performed better, since both heuristics were able to find the shortest path (11.48 units) but the run with Manhattan distance needed to be called 19 times and took 667 ms and the one with Euclidean distance 22 times and took 817 ms.

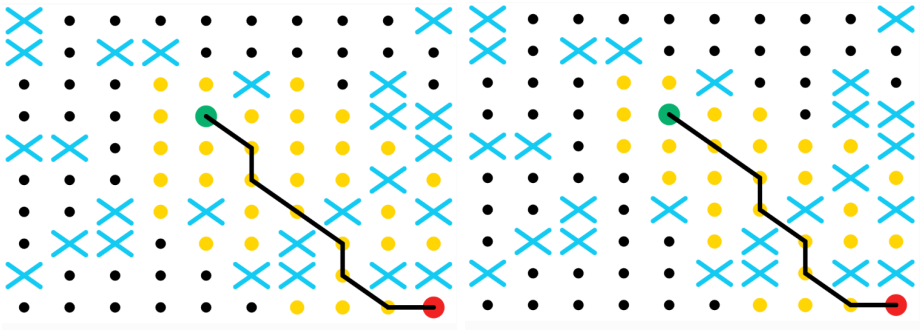


Figure 25: Another example of A* with Euclidean distance (left) and Manhattan distance (right)

In Figure 25 we can see another map where Manhattan heuristic outperformed Euclidian heuristic. In this case the algorithm with Manhattan distance was called 12 times and took 426 ms and Euclidian distance was called 18 times and took 662 ms.

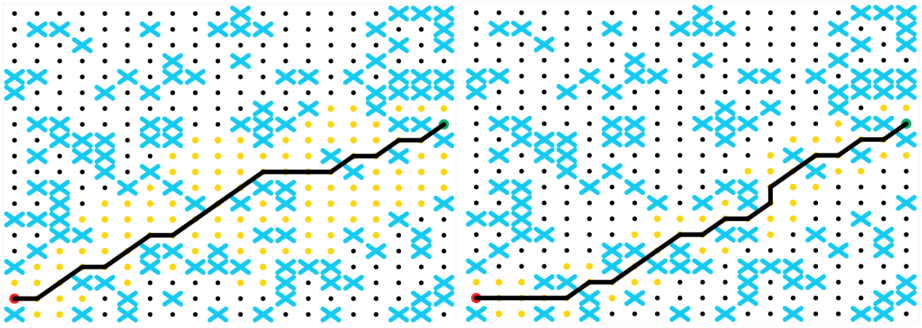


Figure 26: Example of A* with Euclidean distance (left) and Manhattan distance (right)

In Figure 26 we tried slightly more complicated map 20x20 where the Manhattan heuristic deviated from the shortest path. The shortest path found by the Euclidean heuristic was 23.56 units and the path from Manhattan heuristic was 24.14 units long. Although we can conclude that Manhattan heuristic is therefore not ideal if we focus on finding the shortest path it is still able to find a path that has a small error from the shortest path with much less effort (which can be seen by the number of explored yellow nodes) than the Euclidean heuristic does. In this case the algorithm with Euclidean heuristic had to be called 80 times with an execution time 2706.7 ms and the algorithm with Manhattan heuristic just 20 times with an execution time 799.6 ms. Therefore it is very much up to the field where we apply those algorithms, if we wanted to find a path that has a minimum error but finds a path much more quickly with less effort we would go for the Manhattan distance, if we wanted to be more sure to find the shortest path and did not care about the time that much we would choose the Euclidean heuristic.

Furthermore we can also conclude that trying those 2 heuristics on rather simple maps (3x3, 5x5, 10x10) did not yield a deviation from the shortest path by the Manhattan heuristic. Mainly because there are not that many nodes to evaluate and therefore a less chance of making an error. Therefore ideally we could set different heuristic based on the map size that we are running on, that is how we would still be able to find the shortest path with A* but we would have quicker results on simple maps using more suitable heuristic.

In some cases it is also appropriate to multiply the heuristic by some constant C (eq. 9).

$$H(n) = C * \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{9}$$

Doing this we can increase or decrease "the importance" of the heuristic so it ends up contributing more or less to the function (6). We can give up the optimal paths to make A* run faster by increasing C or we can make A* more precise in finding the shortest paths by decreasing C. Where the more C approaches 0 the more A* exhibits Dijkstra-like behaviour.

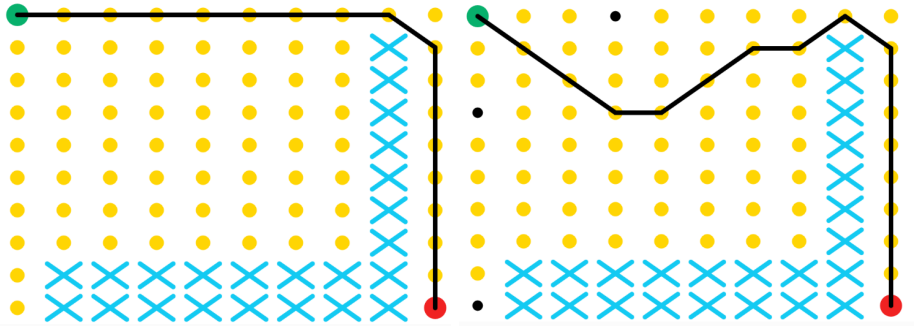


Figure 27: Example of A* with Heuristic with C = 1 (left) and Heuristic with C = 1000 (right)

In Figure 27 we can see that when we chose the constant C to be 1000, A* gave up the optimal path to save time and effort. Quantitatively A* with unchanged heuristic (left) had to be called 75 times with an execution time 2545 ms and the one with modified heuristic (right) 60 times with an execution time 2050.14 ms.

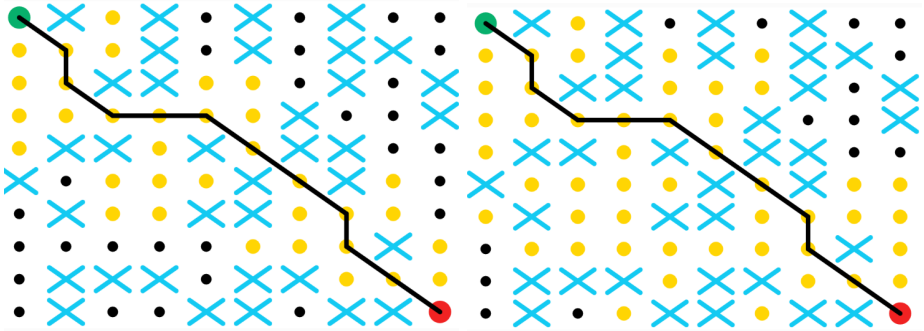


Figure 28: Example of A* with Heuristic with C = 1 (left) and Heuristic with C = 0.8 (right)

In Figure 28 we see the option where we decreased the importance of Heuristic to 80 %. Here we can conclude that the algorithm (on the right) with the decreased heuristic had similar behaviour as Dijkstra would, where it needed to explore more nodes to reach the end point and therefore took more time and effort. In this case the algorithm with unchanged heuristic needed to be called 21 times with an execution time 759 ms. And the modified algorithm 38 times with an execution time 1323.27 ms. But both found the shortest path which was expected.

To conclude this section, A* is heavily dependent on the heuristic it uses and therefore can be easily adjusted to different scenarios. Picking a constant C enables us to choose whether we want to focus on finding the shortest path (choosing $C < 1$) or finding a path with minimum time and effort (choosing $C > 1$).

7 Experimental Results

All Experimental data was collected by testing each algorithm 250 times on each map size. The map sizes that were tested were 3x3, 5x5, 10x10, 15x15, 20x20, 25x25, 30x30, 35x35, 40x40, 45x45, 50x50. In all these cases each map had a randomness of 30% meaning that each node had a 30% chance of being blocked when the map was created. As stated before JavaScript was run locally, therefore it was important to do all the test runs on one computer to make the data comparable. All the data was collected on a machine running a 6 core 12 thread processor clocking 2.2 GHz with 4.1 GHz turbo with active cooling, 8 GB of DDR4 ram memory and the computer has an SSD with reading and writing speeds of 1 GB/sec.

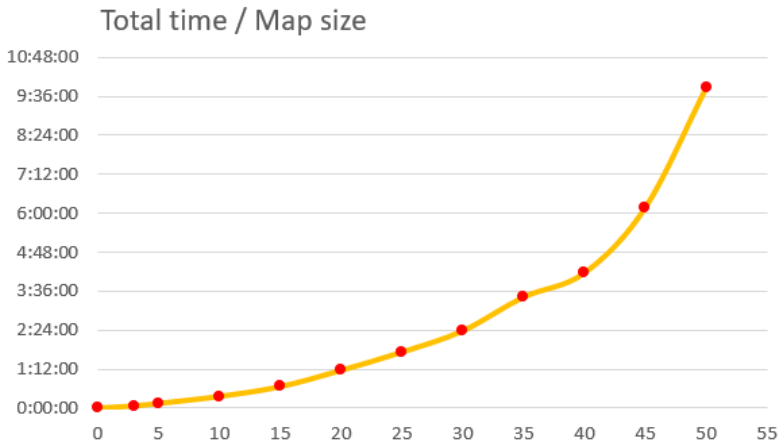


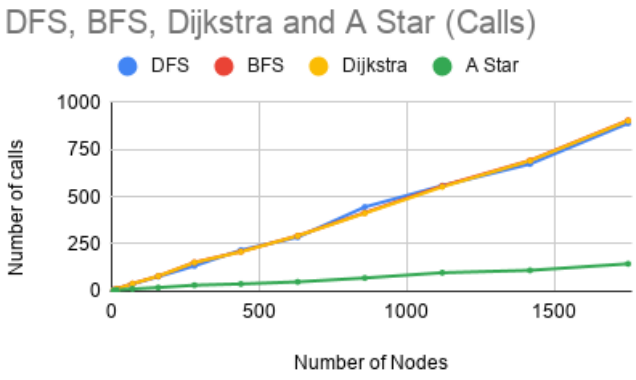
Figure 29: Total time of 1000 runs (250 random maps * 4 algorithms = 1000 runs) against different map sizes (hh:mm:ss)

In Figure 29 the duration of time per each map size changes in an exponential matter as we expected it to do since each time (not including the 3x3 map) the map size increases by 5 and so does the total number of nodes (proportionally).

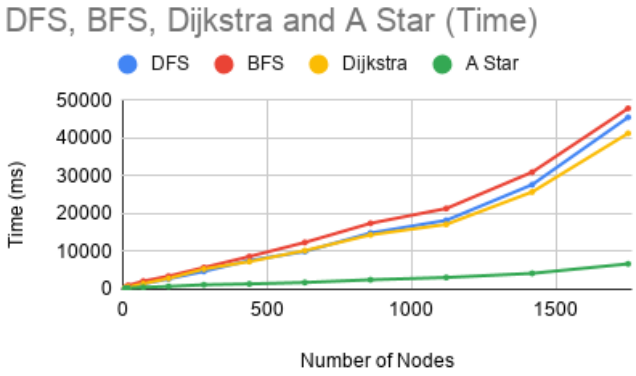
Name	Time[ms]	Times called	Length
DFS	7548.73	216.5	257.32
BFS	8587.2	208.6	15.38
Dijkstra	7249.82	207.7	14.08
A*	1319.99	36.8	14.08

Table 2: Average from 250 runs on a 25x25 map with 30 % randomness

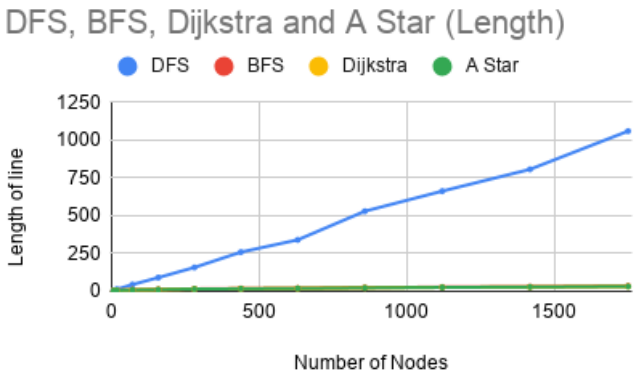
An example of how the data was stored can be seen in table 2, where for each of the algorithms we collected the averages of the run-time, the amount of times the algorithm was called and the length of the final path. Finding the right performance indicators is crucial for analysing any kind of problem. We focused on those three performance indicators, because we found them the most useful when it comes to describing the performance of a path-finding algorithm.



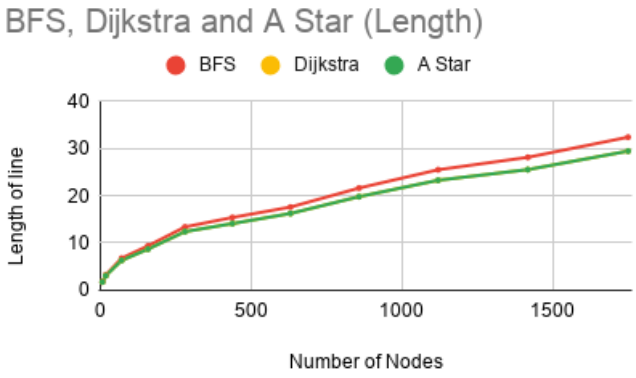
(a) Average number of nodes vs Calls



(b) Average number of nodes vs Time



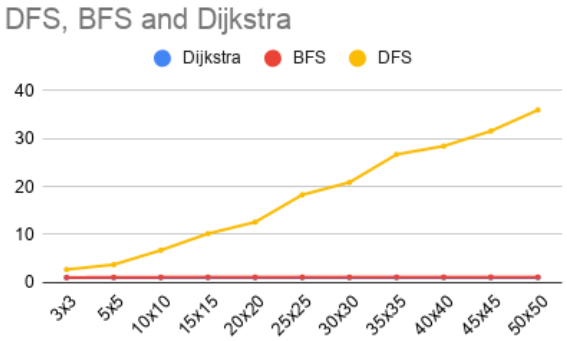
(c) Average number of nodes vs Length



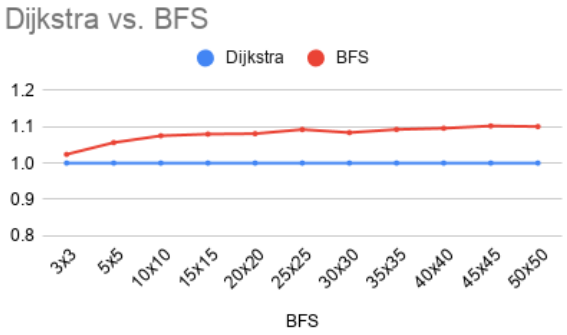
(d) Average number of nodes vs Length without DFS

Figure 30: 4 Images showing how the algorithms perform as map size grows

Figure 30 shows how the number of calls, time and length of the path change as the number of available nodes on the map increases. The number of available nodes is calculated by taking the chance a note is not blocked (70%) and multiplying it by the number of nodes on the map. In all the graphs (a,b,c,d) A* (green) performs the best. In (a) and (b) DFS, BFS and Dijkstra follow the same trend, where the amount of calls and the time [ms] do not deviate by much. In (c) we can see the flaws of DFS that were described in the section 3.5. (d) shows that A*, Dijkstra and BFS follow exactly the same trend, with a difference that BFS did not always find the shortest path and therefore is slightly above the other two.



(a) Length/Optimal length

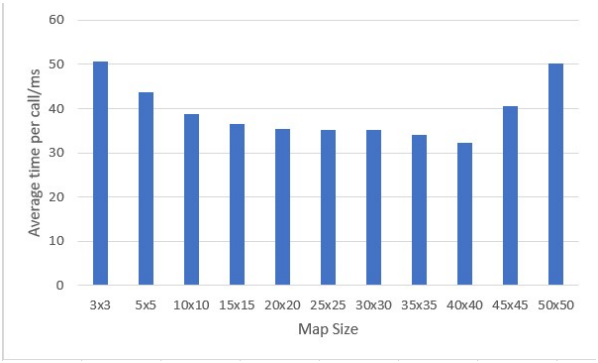


(b) Length/Optimal Length

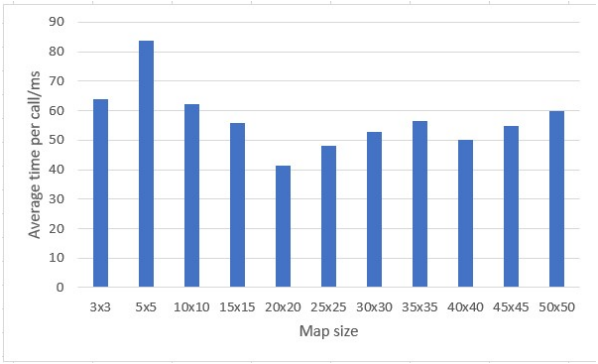
Figure 31: Charts showing how the error in line length grows with map size

Figure 31 shows the length of the line compared to the optimal length as the map size increases. Since we know Dijkstra always finds the shortest path this value is calculated by dividing the length of the path by the length of the Dijkstra path. A* was able to find the shortest path on each of the maps and therefore is here not shown (the error was 0%). For DFS (yellow) the bigger the map the bigger the error, which means that the deviation of the path found by DFS is directly proportional to the map size. BFS performed well, where the maximum error was approximately 10 %.

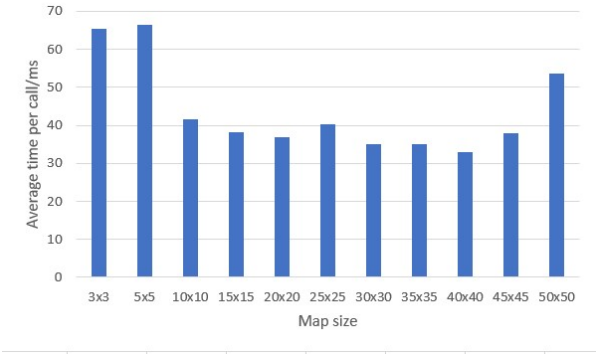
Figure 32 shows the ratio between the number of calls and time as the map grows. This is done by calculating the average time per call on each map for each algorithm and comparing them to the map size.



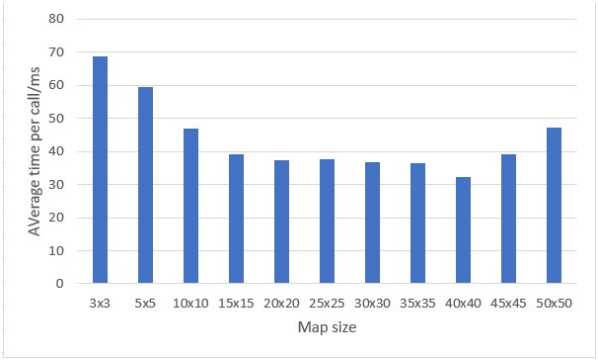
(a) DFS



(b) BFS



(c) Dijkstra



(d) A*

Figure 32: 4 Charts showing the average time per call divided by map size and algorithm

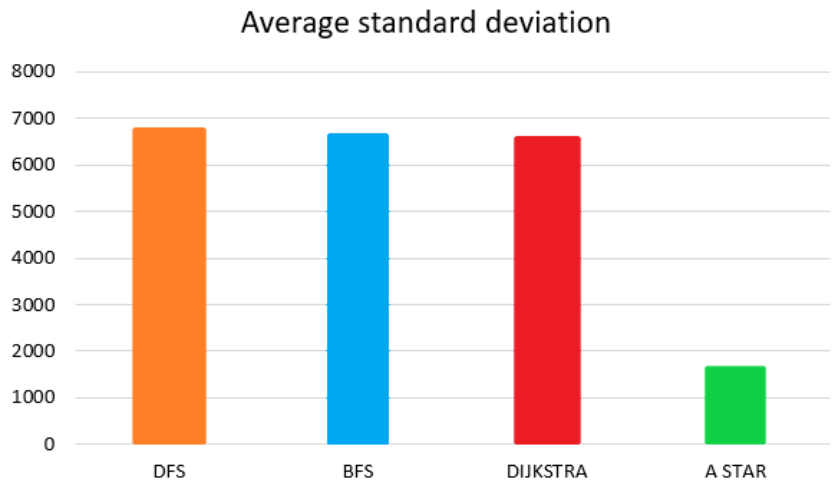


Figure 33: Average standard deviations of time where orange is DFS, blue is BFS, red is Dijkstra and green is A*

Another relevant comparison between the four algorithms is represented by the standard deviations of time for each of them. For our comparison we made an average between all the standard deviations calculated for each set of maps and for each individual algorithm. As it can be seen from the figure 33, A* outperforms the other three by a large margin when it comes to time deviation between different maps since its runtimes are more consistent.

8 Discussion

There are many different ways that we can search for the shortest path each with there own advantages and disadvantages. The 4 that we looked at are unique in how they need to be implemented and in how they perform. DFS and BFS are examples of programs that are fairly easy to implement and will get the job done but are not as efficient as others. These are very basic programs that only differ in searching an adjacency list FIFO (BFS) and LIFO (DFS). These do not look at weights of the lengths between nodes and only care about the position in the adjacency list. Dijkstra is similar to BFS but it looks at weights of the length between nodes which makes it harder to implement. This makes it always find the shortest path because it works by finding the shortest path to all nodes untill it reaches the end node. In Figure 30 we can see DFS has a similar number of calls and time to BFS and Dijkstra but as we look at the length of the line we can see where it really falls behind. If we remove DFS from the graph we get figure 30d and can see the difference between BFS, Dijkstra and A*. In this figure Dijkstra and A* follow the same line. BFS is close to finding the best line but is always longer. If we look back at figures 30a and 30b we see A* performing far better than all the other algorithms because of how it searches in the expected direction of the shortest path. This makes it work far better as the graph gets bigger and the difference between the performance grows. The largest graphs we looked at were 50x50 with a 30% chance of each node being blocked but if we went larger the gap would keep getting bigger.

Upon first glance none of the data we have reflects the big O notations we derived for each algorithm. This is because we only counted the number of times an algorithm would be called and not the number of times the methods inside the algorithms are called. This is because the overhead complexity, the complexity caused by all the drawing and input, vastly overshadows that of the actual algorithms. This overshadowing complexity caused by the drawing is why on Figure 30b the time measured seemingly follows a quadratic curve, and not the shape expected by the calculated big O.

The consequence of counting the number of times the algorithm was called is that the complexity of the data structures as well as the methods called by the algorithms is not accounted. Making them effectively O(1) so they dont play an effect [13]. When this is taken into account the complexity of the algorithm's is O(n), linear. This can be seen in BFS(calls) where every graph is linear. While this does explain DFS, BFS and Dijkstra, the obvious outlier is A star as it has far less calls than any other algorithm.

In AI a similar issue arrises when using pathfinding algorithms [13], this is because the complexity of $O(V + E)$ does not take into account the fact that graph could be huge or infinite, nor the fact that an algorithm contains a heuristic to increase its accuracy or that nodes are only traversed a set number of times.

Another way of measuring complexity is by $O(b^d)$, here b is the branching factor, and d is the shortest path [13]. The logic behind this is simple but effective, by branching out in

b directions with a depth of d the number of nodes explored becomes b^d. The branching factor is the average amount of new nodes discovered per step [13].

To test whether this measure is more representative of the data, we took d to be the average distance between 2 points in a square. For a square that is 1 unit long the average distance is calculated to be approximately 0.52140, however this is not applicable to our scenario because instead of having a continuous spectrum of numbers to average, our specific case requires that we use discrete values to calculate this average distance. We, however, did use this value to make sure our estimation method was at least up to 3 significant figures correct. This is because our minimum graph size is 3x3, where a point has to have discrete values for its coordinates (that is to say (1.1, 2) is an invalid coordinate but (1, 2) is valid). Because of this we averaged the distances between 2 random points in a discrete squares of dimensions that were the size of the graphs used for data collection. The issue with doing this is that in our graphs the shortest path is not necessarily a line that connects the start node to the end node as this is the best case scenario, but it is a close enough approximation to roughly test how accurate the $O(b^d)$ value is.

To calculate b, the branching factor, we used the best case scenario again. In this case every node around the starting node can be explored, when this node is explored every node around that is explored, the graph is considered to be a grid, but other than that is infinite (none of the free spaces are walls, and upon exploring a new area more free spaces are available to explore). In this case 16 spaces are seen and 8 spaces are explored, so the branching factor is 2. In practise, the branching factor would be less because this is assuming the nodes can always move into free space and there are no walls.

Using this information we can see that the calculated big O becomes $O(2^d)$. While this appears to be an exponential complexity it is actually linear, it is exponential with the depth of the search, but linear with the size of the graph, just like the previously calculated complexities of value $O(V)$. This also explains why A* shows a better time because, when considering a heuristic the branching factor decreases, this is known as the effective branching factor $B*$. This makes the algorithm $O((B*)^d)$.

B* is determined by the formula:

$$N + 1 = \sum_{n=0}^{d} ((B*)^d) \tag{10}$$

Where $B*$ is the effective branching factor, N is the number of explored nodes, and d is the length/depth of the shortest path.

When $B*$ is taken to be 1 it means that a very effective heuristic is being used, and this causes the complexity of the A* algorithm to become $O(1)$, which is constant time. Assuming the complexity of the drawing to only be $O(V)$, the complexity for A* becomes $O(V) \cdot O(1) = O(V)$. But the complexity for all the other algorithms become $O(V) \cdot O(2^d)$. Explaining why the curves for BFS, DFS, and Dijkstra have some sort of exponentiation, as opposed to the linear line (for A*) shown in Figure 30b.

When making these assumptions and testing them on all kinds of maps, the upper bound for complexity calculated by b^d is just above what the number of calls performed is (for any algorithm in the worst case), showing that even though a lot of assumptions were made the end result is still somewhat accurate.

This does not fully explain why Figure 30a shows linear time for A* when it should show constant time. This is because these calculations were made with the assumption that there were no obstacles and graph is a square grid. This has the effect of making the effective branching factor: 1. Furthermore, we also assume the average distance between 2 points in the grid as the shortest possible path.

When actually testing the algorithms this assumed maximum bound seemed accurate but it is not completely accurate. This is because when applied on actual maps the effective branching factor will be slightly increased, as well as the shortest path d being a bit longer. For the other algorithms the branding factor would be a bit less than 2 because we assumed that every step would reveal a new node to branch towards, which is just not the case when dealing with the actual maps. This has the effect of making $O(b^d) = O(V)$ as the input is the size of the map.

Since looking at the length of the path is very important we can look in more detail at the error in length of BFS and DFS. This can be seen in figure 31 where 2 graphs show the length of the path / length of the optimal path. Since Dijkstra and A* both find the shortest path on this style of map they are linear at 1. In figure 31a DFS already fails to find the shortest path on a 3x3 map and the error grows very fast. At a 15x15 map the length of the line is already on average 10x longer than Dijkstra and at 50x50 it is 37x the optimal length on average. If we remove DFS we get figure 31b and can more clearly see the difference between Dijkstra and BFS. The error BFS has grows very slowly as the only difference in our case is diagonals. At the largest map we measured (50x50) the line was only 1.1x longer than Dijkstra on average. If a map was more complicated and had weighted edges the difference would be larger. A* is not displayed in this graph as it follows the same path as Dijkstra on our maps. This would not be the case on other styles of maps and also if other heuristic values were used. This is explained more in detail in section 6.6.

If we look at the ratio of execution time and number of calls, DFS seems to execute a call the fastest at 39.3 ms, whereas Dijkstra and A* would be 2$^{\text{nd}}$ in this metric with 43.9 and 43.8 ms respectively arguably to small of a time difference to say anything, BFS is surprisingly much slower at 57.2 ms. If we look at Figure 32 we see that DFS, Dijkstra, and A* (and somewhat BFS) all have a longer time per call on the smallest maps, and when we increase the map size the time per call is reducing until a certain point when it starts increasing again. The initial long time per call can be attributed to the construction of the algorithm (declaration of variables and such), because on smaller maps there are few calls so this effect can be more pronounced. One might think that A* and Dijkstra's increase in time per call might be attributed to the fact that they don't run in linear time, but considering that this effect can be seen also on DFS and BFS, maybe a better explanation could be

memory reallocation for the stack, queue and priority queue, because they are implemented on JavaScript's Array.prototype and not based on a linked list.

9 Conclusion

After analysis of the data we created and research of the different algorithms we have come to the conclusion that A* using the Euclidean heuristic works the best for our map. Not only did it always find the shortest path on our maps, it did that in a much faster time than all the other algorithms. It should be noted that this might not be true for other map styles or larger more complex maps. Dijkstra has its place for complex maps where you want to be 100% sure that you are finding the shortest path and the extra processing time does not matter. BFS's accuracy significantly falls behind Dijkstra in a very simple map on small scales so it does not seem like a viable option for accurate path finding when weights are used; However, if used on an unweighted graph BFS is equally efficient. DFS does not have a place in searching for shortest path as it would always have a large error that continues to grow as the map gets bigger.

When calculating the big O complexities for our algorithms, we determined that BFS and DFS are $O(V)$ where V is the number of nodes, and Dijkstra to be $O(VlogV$. Overall it was hard for us to detect the subtle differences between the two because of how we recorded our data, and the fact that the complexity of our drawing operations was so large that the complexity of the priority queue $O(logV)$ by comparison is constant($O(1)$) making it extremely difficult to detect the difference by comparing the time. This has the effect of giving Dijkstra the appearance of $O(V)$. Because A star without a heuristic of 0 is Dijkstra the complexity of this algorithm would have the same complexity multiplied by the complexity of its heuristic algorithm.

So, at least according to how we measured the data, all the algorithms should have the same complexity, however this is not the case. While BFS, DFS, and Dijkstra all exhibit the same behaviour (complexity wise), A* shows significantly better complexity even though it is (by our calculations just as good as the others). The reason for this is because when the graph is huge or infinite the complexity of $O(V)$ can also be written as $O(b^d)$, where b is the branching factor and d is the shortest distance. This is because the estimates for $O(V)$ include traversing along edges the maximum number of times, as well as not necessarily searching the entire area, and the fact that an algorithm is using a heuristic, whereas $O(b^d)$ accounts for this.

This new way of representing the big O notation is vital for explaining why A* behaves so differently in our data. Because A* uses a heuristic it's effective branching factor is less than the branching factor for other algorithms. This leads to it having a complexity that is more constant than the others. We would also change the way the number of calls for each algorithm is counted to take into account the use of the internal data structures. Possibly we would also investigate DFS with the alternate parent/child path distance calculated.

Looking back after the report we could have improved our results by increasing the map size to see how results change as they get very large. This would require a lot of time as with 50x50 maps it already took almost 10 hours to get results. As well as this we could have done more testing to see the effect of the randomness on the algorithms.

References

[1] John Adrian Bondy. *Graph Theory with Applications*. North Holland, 1976. ISBN: 9780444194510.

[2] Thomas H Cormen. *Introduction to Algorithms*. eng. 3rd ed.. The MIT Press Ser. 2009. ISBN: 9780262270830.

[3] Simon Rubinstein-Salzedo. "Big O Notation and Algorithm Efficiency". In: *Cryptography*. Cham: Springer International Publishing, 2018, pp. 75–83. ISBN: 978-3-319-94818-8. DOI: 10.1007/978-3-319-94818-8_8.

[4] Robert Sedgwick. *Algorithms in C++*. Addison-Wesley. ISBN: 0201510586.

[5] User: Boivie. *Representation of a Stack*. [Online; accessed April 28, 2020]. 2006.

[6] Sally A. (Sally Ann) Goldman. *A practical guide to data structures and algorithms using Java*. eng. Chapman & Hall/CRC applied algorithms and data structures series. Boca Raton: Chapman & Hall/CRC. ISBN: 0-429-14710-4.

[7] Cormen Thomas H.; Leiserson Charles E. Rivest Ronald L. *Introduction to Algorithms*. 3rd ed. MIT Press and McGraw-Hill, 2009[1990]. ISBN: 0-262-03384-4.

[8] User:Boivie. *Representation of a Queue*. [Online; accessed April 30, 2020]. 2006.

[9] Blake Matheny. *Animated BFS*. [Online; accessed May 20, 2020]. 2007.

[10] Edsger W Dijkstra. *A Note on Two Problems in Connexion with Graphs*. 1959.

[11] Dapeng Sun and Min Li. "Evaluation function optimization of A-star algorithm in optimal path selection". In: *Revista Tecnica de la Facultad de Ingenieria Universidad del Zulia* 39.4 (2016), pp. 105–111. ISSN: 02540770. DOI: 10.21311/001.39.4.14.

[12] The Operating. "The Accounting Review , July , 1964- University of Missouri , Columbia A simple exercise to illustrate heuristics ." In: 2 (1964), pp. 2–4.

[13] Peter Russell Stuart; Norvig. *Artificial Intelligence: A Modern Approach*. 3rd ed. Prentice Hall, 2009[1995]. ISBN: 978-0-13-604259-4.