



Prifysgol Cymru
Y Drindod Dewi Sant
University of Wales
Trinity Saint David

A Study of Comparing and Implementing AI Pathfinding Methods

By

Jake Russell

Supervisor: Gordon Dickers

Secondary Marker: Tim Bashford

Project submitted as part of the requirements for the
award of BSc (Hons) Computing

April 2022

DECLARATION

I, Jake Russell declare that I am the sole author of this Project; that all references cited have been consulted; that I have conducted all work of which this is a record, and that the finished work lies within the prescribed word limits.

This work has not previously been accepted as part of any other degree submission.

FORM OF CONSENT

I Jake Russell hereby consent that my Project, submitted in candidature for the B.Sc. (Hons) Computing degree, if successful, may be made available for inter-library loan or photocopying (subject to the law of copyright), and that the title and abstract may be made available to outside organisations.

ABSTRACT

The following project is intended implement and compare different AI pathfinding methods by getting them to solve a maze, it attempts this through the objectives of: finding a way to convert a maze into a format that could be used within code, learning what types of graph traversal and search algorithms are out there, learning how Dijkstra's method operates, successfully transfer that into a code format and run it with the converted maze, learning how A* Search method operates, successfully transfer that into code format and run it with the converted maze, and comparing how each algorithm operates when tasked to solve the maze. The method used for research is Qualitative and the development method used is the waterfall method. An important recommendation is to use for future projects like this one is if able to create way to generate mazes of different sizes in order to expand the range of testing capable.

Contents

ABSTRACT	3
INTRODUCTION	5
RESEARCH & REVIEW OF LITERATURE	6
METHODOLOGY	21
DESIGN, IMPLEMENTATION AND RESULTS	22
EVALUATION	27
CONCLUSIONS AND RECOMMENDATIONS.....	28
REFLECTION	30
REFERENCES	31
APPENDIX	32

INTRODUCTION

This project is about implementing different pathfinding algorithms used in artificial intelligence and comparing their capabilities to one another. The main aim is to use these algorithms in order to solve a maze and figure out whether one should be prioritised over the other for this task. In order to accomplish this a set of objectives were set. First is to find a way to convert a maze into a format that could be used within code. Next is to learn what types of graph traversal and search algorithms are out there. Next is to learn how Dijkstra's method operates, successfully transfer that into a code format and run it with the converted maze. After that is to learn how A* Search method operates, successfully transfer that into code format and run it with the converted maze. Lastly is to compare how each algorithm operates when tasked to solve the maze. The following chapters will be a literature review: showcasing meanings and explanations behind shortest path problems, Graph structures and their representations, search strategies, and shortest path algorithms. An explanation on the choice methodology used when it came to researching and developing. A showcase of the designs, implementations and results of the algorithms. The conclusions and recommendations of the project. And finally, a personal reflection from the author on the project.

RESEARCH & REVIEW OF LITERATURE

The following literature review will include the following sections:

- Shortest Path Problems
- Graph structures and Representation
- Search Strategies
- Shortest Path Algorithms

The first section will go over the four types of shortest path problems. The next section will be describing what graphs are made of, the types of graphs, their terminology and how they can be represented within code. The third section will go over Search strategies for graphs such as Breadth-first, Depth-first, and Best-First Search. Finally, the last section will describe the algorithms used to find the shortest path for the different problems.

Shortest Path Problems

The root of all pathfinding is the idea of the shortest path problem. Author Akshay Singhal [1] describes there to be four types of shortest path problems:

- Single-pair shortest path problem
- Single-source shortest path problem
- Single-destination shortest path problem
- All pairs shortest path problem

Singhal explains single-pair problems as “where the shortest path between a given pair of vertices is computed”. They go on to explain single-source as “where the shortest path from a given source vertex to all other remaining vertices is computed”. Next is single-destination which they put as “where the shortest path from all the vertices to a single destination vertex is computed” as well stating “By reversing the direction of each edge in the graph, this problem reduces to single-source shortest path problem”. Lastly Singhal describes all pairs problem as “where the shortest path between every pair of vertices is computed”.

Singhal's statements are corroborated by another article from the site brilliant.org [2], here in the article it also goes on to describe shortest path problems but focusing on what it believes to be the main two problems: Single-source and All-pairs. It gives the definition of single-source as this: "Given a graph G , with vertices V , edges E with weight function $w(u, v) = w_{u,v}$, and a single source vertex, s , return the shortest paths from s to all other vertices in V ." The article goes on to give this statement on single-pair problems: "If the goal of the algorithm is to find the shortest path between only two given vertices, s and t , then the algorithm can simply be stopped when that shortest path is found. Because there is no way to decide which vertices to "finish" first, all algorithms that solve for the shortest path between two given vertices have the same worst-case asymptotic complexity as single-source shortest path algorithms."

Graph Structures and Representation

Graphs are a core form of data structure used when implementing search strategies. In the document Data structures and algorithms written and revised by John Bullinaria [3] a graph is described as such: "A graph consists of a series of nodes (also called vertices or points), displayed as nodes, and edges (also called lines, links or, in directed graphs, arcs), displayed as connections between the nodes." Bullinaria expands upon the structures of graphs with "A graph is said to be simple if it has no self-loops (i.e., edges connected at both ends to the same vertex) and no more than one edge connecting any pair of vertices." And "If there are labels on the edges (usually non-negative real numbers), we say that the graph is weighted. We distinguish between directed and undirected graphs. In directed graphs (also called digraphs), each edge comes with one or two directions, which are usually indicated by arrows." Bullinaria proceeds to

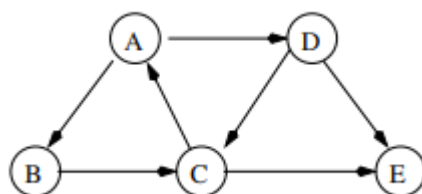


Figure 1: unweighted digraph [3]

present examples of these graph structures with an example of an unweighted digraph in figure 1 and a weighted digraph in figure 2.

Ortega-Arranz et al [4] also states “A graph $G = (V, E)$ is composed by a set of vertices V , also called nodes, and a set of edges E , also called arcs. Every node v is usually depicted as a point in the graph. Every edge e is usually depicted as a line that connects two and only two nodes. An

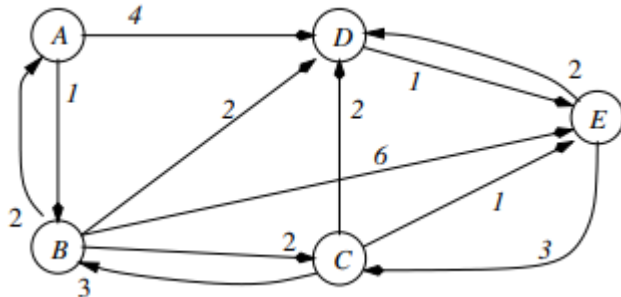


Figure 2: Weighted digraph [3]

edge is a tuple (u, v) that represents a link between nodes u and v . The number of edges connected to a node v is called the degree of v . In an undirected graph all edges can be traversed in both directions, whereas an edge (u, v) of a directed graph only can be traversed from u to v .

Associated to each edge there is a weight function $w(u, v)$, that represents the cost of traversing this edge.”

An organisation by the name of Geeksforgeeks.org [5] has an article that describes graphs as representing many real-life applications, particularly networks. The article then elaborated with: “The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender, and locale.” In terms of representation of graphs within code the article states that there are two representations that are the most common, being Adjacency matrix and Adjacency List. However, the article also mentions that there is Incidence Matrix and Incidence List as well and that “The choice of graph representation is situation-specific. It totally depends on the type of operations to be performed and ease of use.”

According to the article [5] an Adjacency matrix is : "a 2D array of size $V \times V$ where V is the number of vertices in a graph. Let the 2D array be $adj[i][j]$, a slot $adj[i][j] = 1$ indicates that there is an edge from vertex i to vertex j . Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If $adj[i][j] = w$, then there is an edge from vertex i to vertex j with weight w ." The article provides an example unweighted graph (figure 3) and its corresponding adjacency matrix (figure 4) to showcase its explanation.

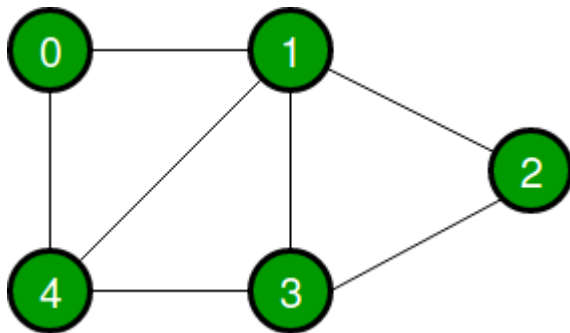


Figure 3: Unweighted Graph [5]

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Figure 4: Adjacency Matrix [5]

Within the same article [5] it also explains the other most common representation: Adjacency list. The article states it as being "An array of lists is used. The size of the array is equal to the number of vertices. Let the array be an $array[i]$. An entry $array[i]$ represents the list of vertices adjacent to the i th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs. Following is the adjacency list representation of the above graph." Like before it provides a visual example (figure 5).

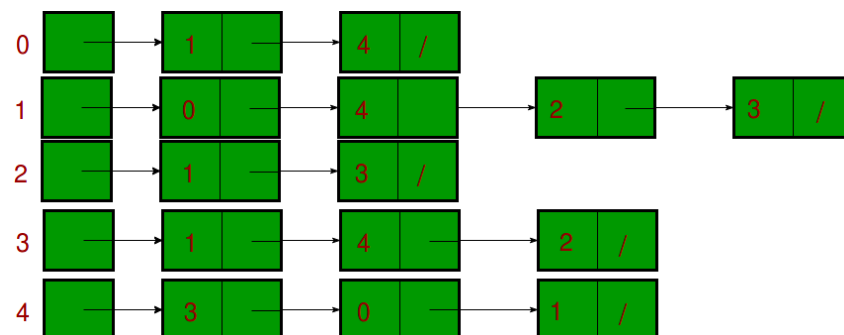


Figure 5: Adjacency List [5]

Search Strategies

First, to understand what makes certain search strategies and algorithms better than others for a particular task a criteria in which to measure must be set, authors Norvig and Russell [6] define this criteria as such: “**Completeness**: Is the algorithm guaranteed to find a solution when there is one? **Optimality**: Does the strategy find the optimal solution? **Time complexity**: How long does it take to find a solution? **Space complexity**: How much memory is needed to perform the search?” To indicate time complexities a system called the “Big Oh notation” is used. Authors Granville Barnett and Luca Del Tongo [7] explains some of the most common big Oh notations: “ $O(1)$ constant: the operation doesn’t depend on the size of its input, e.g. adding a node to the tail of a linked list where we always maintain a pointer to the tail node. $O(n)$ linear: the run time complexity is proportionate to the size of n . $O(\log n)$ logarithmic: normally associated with algorithms that break the problem into smaller chunks per each invocation, e.g. searching a binary search tree. $O(n \log n)$ just $n \log n$: usually associated with an algorithm that breaks the problem into smaller chunks per each invocation, and then takes the results of these smaller chunks and stitches them back together, e.g. quick sort. $O(n^2)$ quadratic: e.g. bubble sort. $O(n^3)$ cubic: very rare. $O(2^n)$ exponential: incredibly rare.”

In a general sense there are two types of search strategies when searching through a graph, uninformed (also called blind search) and informed (also called heuristic search), Norvig and Russell [6] describe uninformed and meaning “the strategies have no additional information about states beyond that provided in the problem definition. All they can do is generate successors and distinguish a goal state from a non-goal state.” Whereas they describe informed search strategies as “Strategies that know whether one non-goal state is “more promising” than another”. The following are example strategies:

Breadth-first search

Norvig and Russell [6] state: “Breadth-first search is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on. In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.” Norvig and Russell expand upon this further with: “Breadth-first search is an instance of which the shallowest unexpanded node is chosen for

expansion. This is achieved very simply by using a First In First Out queue for the frontier. Thus, new nodes (which are always deeper than their parents) go to the back of the queue, and old nodes, which are shallower than the new nodes, get expanded first. The goal test is applied to each node when it is generated rather than when it is selected for expansion. Note also that the algorithm discards any new path to a state already in the frontier or explored set; it is easy to see that any such path must be at least as deep as the one already found. Thus, breadth-first search always has the shallowest path to every node on the frontier.”

In terms of the rating the breadth-first search according to the four criteria stated earlier, Norvig and Russell write “We can easily see that it is complete—if the shallowest goal node is at some finite depth d , breadth-first search will eventually find it after generating all shallower nodes (provided the branching factor b is finite). Note that as soon as a goal node is generated, we know it is the shallowest goal node because all shallower nodes must have been generated already and failed the goal test. Now, the shallowest goal node is not necessarily the optimal one; technically, breadth-first search is optimal if the path cost is a nondecreasing function of the depth of the node.” Time complexity and space complexity however is where Norvig and Russell state the breadth-first search fails, presenting the table shown in figure 6 in how the breadth-first search rapidly becomes inefficient in both time and space. Provided as well by Norvig and Russell is the pseudo code for a breadth-first search on a graph (figure 7).

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Figure 6: Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node. [6]

```

function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier) /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← INSERT(child, frontier)

```

Figure 7: Breadth-first search on a graph [6]

The article from Geeksforgeeks [5] within its breadth-first search section states the time complexity as being “ $O(V+E)$ where V is a number of vertices in the graph and E is a number of edges in the graph.”

Depth-first search

Norvig and Russell [6] state: “Depth-first search always expands the deepest node in the current frontier of the search tree. The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. As those nodes are expanded, they are dropped from the frontier, so then the search “backs up” to the next deepest node that still has unexplored successors.” Along with comparing it to the previously showcased breadth-first search: “whereas breadth-first-search uses a FIFO queue, depth-first search uses a LIFO queue. A LIFO queue means that the most recently generated node is chosen for expansion. This must be the deepest unexpanded node because it is one deeper than its parent—which, in turn, was the deepest unexpanded node when it was selected.” The authors proceed to explain that when it comes to searching graphs a depth-first search hold no advantage over a breadth-first search.

Geeksforgeeks [5] has the time complexity as “ $O(V+E)$ ” and the space complexity “ $O(V)$ ”, since an extra visited array of size V is required.”

Best-first search

Best-first search is an expansion from Breadth-first search and Depth-first search which is informed as apposed to uninformed. Geekforgeeks article [5] writes: “In BFS(Breadth-first search) and DFS(Depth-first search), when we are at a node, we can consider any of the adjacent as next node. So both BFS and DFS blindly explore paths without considering any cost function. The idea of Best First Search is to use an evaluation function to decide which adjacent is most promising and then explore. Best First Search falls under the category of Heuristic Search or Informed Search.” Norvig and Russell [6] describe it as: “Best-first search is an instance of the general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an evaluation function, $f(n)$. The evaluation function is construed as a cost estimate, so the node with the lowest evaluation is expanded first. The implementation of best-first graph search is identical to that for uniform-cost search except for the use of f instead of g to order the priority queue. The choice of f determines the search strategy. Most best-first algorithms include as a component of f a heuristic function, denoted $h(n)$: $h(n)$ = estimated cost of the cheapest path from the state at node n to a goal state.” The article [5] gives an analysis of Best-first search, it states: “worst case time complexity for Best First Search is $O(n * \log n)$ where n is number of nodes. In worst case, we may have to visit all nodes before we reach goal. Note that priority queue is implemented using Min(or Max) Heap, and insert and remove operations take $O(\log n)$ time.” The article then also provides pseudo code of the Best-first search shown in figure 8.

```

Best-First-Search(Graph g, Node start)
  1) Create an empty PriorityQueue
     PriorityQueue pq;
  2) Insert "start" in pq.
     pq.insert(start)
  3) Until PriorityQueue is empty
     u = PriorityQueue.DeleteMin
     If u is the goal
       Exit
     Else
       Foreach neighbor v of u
         If v "Unvisited"
           Mark v "Visited"
           pq.insert(v)
       Mark u "Examined"
End procedure

```

Figure 8: Best-first search pseudo code [5]

Shortest Path Algorithms

Dijkstra's Algorithm

Dijkstra's is a classic shortest path algorithm specialising in solving single-source problems but can solve single-pair problems as well. Ortega-Arranz et al [4] explains Dijkstra's with the following: "The algorithm is the basic solution for the shortest-path problem. Dijkstra's algorithm constructs minimal paths from a source node s to the remaining nodes, exploring adjacent nodes following a proximity criterion until the shortest path to goal node t is found. The exploring process is known as edge relaxation. When an edge $u; v$ is relaxed from a node u , it is said that node v has been reached. Therefore, there is a path from source through u to reach v with a tentative shortest distance. Node v will be considered settled when the algorithm has found the shortest path from source node s to v . The algorithm finishes when t is settled." Expanding upon this further Ortega-Arranz et al write "Thee algorithm uses an array, D , that stores all tentative distances found from source node s to the rest of nodes. At the beginning of the algorithm, every

node is unreachable and no distances are known, so $D[i] = \infty$ for all nodes i , except current source node $D[s] = 0$. The algorithm proceeds as follows:

1. (Initialization) It starts on the source node s , initializing distance array $D[i] = \infty$ for all nodes i and $D[s] = 0$. Node s is settled and considered as the current node c ($c \leftarrow s$).
2. (Edge relaxation) For every node v adjacent to c that has not been settled, a new distance from source is found using the path through c , with value $D[c] + w(c, v)$. If this distance is lower than previous value $D[v]$, $D[v] \leftarrow D[c] + w(c, v)$.
3. (Settlement) The node u with the lowest value in D is taken as the new current node ($c \leftarrow u$). After this, current node c is now considered as settled.
4. (Termination criteria) If u is the target node the algorithm finishes, since the tentative path to u is now considered the minimum one. Otherwise, the algorithm proceeds to step 2.

In order to recover the path, every node reached stores its predecessor, so at the end of the query phase the algorithm just runs back from target through stored predecessors till the source is reached.” Geekforgeeks article [5] supports this with it’s explanation on the algorithm: “Given a graph and a source vertex in the graph, find the shortest paths from the source to all vertices in the given graph. We generate a SPT (shortest path tree) with a given source as a root. We maintain two sets, one set contains vertices included in the shortest-path tree, other set includes vertices not yet included in the shortest-path tree. At every step of the algorithm, we find a vertex that is in the other set (set of not yet included) and has a minimum distance from the source. Below are the detailed steps used in Dijkstra’s algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.

- 1) Create a set sptSet (shortest path tree set) that keeps track of vertices included in the shortest-path tree, i.e., whose minimum distance from the source is calculated and finalized. Initially, this set is empty.
- 2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.
- 3) While sptSet doesn’t include all vertices

....a) Pick a vertex u which is not there in $sptSet$ and has a minimum distance value.

....b) Include u to $sptSet$.

....c) Update distance value of all adjacent vertices of u . To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v , if the sum of distance value of u (from source) and weight of edge $u-v$, is less than the distance value of v , then update the distance value of v ." Below in figure 9 is pseudo code wrote by Bullinaria [3] describing showing Dijkstra's code implementation.

```
// Input: A directed graph with weight matrix 'weight' and
//        a start vertex 's'.
// Output: An array 'D' of distances as explained above.

// We begin by buiding the distance overestimates.

D[s] = 0 // The shortest path from s to itself has length zero.

for ( each vertex z of the graph ) {
    if ( z is not the start vertex s )
        D[z] = infinity // This is certainly an overestimate.
}

// We use an auxiliary array 'tight' indexed by the vertices,
// that records for which nodes the shortest path estimates
// are "known" to be tight by the algorithm.

for ( each vertex z of the graph ) {
    tight[z] = false
}

// We now repeatedly update the arrays 'D' and 'tight' until
// all entries in the array 'tight' hold the value true.

repeat as many times as there are vertices in the graph {
    find a vertex u with tight[u] false and minimal estimate D[u]
    tight[u] = true
    for ( each vertex z adjacent to u )
        if ( D[u] + weight[u][z] < D[z] )
            D[z] = D[u] + weight[u][z] // Lower overestimate
exists.
}
// At this point, all entries of array 'D' hold tight estimates.
```

Figure 9: Dijkstra's Algorithm Pseudo code [3]

According to the Geeksforgeeks article [5] the time complexity of Dijkstra's is " $O(v^2)$ ". This is also supported by Ortega-Arranz et al [4] who state "The linear search in D for the node u with the lowest value (step 3) is performed in $O(n)$. Since it should be done n times in the worst case,

this leads to a total upper bound of $O(n + m + n^2) \approx O(n^2)$. Geeksforgeeks [5] also states that: “Dijkstra’s algorithm doesn’t work for graphs with negative weight cycles. It may give correct results for a graph with negative edges but you must allow a vertex can be visited multiple times and that version will lose its fast time complexity.

Floyd Warshall’s Algorithm

Floyd Warshall’s algorithm specialises in solving all-pairs shortest path problems for an edge weighted graph. Geekforgeeks [5] describes it as such: “We initialize the solution matrix same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered vertices $\{0, 1, 2, \dots, k-1\}$ as intermediate vertices. For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.

- 1) k is not an intermediate vertex in shortest path from i to j . We keep the value of $\text{dist}[i][j]$ as it is.
- 2) k is an intermediate vertex in shortest path from i to j . We update the value of $\text{dist}[i][j]$ as $\text{dist}[i][k] + \text{dist}[k][j]$ if $\text{dist}[i][j] > \text{dist}[i][k] + \text{dist}[k][j]$ ”

In regard to time complexity the article [5] states Floyd Warshall’s is an “ $O(v^3)$ ” time complexity. Bullinaria [3] supports this with his own description of Floyd’s algorithm: “If we are not only interested in finding the shortest path from one specific vertex to all the others, but the shortest paths between every pair of vertices, we could, of course, apply Dijkstra’s algorithm to every starting vertex. But there is actually a simpler way of doing this, known as Floyd’s algorithm. This maintains a square matrix ‘distance’ which contains the overestimates of the shortest paths between every pair of vertices, and systematically decreases the overestimates using the same shortcut idea as above. If we also wish to keep track of the routes of the shortest paths, rather than just their lengths, we simply introduce a second square matrix ‘predecessor’ to keep track of all the ‘previous vertices’.” Bullinaria then provides pseudo code for Floyd’s algorithm shown in figure 10 below.

```

// Store initial estimates and predecessors.

for ( each vertex s ) {
    for ( each vertex z ) {
        distance[s][z] = weight[s][z]
        predecessor[s][z] = s
    }
}

// Improve them by considering all possible shortcuts u.

for ( each vertex u ) {
    for ( each vertex s ) {
        for ( each vertex z ) {
            if ( distance[s][u]+distance[u][z] < distance[s][z] ) {
                distance[s][z] = distance[s][u]+distance[u][z]
                predecessor[s][z] = predecessor[u][z]
            }
        }
    }
}

```

Figure 10: Floyd Warshall's Algorithm Pseudo Code [3]

Much like Geekforgeeks, Bullinaria makes statement on the time complexity while also comparing it to Dijkstra's algorithm: "The time complexity here is clearly $O(n^3)$, since it involves three nested for loops of $O(n)$. This is the same complexity as running the $O(n^2)$ Dijkstra's algorithm once for each of the n possible starting vertices. In general, however, Floyd's algorithm will be faster than Dijkstra's, even though they are both in the same complexity class, because the former performs fewer instructions in each run through the loops. However, if the graph is sparse with $e = O(n)$, then multiple runs of Dijkstra's algorithm can be made to perform with time complexity $O(n^2 \log_2 n)$, and be faster than Floyd's algorithm."

A* Search Algorithm

The A* search algorithm is an extension of Dijkstra's which specialises in solving single-pair shortest path problems, Where Dijkstra's looks at $g(n)$ (the cost to reach the next node) and

chooses the lowest, A* includes an extra heuristic to make it more informed. Norvig and Russell [6] had this to say about A* Search: “The most widely known form of best-first search is called A* search (pronounced “A-star search”). It evaluates nodes by combining $g(n)$, the cost to reach the node, and $h(n)$, the cost to get from the node to the goal: $f(n) = g(n) + h(n)$. Since $g(n)$ gives the path cost from the start node to node n , and $h(n)$ is the estimated cost of the cheapest path from n to the goal, we have $f(n)$ = estimated cost of the cheapest solution through n . Thus, if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of $g(n) + h(n)$. It turns out that this strategy is more than just reasonable: provided that the heuristic function $h(n)$ satisfies certain conditions, A* search is both complete and optimal.”

According to Norvig and Russell however, A* search comes with conditions, they state them as being: “The first condition we require for optimality is that $h(n)$ be an admissible heuristic. An admissible heuristic is one that never overestimates the cost to reach the goal. Because $g(n)$ is the actual cost to reach n along the current path, and $f(n) = g(n) + h(n)$, we have as an immediate consequence that $f(n)$ never overestimates the true cost of a solution along the current path through n .” As well as: “A second, slightly stronger condition called consistency (or sometimes monotonicity) is required only for applications of A* to graph search. A heuristic $h(n)$ is consistent if, for every node n and every successor n' of n generated by any action a , the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n' .” According to a Geeksforgeeks article on A* Search [8] one of A* search’s limitations is: “Although being the best path finding algorithm around, A* Search Algorithm doesn’t produce the shortest path always, as it relies heavily on heuristics / approximations to calculate h .” In terms of calculating h the A* search article from Geeksforgeeks describes multiple methods but the main ones to focus on are the approximation heuristics as they are the least time consuming, the article states the following: “There are generally three approximation heuristics to calculate h – 1) Manhattan Distance – It is nothing but the sum of absolute values of differences in the goal’s x and y coordinates and the current cell’s x and y coordinates respectively. When to use this heuristic? – When we are allowed to move

only in four directions only (right, left, top, bottom).” The figure below (figure 11) is the figure provided for example.

```
h = abs (current_cell.x - goal.x) +  
    abs (current_cell.y - goal.y)
```

Figure 11: Manhattan Distance [8]

Continuing on within the article it states the next approximation heuristic as: “Diagonal Distance- It is nothing but the maximum of absolute values of differences in the goal’s x and y coordinates and the current cell’s x and y coordinates respectively. When to use this heuristic? – When we are allowed to move in eight directions only (similar to a move of a King in Chess).” The figure below (figure 12) is the figure provided for example with the add statement for context: “where D is length of each node(usually = 1) and D2 is diagonal distance between each node (usually = $\sqrt{2}$).”

```
dx = abs(current_cell.x - goal.x)  
dy = abs(current_cell.y - goal.y)  
  
h = D * (dx + dy) + (D2 - 2 * D) * min(dx, dy)
```

Figure 12: Diagonal Distance [8]

Lastly the article describes the third type of approximation heuristic: “Euclidean Distance- As it is clear from its name, it is nothing but the distance between the current cell and the goal cell using the distance formula. When to use this heuristic? – When we are allowed to move in any directions.” The figure below (figure 13) is the figure provided for example.

```
h = sqrt ( (current_cell.x - goal.x)2 +  
           (current_cell.y - goal.y)2 )
```

Figure 13: Euclidean Distance [8]

METHODOLOGY

The research methodology used was Qualitative research, this reason this method was used was because when it comes to shortest path solving and pathfinding algorithms there is already a large plethora of corroborating information on the main algorithms that have been around for decades now and are still incredibly used to this day. Thus, due to this, a greater emphasis on further understanding and the already present and popular algorithms held priority over attempting to look at potential and experimental new variations that are being tinkered with today. The advantages of this choice with this project were that because the goal was to be able to implement and compare these older more popular algorithms a greater understanding with their roots and inner workings assists greatly.

In terms of evaluating these algorithms, the criteria's disclosed from Norvig and Russell [6] from earlier will be used, these being: Completeness, optimality, time complexity and space complexity. In order to learn these, the diagnostic tools from visual studio shall be used.

For the development method, the chosen method shall be the Waterfall method. This reason for this choice is because it sets easy to comprehend step by step goals to work through. The advantages of this chosen method are as stated earlier, it is easy to comprehend, also it allows for a more focussed mind on objectives as a task is only moved on from once completed. The disadvantage to this method however is that it can be rigid to go back to a previous objective if required.

The software used to develop this program will be visual studio, this is due to visual studio being an extensive and easy to use compiler with sets of diagnostic tools in order to test performance. Visual studio has the additional benefit of having a free student licence to be used by students at the university. Other than visual studio acting as the compiler for this project no other software is required to execute this project aims and objectives. In terms of coding the language used will be C++, this is due to the language being the most taught to the author therefore they are most familiar with said language.

DESIGN, IMPLEMENTATION AND RESULTS

In order to showcase the pathfinding ability of certain algorithms, they will be used to solve a simple maze, the maze in question to be solved is one grabbed from online [9] which looks like this:

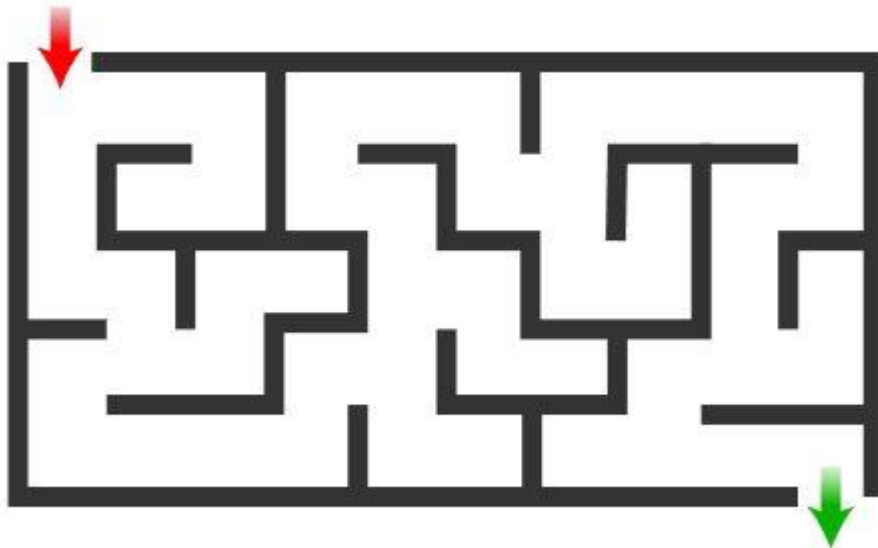


Figure 14: Simple Maze [9]

First the maze needed to be turned into something able to be ran through an algorithm and solved by a computer. Therefore, the maze was turned into a graph equivalent where each node is a corner and the edge's distance is how many "squares" in between each node. The end product looked like this:

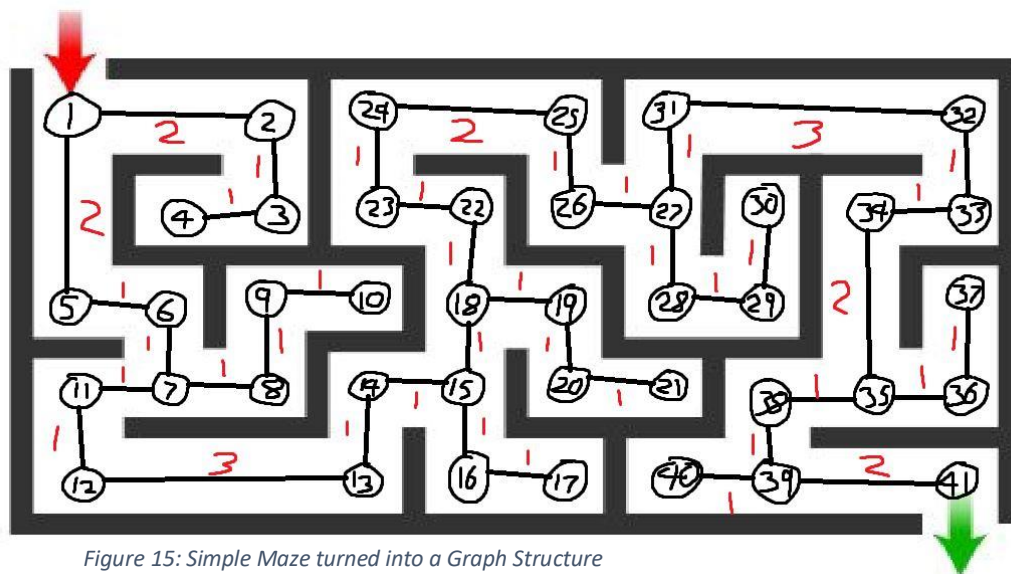


Figure 15: Simple Maze turned into a Graph Structure

Now that the maze could be interpreted as undirected weighted graph it was able to be made into an adjacency matrix:

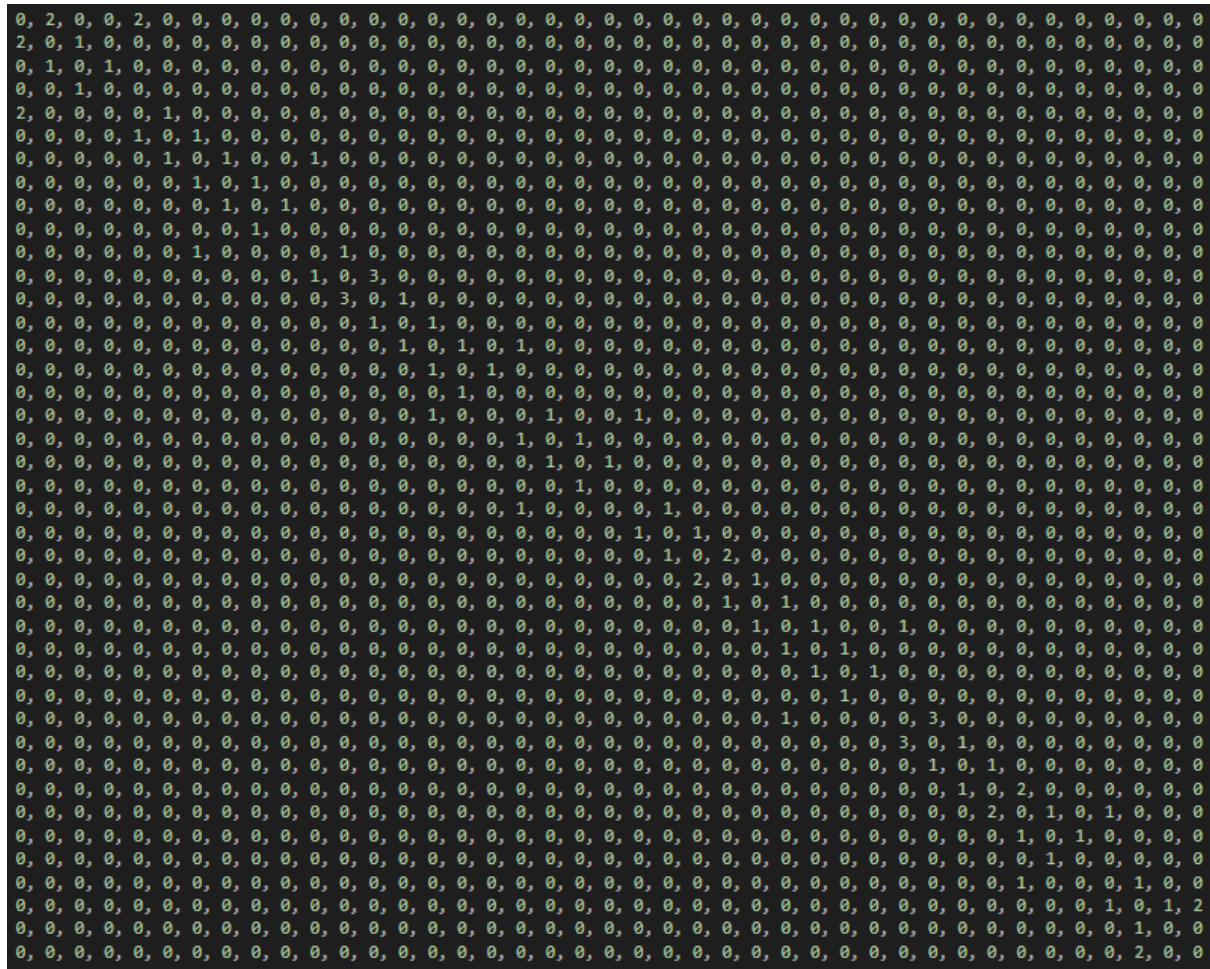


Figure 16: Adjacency matrix of Simple maze

This adjacency matrix was then placed into a Dijkstra's algorithm implementation.

The following figure (figure 17) is the results of Dijkstra's algorithm:

```

Vertex Distance from Source
1 <- ::::Distance = 0
2 <- 1 <- ::::Distance = 2
3 <- 2 <- 1 <- ::::Distance = 3
4 <- 3 <- 2 <- 1 <- ::::Distance = 4
5 <- 1 <- ::::Distance = 2
6 <- 5 <- 1 <- ::::Distance = 3
7 <- 6 <- 5 <- 1 <- ::::Distance = 4
8 <- 7 <- 6 <- 5 <- 1 <- ::::Distance = 5
9 <- 8 <- 7 <- 6 <- 5 <- 1 <- ::::Distance = 6
10 <- 9 <- 8 <- 7 <- 6 <- 5 <- 1 <- ::::Distance = 7
11 <- 7 <- 6 <- 5 <- 1 <- ::::Distance = 5
12 <- 11 <- 7 <- 6 <- 5 <- 1 <- ::::Distance = 6
13 <- 12 <- 11 <- 7 <- 6 <- 5 <- 1 <- ::::Distance = 9
14 <- 13 <- 12 <- 11 <- 7 <- 6 <- 5 <- 1 <- ::::Distance = 10
15 <- 14 <- 13 <- 12 <- 11 <- 7 <- 6 <- 5 <- 1 <- ::::Distance = 11
16 <- 15 <- 14 <- 13 <- 12 <- 11 <- 7 <- 6 <- 5 <- 1 <- ::::Distance = 12
17 <- 16 <- 15 <- 14 <- 13 <- 12 <- 11 <- 7 <- 6 <- 5 <- 1 <- ::::Distance = 13
18 <- 15 <- 14 <- 13 <- 12 <- 11 <- 7 <- 6 <- 5 <- 1 <- ::::Distance = 12
19 <- 18 <- 15 <- 14 <- 13 <- 12 <- 11 <- 7 <- 6 <- 5 <- 1 <- ::::Distance = 13
20 <- 19 <- 18 <- 15 <- 14 <- 13 <- 12 <- 11 <- 7 <- 6 <- 5 <- 1 <- ::::Distance = 14
21 <- 20 <- 19 <- 18 <- 15 <- 14 <- 13 <- 12 <- 11 <- 7 <- 6 <- 5 <- 1 <- ::::Distance = 15
22 <- 18 <- 15 <- 14 <- 13 <- 12 <- 11 <- 7 <- 6 <- 5 <- 1 <- ::::Distance = 13
23 <- 22 <- 18 <- 15 <- 14 <- 13 <- 12 <- 11 <- 7 <- 6 <- 5 <- 1 <- ::::Distance = 14
24 <- 23 <- 22 <- 18 <- 15 <- 14 <- 13 <- 12 <- 11 <- 7 <- 6 <- 5 <- 1 <- ::::Distance = 15
25 <- 24 <- 23 <- 22 <- 18 <- 15 <- 14 <- 13 <- 12 <- 11 <- 7 <- 6 <- 5 <- 1 <- ::::Distance = 17
26 <- 25 <- 24 <- 23 <- 22 <- 18 <- 15 <- 14 <- 13 <- 12 <- 11 <- 7 <- 6 <- 5 <- 1 <- ::::Distance = 18
27 <- 26 <- 25 <- 24 <- 23 <- 22 <- 18 <- 15 <- 14 <- 13 <- 12 <- 11 <- 7 <- 6 <- 5 <- 1 <- ::::Distance = 19
28 <- 27 <- 26 <- 25 <- 24 <- 23 <- 22 <- 18 <- 15 <- 14 <- 13 <- 12 <- 11 <- 7 <- 6 <- 5 <- 1 <- ::::Distance = 20
29 <- 28 <- 27 <- 26 <- 25 <- 24 <- 23 <- 22 <- 18 <- 15 <- 14 <- 13 <- 12 <- 11 <- 7 <- 6 <- 5 <- 1 <- ::::Distance = 21
30 <- 29 <- 28 <- 27 <- 26 <- 25 <- 24 <- 23 <- 22 <- 18 <- 15 <- 14 <- 13 <- 12 <- 11 <- 7 <- 6 <- 5 <- 1 <- ::::Distance = 22
31 <- 27 <- 26 <- 25 <- 24 <- 23 <- 22 <- 18 <- 15 <- 14 <- 13 <- 12 <- 11 <- 7 <- 6 <- 5 <- 1 <- ::::Distance = 20
32 <- 31 <- 27 <- 26 <- 25 <- 24 <- 23 <- 22 <- 18 <- 15 <- 14 <- 13 <- 12 <- 11 <- 7 <- 6 <- 5 <- 1 <- ::::Distance = 23
33 <- 32 <- 31 <- 27 <- 26 <- 25 <- 24 <- 23 <- 22 <- 18 <- 15 <- 14 <- 13 <- 12 <- 11 <- 7 <- 6 <- 5 <- 1 <- ::::Distance = 24
34 <- 33 <- 32 <- 31 <- 27 <- 26 <- 25 <- 24 <- 23 <- 22 <- 18 <- 15 <- 14 <- 13 <- 12 <- 11 <- 7 <- 6 <- 5 <- 1 <- ::::Distance = 25
35 <- 34 <- 33 <- 32 <- 31 <- 27 <- 26 <- 25 <- 24 <- 23 <- 22 <- 18 <- 15 <- 14 <- 13 <- 12 <- 11 <- 7 <- 6 <- 5 <- 1 <- ::::Distance = 27
36 <- 35 <- 34 <- 33 <- 32 <- 31 <- 27 <- 26 <- 25 <- 24 <- 23 <- 22 <- 18 <- 15 <- 14 <- 13 <- 12 <- 11 <- 7 <- 6 <- 5 <- 1 <- ::::Distance = 28
37 <- 36 <- 35 <- 34 <- 33 <- 32 <- 31 <- 27 <- 26 <- 25 <- 24 <- 23 <- 22 <- 18 <- 15 <- 14 <- 13 <- 12 <- 11 <- 7 <- 6 <- 5 <- 1 <- ::::Distance = 29
38 <- 35 <- 34 <- 33 <- 32 <- 31 <- 27 <- 26 <- 25 <- 24 <- 23 <- 22 <- 18 <- 15 <- 14 <- 13 <- 12 <- 11 <- 7 <- 6 <- 5 <- 1 <- ::::Distance = 28
39 <- 38 <- 35 <- 34 <- 33 <- 32 <- 31 <- 27 <- 26 <- 25 <- 24 <- 23 <- 22 <- 18 <- 15 <- 14 <- 13 <- 12 <- 11 <- 7 <- 6 <- 5 <- 1 <- ::::Distance = 29
40 <- 39 <- 38 <- 35 <- 34 <- 33 <- 32 <- 31 <- 27 <- 26 <- 25 <- 24 <- 23 <- 22 <- 18 <- 15 <- 14 <- 13 <- 12 <- 11 <- 7 <- 6 <- 5 <- 1 <- ::::Distance = 30
41 <- 39 <- 38 <- 35 <- 34 <- 33 <- 32 <- 31 <- 27 <- 26 <- 25 <- 24 <- 23 <- 22 <- 18 <- 15 <- 14 <- 13 <- 12 <- 11 <- 7 <- 6 <- 5 <- 1 <- ::::Distance = 31

```

Figure 17: Dijkstra's Results

Due to Dijkstra's algorithm being a single-source pathfinding solution it returned the shortest from the source to all other nodes. You could however set a goal node and have the loop break once goal node's shortest path has been found if you wanted this to only solve single-pair solutions. There is a decent argument to do this for this implementation due to mazes typically only having one exit, but the author decided it was better to showcase Dijkstra's full capability instead. As for whether the implementation was successful, we can check the path given to the exit node which in this mazes case is node 41. Now as this is a simple maze as well as a perfect maze (meaning there is only one solution) it's quite easy to see what the solution of the shortest path is by looking at it, below is a figure highlighting the solution (figure 18):

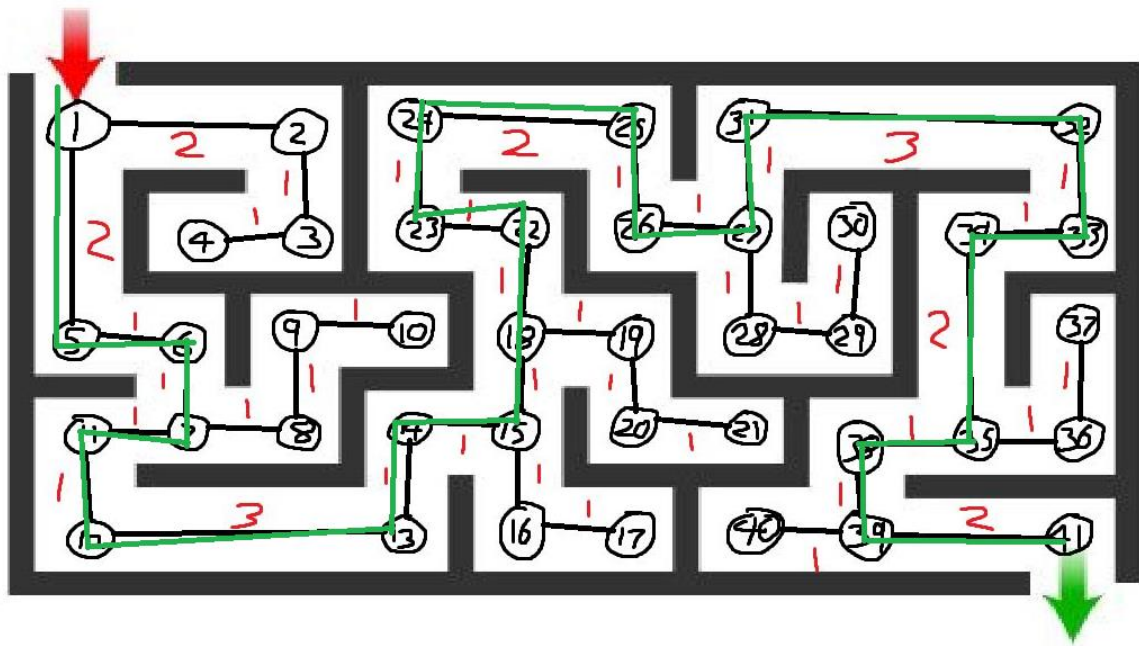


Figure 18: Simple Maze Solution

In written terms the path is (1, 5, 6, 7, 11, 12, 13, 14, 15, 18, 22, 23, 24, 25, 26, 27, 31, 32, 33, 34, 35, 38, 39, 41) with a total distance of $2 + 1 + 1 + 1 + 1 + 3 + 1 + 1 + 1 + 1 + 1 + 1 + 2 + 1 + 1 + 1 + 3 + 1 + 1 + 2 + 1 + 1 + 2 = 31$, which we can see from figure 17 that the algorithm implementation had returned the correct shortest path. In terms of time elapsed, the program with 41 nodes took 40ms as well as using up ≈ 900 KBs and shown in figure 19:

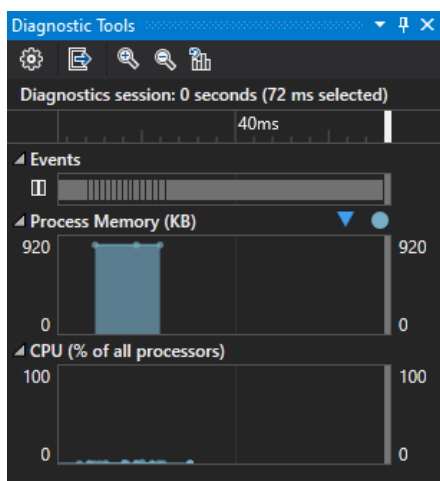


Figure 19: Dijkstra's run diagnostics

For implementing A* a heuristic must be created, the obvious choice of heuristic here is the Manhattan distance heuristic due to it being mainly suited for movement only allowed in the four

core directions, which the Maze used happens to have this restriction. A problem though is that the Manhattan distance formula requires coordinates, in order to turn these nodes to coordinates the maze must be turned into a grid. The following figure (figure 20) is the maze with a makeshift grid and axis placed over it.

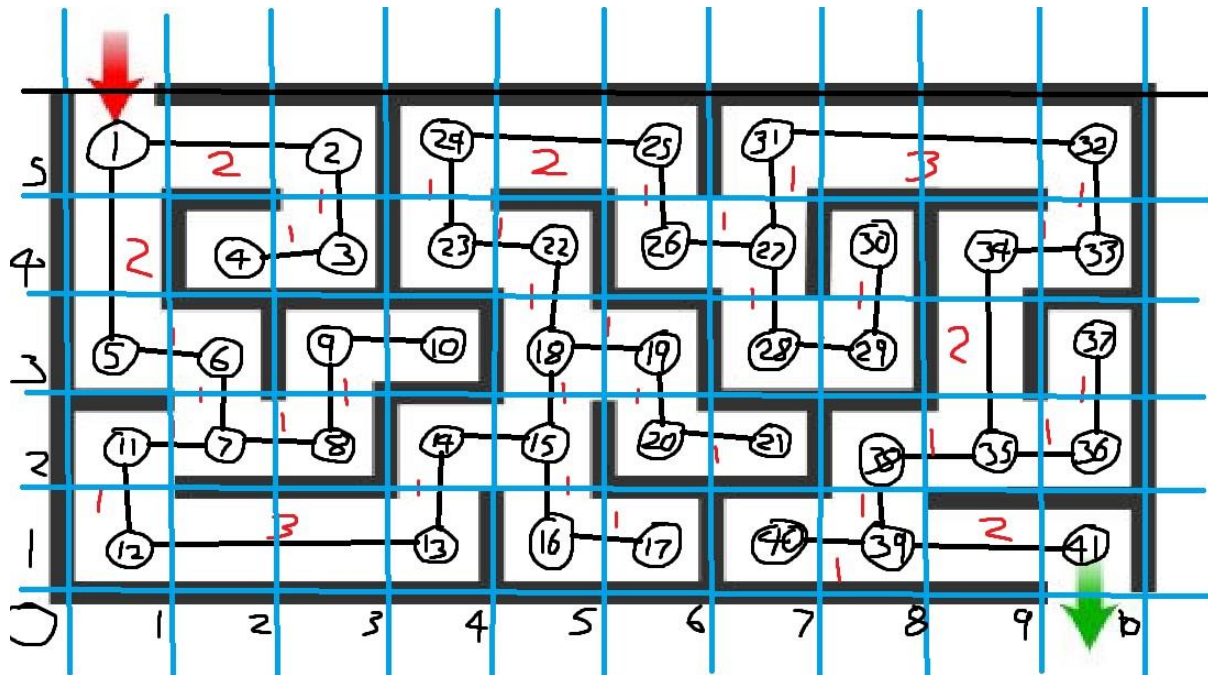


Figure 20: Simple maze grid

Using this grid you can assign a coordinate to each node, for example node 1 would have the coordinates (1, 5) or node 22 which would have the coordinates (5, 4). With this you can now give each node a heuristic using the Manhattan distance method shown in figure 11. Using node 22 still, it would gain a $h(n)$ of 8, therefore the $f(n)$ to move to node 22 from node 18 would be $1 + 8 = 9$. With this you can expand upon the Dijkstra's algorithm by including a table to contain each node's respective coordinates and inputting a goal node you can automatically calculate their $h(n)$ values. Thus, by adding the $h(n)$ values to the $g(n)$ values already checked by Dijkstra's you can get the new $f(n)$ values which are informed. Lastly by including a goal node for which the algorithm to cease operations once it has reached it, you have effectively converted Dijkstra's to A* search. Unfortunately due to the authors own inability they were unable to fully translate this to a working code implementation to be able to test and evaluate, though they still stand by their proposed method as being capable of implementing A* search.

EVALUATION

In terms of meeting the objectives set out at the beginning, some were met, some were not. The objective to find a way to convert a maze into a format that could be used within code was met. It was learnt that mazes could be represented by an undirected weighted graph, and then from there converted into an adjacency matrix which could then be placed into an array with the code. The second objective to discover types of graph traversal and search algorithms was met. Methods such as the Breadth-first search, Depth-first search, and Best-first search were learnt. The third objective of learning how Dijkstra's algorithm works and to implement it solving a maze was met, the author understands Dijkstra's well and was capable of creating a program that successfully solved the maze using Dijkstra's. The last objective of learning how A* search works and to implement it solving a maze was not met, while the author does understand A* search and can work it out manually on paper using a graph or grid, they were unable to translate it into working code. This unfortunately meant no comparison of maze solving between the two algorithms could be made and thus by proxy, the aim of comparing and implementing AI pathfinding methods was not met either.

CONCLUSIONS AND RECOMMENDATIONS

In summary, it was learnt that there are four types of shortest path problems: Single – pair, Single – source, Single – destination, and All – pairs. Single – pair is the shortest path between a source node and a destination node, Single – source is the shortest path between a source node and all other nodes with Source – destination being essentially the same but flipped for the paths to a destination node instead, and All – pairs being the shortest path between every pair of nodes. It was then learnt that graphs follow the following structures: Directed, undirected, weighted, and unweighted. It was also discovered that graphs can be used to represent networks and can be converted into a data structure called an adjacency matrix and an adjacency list. Next it was discovered that search algorithms have criteria in which they are judged being completeness, optimality, time complexity and space complexity. It was also learnt that these search algorithms come in two forms: uninformed and informed. Examples of uninformed are Breadth-first search and Depth-first search whereas informed is that of Best-first search. Dijkstra's algorithm was learnt with the aspect of it being a single-source pathfinding algorithm that can be made into single-pair but it's not its speciality. A* search was learnt to be an extension of Dijkstra's which uses a heuristic function to allow it to make more informed decisions, it also more favours single-pair problems.

When designing the programs, a simple maze was taken from online and then manually converted into an appropriate graph structure which was then converted into an adjacency matrix for implementation into an array with the code which was enough to allow a working Dijkstra's algorithm solution. When it came to A* however the maze/graph also needed to be converted into a makeshift grid in order for the nodes to have coordinates, as coordinates were required in calculating the heuristics. Though in the end a working A* program was unable to be created.

If recommendations could be made for future attempts at this project or something similar, they are as follows: Instead of manually grabbing a maze and converting it to a graph, perhaps a form of maze generation would be a more elegant way of carrying this out, this way the maze could be implemented to have different sizes and variety, in doing so would allow for a more vigorous test

on how much better or worse the algorithms get with more or less nodes, enabling a stronger comparison.

REFLECTION

Through doing this project I learnt of how integrated aspect of finding the shortest path is in daily life, satellite navigation, video games, flights, self-driving cars, they all use forms of pathfinding methods and as technology becomes more automated, pathfinding methods will become more used and crucial. I also discovered gaps in my knowledge when it came to coding, especially in C++, I was unable to find a to get a working implementation of an A* search algorithm and cost me dearly when it came to completing my aims and objectives. From what I could see when doing my research on these pathfinding algorithms a lot of other programmer would usually lean towards python to code them, I don't know whether that's because the language is better at performing these task though, it did seem simpler at a glance however so maybe that's why, because of this I think for future projects attempting coding for AI aspects I will attempt to learn python and use that as my language of choice. I am pleased with how I converted the maze into a graph and then into an adjacency matrix but I can see how it would be better to create some kind of automated Maze generation in which you can alter its size before creation, as that should allow for a better variety of tests as with just one sized maze it's hard to judge how much better or worse and algorithm is in this scenario since an algorithm could be more efficient than another when there are a small amount of nodes but become worse than the other after a certain threshold.

REFERENCES

- [1] A. Singhal, "gatevidyalay," [Online]. Available: <https://www.gatevidyalay.com/shortest-path-algorithms-shortest-path-problems/>. [Accessed 5 January 2022].
- [2] "Shortest Path Algorithms," Brilliant.org, [Online]. Available: <https://brilliant.org/wiki/shortest-path-algorithms/#types-of-shortest-path-algorithms>. [Accessed 5 January 2022].
- [3] J. Bullinaria, "Data Structures and Algorithms," Birmingham, 2018.
- [4] H. Ortega-Arranz, D. R. Llanos and A. Gonzalez-Escribano, The Shortest-Path Problem, Morgan & Claypool, 2014.
- [5] "Geeksforgeeks," [Online]. Available: <https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/>. [Accessed 12 January 2022].
- [6] S. J. R. a. P. Norvig, Artificial Intelligence A Modern Approach Third Edition, Pearson, 2009.
- [7] G. Baranett and L. D. Tongo, Data Structures and Algorithms: Annotated Reference with Examples First Edition, 2008.
- [8] "Geeksforgeeks," [Online]. Available: <https://www.geeksforgeeks.org/a-search-algorithm/?ref=gcse>. [Accessed 7 February 2022].
- [9] [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/8/88/Maze_simple.svg.

APPENDIX

Dijkstra's Code:

```
#include <iostream>
#include <limits.h>
#define V 41 //Number of vertices in the graph
using namespace std;

int minDist(int dist[], bool shortestSet[]) // Function to find the vertex with the
minimum distance from the vertices not in the shortest set
{
    //Initialises the minimum value
    int min = INT_MAX; //Sets min as a very large number to represent infinity
    int minIndex;

    for (int i = 0; i < V; i++)
    {
        if (shortestSet[i] == false && dist[i] <= min)
        {
            min = dist[i], minIndex = i;
        }
    }
    return minIndex;
}

void Display(int dist[], int parent[]) // Function to print the results
{
    cout << "Vertex \t Distance from Source" << endl;
    for (int i = 0; i < V; i++)
    {
        int temp = parent[i];
        cout << i + 1 << " <- " ;
        while (temp != -1)
        {
            cout << temp + 1 << " <- ";
            temp = parent[temp];
        }
        cout << "::::Distance = " << dist[i];
        cout << endl;
    }
}

void dijkstra(int graph[V][V], int source) //runs dijkstra's as a single source
problem solution for an adjacency matrix representing a graph converted from the maze
{
    int dist[V]; //Output array
    int parent[V]; //Parent Array
    bool shortestSet[V]; // shortestSet set as True

    for (int i = 0; i < V; i++)
    {
        dist[i] = INT_MAX, shortestSet[i] = false; //initialises the distances
as infinite
    }
    dist[source] = 0; //distance from source is 0
    parent[source] = -1; //Parent node is technically 1 behind the source to start
with
    for (int count = 0; count < V - 1; count++) //Find shortest path of all
vertices
```