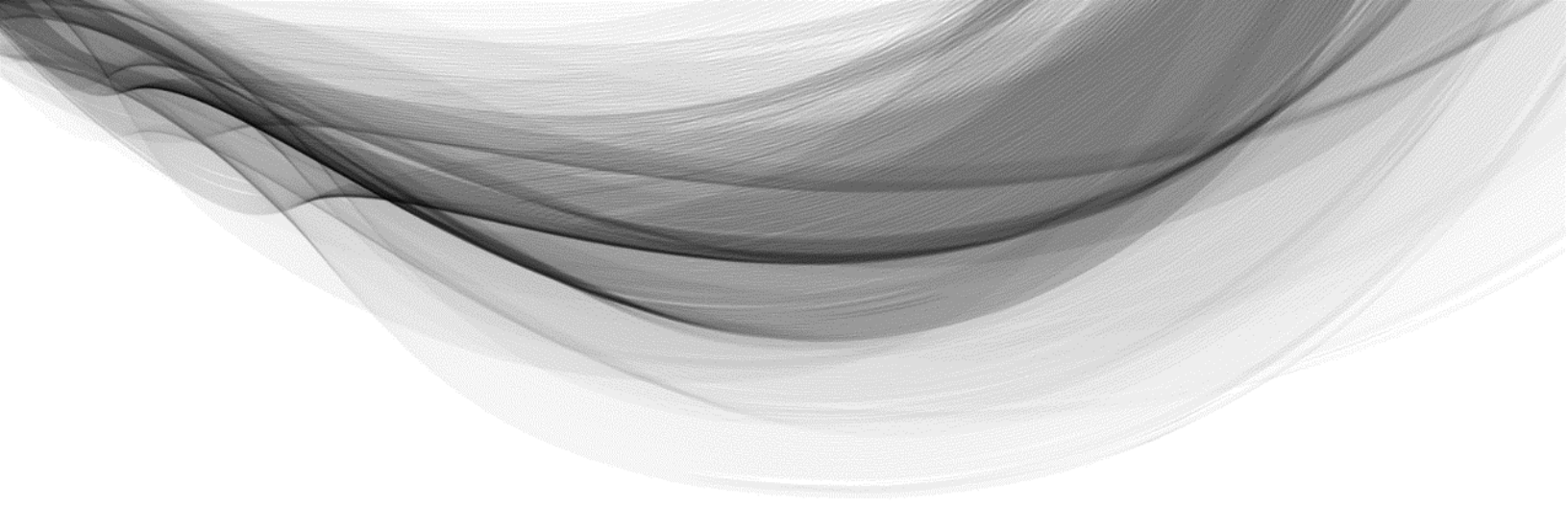




Maze Pathfinding Implementation

L5 – Advanced Software Engineering

Module Code: ACCB5011

BSc Computing (Software Engineering)

Contents

1. Introduction	3
2. Dijkstra's Algorithm and the A* Algorithm	3
2.1. Dijkstra's Algorithm	3
2.2. The A* Algorithm	3
2.3. Comparing Dijkstra's Algorithm and the A* Algorithm	4
3. Implementation of Maze-Based Solutions	4
3.1 Understanding the Problem and Constraints	4
3.2 Algorithm Choice and Design Decisions	5
3.3 Implementation Details	5
3.4 Results	7

1. Introduction

Pathfinding algorithms play an important role in the world of computer science, enabling navigation through complex environments such as digital maps, games and networks [1]. This report examines two of the most widely used pathfinding algorithms, Dijkstra's Algorithm and the A* Algorithm, discussing how they function and comparing the two's performance within a grid-based, maze scenario. The second part of this report reflects on the implementation of my own maze solving solution, including the design process, challenges and its performance.

2. Dijkstra's Algorithm and the A* Algorithm

This section introduces Dijkstra's Algorithm and the A* Algorithm, describing how they function as well as their typical applications. They are later compared with the goal of highlighting their strengths, weaknesses, and suitability within a maze-based pathfinding scenario.

2.1 Dijkstra's Algorithm

Dijkstra's Algorithm is a graph-based pathfinding algorithm used to find the shortest path from a single source node to all other nodes within a weighted graph [2]. The algorithm explores nodes in order of increasing cumulative cost with the goal of ensuring the shortest possible path to each node is found. This is achieved by storing a set of unvisited nodes, each associated with a value representing their distance, and selecting the node with the lowest cost [3].

At the start of its execution, the source node is assigned a value of zero, while all other nodes are given a distance value representing infinity. As the algorithm progresses, adjacent nodes are examined, and their distance values updated whenever a shorter path is identified. Once a node's shortest path has been found, it is marked as visited and removed, meaning it shall no longer be considered moving forward. This process continues until the destination node has been reached or all other nodes within reach have been explored [3].

A key advantage of Dijkstra's Algorithm is that it guarantees that the shortest path to the destination node (when all edges have a non-negative cost) [4]. This makes the algorithm suitable for a wide range of routing and navigational problems. However, due to the fact Dijkstra's Algorithm does not provide any guidance towards the destination node, it may also explore a large number of unnecessary nodes. This can reduce its overall efficiency in grid-based environments such as the maze scenario, where the target destination is already known.

Despite this limitation Dijkstra's Algorithm remains widely used due to its simplicity and guarantee of finding a shortest path. These characteristics make it a useful baseline for comparison against a more complex algorithm such as the A* Algorithm.

2.2 The A* Algorithm

The A* Algorithm is an informed pathfinding algorithm that works as an extension of Dijkstra's Algorithm. A key difference between the two is A*'s use of heuristic information, guiding its search towards a specific target node [4]. Rather than relying solely on cumulative cost travelled so far, A* evaluates nodes based on a combination of a cost function that accounts for both the known path cost, and an estimated remaining distance to its goal. This allows for the algorithm to prioritise paths that have a higher likelihood to reach its destination node at a lower cost.

Within A*, each node is assigned a value defined by the function $f(n) = g(n) + h(n)$, where:

- $f(n)$ represents the total estimated cost of a path via node n . Nodes with the lowest value are explored first.
- $g(n)$ represents the number of moves made so far.
- $h(n)$ represents the estimated number of moves remaining.

When the heuristic function $h(n)$ never overestimates the true remaining cost to the target, A* is guaranteed to find the shortest path to its destination node [5].

One drawback of A* is its dependence on the quality of the heuristic information it is provided. Poorly chosen heuristics may reduce its effectiveness and lead to behavior characteristic of uninformed search algorithms. Despite this dependency, the A* Algorithm remains widely used within major platforms such as Google Maps, utilizing it for applications such as finding the fastest route to a destination [6].

2.3 Comparing Dijkstra's Algorithm and the A* Algorithm

While Dijkstra's Algorithm and the A* Algorithm both share a common foundation, using cumulative costs to find a shortest path, they differ greatly in their approach to finding a shortest path.

Dijkstra's Algorithm guarantees an optimal solution by expanding nodes in order of increasing cumulative cost, excelling within scenarios where no specific destination node is provided. However, because the algorithm does not incorporate any heuristic guidance, nodes are explored uniformly in all directions. This can result in Dijkstra's Algorithm examining nodes that do not directly contribute to reaching a target in an efficient manner.

In contrast, the A* Algorithm improves upon Dijkstra's Algorithm by incorporating heuristic information. By combining factors such as estimated remaining distance and cumulative cost so far, A* is able to prioritise more promising paths. However, when the heuristic function overestimates the true remaining cost, the algorithm may prioritise less efficient routes. This leads to increased node exploration and decreased efficiency. In such a case, the performance of A* over Dijkstra's Algorithm is reduced.

In the case of the maze-based scenario, Dijkstra's Algorithm can be expected to explore a larger portion of the maze before reaching its destination node. In comparison, A* can be expected to direct its search towards the target, leading to fewer explored nodes and a faster solution. Using this information, it can be deduced that A* is better suited for maze-based pathfinding where a target destination is known.

3. Implementation of Maze-Based Solutions

This section of the report covers my own implementation of a pathfinding solution for the provided maze-based scenario, aiming to showcase the design decisions, reasoning, and development process behind the final solution.

3.1 Understanding the Problem and Constraints

Before implementing a solution, I first separated the scenario into two categories, one containing information relating to the problem itself, and the other describing constraints.

Problem Overview:

- The maze consists of a 20 x 20 grid with cells containing either a player, a target, a wall, or empty space.
- The player can move one cell at a time in directions, left, right, up, and down.
- The objective is to reach the target from the starting position.
- Both the maze and individual cells can be viewed.
- The maze is randomly generated but always solvable.

Constraints Overview:

- Only the `GridNavigatorAutomatic::Navigate()` method may be modified.
- Player movement must be performed using the provided methods.
- Collisions with walls must be avoided.

3.2 Algorithm Choice and Design Decisions

Based on the problem requirements and constraints, Dijkstra's Algorithm was selected. This was due to its guarantee of finding the shortest path, keeping the number of grid moves as low as possible. While the A* Algorithm was considered, its dependency on quality heuristics introduces additional complexity and potential sources for error.

3.3 Implementation Details

This section describes how Dijkstra's Algorithm was implemented within the constraints of the provided framework.

Finding Player and Target Positions:

```
21 // fallback for player and target nodes in the case they change
22 bool foundPlayer = (grid->GetGridSquare(startX, startY) == griditem::player);
23 bool foundGoal = (grid->GetGridSquare(goalX, goalY) == griditem::target);
24
25 if (!foundPlayer || !foundGoal)
26 {
27     for (int x = 0; x < gridSize; x++)
28     {
29         for (int y = 0; y < gridSize; y++)
30         {
31             griditem cell = grid->GetGridSquare(x, y);
32
33             if (!foundPlayer && cell == griditem::player)
34             {
35                 startX = x;
36                 startY = y;
37                 foundPlayer = true;
38             }
39             else if (!foundGoal && cell == griditem::target)
40             {
41                 goalX = x;
42                 goalY = y;
43                 foundGoal = true;
44             }
45         }
46     }
47     if (foundGoal && foundPlayer)
48         break;
49 }
50 }
```

[Figure. 1]

Although the framework provides fixed coordinates for the player and target starting positions, the implementation scans the grid for both nodes in the event that the coordinates are changed.

Data Structure:

```
72 // initialize nodes
73 for (int y = 0; y < gridSize; y++)
74 {
75     for (int x = 0; x < gridSize; x++)
76     {
77         cost[x][y] = inf;
78         closed[x][y] = false;
79         inOpen[x][y] = false;
80         parentX[x][y] = -1;
81         parentY[x][y] = -1;
82     }
83 }
```

[Figure. 2]

Figure 2 shows the initialization of all nodes within the grid, assigning data such as cost, visitation status and parent coordinates via the use of a nested loop.

Core Pathfinding Logic:

```
120 // check adjacent cells and update shortest path
121 // check up
122 {
123     int adjacentX = current.x;
124     int adjacentY = current.y - 1;
125     movedirection dir = movedirection::up;
126
127     if (adjacentX < 0 || adjacentX >= gridSize || adjacentY < 0 || adjacentY >= gridSize) // stop if outside adjacent cell outside of grid
128     {
129     }
130     else if (closed[adjacentX][adjacentY]) // stop if cell is already finalized
131     {
132     }
133     else if (grid->GetGridSquare(adjacentX, adjacentY) == griditem::wall) // stop if cell is a wall
134     {
135     }
136     else
137     {
138         int totalCost = cost[current.x][current.y] + 1; // increase cumulative cost by 1
139
140         // update cell if cheaper path found
141         if (totalCost < cost[adjacentX][adjacentY])
142         {
143             cost[adjacentX][adjacentY] = totalCost;
144             parentX[adjacentX][adjacentY] = current.x;
145             parentY[adjacentX][adjacentY] = current.y;
146             parentMove[adjacentX][adjacentY] = dir;
147
148             // if cell not in open list, add node
149             if (!inOpen[adjacentX][adjacentY])
150             {
151                 open.push_back({adjacentX, adjacentY, totalCost});
152                 inOpen[adjacentX][adjacentY] = true;
153             }
154             else
155             {
156                 for (auto &n : open)
157                 {
158                     if (n.x == adjacentX && n.y == adjacentY)
159                     {
160                         n.cost = totalCost;
161                         break;
162                     }
163                 }
164             }
165         }
166     }
167 }
```

[Figure. 3]

Figure 3 contains the logic regarding adjacent cells. A series of conditional if statements first check if the selected adjacent cell is valid, if so, its cumulative cost is increased by 1, representing the cost of a single move within the grid. If the current path to reach the cell is found to be the cheapest, the cost value, parent coordinates, and movement

direction are updated accordingly. If the cell has not yet been added to the open list, it is inserted for future evaluation. This block is repeated three more times in order to check all four directions of travel.

Path Reconstruction:

```
315 // reconstruct path
316 std::vector<movedirection> path;
317 int x = goalX;
318 int y = goalY;
319
320 // follow parent pointers from goal to start and store moves taken
321 while (!(x == startX && y == startY))
322 {
323     path.push_back(parentMove[x][y]);
324     int px = parentX[x][y];
325     int py = parentY[x][y];
326     x = px;
327     y = py;
328 }
329
330 // reverse path from goal -> start to start -> goal
331 std::reverse(path.begin(), path.end());
332
333 // execute path
334 for (movedirection dir : path)
335 {
336     moveresults r = grid->MovePlayer(dir);
337     if (r == moveresults::complete)
338         break;
339 }
340 }
```

[Figure. 4]

Figure 4 depicts the logic used to reconstruct the shortest path. A while loop is first used to backtrack through the maze from the goal position, pushing the movement direction taken at each step to a vector. This list of move directions is then reversed so that it now travels from the start to the goal. A final for loop is then used to execute these moves until reaching the target cell.

3.4 Results

The benchmark results of ten thousand attempts demonstrates the pathfinding algorithm's consistent performance across multiple runs, with the range of accesses being 236 with a mean of 513. While this is close to a target of 400 (representing every cell within the grid), this number could have been reduced further by scanning the grid once completely at the start of a run, and then saving the results to memory.

The efficiency of movement varied more significantly with the minimum and maximum number of moves being 34 and 90 respectively. The mean however remained on the lower end at 47. Importantly there were no wall hits recorded in any attempts, demonstrating the algorithm's reliability in respect to obstacle collision.

```
Minimum accesses: 353
Mean accesses: 513
Maximum accesses: 589
Minimum moves: 34
Mean moves: 47
Maximum moves: 90
Minimum wall hits: 0
Mean wall hits: 0
Maximum wall hits: 0
```

[Figure. 5]

8. References

- [1] "Use-Cases Of The Shortest Path Algorithm," *Memgraph.com*, 2022. <https://memgraph.com/blog/use-cases-of-the-shortest-path-algorithm> (Accessed Dec. 2, 2025)
- [2] D. Fan and P. Shi, "Improvement of Dijkstra's algorithm and its application in route planning," *IEEE Xplore*, Aug. 01, 2010. <https://ieeexplore.ieee.org/abstract/document/5569452> (Accessed Dec. 5, 2025)
- [3] D. Rachmawati and L. Gustin, "Analysis of Dijkstra's Algorithm and A* Algorithm in Shortest Path Problem," *Journal of Physics: Conference Series*, vol. 1566, p. 012061, Jun. 2020, doi: <https://doi.org/10.1088/1742-6596/1566/1/012061> (Accessed Dec. 5, 2025)
- [4] A. Patel, "Introduction to A*," *theory.stanford.edu*, 1997. <https://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html> (Accessed Dec. 5, 2025)
- [5] J. Yao, C. Lin, X. Xie, A. J. Wang, and C.-C. Hung, "Path Planning for Virtual Human Motion Using Improved A* Star Algorithm," *IEEE Xplore*, Apr. 01, 2010. <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5501636> (Accessed Dec. 6, 2025)
- [6] A. Deshmukh, "Towards AI," *Towardsai.net*, Apr. 20, 2025. <https://towardsai.net/p/artificial-intelligence/from-a-to-b-algorithms-that-power-google-maps-navigation> (Accessed Dec. 7, 2025)