

FinalReport

by Robert Jamieson

Submission date: 11-Jan-2022 08:41PM (UTC+0000)

Submission ID: 168499898

File name: 24101_Robert_Iwan_Jamieson_FinalReport_394416_1069757978.docx (381.17K)

Word count: 2060

Character count: 11472

Robert Iwan Jamieson (2004728)

The University of Wales Trinity Saint David
School of Applied Computing
Wales Institute of Science and Art

Advanced Software Development

Maze pathfinding implementation (ACCB5011)
Robert Iwan Jamieson (2004728)
Date of submission: 11.01.2022

Contents

Chapter 1 Introduction.....	4
Chapter 2 Algorithms	5
2.1 Dijkstra's algorithm.....	5
2.2 A*'s algorithm.....	6
2.3 Dijkstra and A* algorithm comparison	6
Chapter 3 Algorithm pathfinding implementation for a maze.....	7
3.1 Reflection on the development process of the implementation.....	7
3.2 Results of the implementations	7
3.3 Interesting portions of the code	8
Chapter 4 References.....	10
Chapter 5 Appendix – Appropriate code from the pathfinding program.....	11

List of figures

Figure 1: Dijkstra algorithm example. Inspired from source: [1].....5

Figure 2: A* algorithm example. Inspired from source [5].....6

Figure 3: Benchmark results from the unfinished algorithm.....7

Figure 4: Benchmark results from the finalized algorithm.....8

Figure 5: “MoveLeftIfPossible()” method showcase. 8

Figure 6: The main loop with methods.....8

Figure 7: Showcasing the "_troubleshoot" variable.....9

Chapter 1 Introduction

This report will first focus on describing the Dijkstra [1] and A* [5] algorithms and how they compare to each other. After this, a reflection on the development process regarding the implementation of a pathfinding algorithm to complete a random maze will be discussed. This includes a discussion on the algorithm chosen, the results of the implementations and the interesting sections of the code.

Chapter 2 Algorithms

Described below is the way the Dijkstra algorithm and the A* algorithm perform and how they compare to each other.

2.1 Dijkstra's algorithm

Dijkstra's algorithm exists to find the shortest path from a destination vertex to every other vertex or to a specific vertex. It's a greedy algorithm [2] because it first looks at the closest adjacent vertices and goes from there. The way the algorithm works is illustrated in figure 1 below.

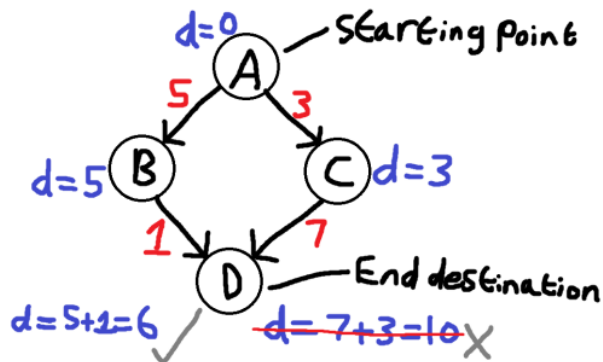


Figure 1: Dijkstra algorithm example. Inspired from source: [1].

All vertices except the starting vertex are initially considered an infinite length as they are unknown. The start vertex is always 0 distance [3].

How the Dijkstra's algorithm operates (with reference to figure 1):

1. A close adjacent vertex [3] that is unvisited as of yet is chosen and the cost between the **current vertex** [1] (starting vertex) and that adjacent vertex is calculated to find the distance between them. This process is repeated for all adjacent vertices close to the current vertex. The current vertex (A vertex) has now been visited – visited nodes are never visited again [1].
2. The **current vertex** is now the C vertex as it is the shortest path found so far [4]. The process in step 1 is repeated.
3. The distance between vertex C and D has been calculated and now the algorithm goes to the next unvisited vertex with the next shortest path found so far [1]: vertex B. B is now the **current vertex**. The process in step 1 is repeated.

After checking all routes, it has been found that A to B to D costs a total of 6 distance, whilst A to C to D costs a total of 10. So, although vertex C seemed like the shortest route to take at the beginning, it ended up being the longest route.

2.2 A*'s algorithm

The A* algorithm [5] is similar to the Dijkstra algorithm except it has a “heuristic element” [5]. This means the algorithm also takes into consideration how far approximately any vertex is away from the end destination [6]. This algorithm therefore does not always take the shortest path from one vertex to the next. It instead follows the (generally) more efficient [5] overall path – the vertex that is approximately closest to the end destination every time.

Figure 2 below is a demonstration on how the A* algorithm takes elements from the Dijkstra algorithm but considers the cost of estimated heuristic too.

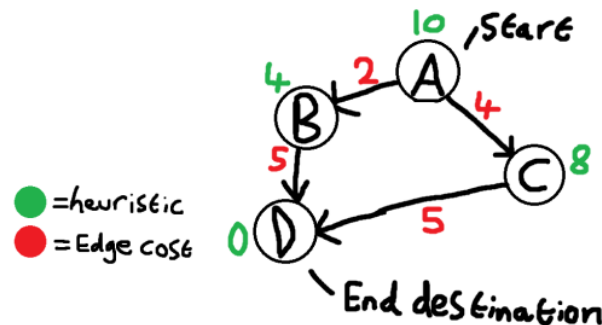


Figure 2: A* algorithm example. Inspired from source [5].

In Figure 2, the A* algorithm would quickly find that vertex B is approximately closer to the end destination than vertex C due to the heuristic value and would therefore pursue that path.

2.3 Dijkstra and A* algorithm comparison

As the Dijkstra algorithm follows vertices that are short distances away from each other without considering the bigger picture [5], the A* algorithm is more efficient. It instead calculates the most optimal [5] pathing and does so faster. The Dijkstra algorithm can get stuck down a long rabbit-hole as it is uninformed; it does not know where the end destination is [7]. This means it follows paths “that will never provide the shortest option” [5]. Following paths that are not the shortest way to the end destination is not only inefficient but wastes computing time and resources.

The A* algorithm on the other hand does not have this disadvantage. It is informed [7] of the end destination and can use that information to follow optimal paths that lead closer and closer to the end destination, using a heuristic function [5]. It does not necessarily follow shorter paths simply because they're shorter – it thinks about the bigger picture.

Dijkstra differs further from the A* algorithm because it can be used to find the shortest path between nodes as well as finding the end destination [3]. Whereas the A* algorithm is only interested in finding the optimal [5] path to the end destination node and is therefore more specific in its capabilities. But even though it has a more specific purpose, it is generally more efficient [5] than the Dijkstra algorithm.

Chapter 3 Algorithm pathfinding implementation for a maze

A custom algorithm has been created to solve a randomly generated maze 100% of the time (in 10,000 runs). Below is a reflection on the development process, results obtained and interesting parts of the code.

3.1 Reflection on the development process of the implementation

The algorithm implemented has been coded so that no matter where the walls spawn or where the entrances are to pass through those walls, the player will always reach the target. Neither the Dijkstra nor the A* algorithm have been implemented, instead a custom-made algorithm has been used.

Although the maze generation is random, the target will always spawn at the highest point in the maze. This knowledge was used to ensure that the player would go up at every opportunity. The player is never in danger of going out of bounds as there is always walls encapsulating the maze which the player can't pass through.

3.2 Results of the implementations

The first algorithm implemented went like this:

1. Player goes up as much as possible to a max of 20 times in a loop.
2. Player attempts to go up and right 20 times.
3. Player attempts to go up and left 20 times.
4. Repeat 2 and 3.

This was very inefficient, very maze specific and didn't always work. It was therefore scrapped.

The next algorithm implemented was made to be more efficient; it did not simply loop a direction 20 times. Instead a while loop was used to identify the position of walls one move at a time.

Benchmark results:

```
Minimum accesses: 37
Mean accesses: 82
Maximum accesses: 210
Minimum moves: 34
Mean moves: 78
Maximum moves: 194
Minimum wall hits: 17
Mean wall hits: 63
Maximum wall hits: 185
```

Figure 3: Benchmark results from the unfinished algorithm.

Unfortunately, there was a bug that wasn't calculating the correct player Y and the player was still hitting walls occasionally, causing the algorithm to be less efficient as seen in figure 3 above.

After optimizations, figure 4 below reveals a benchmark of the **finalized** version of the algorithm:

```
Minimum accesses: 54
Mean accesses: 111
Maximum accesses: 266
Minimum moves: 34
Mean moves: 56
Maximum moves: 130
Minimum wall hits: 0
Mean wall hits: 0
Maximum wall hits: 0
```

Figure 4: Benchmark results from the finalized algorithm.

The aim with this version was to ensure that above all else, the player never hit any walls. To do this, before moving in any direction, the algorithm always checks ahead for an immediate wall. By doing so, the mean moves it takes the algorithm to reach the target has improved drastically. Walls are never hit, although the mean accesses has increased a bit.

3.3 Interesting portions of the code

Below in figure 5 is code within the method “MoveLeftIfPossible()”. The interesting part of this code is that the player will only ever move left if there is no immediate wall and if there is an immediate wall above the player.

```
//Keeps moving player left if there is no immediate wall to the left...
//but an immediate wall does need to be above it.
void GridNavigatorAutomatic::MoveLeftIfPossible()
{
    _left = true;
    while (grid->GetGridSquare(_playerx - 1, _playery) != griditem::wall
        && grid->GetGridSquare(_playerx, _playery - 1) == griditem::wall)
    {
        grid->MovePlayer(movedirection::left);
        _playerx = _playerx - 1;
        PrintInfo(); //Only prints if troubleshoot is true;
    }
    _left = false;
}
```

Figure 5: “MoveLeftIfPossible()” method showcase.

As seen in figure 6 below, the move right method is always executed after the move up method. This ensures that after every entrance the player passes through, it always attempts to go right first. This is intentional because when the player’s coordinate $y = 1$, the player is almost always to the left of the target. By ensuring the player always goes right first, it takes fewer mean moves to reach the target.

```
//Keep looping for as long as the target has not been reached
while (grid->CheckComplete() == false)
{
    MoveLeftIfPossible();
    MoveUpIfPossible();
    MoveRightIfPossible();
}
```

Figure 6: The main loop with methods.

Robert Iwan Jamieson (2004728)

One of the most interesting parts of the code is when the troubleshooting code is executed, AKA when the variable “_troubleshoot” equals “true”. This is intended to be used for automatic mode, not benchmark mode.

```
// ~~~~~  
/*For "_troubleshoot":  
true CAN be used for Automatic mode (2). Prints Grid, movement direction...  
and Player X and Y coordinates so it's easy to troubleshoot.  
false is for Benchmark mode (3). No cout's.*/  
_troubleshoot = true;  
// ~~~~~
```

Figure 7: Showcasing the "_troubleshoot" variable.

As seen in figure 7, once the variable is enabled, it prints out the grid; the player's coordinates and the direction the player is moving. This helped with identifying that the player's Y coordinate was incorrect in some circumstances. Once that bug was resolved the mean moves lowered significantly, as illustrated in figure 3 and 4 from chapter 2.2.

Chapter 4 References

- [1] Isaac Computer Science, "Dijkstra's algorithm," Isaac Computer Science, [Online]. Available: https://isaacomputerscience.org/concepts/dsa_search_dijkstra?examBoard=all&stage=all. [Accessed 2 Jan 2022].
- [2] GeeksForGeeks, "Greedy Algorithms," GeeksForGeeks, 31 Jul 2021. [Online]. Available: <https://www.geeksforgeeks.org/greedy-algorithms/>. [Accessed 2 Jan 2022].
- [3] Baeldung, "Understanding Time Complexity Calculation for Dijkstra Algorithm," Baeldung, 13 Oct 2021. [Online]. Available: <https://www.baeldung.com/cs/dijkstra-time-complexity>. [Accessed 3 Jan 2022].
- [4] GeeksForGeeks, "Dijkstra's shortest path algorithm | Greedy Algo-7," GeeksForGeeks, 10 Sep 2021. [Online]. Available: <https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/>. [Accessed 3 Jan 2022].
- [5] Isaac Computer Science, "A* search algorithm," Isaac Computer Science, [Online]. Available: https://isaacomputerscience.org/concepts/dsa_search_a_star?examBoard=all&stage=all. [Accessed 4 Jan 2022].
- [6] GeeksForGeeks, "A* Search Algorithm," GeeksForGeeks, 24 Dec 2021. [Online]. Available: <https://www.geeksforgeeks.org/a-search-algorithm/>. [Accessed 4 Jan 2022].
- [7] M. Simic, "Dijkstra vs. A* – Pathfinding," Baeldung, 12 Nov 2021. [Online]. Available: <https://www.baeldung.com/cs/dijkstra-vs-a-pathfinding>. [Accessed 4 Jan 2022].

Chapter 5 Appendix – Appropriate code from the pathfinding program

```
#pragma once
#include "GridNavigator.h"
class GridNavigatorAutomatic : public GridNavigator
{
private:
    int _playerx;
    int _playery;
    bool _troubleshoot;
    bool _left;
    bool _up;
    bool _right;
public:
    GridNavigatorAutomatic();

    void MoveLeftIfPossible();
    void MoveUpIfPossible();
    void MoveRightIfPossible();
    void PrintInfo();

    void Navigate();
};

+++++

#include "GridNavigatorAutomatic.h"

GridNavigatorAutomatic::GridNavigatorAutomatic() : GridNavigator()
{
    //Starting player X and Y coordinates
    _playerx = 1;
    _playery = 18;
    //~~~~~
    /*For "_troubleshoot":
    true CAN be used for Automatic mode (2). Prints Grid, movement direction...
    and Player X and Y coordinates so it's easy to troubleshoot.
    false is for Benchmark mode (3). No cout's.*/
    _troubleshoot = false;
    //~~~~~
    //The below variables help in troubleshoot mode:
    _left = false;
    _up = false;
    _right = false;
}

//Keeps moving player left if there is no immediate wall to the left...
//but an immediate wall does need to be above it.
void GridNavigatorAutomatic::MoveLeftIfPossible()
{
    _left = true;
    while (grid->GetGridSquare(_playerx - 1, _playery) != griditem::wall
        && grid->GetGridSquare(_playerx, _playery - 1) == griditem::wall)
    {
        grid->MovePlayer(movedirection::left);
        _playerx = _playerx - 1;
        PrintInfo(); //Only prints if troubleshoot is true;
    }
    _left = false;
}
```

Robert Iwan Jamieson (2004728)

```
//Keeps moving player up if there is no immediate wall above.
void GridNavigatorAutomatic::MoveUpIfPossible()
{
    _up = true;
    while (grid->GetGridSquare(_playerx, _playery - 1) != griditem::wall)
    {
        grid->MovePlayer(movedirection::up);
        _playery = _playery - 1;
        PrintInfo();
    }
    _up = false;
}

//Keeps moving player right if there is no immediate wall to the right...
//but an immediate wall does need to be above it.
void GridNavigatorAutomatic::MoveRightIfPossible()
{
    _right = true;
    while (grid->GetGridSquare(_playerx + 1, _playery) != griditem::wall
        && grid->GetGridSquare(_playerx, _playery - 1) == griditem::wall)
    {
        grid->MovePlayer(movedirection::right);
        _playerx = _playerx + 1;
        PrintInfo();
    }
    _right = false;
}

//Prints info to help troubleshoot any issues (only when _troubleshoot = true)
void GridNavigatorAutomatic::PrintInfo()
{
    if (_troubleshoot == true)
    {
        if (_left == true)
        {
            cout << endl << "MOVE LEFT..." << endl;
        }
        else if (_up == true)
        {
            cout << endl << "MOVE UP..." << endl;
        }
        else if (_right == true)
        {
            cout << endl << "MOVE RIGHT..." << endl;
        }
        grid->PrintGrid();
        cout << endl << "Player Y: " << _playery;
        cout << endl << "Player X: " << _playerx << endl;
        cout << "+++++" << endl;
    }
}

void GridNavigatorAutomatic::Navigate()
{
    //Keep looping for as long as the target has not been reached
    while (grid->CheckComplete() == false)
    {
        MoveLeftIfPossible();
        MoveUpIfPossible();
        MoveRightIfPossible();
    }
}
```

FinalReport

GRADEMARK REPORT

FINAL GRADE

68/100

GENERAL COMMENTS

Instructor

A good report, , well written and considered. I like your use of self-drawn figures - worth mentioning that you can take non-copyrighted figures from other sources and incorporate them into your work (citing them of course), but I really like the ones you created.

Your approach to solving the problem was nice and systematic. Not perfect in that it didn't find an optimal path, but it arrived at a solution in a reasonable number of steps and is, overall, a good effort.

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12