

1910716 Dijkstra's Maze Solving Solution FINAL

by Ollie Bird

Submission date: 13-Jan-2022 05:47PM (UTC+0000)

Submission ID: 168839236

File name: 18246_Ollie_Bird_1910716_Dijkstras_Maze_Solving_Solution_FINAL_394416_119014840.docx
(141.81K)

Word count: 1710

Character count: 8552

Description, Comparison and Implementation of Dijkstra's and the A* Shortest Path Search Algorithm

Oliver Bird

Apprentice Software Engineer – BT , Data Services

1910716@student.uwtsd.ac.uk

ABSTRACT – A short description of both the A* and Dijkstra's shortest path search algorithms are provided. Both the time and space complexities of each are evaluated along with a "pros" and "cons" considered for each. The use cases for each are compared. It is detailed that A* is a more efficient algorithm due to $O(V^2)$ time complexity in the worst case. An implantation of Dijkstra's is then demonstrated on a simple 20x20 randomly generated maze program.

1. Dijkstra's

A) Description - For a given graph or 2D array, the following steps will find the shortest path between 2 given nodes [1]:

- Flag a single node as the starting position
- Set the weight of that node to 0
- Set the weights of all other graph nodes to infinity
- For each cell, find all possible navigable cells
- For each neighbour, evaluate the cost of reaching it
- Repeat steps 4 and 5 for each node within the graph, without evaluating cells which have already been visited, as their weights have already been evaluated
- Upon completion, evaluate each cell starting from the target, choosing only the lowest recorded path weight when selecting the next cell in the shortest path.
- Repeat step 7 until the starting node is reached.

The algorithm maps all nodes across the graph, discovering all possible paths across it. It then constructs the shortest path from the end node to the start, which can be reversed to discern the final shortest path.

B) Complexities – Depending on the exact implementation of the algorithm, it can have various time and space complexities. Fig.1 details the examples of this where V is the number of vertices and E is the number of edges:

	Best Case	Worst Case
Time	$O((V + E) \log V)$	$O(V^2)$
Space	$O(V + E)$	$O(V + E)$

Fig 1. Table of complexities for Dijkstra's

C) Pros – Dijkstra's has several positive benefits when searching for the shortest path:

- Multiple paths are found throughout the maze structure. The algorithm can be slightly modified to allow for other paths to be recorded. In the case of the maze changing, additional routes can be used to allow the maze to be solved.
- The algorithm will always return the shortest path in the case for which this is an absolute requirement.
- On a small graph, the algorithm is efficient, and in the best case does not grow exponentially more difficult as nodes and edges increase.

D) Cons – There are a number of negative use cases and issues when using Dijkstra's:

- Dijkstra's cannot be used to evaluate a negative edge weight.
- It performs a complete search of the entire graph, regardless of if a node / path can actually lead to the end node. This is computationally inefficient as redundant nodes are evaluated.
- In the worst case, the algorithm becomes heavily inefficient as the number of vertices increases. This means, for large graphs the time complexity hinders the usefulness of the algorithm in the case of which time is a valued resource.

It has been found previously that a minor modification to the algorithm storage methodology can increase its efficiency further [2] [3]

2. A*

A) Description – For a given graph or 2D array, the following steps will find the shortest path between 2 nodes.

- Begin on the starting node where $Gcost$ is 0 and $Hcost$ is unknown.
- Evaluate all nodes surrounding the current node if they are able to be traversed to and have not already been evaluated in this method.

- For each available node, evaluate the *Fcost* value and overwrite any current value stored there in the case of which the new value is lower.
- Once a node has been evaluated in step 2, mark the node as previously evaluated.
- Repeat the above steps for the node with the lowest current *Fcost* until the final goal is reached.

The algorithm works by choosing the current best guess path found via its *Fcost*. It continuously loops through the graph selecting the node with the lowest estimated cost and evaluating its neighbours until the end node is found. This guarantees the shortest path but will only find a single path.

The following definitions must be noted [4]:

- *Fcost* – The estimated cost of reaching the end goal via a given node. This is calculated via $Hcost + Gcost$
- *Gcost* – The cost so far to reach the current node
- *Hcost* – The estimated cost to reach the goal from the current node.

B) Complexities – Due to the heuristic (guess like) nature of the algorithm, there is no definitive ‘best case’ description of its complexities. In the worst cases for both time and space, the algorithm follows a pattern of exponential growth of $O(b^d)$ where d is the current shortest path to the node and b is the branching factor of the current tree [5]

C) Pros – The A* algorithm has several positive benefits when searching for the shortest path:

- It is extremely efficient at determining the shortest paths when compared to other search algorithms [6].
- It is guaranteed to find the shortest path through a maze with a possible solution, and in the case of which a solution is not possible, will find the closest node to the end node with a traversable path.
- Due to being heuristic, the algorithm hones in on the best possible paths with priority, meaning redundant paths are less likely to be analysed.

D) Cons

- The algorithms’ efficiency is heavily dependent on the implementation of the heuristic function to determine *Hcost*. The evaluation of the Euclidean distance between a given node and the final node is the most optimal however is not always obvious or implementable solution, certainly in higher dimensional spaces.
- Both time and space complexity of the algorithm rise quickly as the number of nodes and available traversable paths increase.

3. Comparison

As shown previously, A* is more efficient at finding a single traversable short path through a given network [6], in both computational time and space. In the absolute worst case of its implementation, A* is the same as Dijkstra’s, therefore Dijkstra’s is the ‘worst case’ implementation of the A* algorithm.

For small graphs [6], it is found that both algorithms perform similarly enough to each other in finding the shortest path, however for maximum efficiency, a good implementation of A* will always be faster than Dijkstra’s, as it is not required to access each node to determine the final solution.

Dijkstra’s has the potential advantage of finding multiple paths as the graph is evaluated in full. Rather than in A* which finds an individual solution and disregards other routes, Dijkstra’s can be adapted to solve minor changes in the graph structure.

4. Dijkstra’s Code Implementation

The implementation of Dijkstra’s algorithm to solve a 20x20 maze within C++ follows the below pattern:

- Map each node of the maze once to a 2D array of integer values where walls are 0, empty space is INT_MAX (a very large integer value to simulate infinity within C++), the starting position is 0 and for simplicity, the goal is 100.
- Create a copy binary matrix of solved and unsolved coordinates which matches the current maze where solved =1 and unsolved =0.
- Mark the initial path weight of the starting node as 0 and all other nodes as having an infinite path weight.
- For each node, evaluate the weights for each adjacent node, given that they must not be a wall or a previously traversed node.
- Choose the lowest adjacent weight and fill out the current node weight with this weight plus 1, and mark its coordinate as solved in the binary matrix.
- Continue until the final node is reached.
- Select the end node of the graph and choose its lowest weight neighbour.
- Record the direction of movement between the two nodes and add its inverse to the current path (up is down, left is right etc).
- Set the new node as the new end node and repeat until the starting node is reached.
- Feed the recorded path in reverse to the player to cause it to traverse the graph.

The above algorithm results in a constant number of grid accesses for any maze supplied by the program of 400, as each node is accessed only once. Fig2. And 3. show examples of the solution outputs, when ran 10,000 times.

```

Minimum accesses: 400
Mean accesses: 400
Maximum accesses: 400
Minimum moves: 34
Mean moves: 49
Maximum moves: 90
Minimum wall hits: 0
Mean wall hits: 0
Maximum wall hits: 0

```

Fig 2. Example program output

```

Minimum accesses: 400
Mean accesses: 400
Maximum accesses: 400
Minimum moves: 34
Mean moves: 49
Maximum moves: 86
Minimum wall hits: 0
Mean wall hits: 0
Maximum wall hits: 0

```

Fig 3. Example program output

The program will never hit a wall, as paths containing walls are never evaluated as they are skipped in the node evaluation loop. Nodes which are currently not accessible via a traversable path are also skipped to later iterations of the loop. This is shown in Fig. 4.

```

if ((above == 99 || above == INT_MAX) &&
    (below == 99 || below == INT_MAX) &&
    (left == 99 || left == INT_MAX) &&
    (right == 99 || right == INT_MAX)) {
    // Must be flagged to true in order for the loop to continue
    changesMade = true;
    continue;
}

```

Fig. 4 Non-traversable Node exclusion

The use of a binary matrix to mark nodes as visited allows a great reduction in computational time as the loops continue to solve all traversable paths.

```

if (maze[x][y] != 99 && solved[x][y] == 0)

```

Fig 5. Already visited node exclusion

As the loop calculates the current node path weight as an incrementation of the lowest neighbour path weight, the starting node is given a special skip over condition to allow it to remain as a 0 value.

```

// Corrects for the starting position
if (maze[x][y] == 0) {
    fillValue.push_back(-1);
}

```

Fig 6. Start position correction

The path is recorded as a string of instruction characters which are converted into movements for the player. This is processed by reversing the shortest path options from the end node and working backwards to the starting node. This works as all path weights are relative to the starting node.

```

vector<int> lowestPathPerCord;
int above = maze[xPos][yPos - 1], below = maze[xPos][yPos + 1],
    left = maze[xPos - 1][yPos], right = maze[xPos + 1][yPos];
lowestPathPerCord.push_back(above);
lowestPathPerCord.push_back(below);
lowestPathPerCord.push_back(left);
lowestPathPerCord.push_back(right);
std::sort(std::begin(lowestPathPerCord), std::end(lowestPathPerCord));
int lowestCell = lowestPathPerCord[0];
// Converts the iteration through the maze into an instruction for the character to move
if (lowestCell == above) {
    startPosition[1] = yPos - 1;
    path.push_back("d");
}
else if (lowestCell == below) {
    startPosition[1] = yPos + 1;
    path.push_back("u");
}
else if (lowestCell == left) {
    startPosition[0] = xPos - 1;
    path.push_back("l");
}
else if (lowestCell == right) {
    startPosition[0] = xPos + 1;
    path.push_back("r");
}
}

```

Fig 7. Path to character conversion

References

- [1] 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 08 12 2021].
- [2] R. P. S. M. Y. RK Arjun, "RESEARCH ON THE OPTIMIZATION OF DIJKSTRA'S," *International Journal of Science, Technology & Management*, vol. 04, no. 01, 2015.
- [3] S. Saunders, "A Comparison of Data Structures for Dijkstra's Single Source Shortest Path Algorithm," University of Canterbury. Computer Science, Canterbury, 1999.
- [4] S. Lague, "A* Pathfinding (E01: algorithm explanation)," 14 12 2014. [Online]. Available: https://www.youtube.com/watch?v=L-WgKMFuhE&t=531s&ab_channel=SebastianLague. [Accessed 01 01 2022].

[5] S. J. N. Peter Russell, Artificial Intelligence: A Modern Approach (2nd Edition), Prentice Hall, 2002.

[6] L. G. Dian Rachmawati, "Analysis of Dijkstra's Algorithm and A* Algorithm in," in *J. Phys.: Conf. Ser.* *1566 012061*, 2020.

1910716 Dijkstra's Maze Solving Solution FINAL

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

92/100

Instructor

A great effort, Ollie - an excellent report and a really impressive implementation. Really well done.

PAGE 1

PAGE 2

PAGE 3

PAGE 4
