

Maze Pathfinding implementation Ass 2

by Julian Milligan

Submission date: 11-Jan-2022 04:43PM (UTC+0000)

Submission ID: 168470362

File name: 16671_Julian_Milligan_Maze_Pathfinding_implementation_Ass_2_394416_394584061.docx
(252.06K)

Word count: 1616

Character count: 7274

Name – Julian Milligan

Student number – 1904804

Course – Bachelor of Science (Part-time): Computing (Software Engineering)

Module – L5 - Advanced Software Development

Module code - ACCA5011

Assignment title - Maze Pathfinding implementation

Date – 01/12/2021

Table of Contents

Introduction	3
Comparison between Dijkstra`s Algorithm and A* Algorithm.....	3
Dijkstra`s Algorithm.....	3
A* Algorithm	4
A comparison between both.....	5
Reflection of the code used	6
Code Testing.....	6
Bibliography	7
Other reading and viewing.....	8
Appendix	8
Figure 1.....	9
Figure 2.....	9
Figure 3.....	9
Figure 4.....	9

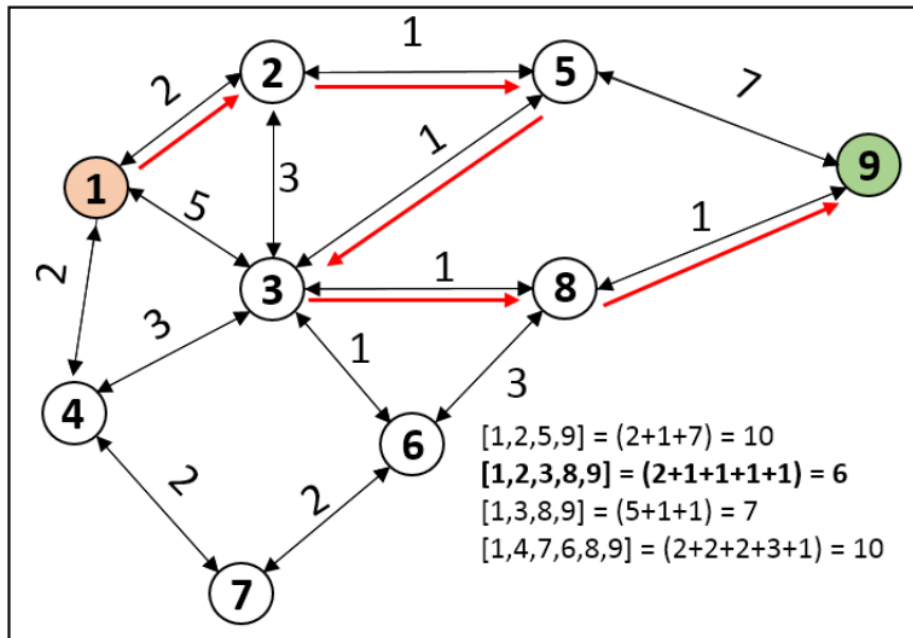
Introduction

This report and the following code are to describe the both the Dijkstra's and A* Algorithm. An algorithm is a process that follows a set of rules to solve a problem. Both the algorithms in this report are path finder algorithms and can be found in GPS devices , various on-line map applications and games. These two algorithms are designed to find the shortest path between two points based on the weight / distance between the start and end point. To archive this a graph is used as an abstract mathematical object [1]

Comparison between Dijkstra's Algorithm and A* Algorithm

Dijkstra's Algorithm

Dijkstra's Algorithm was created in 1956 and published in 1959 by Edsger W Dijkstra to help him find the shortest path between Rotterdam to Groningen . This algorithm is classed as a greedy algorithm [2]. The idea is to plot all the points or nodes into a graph and calculating the distance or weight between each node. Dijkstra's algorithm finds the shortest path from the start node to every other node. This method works well for small to medium sized graphs but for larger ones it becomes slower because of the method it uses.



Once a node has been visited and the weight calculated to the next node the algorithm will mark the shortest route. The above figure shows the weight between nodes. The nodes are exits for miners and the above figure is part of project to use the Dijkstra's Algorithm to help the miners escape in an emergency. [3]

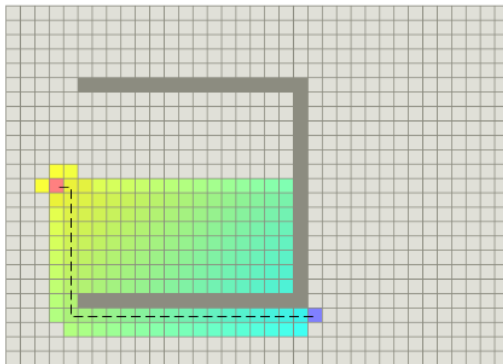
Below is the step the algorithm would use to find the shortest path:

1. Set all nodes except the start node's distance to infinity.
2. Set all nodes to non-visited.
3. Set the node with the smallest weight as the current node (C).
4. For each neighbouring (N) node of C add the weight of C with the weight of the edge connecting C – N . If it is smaller than the current weight of N set it as the new weight of N
5. Mark C as visited (V)
6. Repeat from step 3 until the destination node is set to V.

A* Algorithm

A* builds on Dijkstra's algorithm but unlike Dijkstra's A* does not look for the weight / distance between the start and every other node, it is only interested in the start and end node. This makes this algorithm more flexible and more widely used today. The reason is more flexible is because it can use heuristics to help guide its path.

As quoted from Amit's Thoughts on Pathfinding



"In the standard terminology used when talking about A*, $g(n)$ represents the exact cost of the path from the starting point to any vertex n , and $h(n)$ represents the heuristic estimated cost from vertex n to the goal. In the above diagrams, the yellow (h) represents vertices far from the goal and teal (g) represents vertices far from the starting point. A* balances the two as it moves from the starting point to the goal. Each time through the main loop, it examines the vertex n that has the lowest $f(n) = g(n) + h(n)$." [4]

A* algorithm steps:

1. Set all nodes except the start node's distance to infinity.

2. Set all nodes to non-visited.
3. Set the node with the smallest weight as the current node (C).
4. For each N of C add the current weight with weight of the edge connecting C – N and the weigh to the destination node (D) (heuristic). If it is smaller than N set as the new N.
5. Mark C as visited (V)
6. Repeat from step 3 until one of the N nodes is D.

The steps between the two are not that different and will be discussed further in the comparison.

A comparison between both.

Both algorithms are designed to find the quickest route between two points. Look at data from the paper by Dian Rachmawati and Lysander Gustin 2020 [5] for some small amounts of data. There was not a great deal of difference in the running time but there was in the number of loops needed to find the shortest route. See [figure 1](#) and [figure 2](#). 103 nodes were used in this dataset so 10 less loops by A* is significant but the time both took to complete with this small data set is around the same. There is a slight difference in the time 1 nanosecond. So, when it does come to datasets with millions of nodes the difference in time and loops should show that A* is faster.

Based on 1,030,000 nodes

The loop count would be

Dijkstra 3,600,000

A* 2,600,000

Time to complete

Dijkstra = 1.027116 seconds

A* = 1.026704 seconds

Reflection of the code used

The algorithm used to solve the maze was the wall Follower Algorithm.

“The wall follower algorithm is considered as the simplest maze solving technique since its algorithm, as the name states, only follows the wall” [6]

The algorithm is designed to look for a wall in the direction it's moving and keep one hand on a wall. Left or right hand , in this project the left hand was chosen.

The code moves the player up with the wall to their left and if there is wall in front of them, they keep moving up until the code encounters a wall in front of them. If a wall is found then code then looks if the player can move right , an if statement is then used to check if the player moves up and there is a wall.

If there is a wall a do / while loop is user to move the player right while checking for a hole in the wall above the player.

When the player hits a wall to the left the player is turned around and another do / while loop is used to progress the player forward again looking up for a hole in the wall.

The full code can be seen in [figure 4](#).

The code can be downloaded from GitHub at <https://github.com/1904804/ASDPathfindingAssignment.git>

Code Testing

Running the code option 2 , 10 times.

Run	Completed	Grid Access	Grid Moves	Wall hits
1	Yes	0	68	57
2	Yes	0	34	17
3	Yes	0	68	57
4	Yes	0	68	57
5	Yes	0	68	57
6	Yes	0	68	57
7	Yes	0	34	17
8	Yes	0	34	17
9	Yes	0	68	57
10	Yes	0	34	17

Due to results in the grid->PrintGrid() was used to view the path of the player through the grid. There seems to be two paths used to complete the maze.

1. When the player can move up and then right and pass through an opening on to the next level with out the need to go left.
2. The other is where the player moves up and needs to go left to get to an opening to the next level.

Running option 3 would not produce any results so the value was reduced to 10 and it ran. See [figure 3](#) as per the manual test the same pattern of results is seen. Even though run the code via option 2 would not produce any errors or failed maze attempts.

During testing the lines

```
cout << "Move right ok " << " and loop nun = " << i << endl;  
grid->PrintGrid();
```

Where used to show the path of the player moving and the top line was used as an indication if a wall had been seen and a turn made.

Bibliography

- [a. M. J. H. Kairanbay M, "A Review And Evaluations Of Shortest Path Algorithms.," *Int J. of Sci. & Tech. Res*, vol. 6, no. 2, p. 99, 2013.
1
]
- ["W3 Schools," W3 Schools, [Online]. Available: <https://www.w3schools.in/data-structures-tutorial/greedy-algorithm/>. [Accessed 01 01 2022].
2
]
- [M. U. o. S. & T. R. M. K. A.-O. M. U. o. S. & T. R. M. D. B. M. U. o. S. & T. R. M. D. B. M. U.
3 o. S. & T. R. M. A. Ur Rehman, "EMERGENCY EVACUATION GUIDANCE SYSTEM FOR
] UNDERGROUND MINERS," 24 02 2019. [Online]. Available:
https://www.researchgate.net/publication/331484960_EMERGENCY_EVACUATION_GUIDANCE_SYSTEM_FOR_UNDERGROUND_MINERS. [Accessed 01 01 2022].
- [[Online]. Available:
4 <http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html>. [Accessed
] 01 01 2022].
- [D. R. a. L. Gustin, "Analysis of Dijkstra's Algorithm and A* Algorithm in Shortest Path
5 Problem," *Journal of Physics: Conference Series*, 2020.
]
- [J. G. S. A. M. L. P. S. L. U. A. J. C. B. M. A. D. C. A. Z. A. D. ay Robert B. del Rosario,
6 "Modelling and Characterization of a Maze-Solving Mobile Robot Using Wall Follower
] Algorithm," *Applied Mechanics and Materials (Volumes 446-447)*, pp. 1245 - 1249, 2013.

Other reading and viewing

<https://uwtsd.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=5151ad63-a4b5-4716-b8e0-adf301215767>

[Maze Solving Algorithm - Wall Follower \(liquiseach.com\)](#)

[Buckys C++ Programming Tutorials - 36 - Multidimensional Arrays](#)

[Graph Data Structure 4. Dijkstra's Shortest Path Algorithm - YouTube](#)

[C++ Data Structures: Dijkstra's Algorithm - YouTube](#)

[Dijkstra Algorithm Coding C++ | Shortest Path | Implementation | Graphs | Greedy - YouTube](#)

[Path Planning - A* \(A-Star\) - YouTube](#)

[Sorting in C++ - YouTube](#)

Appendix

Figure 1

Table 4. Running Time from starting point to 1st destination point using Dijkstra and A* algorithm

Test Number	Running Time Dijkstra (nanoseconds)	Running Time A* (nanoseconds)	Total Distance (in km)
1	997.200	996.800	4.504
2	998.200	996.900	4.504
3	997.800	997.400	4.504

Figure 2

Loop Counts

We also take 3 run of each algorithm and the result shown below.

Table 5. Loop counts from starting point to 1st destination point using Dijkstra and A* algorithm

Test Number	Loop Count Dijkstra	Loop Count A*	Total Distance (in km)
1	36	26	4.504
2	36	26	4.504
3	36	26	4.504

Figure 3

```

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: 0
Mean accesses: 0
Maximum accesses: 0
Minimum moves: 34
Mean moves: 64
Maximum moves: 68
Minimum wall hits: 17
Mean wall hits: 53
Maximum wall hits: 57

```

Figure 4

```
void GridNavigatorAutomatic::Navigate()
{
    moveresults mr{};

    do
    {
        if (grid->MovePlayer(movedirection::up) == moveresults::ok)
        {
        }
        else
        {
            if (grid->MovePlayer(movedirection::right) == moveresults::ok)
            {
                do
                {
                    if (grid->MovePlayer(movedirection::up) == moveresults::wall)
                    {
                        mr = grid->MovePlayer(movedirection::right);
                    }
                } while (mr == moveresults::ok);
            }
            else
            {
                do
                {
                    if (grid->MovePlayer(movedirection::up) == moveresults::wall)
                    {
                        mr = grid->MovePlayer(movedirection::left);
                    }
                } while (mr == moveresults::ok);
            }
        }
    } while (mr != moveresults::complete);
}
```

Maze Pathfinding implementation Ass 2

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

Instructor

50/100

A good report and reasonable implementation using the hand method. I would suggest, however, that accessing the grid would have been preferable to bumping into walls, but nonetheless, a reasonable implementation.

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10