



ACCB5011: ADVANCED SOFTWARE DEVELOPMENT

Assignment: Maze Pathfinding Implementation

Joe Frohwein

2315545

2315545@student.uwtsd.ac.uk
Computing (Software Engineering)

Contents

1. Dijkstra's and A* Algorithms	2
Dijkstra's Algorithm	2
A* Algorithm.....	2
Comparison	3
Conclusion	3
2. Reflection on the Pathfinding Implementation.....	3
The "Hand on the left wall" method	4
Depth-first Search (DFS)	5
Breadth-first Search (BFS)	7
Dijkstra's Algorithm	8
References	10

1. Dijkstra's and A* Algorithms

Dijkstra's Algorithm

Dijkstra's algorithm "solves the single-source shortest-paths problem on a weighted, directed graph $G = (V, E)$ for the case in which all edge weights are nonnegative" [1].

Dijkstra's algorithm works by computing the lightest path from a starting node to every other node in a graph. This is done by maintaining a set of vertices whose final shortest-path weights from the source have already been determined. Then, a 'priority' queue [2] is used to repeatedly select the vertex with the lowest path weight/distance and relaxes all of its edges [1].

Dijkstra's algorithm is referred to as a 'greedy' strategy, as it always chooses the lightest possible vertex from the current node. "Greedy strategies do not always yield optimal results in general, but as the following theorem and its corollary show, Dijkstra's algorithm does indeed compute shortest paths" [1].

Dijkstra's algorithm will always find the shortest, most optimal path to each node, however several nodes will be checked more than once, due to the algorithms behaviour to expand outwards from the source, level by level, without any knowledge on un-visited nodes in the graph. This drawback can be seen in section 2 below.

"The time complexity of Dijkstra's Algorithm is typically $O(V^2)$ when using a simple array implementation or $O((V + E) \log V)$ with a priority queue, where V represents the number of vertices and E represents the number of edges in the graph" [3]

A* Algorithm

The A* algorithm is used to find the shortest path from a specific source node to a specific goal node in a graph. The algorithm is "an extension of Dijkstra's shortest path algorithm. It uses a heuristic function to guide the search in order to avoid visiting nodes that are unlikely to be a part of the optimal path". To do this, the A* algorithm "compares $f_i = g_i + h(i)$, where f_i is a heuristic approximation of the length of the whole path and $h(i)$ is the heuristic function" [4].

The heuristic function allows the A* algorithm to be an improvement on Dijkstra's algorithm as it "shall never overestimate the length of the remaining path. This condition is known as the admissibility condition. An admissible heuristic function guarantees that $h(i)$ for the remaining path is lower or equal to the length of the actual optimal path" [4] guaranteeing that it'll always find the most optimal path through the graph.

"In the worst case, A* operates in $O(b^d)$ time, where b is the branching factor and d is the depth of the solution" [5].

Comparison

These descriptions of Dijkstra's and A* algorithms show several similarities and differences.

The main similarity is that they are both algorithms that are used to determine the shortest path through weighted graphs, whilst both guaranteeing the optimal path, if used correctly. However, they also differ in their specific use, as Dijkstra's algorithm is used to find the shortest path from a starting node to every other node, whereas the A* algorithm is used to find the shortest path from a starting node to a specific goal node.

Some more similarities are that they are both 'greedy' algorithms that require 'priority' queues, non-negative edge weights, and the ability to relax edges to work. These similarities all apply because, as previously stated, the A* algorithm is "an extension of Dijkstra's shortest path algorithm" so share several of the same properties and functions.

The main difference is the use of heuristics in the A* algorithm. "Whereas Dijkstra's algorithm uses the length of the shortest path from the beginning, to select edges the A* instead compares $f_i = g_i + h(i)$ ", as previously stated [4].

The other main difference would be the algorithms' behaviour and efficiency. As previously stated, Dijkstra's algorithm works by expanding outwards from the source, level by level, without any knowledge on un-visited nodes in the graph, leading to more nodes than necessary being checked. However, the A* algorithm "avoids visiting nodes that are unlikely to be part of the optimal path" through its use of heuristics, therefore leading to only the necessary nodes being checked, leading to greater efficiency.

Conclusion

To summarise, Dijkstra's and A* algorithms are both efficient algorithms guarantee finding the shortest path between nodes in a graph. Although it can be case dependent, the A* algorithm is generally more computationally efficient than Dijkstra's algorithm, due to its use of heuristics, reducing the unnecessary checking of nodes.

2. Reflection on the Pathfinding Implementation

Each method detailed below begins with the initialisation code that defines the grid dimensions, the starting coordinates of the player, and the 'moveresults' object:

```
const int height = 20;
const int width = 20;

const int xStart = 1;
const int yStart = 18;

moveresults mr;
```

The “Hand on the left wall” method

In this method, the player keeps their hand on the left wall and follows it until the exit is reached. Although this method involves back-tracking and is inefficient, it provides a simple starting strategy.

The method begins by initialising the player’s facing direction, then assigns integers to each direction (0 = Up, 1 = Right, 2 = Down, 3 = Left) as they’re simpler to compare than strings, using the modulus of 4 to control this, as there are 4 possible directions.

```
int left = (direction + 3) % 4;

int xChange = 0;
int yChange = 0;

if (left == 3)
{
    xChange = -1;
}
else if (left == 0)
{
    yChange = -1;
}
else if (left == 1)
{
    xChange = 1;
}
else if (left == 2)
{
    yChange = 1;
}
```

In a loop that repeats until the player reaches the target, the player’s left side is checked for a wall. If there is no wall, the player is turned and moved to their left.

```
if (direction == 0)
{
    mr = grid->MovePlayer(movedirection::left);
    x--;
    direction = 3;
}
else if (direction == 1)
{
    mr = grid->MovePlayer(movedirection::up);
    y--;
    direction = 0;
}
else if (direction == 2)
{
    mr = grid->MovePlayer(movedirection::right);
    x++;
    direction = 1;
}
else if (direction == 3)
{
    mr = grid->MovePlayer(movedirection::down);
    y++;
    direction = 2;
}
```

If there is a wall, the space in front of the player is now checked for a wall. If there is no wall, the player is moved forward, else the player is turned to the right, and the code returns to the start of the loop.

```

if (grid->GetGridSquare(x + xChange, y + yChange) == griditem::wall)
{
    int xForward = 0;
    int yForward = 0;

    if (direction == 0)
    {
        yForward--;
    }
    else if (direction == 1)
    {
        xForward++;
    }
    else if (direction == 2)
    {
        yForward++;
    }
    else if (direction == 3)
    {
        xForward--;
    }

    if (grid->GetGridSquare(x + xForward, y + yForward) == griditem::wall)
    {
        direction = (direction + 1) % 4;
    }
    else
    {
        if (direction == 0)
        {
            mr = grid->MovePlayer(movedirection::up);
            y--;
        }
        else if (direction == 1)
        {
            mr = grid->MovePlayer(movedirection::right);
            x++;
        }
        else if (direction == 2)
        {
            mr = grid->MovePlayer(movedirection::down);
            y++;
        }
        else if (direction == 3)
        {
            mr = grid->MovePlayer(movedirection::left);
            x--;
        }
    }
}
}

```

As previously stated, this method is inefficient but provides a basic starting strategy and a benchmark for comparison.

```

Minimum accesses: 69
Mean accesses: 172
Maximum accesses: 336
Minimum moves: 34
Mean moves: 82
Maximum moves: 162
Minimum wall hits: 0
Mean wall hits: 0
Maximum wall hits: 0

```

Depth-first Search (DFS)

DFS explores as far as possible down one path and backtracks upon reaching a dead-end [6].

This code involves an array, used to track which squares have been visited; to avoid re-visiting, A stack is utilised to perform this method as a DFS adopts a LIFO system, allowing each of the player's movements to be backtracked in reverse order.

```

int x = xStart;
int y = yStart;

bool visited[width][height] = { false };

std::stack<int> stack;

```

In a loop, the player moves forward in each direction until a wall is reached, checking in the order of up, right, left, down (prioritising upwards and rightwards due to the starting position of the player and the target). If all surrounding squares of the player are walls or visited, the player backtracks by accessing their last move off the top of the stack and trying to find an un-visited square from that previous position.

```

// 0 = UP, 1 = RIGHT, 2 = LEFT, 3 = DOWN
if (!visited[xUp][yUp] && grid->GetGridSquare(xUp, yUp) != griditem::wall)
{
    mr = grid->MovePlayer(movedirection::up);
    y--;
    stack.push(0);
}
else if (!visited[xRight][yRight] && grid->GetGridSquare(xRight, yRight) != griditem::wall)
{
    mr = grid->MovePlayer(movedirection::right);
    x++;
    stack.push(1);
}
else if (!visited[xLeft][yLeft] && grid->GetGridSquare(xLeft, yLeft) != griditem::wall)
{
    mr = grid->MovePlayer(movedirection::left);
    x--;
    stack.push(2);
}
else if (!visited[xDown][yDown] && grid->GetGridSquare(xDown, yDown) != griditem::wall)
{
    mr = grid->MovePlayer(movedirection::down);
    y++;
    stack.push(3);
}
else
{
    int lastMove = stack.top();
    stack.pop();

    if (lastMove == 0)
    {
        mr = grid->MovePlayer(movedirection::down);
        y++;
    }
    else if (lastMove == 1)
    {
        mr = grid->MovePlayer(movedirection::left);
        x--;
    }
    else if (lastMove == 2)
    {
        mr = grid->MovePlayer(movedirection::right);
        x++;
    }
    else if (lastMove == 3)
    {
        mr = grid->MovePlayer(movedirection::up);
        y--;
    }
}

```

This method is slightly better than the previous method but still involves a lot of dead-ends and backtracking, therefore not finding the shortest path.

```

Minimum accesses: 51
Mean accesses: 126
Maximum accesses: 463
Minimum moves: 34
Mean moves: 87
Maximum moves: 410
Minimum wall hits: 0
Mean wall hits: 0
Maximum wall hits: 0

```

Breadth-first Search (BFS)

BFS involves traversing the maze level by level, until the target is reached in the least number of moves, allowing for the shortest path to be found [7].

Arrays are used to track each squares visited status and parent's coordinates, and a pair of queues is used to track the coordinates of the squares to be checked next, as a BFS adopts a FIFO system, allowing squares to be checked in the order that they were added.

```
int x = xStart;
int y = yStart;

bool visited[width][height] = { false };

std::queue<int> xQueue;
std::queue<int> yQueue;

xQueue.push(x);
yQueue.push(y);

visited[x][y] = true;

int xParent[width][height];
int yParent[width][height];

xParent[x][y] = -1;
yParent[x][y] = -1;

int xEnd = -1;
int yEnd = -1;
```

In a loop, starting from the player's starting position, all adjacent, non-visited, non-wall squares that the player could potentially move into are marked as visited and assigned parent squares, as this will always be the shortest path to this square.

```
while (!xQueue.empty() && !yQueue.empty())
{
    int xFront = xQueue.front();
    xQueue.pop();

    int yFront = yQueue.front();
    yQueue.pop();

    if (grid->GetGridSquare(xFront, yFront) == griditem::target)
    {
        xEnd = xFront;
        yEnd = yFront;
        break;
    }

    int xUp = xFront;
    int yUp = yFront - 1;

    int xRight = xFront + 1;
    int yRight = yFront;

    int xDown = xFront;
    int yDown = yFront + 1;

    int xLeft = xFront - 1;
    int yLeft = yFront;

    if (!visited[xUp][yUp] && grid->GetGridSquare(xUp, yUp) != griditem::wall)
    {
        visited[xUp][yUp] = true;
        xParent[xUp][yUp] = xFront;
        yParent[xUp][yUp] = yFront;

        xQueue.push(xUp);
        yQueue.push(yUp);
    }

    if (!visited[xRight][yRight] && grid->GetGridSquare(xRight, yRight) != griditem::wall)
    {
        visited[xRight][yRight] = true;
        xParent[xRight][yRight] = xFront;
        yParent[xRight][yRight] = yFront;

        xQueue.push(xRight);
        yQueue.push(yRight);
    }

    if (!visited[xDown][yDown] && grid->GetGridSquare(xDown, yDown) != griditem::wall)
    {
        visited[xDown][yDown] = true;
        xParent[xDown][yDown] = xFront;
        yParent[xDown][yDown] = yFront;

        xQueue.push(xDown);
        yQueue.push(yDown);
    }

    if (!visited[xLeft][yLeft] && grid->GetGridSquare(xLeft, yLeft) != griditem::wall)
    {
        visited[xLeft][yLeft] = true;
        xParent[xLeft][yLeft] = xFront;
        yParent[xLeft][yLeft] = yFront;

        xQueue.push(xLeft);
        yQueue.push(yLeft);
    }
}
```

Once the shortest path is found, the path is backtracked from the target back to the player and stored in a stack, which reverses the order due to its LIFO data structure [8], organising the path in the correct order, before the player is moved to the target.

```

std::stack<int> xStack;
std::stack<int> yStack;

int xFront = xEnd;
int yFront = yEnd;

while (xFront != -1 && yFront != -1)
{
    xStack.push(xFront);
    yStack.push(yFront);

    int xNext = xParent[xFront][yFront];
    int yNext = yParent[xFront][yFront];

    xFront = xNext;
    yFront = yNext;
}

while (!xStack.empty() && !yStack.empty())
{
    int xNext = xStack.top();
    int yNext = yStack.top();

    if (xNext == xStart && yNext == yStart)
    {
        xStack.pop();
        yStack.pop();
    }
    else
    {
        if (xNext != x && yNext == y)
        {
            if (xNext > x)
            {
                mr = grid->MovePlayer(movedirection::right);
                x++;
            }
            else if (xNext < x)
            {
                mr = grid->MovePlayer(movedirection::left);
                x--;
            }
        }
        else if (xNext == x && yNext != y)
        {
            if (yNext > y)
            {
                mr = grid->MovePlayer(movedirection::down);
                y++;
            }
            else if (yNext < y)
            {
                mr = grid->MovePlayer(movedirection::up);
                y--;
            }
        }

        xStack.pop();
        yStack.pop();
    }
}

```

BFS ensures the shortest possible path is found before moving the player, removing the need for backtracking, which is clearly reflected in the mean moves. However, grid accesses increase due to checking multiple adjacent squares at each step.

```

Minimum accesses: 396
Mean accesses: 616
Maximum accesses: 692
Minimum moves: 34
Mean moves: 48
Maximum moves: 88
Minimum wall hits: 0
Mean wall hits: 0
Maximum wall hits: 0

```

Dijkstra's Algorithm

Dijkstra's Algorithm methodology is an extension of BFS, but it also considers the weight of each move to find the lowest-weight path [9].

The code, which is a slightly altered version of the previous BFS method, uses visited and parent arrays, as well as another array to track the cost to get to each square.

```
bool visited[width][height] = { false };
int pathWeight[width][height] = {};

for (int i = 0; i < width; i++)
{
    for (int j = 0; j < height; j++)
    {
        pathWeight[i][j] = INT_MAX;
    }
}

pathWeight[x][y] = 0;

int xParent[width][height];
int yParent[width][height];

xParent[x][y] = -1;
yParent[x][y] = -1;
```

In a loop, the grid is scanned to find the un-visited square with the lowest current cost to get to, which initially is the player's starting position.

```
for (int i = height - 1; i >= 0; i--)
{
    for (int j = 0; j < width; j++)
    {
        if (!visited[j][i] && pathWeight[j][i] < minWeight)
        {
            minWeight = pathWeight[j][i];
            xCurrent = j;
            yCurrent = i;
        }
    }
}
```

Each lowest-cost square's neighbours are checked and updated if a lower cost is found, before moving onto the next square, until the target is found.

<pre>if (xCurrent != -1 && yCurrent != -1) { visited[xCurrent][yCurrent] = true; if (grid->GetGridSquare(xCurrent, yCurrent) == griditem::target) { xEnd = xCurrent; yEnd = yCurrent; } int xUp = xCurrent; int yUp = yCurrent - 1; int xRight = xCurrent + 1; int yRight = yCurrent; int xDown = xCurrent; int yDown = yCurrent + 1; int xLeft = xCurrent - 1; int yLeft = yCurrent; if (!visited[xUp][yUp] && grid->GetGridSquare(xUp, yUp) != griditem::wall) { int weight = pathWeight[xCurrent][yCurrent] + 1; if (weight < pathWeight[xUp][yUp]) { pathWeight[xUp][yUp] = weight; xParent[xUp][yUp] = xCurrent; yParent[xUp][yUp] = yCurrent; } } }</pre>	<pre>if (!visited[xRight][yRight] && grid->GetGridSquare(xRight, yRight) != griditem::wall) { int weight = pathWeight[xCurrent][yCurrent] + 1; if (weight < pathWeight[xRight][yRight]) { pathWeight[xRight][yRight] = weight; xParent[xRight][yRight] = xCurrent; yParent[xRight][yRight] = yCurrent; } } if (!visited[xDown][yDown] && grid->GetGridSquare(xDown, yDown) != griditem::wall) { int weight = pathWeight[xCurrent][yCurrent] + 1; if (weight < pathWeight[xDown][yDown]) { pathWeight[xDown][yDown] = weight; xParent[xDown][yDown] = xCurrent; yParent[xDown][yDown] = yCurrent; } } if (!visited[xLeft][yLeft] && grid->GetGridSquare(xLeft, yLeft) != griditem::wall) { int weight = pathWeight[xCurrent][yCurrent] + 1; if (weight < pathWeight[xLeft][yLeft]) { pathWeight[xLeft][yLeft] = weight; xParent[xLeft][yLeft] = xCurrent; yParent[xLeft][yLeft] = yCurrent; } }</pre>
---	---

Like in BFS, the optimal path is found, however the grid accesses were greater because squares are checked more than once to compare the weights.

```
Minimum accesses: 517
Mean accesses: 720
Maximum accesses: 833
Minimum moves: 34
Mean moves: 47
Maximum moves: 92
Minimum wall hits: 0
Mean wall hits: 0
Maximum wall hits: 0
```

Since all moves in this particular problem have an equal cost of 1, Dijkstra's algorithm is unnecessarily rigorous, as the shortest path found with a BFS will be identical to the lowest cost path found with Dijkstra's algorithm [10]. Therefore, for this particular problem, Breadth-first Search would be the most efficient method achieving fewer moves than a DFS and fewer grid accesses than Dijkstra's algorithm.

References

- [1] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, Introduction to Algorithms, The MIT Press, 2022.
- [2] M. Cryan, "Introduction to Algorithms and Data Structures," [Online]. Available: https://opencourse.inf.ed.ac.uk/sites/default/files/2025-01/iads_16_dijkstra_0.pdf. [Accessed 03 01 2026].
- [3] "Time and Space Complexity of Dijkstra's Algorithm," 23 07 2025. [Online]. Available: <https://www.geeksforgeeks.org/dsa/time-and-space-complexity-of-dijkstras-algorithm/>. [Accessed 03 01 2026].
- [4] J. Haapala, "Routing Military Aircraft by Solving," Aalto University, 2015.
- [5] Suf, "A* Algorithm: A Comprehensive Guide," The Research Scientist Pod, [Online]. Available: <https://researchdatapod.com/a-star-algorithm/>. [Accessed 03 01 2026].
- [6] "Depth First Search or DFS for a Graph," GeeksForGeeks, 25 10 2025. [Online]. Available: <https://www.geeksforgeeks.org/dsa/depth-first-search-or-dfs-for-a-graph/>. [Accessed 03 01 2026].
- [7] "Breadth First Search or BFS for a Graph," GeeksForGeeks, 06 12 2025. [Online]. Available: <https://www.geeksforgeeks.org/dsa/breadth-first-search-or-bfs-for-a-graph/>. [Accessed 03 01 2026].

- [8] "Stack Data Structure," GeeksForGeeks, 13 11 2025. [Online]. Available: <https://www.geeksforgeeks.org/dsa/stack-data-structure/>. [Accessed 03 01 2026].
- [9] "Dijkstra's Algorithm," W3 Schools, [Online]. Available: https://www.w3schools.com/dsa/dsa_algo_graphs_dijkstra.php. [Accessed 03 01 2026].
- [10] C. Warokar, "Breadth-first search is a special case of uniform-cost search. Why ?," Medium, 18 09 2024. [Online]. Available: <https://medium.com/@chitrnk971/breadth-first-search-is-a-special-case-of-uniform-cost-search-why-cc7a49250d2e>. [Accessed 03 01 2026].