

Maze Pathfinding Implementation - Liam Smalldon

by Liam Smalldon

Submission date: 13-Jan-2022 08:02PM (UTC+0000)

Submission ID: 168855295

File name: 18581_Liam_Afon_Smalldon_Maze_Pathfinding_Implementation_-_Liam_Smalldon_394416_432240197.docx (397.25K)

Word count: 2149

Character count: 11283

Maze Pathfinding Implementation

Liam Smalldon – 1910726

Table of Contents

Dijkstra’s Algorithm..... 3

A* Algorithm 3

Comparison..... 3

Implementation Reflection 4

References 10

Appendices..... 11

Dijkstra's Algorithm

Dijkstra's algorithm is an algorithm invented by Edsger Dijkstra. It finds the shortest path between two points using nodes that have a cost to get to them. The algorithm starts by expanding out equally in every direction and will explore each of the shortest paths first until it gets to the final node. Then, the lowest cost path is determined between each node that reaches the final node resulting in the shortest path to the goal [1].

A* Algorithm

The A* algorithm is a searching algorithm formulated by weighted graphs that finds the shortest path between the initial node and the final node. It is a modified version of Dijkstra's algorithm. Like Dijkstra's, it works when the path is represented as a graph. The graph is explored and the cost of each node is evaluated on all possible routes. It gives priority to nodes it thinks should be faster as the heuristic function estimates the quickest path from the initial node to the final node [2]. The heuristic function could be, for example, the Euclidian distance.

Comparison

Both algorithms should know the current state of the problem. The possible actions should be programmed in order to change the state of the problem that needs to be implemented. And finally, the ability to recognise the shortest path from the initial to the final node [3].

Dijkstra's algorithm is an uninformed algorithm, meaning it does not have any additional information to help determine where it should go, thus, must explore every possible route to find the shortest path [3].

The A* algorithm is an informed search algorithm. What is meant by informed is that it uses additional information to direct the search to the final node. This additional information a type of score that is only an estimation. This estimation is called a heuristic estimated cost, which estimates the cost between two nodes [3]. Therefore, it does not need to mark each node with infinity as it has some idea of the cost to each node.

As Dijkstra's algorithm it has no additional information, each node is marked as an infinite distance from the start node to the end node initially. The cost is not actually infinite, but this is used to mark the nodes as unvisited. The starting node is initialised with 0. The starting node is then marked as visited in the visited list to remember where it has been. The distance to each neighbouring node is calculated by adding the cost of the edges that belong to the path. The lowest cost node that is in the unvisited list is chosen; this then becomes the current node. Each cost to the node will always be less than infinity so these costs are then recorded as the cost to get to the nodes. This is looped over until the unvisited queue has reached 0 and the destination node is reached. From the destination, you check which path is the shortest to the previous node. This is repeated until you are back at the source node, giving you the shortest path [4].

The A* algorithm works by initialising the cost of all nodes to infinity, just like Dijkstra's, to show they have not been visited. The starting node is also initialised with the cost of 0. Differently to Dijkstra's, the lowest heuristic score is then used to find the potential closest node. This works by examining the vertex that has the lowest cost based on its exact cost, plus it's heuristic estimated cost from the vertex to the final node. It basically moves into each optimal node and continues till the end keeping track of where it has been via the visited list, just like Dijkstra's. A* gives priority to nodes that should be better than others. It will pursue these routes first, making it faster than Dijkstra's [5]. This optimal node then becomes the current node. The neighbours of this node are

examined and the neighbouring node is calculated again by the cost of the edges, plus the heuristic score is added to the cost to get to the optimal node. The lowest heuristic scored node in the unvisited list will become the next node to move to and the current one will be marked as visited. This is repeated until the destination node is found. Then like Dijkstra's follow the shortest path the previous nodes and work backwards to the source node to get the shortest path [6].

Both algorithms use costs between nodes to determine the most efficient path. When programming both algorithms, two lists are created. The open list, which has all the unvisited nodes. The other list is the closed list, which has all the visited nodes [7]. Each algorithm can also only work if all the costs between nodes are positive numbers.

Implementation Reflection

Here is the author of the document's short reflection on the development process of their maze solving algorithm.

```

// grid->PrintGrid();

// Initialise copy maze
int maze[20][20];

// Array to keep track of visited coordinates
int visited[20][20];

// Get maze as integer array as grid->GetGrid() is too hard to work with
for (int y = 0; y < 20; y++) {
    for (int x = 0; x < 20; x++) {
        if (grid->GetGridSquare(x, y) == griditem::empty) {
            // Initialise possible moves as "Infinite"
            maze[x][y] = INT_MAX;
            visited[x][y] = 0;
        }
        else if (grid->GetGridSquare(x, y) == griditem::wall) {
            // Set walls as out of range number
            maze[x][y] = 99;
            // maze[x][y] = -1;
            // Set walls as 2, to be ignored
            visited[x][y] = 2;
        }
        else if (grid->GetGridSquare(x, y) == griditem::player) {
            maze[x][y] = 100;
            visited[x][y] = 1;
        }
        else if (grid->GetGridSquare(x, y) == griditem::target) {
            maze[x][y] = 0;
            visited[x][y] = 0;
        }
    }
}

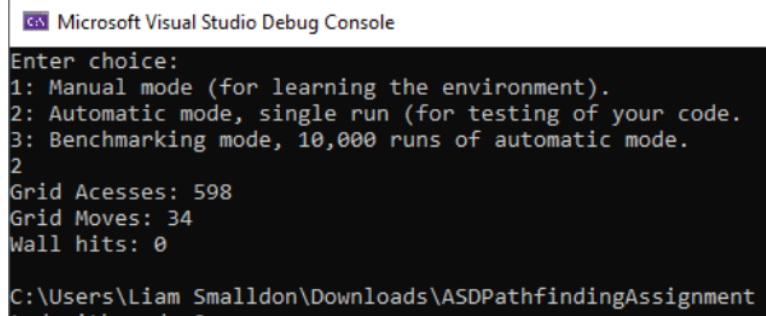
```

Figure 1

The author of this document opted to convert the maze to a 2D array made of integers as seen in Figure 1. This was done because they couldn't use the "grid->GetGrid()" as it returned it as type "griditem". Working with primitive types made creating the solution much easier.

Doing it this way also means that the code is not very dynamic. It won't be able to cope if the grid size changes and will require some manual intervention. If it were to be dynamic, there would have to be a method to always get the "gridsize", from here calculations can be put into the array indexes to always, for example, the start and end positions.

The maze is remapped from the set enum class. Empty spaces are now infinite, following Dijkstra's algorithm. Walls are set to a number out of range, so they are ignored. The player's position is set to 100 and the target is set to 0. See Figure 1.

A screenshot of the Microsoft Visual Studio Debug Console. The window title is "Microsoft Visual Studio Debug Console". The text inside shows a program prompt "Enter choice:" followed by three numbered options: "1: Manual mode (for learning the environment).", "2: Automatic mode, single run (for testing of your code).", and "3: Benchmarking mode, 10,000 runs of automatic mode.". The user has entered "2". The program then outputs "Grid Acesses: 598", "Grid Moves: 34", and "Wall hits: 0". At the bottom, the file path "C:\Users\Liam Smalldon\Downloads\ASDPathfindingAssignment" is visible.

```
Microsoft Visual Studio Debug Console
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.
2
Grid Acesses: 598
Grid Moves: 34
Wall hits: 0
C:\Users\Liam Smalldon\Downloads\ASDPathfindingAssignment
```

Figure 2

There is a strange flaw here as the grid access counter reports over 400 accesses each time the program runs as seen in Figure 2. It is suspected that the if conditions are to blame and there is uncertainty on how to fix. This does cause the algorithms efficiency to decrease, although, not by much.

This implementation starts from the destination and moves to the source, this works as start and end positions in the maze are always the same.

```

// Loop through vertices
for (int y = 1; y < 19; y++) {
    bool hasBeenVisited = true;
    while (hasBeenVisited == true) {
        hasBeenVisited = false;
        for (int x = 18; x > 0; x--) {
            // If cell is valid and empty
            if (maze[x][y] != 99 && visited[x][y] == 0) {
                int north = maze[x][y - 1];
                int south = maze[x][y + 1];
                int west = maze[x - 1][y];
                int east = maze[x + 1][y];
                vector<int> incrementPath;

                if (north != 99 && north != INT_MAX) {
                    incrementPath.push_back(north);
                } else if (south != 99 && south != INT_MAX) {
                    incrementPath.push_back(south);
                } else if (west != 99 && west != INT_MAX) {
                    incrementPath.push_back(west);
                } else if (east != 99 && east != INT_MAX) {
                    incrementPath.push_back(east);
                }

                // Check neighbours.                                Ignore starting position.
                if (canMakeChange(north, south, west, east) == true && maze[x][y] != 0) {
                    // Sort numbers into chronological order
                    std::sort(std::begin(incrementPath), std::end(incrementPath));

                    // Add increments to the maze
                    maze[x][y] = incrementPath[0] + 1;
                    //cout << incrementPath[0] + 1 << endl;

                    // Record cell as visited
                    visited[x][y] = 1;
                    hasBeenVisited = true;
                }
            }
        }
    }
}

```

Figure 3

Using the altered maze, Figure 3, each column will be looped over. Then each row will be looped over while it has not been visited. The neighbouring cells are then checked to see if they are valid and can be mapped. The mapping was done using incrementation, which leaves a chronological path from the destination to the source in the maze which can be followed later [8]. To keep track of this mapping, a second array was used to track where the algorithm had been.

```

// Starting coordinates of player
vector<int> startPosition = { 1, 18 };

// Check if complete on each loop
while (grid->CheckComplete() == false) {
    int x = startPosition[0];
    int y = startPosition[1];
    vector<int> incrementalPath;

    // Initialise vector
    incrementalPath.push_back(maze[x][y - 1]); // North
    incrementalPath.push_back(maze[x][y + 1]); // South
    incrementalPath.push_back(maze[x - 1][y]); // West
    incrementalPath.push_back(maze[x + 1][y]); // East

    //for (int pathCheck : incrementalPath) {
    //    cout << pathCheck << endl;
    //}

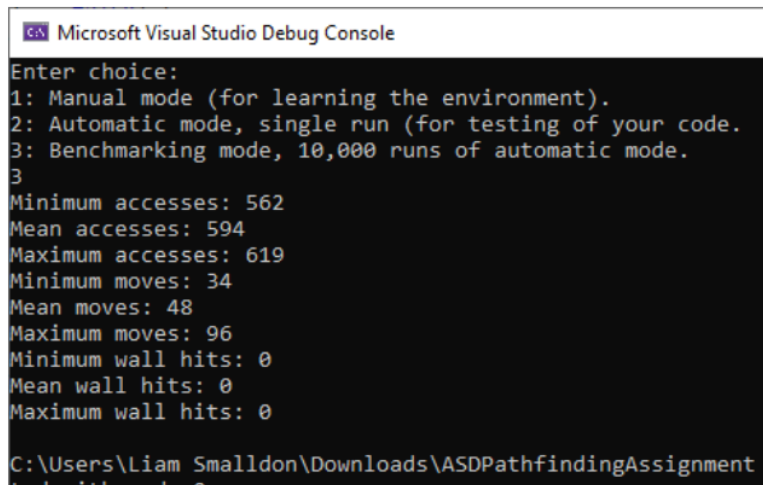
    // Sort path in chronological order
    std::sort(std::begin(incrementalPath), std::end(incrementalPath));

    // if next increment in path is equal to next increment in array, move to it
    if (incrementalPath[0] == maze[x][y - 1]) {
        startPosition[1] = y - 1;
        grid->MovePlayer(movedirection::up);
    }
    else if (incrementalPath[0] == maze[x][y + 1]) {
        startPosition[1] = y + 1;
        grid->MovePlayer(movedirection::down);
    }
    else if (incrementalPath[0] == maze[x - 1][y]) {
        startPosition[0] = x - 1;
        grid->MovePlayer(movedirection::left);
    }
    else if (incrementalPath[0] == maze[x + 1][y]) {
        startPosition[0] = x + 1;
        grid->MovePlayer(movedirection::right);
    }
}
// grid->PrintGrid();

```

Figure 4

Once the maze has been mapped, a chronological number line is in place in the maze to be followed. As you can see in Figure 4, the starting coordinates are set and the directions to check are pushed to a vector. These directions are then sorted from smallest to largest to show the path to follow. Next if a direction in the maze matches the incrementing path, the player will move to it.



```
Microsoft Visual Studio Debug Console
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: 562
Mean accesses: 594
Maximum accesses: 619
Minimum moves: 34
Mean moves: 48
Maximum moves: 96
Minimum wall hits: 0
Mean wall hits: 0
Maximum wall hits: 0
C:\Users\Liam Smalldon\Downloads\ASDPathfindingAssignment
```

Figure 5

Using benchmarking mode as seen in Figure 5, the algorithm is successfully run 10,000 times. Walls are never hit, meaning the algorithm knows where the walls are and the path it follows avoids them causing it to be fairly efficient. If there was no issue in accessing the grid, its access count would be at 400.

References

- [1] "Dijkstra's Algorithm: Definition, Applications & Examples," study.com/, [Online]. Available: <https://study.com/academy/lesson/dijkstra-s-algorithm-definition-applications-examples.html>. [Accessed 06 01 2022].
- [2] A. Singhal, "A* Algorithm | A* Algorithm Example in AI," www.gatevidyalay.com, [Online]. Available: <https://www.gatevidyalay.com/a-algorithm-a-algorithm-example-in-ai/>. [Accessed 06 01 2022].
- [3] M. Lukic, "Dijkstra's Algorithm vs A* Algorithm," stackabuse.com, 10 12 2021. [Online]. Available: <https://stackabuse.com/dijkstras-algorithm-vs-a-algorithm/>. [Accessed 06 01 2022].
- [4] isaac computer science, "Dijkstra's algorithm," isaacomputerscience.org, [Online]. Available: https://isaacomputerscience.org/concepts/dsa_search_dijkstra?examBoard=all&stage=all. [Accessed 06 01 2022].
- [5] A. Patel, "Introduction to A*," [Online]. Available: <http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html>. [Accessed 06 01 2022].
- [6] isaac computer science, "A* search algorithm," isaacomputerscience.org, [Online]. Available: https://isaacomputerscience.org/concepts/dsa_search_a_star?examBoard=all&stage=all. [Accessed 06 01 2022].
- [7] GeeksforGeeks, "A* Search Algorithm," www.geeksforgeeks.org, 24 12 2021. [Online]. Available: <https://www.geeksforgeeks.org/a-search-algorithm/>. [Accessed 06 01 2022].
- [8] D. Mack, "Mazes for Programmers: Dijkstra's algorithm," 26 12 2019. [Online]. Available: <https://daemianmack.org/posts/2019/12/mazes-for-programmers-dijkstras-algorithm.html>. [Accessed 04 01 2022].

Appendices

Code

```
#include "GridNavigatorAutomatic.h"

/// <summary>
/// Checks if neighbouring cells in array are blocked
/// </summary>
/// <param name="north">Maze movement</param>
/// <param name="south">Maze movement</param>
/// <param name="west">Maze movement</param>
/// <param name="east">Maze movement</param>
/// <returns>bool if cell is blocked</returns>
bool canMakeChange(int north, int south, int west, int east) {

    bool canMakeChange = false;

    // If neighbour is not a wall or intmax then value can be changed
    if (!(north == 99 || north == INT_MAX)) {
        canMakeChange = true;
    }
    else if (!(south == 99 || south == INT_MAX)) {
        canMakeChange = true;
    }
    else if (!(west == 99 || west == INT_MAX)) {
        canMakeChange = true;
    }
    else if (!(east == 99 || east == INT_MAX)) {
        canMakeChange = true;
    }
    return canMakeChange;
}

GridNavigatorAutomatic::GridNavigatorAutomatic() : GridNavigator()
{
}

void GridNavigatorAutomatic::Navigate()
{
    // grid->PrintGrid();

    // Initialise copy maze
    int maze[20][20];

    // Array to keep track of visited coordinates
    int visited[20][20];

    // Get maze as integer array as grid->GetGrid() is too hard to work with
    for (int y = 0; y < 20; y++) {
        for (int x = 0; x < 20; x++) {
            if (grid->GetGridSquare(x, y) == griditem::empty) {
                // Initialise possible moves as "Infinite"
                maze[x][y] = INT_MAX;
                visited[x][y] = 0;
            }
            else if (grid->GetGridSquare(x, y) == griditem::wall) {
                // Set walls as out of range number
                maze[x][y] = 99;
                // maze[x][y] = -1;
            }
        }
    }
}
```

```

        // Set walls as 2, to be ignored
        visited[x][y] = 2;
    }
    else if (grid->GetGridSquare(x, y) == griditem::player) {
        maze[x][y] = 100;
        visited[x][y] = 1;
    }
    else if (grid->GetGridSquare(x, y) == griditem::target) {
        maze[x][y] = 0;
        visited[x][y] = 0;
    }
}
}

// Somehow getting over 400 grid access :(
//cout << grid->GetCounter()->GetAccessCounter() << endl;

// Loop through vertices
for (int y = 1; y < 19; y++) {
    bool hasBeenVisited = true;
    while (hasBeenVisited == true) {
        hasBeenVisited = false;
        for (int x = 18; x > 0; x--) {
            // If cell is valid and empty
            if (maze[x][y] != 99 && visited[x][y] == 0) {
                int north = maze[x][y - 1];
                int south = maze[x][y + 1];
                int west = maze[x - 1][y];
                int east = maze[x + 1][y];
                vector<int> incrementPath;

                if (north != 99 && north != INT_MAX) {
                    incrementPath.push_back(north);
                }
                else if (south != 99 && south != INT_MAX) {
                    incrementPath.push_back(south);
                }
                else if (west != 99 && west != INT_MAX) {
                    incrementPath.push_back(west);
                }
                else if (east != 99 && east != INT_MAX) {
                    incrementPath.push_back(east);
                }

                // Check neighbours.
                Ignore
                starting position.
                if (canMakeChange(north, south, west, east) == true &&
                    maze[x][y] != 0) {
                    // Sort numbers into chronological order
                    std::sort(std::begin(incrementPath),
                        std::end(incrementPath));

                    // Add increments to the maze
                    maze[x][y] = incrementPath[0] + 1;
                    //cout << incrementPath[0] + 1 << endl;

                    // Record cell as visited
                    visited[x][y] = 1;
                    hasBeenVisited = true;
                }
            }
        }
    }
}
}

```

```

// Print mapped maze
//for (int x = 0; x < 20; x++) {
//    for (int y = 0; y < 20; y++) {
//        cout << maze[y][x] << " ";
//    }
//}
//cout << endl;

// Print visited maze to check each cell being reached
//for (int x = 0; x < 20; x++) {
//    cout << endl;
//    for (int y = 0; y < 20; y++) {
//        cout << visited[x][y] << " ";
//    }
//}
//cout << endl;

// Starting coordinates of player
vector<int> startPosition = { 1, 18 };

// Check if complete on each loop
while (grid->CheckComplete() == false) {
    int x = startPosition[0];
    int y = startPosition[1];
    vector<int> incrementalPath;

    // Initialise vector
    incrementalPath.push_back(maze[x][y - 1]); // North
    incrementalPath.push_back(maze[x][y + 1]); // South
    incrementalPath.push_back(maze[x - 1][y]); // West
    incrementalPath.push_back(maze[x + 1][y]); // East

    //for (int pathCheck : incrementalPath) {
    //    cout << pathCheck << endl;
    //}

    // Sort path in chronological order
    std::sort(std::begin(incrementalPath), std::end(incrementalPath));

    // if next increment in path is equal to next increment in array, move to it
    if (incrementalPath[0] == maze[x][y - 1]) {
        startPosition[1] = y - 1;
        grid->MovePlayer(movedirection::up);
    }
    else if (incrementalPath[0] == maze[x][y + 1]) {
        startPosition[1] = y + 1;
        grid->MovePlayer(movedirection::down);
    }
    else if (incrementalPath[0] == maze[x - 1][y]) {
        startPosition[0] = x - 1;
        grid->MovePlayer(movedirection::left);
    }
    else if (incrementalPath[0] == maze[x + 1][y]) {
        startPosition[0] = x + 1;
        grid->MovePlayer(movedirection::right);
    }
}
// grid->PrintGrid();
}

```

Maze Pathfinding Implementation - Liam Smalldon

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

Instructor

A very good report and a really nice implementation.

Some of the issues you had are quite solvable - you can actually cast between enums and integers, so could've solved this issue trivially. That aside, however, you've done very good work here, well done.

78/100

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13