

Assignment – Advanced Software Development

Ben Roberts (2205382)

Advanced Software Development (FT - ACCB5011)

University of Wales Trinity Saint David – School of Applied Computing

Table of Contents

Introduction.....3

Task 1 – Research.....4

Task 2 – Implementation.....6

References.....9

Introduction

The purpose of this document is to compare the Dijkstra's and A* pathfinding algorithms, and to produce an example pathfinding algorithm to solve a maze.

Task 1 – Research

Dijkstra's algorithm and the A* algorithm are pathfinding algorithms which aim to provide a solution to the shortest path problem [1]. These two algorithms are greedy algorithms, which means that they use the best immediate solution to the issue [2].

Dijkstra's algorithm is a pathfinding algorithm that attempts to find the shortest point to a path between two points across a graph. It builds a minimal path from the starting node to the target by constructing a distance array for each individual point. Nodes are initially marked with a value of infinity, and then have their values reassigned based on how close they are to the start and end node. It then moves to the node with the lowest value and repeats until it reaches the end point [3]. Dijkstra's Algorithm is a breadth-first search, meaning that nodes at the top are computed first [4].

Dijkstra's algorithm uses the following steps:

1. Creates an array for each node on the graph. The distance of each node is infinity by default with the exception of the starting node, which is assigned 0 and as such is considered the current node.
2. A process called edge relaxation is performed wherein the algorithm sets a distance value based on how far each node is from the current and the targeted node. The distance value of any nodes that have to be travelled through are added.
3. The node with the lowest distance value is now considered the current node.
4. If the targeted node has not been reached, repeat the algorithm again from step 2. Otherwise, the algorithm is considered complete and can be terminated.

A* (also known as Astar [5]) is a variation of Dijkstra's algorithm which performs additional heuristics to find the shortest path [3]. Whereas Dijkstra's algorithm only concerns the physical distance between the two nodes, A* also performs an additional heuristic check based on a distance vector [3]. One distance vector that can be used is euclidean distances, which are just a straight line between the current node and the target node. The sum of these two values are then used to find what the shortest path is.

The A* algorithm uses the following steps:

1. Creates two different arrays – one for open (unevaluated) and closed (evaluated) nodes. Add the starting node to the open list.
2. Add the current node to the closed array, then evaluate every neighbouring node. Skip any neighbouring nodes that are in the closed list, and evaluate the weight value of any unevaluated neighbouring nodes. For nodes that are not marked as open but are not closed either, calculate their weight value based on the sum of their Dijkstra edge distance as well as the distance vector being used by this particular flavour of A* (such as Euclidian distance, but other distance vectors can be used depending on the use case). If the node is marked as open, check if the

edge distance of the node from the current node is less than what it is already set as and, if it is lower, reset the distance.

3. Close the current node, then move to the node with the lowest weight value.
4. Repeat these steps until the target node has been reached.

There are further variants of the A* algorithm for specific use cases, such as robotics [6] and network routing [7].

Although Dijkstra and A* are usually both implemented as a breadth first search, Dijkstra's algorithm is not as computationally complex as A* because it does not use a heuristic but it is less accurate as a result. While Dijkstra's algorithm may be preferable on less powerful hardware, A* is preferable on modern hardware. While A* is more computationally complex than Dijkstra's, in more complex graphs (such as road networks) A* scales better and can have specialised heuristic functions for specific use cases [8]. However, in cases where a valid heuristic is not available or necessary, Dijkstra's Algorithm may be better than A* because a heuristic may be expensive to compute and may not be as beneficial on smaller graphs.

Task 2 – Implementation

```
void GridNavigatorAutomatic::Navigate()
{
    moveresults mr;
    bool going_left = false;

    while mr != moveresults::complete {
        grid->PrintGrid();

        mr = grid->MovePlayer(movedirection::up);
        if (mr == moveresults::wall) {
            if (going_left) {
                mr = grid->MovePlayer(movedirection::left);
            } else {
                mr = grid->MovePlayer(movedirection::right);
            }

            // Invert the boolean condition
            going_left = !going_left
        }
    }
}
```

Figure 1: Screenshot of the first proof of concept

The purpose of this program is to algorithmically navigate a randomly generated maze without colliding with a wall. This algorithm exclusively uses heuristics to find the goal, as the mazes that are generated by the maze program are always extremely similar. For example, each maze is always a 20x20 grid, the player character always spawns on the bottom left and the goal is always on the top right. There is also never any need to move downwards.

The first implementation of the program functioned mainly as a proof of concept to test the viability of a heuristic implementation. In this implementation, the player is always moved up by default and, if a wall is hit, moves to the right. If a right wall is hit then the player will begin moving left instead of right until it hits another wall. No checks against the grid are made and many wall collisions are made (including against walls that have already been collided against), which means that this implementation is suboptimal. Still, it confirmed a number of conditions about the maze (including the fact that the goal is always in the same place) and in turn proved the possibility of using a purely heuristic method of finding the goal.

```

void GridNavigatorAutomatic::Navigate()
{
    moveresults mr;
    bool going_left = false;
    griditem tile;

    // Starting position of the player is always X: 1 and Y: 18
    int player_x = 1;
    int player_y = 18;

    while (mr != moveresults::complete) {
        // grid->PrintGrid();

        // Y: 1 is always the goal row
        if (player_y == 1) {
            mr = grid->MovePlayer(movedirection::right);
        }

        else {
            tile = grid->GetGridSquare(player_x, player_y - 1);

            if (tile == griditem::empty) {
                mr = grid->MovePlayer(movedirection::up);
                player_y -= 1;
            } else {
                // Check every tile left and right
                int x_left = player_x - 1;
                int x_right = player_x + 1;
                int target_x;

                while (true) {
                    tile = grid->GetGridSquare(x_right, player_y - 1);
                    if (tile != griditem::empty) {
                        tile = grid->GetGridSquare(x_left, player_y - 1);
                        if (x_right < 18) {
                            x_right += 1;
                            cout << x_right << std::endl;
                        }
                        if (tile == griditem::empty) {
                            target_x = x_left;
                            going_left = true;
                            break;
                        } else if (x_left > 0) {
                            x_left -= 1;
                            cout << x_left << std::endl;
                        }
                    } else {
                        target_x = x_right;
                        going_left = false;
                        break;
                    }
                }

                while (player_x != target_x && mr != moveresults::complete) {
                    // grid->PrintGrid();
                    if (going_left) {
                        mr = grid->MovePlayer(movedirection::left);
                        player_x -= 1;
                    } else {
                        mr = grid->MovePlayer(movedirection::right);
                        player_x += 1;
                    }
                }
            }
        }
    }
}

```

Figure 2: Full screen image of the purely heuristic code

With this knowledge, the second implementation was made to be more optimal in terms of moves. Instead of blindly moving forward, it checks the tile above and, if it is an empty tile, it moves upwards. If the tile above is a wall, it then alternates between scanning right and left until it finds an opening and moves over to it when it has found one. It does not perform any scanning on the top row because that row is always clear and the goal is always in the same place, so it just moves to the right when on that row.

```

void GridNavigatorAutomatic::Navigate()
{
    ...

    // Fix for an issue where the navigator would try to move sideways if a goal tile is above
    it
    if (tile != griditem::wall) {
        mr = grid->MovePlayer(movedirection::up);
        player_y -= 1;
    } else {
        ...
    }
}

```

Figure 3: Truncated code image showing the fix that was made

Due to an if error, it would make a wall collision if the goal is directly above it. The cause of this issue was that the code was checking for an empty space in order to move up, which would flag the goal as an illegal move. This was fixed by changing the check for an empty tile to instead be a check for if a wall is not on the tile.

```

Enter choice:
1: Manual mode (for learning the environment).
2: Automatic mode, single run (for testing of your code.
3: Benchmarking mode, 10,000 runs of automatic mode.
3
Minimum accesses: 17
Mean accesses: 63
Maximum accesses: 131
Minimum moves: 34
Mean moves: 48
Maximum moves: 90
Minimum wall hits: 0
Mean wall hits: 0
Maximum wall hits: 0

```

Figure 4: Output of the navigation program

Running the fixed version of the algorithm in benchmarking mode (which performs ten thousand maze runs) the algorithm performs on average 63 grid accesses, moves 48 times on average and never hits a wall.

Compared to a more conventional pathfinding algorithm, this algorithm does not use container types such as stacks and dequeues. This means that the algorithm is not as complex as a more general pathfinding algorithm, and is therefore less intensive on system resources and is easier to read. However, it is not appropriate for more complicated mazes because it is specifically optimised for this particular maze generator. Given more time and word count, an implementation of A* or Dijkstra's algorithm may have been produced to compare it with this algorithm.

References

- [1] D. Rachmawati and L. Gustin, 'Analysis of Dijkstra's Algorithm and A* Algorithm in Shortest Path Problem', *Journal of Physics: Conference Series*, vol. 1566, no. 1, Jun. 2020, doi: 10.1088/1742-6596/1566/1/012061.
- [2] 'greedy algorithm'. Accessed: Nov. 27, 2025. [Online]. Available: <https://xlinux.nist.gov/dads//HTML/greedyalgo.html>
- [3] Llanos Diego R, Gonzalez-Escribano Arturo, and Ortega-Arranz Hector, *The shortest-path problem: analysis and comparison of methods*. in Synthesis lectures on theoretical computer science, no. 1. San Rafael, California: Morgan & Claypool, 2015. doi: 10.2200/S00618ED1V01Y201412TCS001.
- [4] 'The Life Changing Magic of Dijkstra and A*', 2017. Accessed: Jan. 22, 2026. [Online]. Available: https://web.stanford.edu/class/archive/cs/cs106x/cs106x.1174/lectures/25-Dijkstra_AStar/25-Dijkstra_AStar_Notes.pdf
- [5] H. Li *et al.*, 'Source-seeking multi-robot team simulator as container of nature-inspired metaheuristic algorithms and Astar algorithm', *Expert systems with applications*, vol. 233, pp. 120932-, 2023, doi: 10.1016/j.eswa.2023.120932.
- [6] H. Li *et al.*, 'Source-seeking multi-robot team simulator as container of nature-inspired metaheuristic algorithms and Astar algorithm', *Expert systems with applications*, vol. 233, pp. 120932-, 2023, doi: 10.1016/j.eswa.2023.120932.
- [7] W. Shao, S. Jia, C. Yang, and T. Shangguan, 'Research on Cluster Routing Protocol Based on Genetic Algorithm and Astar Algorithm', *Informatica (Ljubljana)*, vol. 48, no. 14, 2024, doi: 10.31449/inf.v48i14.6264.
- [8] W. Yin and X. Yang, 'A Totally Astar-based Multi-path Algorithm for the Recognition of Reasonable Route Sets in Vehicle Navigation Systems', *Procedia - Social and Behavioral Sciences*, vol. 96, pp. 1069–1078, Nov. 2013, doi: 10.1016/j.sbspro.2013.08.123.