

Assessment 2 - ZE - 1908686

by Zack Evans

Submission date: 11-Jan-2022 12:39PM (UTC+0000)

Submission ID: 168418865

File name: 18253_Zack_Evans_Assessment_2_-_ZE_-_1908686_394416_32925258.docx (92.7K)

Word count: 1218

Character count: 5855

Advanced Software Development

Level 5 Assessment – 2

Zack Evans
1908686

Table of Contents

Introduction 3

Dijkstra’s Algorithm..... 4

The A* Algorithm 4

Comparison 4

Assessment Implementation 6

References 7

Figures..... 8

Introduction

The purpose of the assessment is to go over go over two different algorithm's. Dijkstra's Algorithm and the A* Algorithm. From this, the report needs to reflect on how they work and the comparisons between the two. After the description of the two algorithm's. The reflection on the development process of implementing a algorithm and details about the algorithm.

Dijkstra's Algorithm

This algorithm is used to find the shortest path from the starting position to the target position. For this to work we will need a bunch of different locations (nodes) and roads to connect them together (edges). From this we want to travel from the starting location to the end location in the shortest distance. First the algorithm makes a list of the nodes and records three pieces of information:

- What's the shortest distance
- Which node was the previous in the shortest path?
- Has the node been checked yet?

Following each edge (the roads we use) from the start node (starting location) we can record the distance between each node directly. This would also record that the start node was the previous one. Meaning that the node is done and marked as checked.

Next, we repeat over and over as we pick the node with the shortest distance that hasn't been checked. Jumping into that node and following all the edges that lead from it. Then for each location directly reached from our current node that hasn't been checked is compared with the distance recorded and whether the current node can reach its destination in a shorter distance. If the current path is the shortest then the algorithm updates the tracker with the current shortest path. Once all the edges (roads) have been checked we check the node and move onto the next. It then repeats the step.

Finally, when all the nodes have been checked, it traces back through the previous nodes its recorded to find the shortest path between the start and end node.

The A* Algorithm

This is another algorithm that finds the shortest path between two points. Like Dijkstra's algorithm the A* uses nodes (locations) and edges (roads). If we have a bunch of nodes that are linked via edges. Each edge has a cost, and from this the A* wants to go from one node to another.

From the starting location we don't know the path, so from this the A* algorithm picks a random location and then looks to see where it can go from there. It then repeats this process until it cannot find a new node or finds the destination. This method is the first "depth" search. It goes as deep as it can before it considers planning its route.

After the previous step, the algorithm then performs estimate between the start and end locations between the nodes. From this the algorithm gives different scores to each edge depending on the node. So, if one edge has a better estimate over another edge then the algorithm picks the better estimate. Unlike where the Dijkstra's algorithm calculates everything on its own, user interaction is needed to take in a variable for the heuristic function. This is used for the estimation between nodes and from the start to the end.

Comparison

One of the biggest differences between is that A* requires a user parameter. Unlike Dijkstra that uses the real cost of each path, A* uses both the real and approximate cost from each node to the goal. If the heuristic function is calibrated correctly then this can lead to promising results but if an

Zack Evans
1908686

incorrect value is inserted this can lead to inefficient pathing between the start and end nodes. This means overall that A* can be better if it is tested enough with different heuristic values. Overall Dijkstra's algorithm is faster to implement while A* takes some time calibrating the estimate to get best results.

Next is A* can be faster than Dijkstra's algorithm due to the heuristic value being used to use the best estimate paths instead of all the paths to calculate. This means that it can search less nodes overall to find the final path the best estimate due to A* approximating the future cost to find the cheapest path. But if the puzzle is small enough there wouldn't be enough of a difference to tell between the two algorithms' if there are few nodes.

Assessment Implementation

When implementing the solution algorithm to the maze. The algorithm changed completely once optimisation had to be taken into place. At first an algorithm that works like Dijkstra's or the A* algorithm would've been used but then noticing that the player position doesn't change as well as the goal. A simple algorithm was implemented instead due to the simple course layout.

Before coding the solution, some research was done to see how the maze worked. At first a solution that kept track of each intersection was going to be used to plan and calculate the least cost path but then after noticing that the maze layout is generated like each other a specialised algorithm was used instead. Due to no planning was needed to be taken place within the algorithm the whole code would just move the place to the expected position of the flag.

First the player position is set to default and then detection is done [f1]. This detection only checks the top, left and right squares due to the bottom square isn't needed to be check due there isn't any need to go down. Compared to the first solution which was to return the entire grid back which means that the accesses were default at 400. This reduces the accesses by more than 50% [f2].

Next would be the conditional movement. Due to the flag's destination is always in the top right. The conditional movement leans towards moving up and right. Depending on the surrounding squares the player moves in different directions. By default, the player moves up unless there's a wall above the player. Then they move to the right. If the player is in a corner piece, then the player moves in the other direction until a gap is detected in the above wall [f3].

At the end of every movement cycle, the player checks to see if they have completed the course. If so, then it ends the algorithm. If not, then the player repeats the process until they have reached the goal.

The main reason why a custom algorithm was used over a generic one that finds the most cost-efficient path is due to the layout and position of the generated maze is similar each time. From the captured image [f4], the player is always in the bottom left while the flag is in the top left. Then the block above the player is always generated on the same y position but different x position. And this goes the same with the thin walls above it. Meaning that an algorithm that favours the player moving up and right will achieve minimum accesses and movements required.

References

[1] Dijkstra's Algorithm

<https://brilliant.org/wiki/dijkstras-short-path-finder/>

[2] A* Algorithm

<https://www.geeksforgeeks.org/a-search-algorithm/>

Figures

[f1]

```
class GridNavigatorAutomatic : public GridNavigator
{
private:
    int pos[2] = { 18, 1 };
public:
    GridNavigatorAutomatic();

    void Navigate();
};
```

```
// Grid checks
griditem up = this->grid->GetGridSquare(pos[1], pos[0] - 1);
griditem left = this->grid->GetGridSquare(pos[1] - 1, pos[0]);
griditem right = this->grid->GetGridSquare(pos[1] + 1, pos[0]);
```

[f2]

```
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: 71
Mean accesses: 128
Maximum accesses: 275
Minimum moves: 34
Mean moves: 56
Maximum moves: 124
Minimum wall hits: 0
Mean wall hits: 0
Maximum wall hits: 2
```

[f3]

```

// Conditional movement
if (left == griditem::wall &&
    up == griditem::wall) {
    bool run = true;
    while (run) {
        if (up == griditem::wall) {
            result = this->grid->MovePlayer(movedirection::right); pos[1]++;
            up = this->grid->GetGridSquare(pos[1], pos[0] - 1);
        }
        else run = false;
    }
}
else if (up == griditem::wall &&
    right == griditem::wall) {
    bool run = true;
    while (run) {
        if (up == griditem::wall) {
            result = this->grid->MovePlayer(movedirection::left); pos[1]--;
            up = this->grid->GetGridSquare(pos[1], pos[0] - 1);
        }
        else run = false;
    }
}
else if (up == griditem::wall) {
    result = this->grid->MovePlayer(movedirection::right);
    if (result == moveresults::ok) pos[1]++;
}
else { // Default
    result = this->grid->MovePlayer(movedirection::up);
    if (result == moveresults::ok) pos[0]--;
}

```

[f4]

```

W W W W W W W W W W W W W W W W W
W                                     T W
W   W W W W W W W W W W W W W W W
W                                     W
W W W W   W W W W W W W W W W W W
W                                     W
W W W W W W W W W W W W W W W W
W                                     W
W                                     W
W           W W W W W W W W W W W
W           W W W W W W W W W W W
W           W W W W W W W W W W W
W           W W W W W W W W W W W
W                                     W
W                                     W
W                                     W
W           W W W W W W   W W W W W W
W                                     W
W P                                     W
W W W W W W W W W W W W W W W W W

```

Zack Evans
1908686

Assessment 2 - ZE - 1908686

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

56/100

Instructor

A reasonable report, however your sources are both incorrectly cited and inappropriate for level 5 work (avoid wikipedia especially!)

A good effort at an algorithm, not perfect as I think you know, but of a good standard overall, using some reasonable conditional logic etc.

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9