Pathfinding through Artificial Intelligence

Abstract

The purpose of this project was to examine all possible solutions to finding a path on a map, while only knowing the map, the starting point, and the end point. A variety of algorithms and searches were implemented to find and compare possible solutions from the starting point to the end point. These searches include a breadth-first search, uniform cost search, iterative deepening, and three heuristics (Manhattan, Euclidean, and Greedy). Each algorithm was written using C++ and the solutions generated by each algorithm were compared in terms of movement cost, distance, and time. All searches except for the uniform cost search resulted in the minimum number of moves (distance) but did not result in the lowest cost. No search was able to find the optimal path, but the Manhattan and Greedy searches found the lowest cost path, also in the lowest amount of time. The breadth-first search was found to be slowest and least efficient in terms of time and movement cost. Optimizations and improvements are possible, as well as adding twists or variations to the problem to better test the algorithms.

Introduction

This project attempts to tackle the pathfinding problem. This is a scenario in which a starting point (A) and end point (B) are known, as well as the map that these points are on. The pathfinding problem takes this scenario and tries to find a path from the starting point to the ending point – essentially going from Point A to Point B. This problem is solved by using artificial intelligence to find possible paths from the starting point to the end point. These paths are then compared using multiple metrics, the most noticeable being time (in seconds), cost (movement cost based on the type of tile), and distance (number of moves).

While this problem can be solved in a variety of ways, the simplest and most effective solution started with creating the map. The map was divided up into a 15 by 20 grid, with each square containing a “tile” that holds an x-coordinate, y-coordinate, land type and land cost. Each tile also contained information concerning the four neighbors in each cardinal direction around it. From here, search algorithms could be conducted to find a path from the start to the end, while also counting the number of moves and the cost of each move.

Algorithms

To solve the pathfinding problem, multiple algorithms needed to be implemented. This way, the effectiveness and efficiency of each algorithm could not only be tested, but also compared.

The first algorithm tested was a breadth-first search. This algorithm creates a tree, using the starting map tile as the root of the tree and creating four branches at every level. Each node has four branches: east, west, north, and south. These branches are added to a queue which contains all the tiles that are to be examined. The search repeats, checking every node on the tree level-by-level until the goal tile is found. For example, if the current tile at the front of the queue does not equal the goal tile, then the current tile’s branches will be added to the queue and the next element on the queue. This process checks for a solution at each level, which means that this search will always find a solution, but it may not be the best solution. This is because there may be a better solution in a lower level or further down in the current level.

Next, a uniform cost search was implemented. A uniform cost search works similarly to the breadth-first search with a few important differences. As mentioned before, a breadth-first search may not find the best solution, and this is because all branches are assumed to have the same cost. The uniform cost search factors in the cost of each branch and uses a priority queue instead of a queue. The priority queue allows the branches to be sorted by a factor of cost, effectively choosing the branch with the lowest cost and is closest to the goal. The most difficult aspect of the uniform cost search is determining what lowest cost means, as in the pathfinding problem time, distance, and overall movement cost could all be used. For this problem, the movement cost was the primary means of prioritization and the distance from the current tile to the goal was a secondary means, in the event that the cost was tied.

Another algorithm that was used to solve the pathfinding problem was iterative deepening. Iterative deepening utilizes a depth-first search that has a maximum limit and will increase the limit until a solution is found. Unlike a breadth-first search that checks all branches at a given level, a depth-first search checks all branches in a given node, prioritizing going deeper in the tree, as opposed to checking all the nodes at one level. This means that some branches will be checked at a lower depth before others are checked at a higher depth. This is least efficient when the solution is found in one of the last branches checked, but it will always return a solution and will also make sure that it is the most optimal solution as long as all costs are the same (in this problem not all costs are the same, so iterative deepening will find the path with the fewest number of moves, even if those moves result in higher movement costs than another solution). Iterative deepening uses a depth-first search algorithm, but places a maximum depth limit on the search, meaning that the path must be found at or before a given level. Iterative deepening starts the limit at 0, and repeats the search, slowly incrementing the limit until a solution is found. This increases efficiency and finds the optimal path (in terms of moves, not movement cost). When running this algorithm, there is an option to repeat states. This means that a certain path can either revisit a tile, or once it has been visited it cannot cross it again. The solution to this is creating a visited list for each path, not allowing a path to visit the same tile twice. This solution will decrease time to find a path and find a better path, but also makes the programming more difficult.

The final algorithm used is called the A\* heuristic. This is a unique search because the programmer (or user), gets to define what the heuristic is. A heuristic is basically the function that takes what is known about the situation - how far the current state is from the start and how far the current state is from the end – and then finds the best branch based on the situation. This is an informed search that uses what is known about the situation to find the next best move. How that next best move can be found is ultimately decided by the type of heuristic. A heuristic function has two main parts: g(n), the cost (distance) from the start to current tile, and h(n), the cost (distance) from the current tile to the goal. For the pathfinding problem, three heuristic functions were used: Manhattan, Euclidean, and Greedy.

The Greedy heuristic is likely the simplest and easiest to understand. The function looks at what is branch is closest to the desired result (in terms of x and y from reaching the end point) and takes the branch. If any branches are same distance away, then the branch with the lowest movement cost will be taken. The Manhattan heuristic also uses the x and y distance for each branch but examines both the distance from the start to current, or g(n), and the distance from current to the goal, or h(n). The Euclidean heuristic is calculated the same way as the Manhattan, but the distance is solved by using Pythagorean’s theorem. The following equations for each function are as follows.

*Table 1 – A\* Heuristic Functions*



Results

To test the effectiveness and efficiency of each algorithm, a set of coordinates were established. Each algorithm could then be tested, using a consistent environment where the results are easily comparable. Not only could my personal programs be compared, but they could also be compared to other individuals who tried to solve the pathfinding problem. For this paper, a starting point of (2,0) and an ending point of (13,18) will be assumed for all tests. The following figures displayed will start at “S” which is (2,0) and end at “E” which is (13,18). The path found will be marked by “#” and all checked tiles not on the path are marked with “X” or “F”.

Using the test conditions described above, there were three primary metrics: time (how long did the search take until a path was found), distance (how many moves were taken and how much ground was covered), and movement cost (what was the combined cost of the moves, see Table 2 down below).

*Table 2 – Movement Cost by Tile*

|  |  |
| --- | --- |
| Tile | Movement Cost |
| Road | 1 |
| Field | 2 |
| Forest | 4 |
| Hill | 5 |
| River | 7 |
| Mountain | 10 |
| Water (Lake) | Unpassable |

Each algorithm had unique strengths and weaknesses which will be described in more detail later, but the data can best be displayed through the figure below. This chart shows the number of moves in the solution that each search found, along with the total cost of those moves.

*Figure 1*

All of the A\* heuristic functions as well as the breadth-first search found a solution with the minimum number of moves, 29, but found different paths from the start to the end. These algorithms were programmed to find the most efficient path in terms of distance, but not necessarily movement cost, resulting in different paths and movement costs (see Figures 3 through 5 for more details). This is where the uniform cost path should have stood out from the rest, but it did not.

*Figure 2 – Uniform Cost Search Results with Map*

![Diagram, engineering drawing

Description automatically generated]()



Because of the priority queue structure that the uniform cost search uses, how the branches are prioritized determines the path that is found. Because uniform cost search finds the path with the lowest cost, it will try to account for both distance and movement cost. There is great success in the lower half of the map, starting with (1,8) highlighted in red, all the way until the end goal of (13,18). On this section, the uniform cost search is able to find the road and take it all the way to the end goal. Unfortunately, the uniform cost search does not handle the mountainous terrain in the northwest corner of the map. As a result, the path that is found avoids the mountains at all cost, as well as the hills and some potential shorter paths and instead winds its way to the river. Once the path crosses the river, it is incredibly efficient, utilizing every available road tile to reach the end goal. The other searches did not incorporate cost of each move into their decision making and focused more on how each move would affect the distance traveled and distance remaining.

*Figure 3 – Breadth-First Search Results with Map*

![Diagram, engineering drawing

Description automatically generated]()

*Figure 4 – A\* Greedy and A\* Manhattan Search Results with Map*

*![Diagram, engineering drawing

Description automatically generated]()*

Even though the breadth-first search, Manhattan search, and Greedy search all have unique code and decision making, the result is nearly the same. All of these programs focus on getting from the start to the finish in the fewest moves possible. The breadth-first search goes straight south toward the bottom of the map, and then straight east to the end point. The Manhattan and Greedy searches both go straight east to the right side of the map, and then straight down to the end point. Both of these searches, along with the Euclidean search, result in 29 moves and varied movement costs: 94 for breadth-first, 74 for Manhattan, 74 for Greedy, and 118 for Euclidean.

*Figure 5 – A\* Euclidean Search Results with Map*

*![Diagram, engineering drawing

Description automatically generated]()*

The Euclidean search attempts to take the shortest distance possible, using Pythagoreans theorem. As a result, this solution starts by going through the mountains and hills, making no attempt to avoid them, while also cutting through the forest and adding unnecessary cost. Notice how there are very few roads or fields used in this path resulting in such a high cost.

*Table 3 – Algorithm Cost by Time from (2,0) to (13,18)*



One of the most interesting aspects of the pathfinding problem was the time it took to find each problem. Breadth-first took a substantial amount of time, mostly due to it’s time complexity of O(bd) where b is the branching factor and d is the depth of the search. The informed searches, Manhattan, Euclidean, and Greedy, did not examine every possible outcome and instead found the best path based on the information that was known at the time. The uniform cost search prioritized which branches to examine, in essence, pruning away certain paths that did not fit considering the moves already made. The time differences are difficult to tell from this problem and on an expanded map, the time differences may be clearer. For this project, the time differences do not matter, except for the breadth-first. Every informed search is under 1 millisecond, the uniform cost search is less than a second, and the breadth-first search gets crushed in proportion to the others.

Conclusion

Each algorithm discussed in this paper had significant strengths and weaknesses. The breadth-first search found a path with the fewest number of moves possible but did it at a much slower rate. The Manhattan and greedy searches found identical paths, both finding a solution with the minimum number of moves and the lowest cost discovered by any algorithm (but not the lowest path available). The Euclidean search had the highest movement cost, because while it found a path in the minimum number of moves, it had to cut through high-cost tiles and regions to do so. The lowest cost search was able to find path that used the lowest-cost tiles and reached the end goal but had to backtrack a little to do so. This project was not a failure, but there are optimizations and improvements for the future.

For this project, an attempt was made to stick to the search described as accurately as possible, resulting in very little improvements or changes. Changes could be done on certain algorithms such as pruning, better decision-making, sorting the fringe lists, or combining searches. For example, an A\* breadth first could be used to find all possible solutions using the informed search decision making. Using Manhattan or greedy decision-making without pruning and examining all branches still could help find a more efficient path faster.

Another possibility would be exploring a uniform cost bidirectional search. A bidirectional search runs two searches at the same time, one from the start and from the end, meeting at a point in the middle. The problem with this search, is that it works to find one point in the middle and with pathfinding, there are hundreds of possible “middles”. But, in the uniform cost search there is a point that starts the optimal path and if that is known, that point would become the middle.

Also, while not discussed in this paper, an eight-neighbor breadth-first search was tested. The original breadth-first search described in this paper has four neighbors (north, east, south, west), but a modified version would include four more (northeast, northwest, southeast, southwest). While this allows for more paths to be found, including paths with lower costs, the branching factor is far too high to find a path at higher depths.

And all of these improvements and adjustments assume that the map would look the same. The map could also be adjusted to have hexagonal tiles (like as in Settlers of Catan) or grouped polygons (like as in Risk). These changes would affect how the search is conducted by impacting the number of neighbors, the coordinate system, and how cost is counted. The results may look quite different, as part of the advantage to this problem is the consistency and predictability of the board. With a different coordinate system or different sized map, some algorithms would not be as strong as they are in the current environment and it would be quite interesting to see the impact and the results that follow.