Analysis of Pathfinding Algorithms in Python: Heuristic Implementations

Julian Soto

Department of Computer Science, Northwest Nazarene University

COMP4220: Artificial Intelligence

Professor Levandovsky

February 12, 2023

Table of Contents

Abstract

Several pathfinding algorithms of varying time complexities can be implemented to find different paths on a map. The paths obtained by these algorithms differ according to the search method implemented by the search algorithm. This paper evaluates four unique search algorithms: breadth-first search, lowest cost search, iterative deepening search, and an A* search with two different heuristics. The results of this research show that the breadth search algorithm is the easiest to implement and does ultimately find a path; however, its path is unoptimized. The lowest-cost search algorithm finds an optimal path, but, takes fairly long to execute. The iterative deepening search is less space-complex than the breadth-first search and finds a path quicker, however does not execute on larger maps. The A* algorithm provides a good balance of space and time complexity, while finding a relatively optimal path. Depending on the weight used for the A* heuristic it greatly changes the optimality, where a larger weight equates to a larger path cost, but quicker execution. Overall, the results of this research show that the algorithm that should be implemented depends on the project's situation.

Introduction to Pathfinding Problem

Pathfinding is an intriguing problem that makes use of algorithms to find a path from a start goal to an end goal. In this specific problem two maps were provided; they were made up of characters with the height and width of the map on the first line. The characters included R, f, F, h, r, M, and W. Each of these characters had a corresponding movement cost being 1, 2, 4, 5, 7, 10, and uncrossable respectively. The purpose of this paper is to explore the reasons for implementing unique pathfinding algorithms to reach a desired end goal. While the circumstances for this investigation may be low stakes and simple, it is important to note that this problem is highly important in the field of computer science as it is applicable to real life situations one example being the determination of the flow of oil or water in a system where certain soil and rocks may impeded the flow of water to varying extents.

Search Algorithms

Breadth First Search

The breadth first search is one of the most simple search algorithms. In the given situation this type of search is valid when the characters of the map all have the same movement cost. This search will find the shortest path length towards the goal. However, this algorithm is very space complex whre in it's wost case it is $O(b^d)$ in complexity, where b is the length of the path, and d is the branching factor.

Dijkstra's Algorithm (Lowest Cost Search)

Dijkstra's algorithm makes use of a priority queue in order to explore the node with the smallest path cost first, until it finds the goal node. In a situation such as the one this paper explores, Djikstra's algorithm is perfect for finding the lowest cost path. The algorithm is fairly time complex where it's worst case scenario is $\theta(|E|+|V|\log|V|)$. However, the tradeoff is worth it if you ultimately would like to find the shortest path.

The algorithm takes in the starting node and then checks all of the neighboring nodes as well as their movement cost then adds the parent node cost to each of them. Once this is done they ar added to the priority queue, checking the nodes with the smallest path cost. If a shorter path cost to a specfic node is found later on it will be updated accordingly. Once the algorithm is finished executing it will return the path with the lowest path cost.

Iterative Deepening Search

The iterative deepening search is a search that essentially combines a depth first search with a breadth first search in order to be quick and efficient. This search recursively calls a depth first search but sets a depth limit in order to prevent long run times if a goal is not found in a reasonable amount of time. The search will begin at the starting node and continue down a certain depth until it reaches the goal node, or there are no nodes left to evaluate. This search will not necessarily find the shortest path, but, it is quite efficient. However, It is important to note that this search did not work on the large map as the recursion limit needs to be set very high within python in order for it to execute.

A*

The a star algorithm is essentially an implementation of Dijkstra's algorithm with an added heuristic. Different heuristics may be implemented into this algorithm such as Manhattan distance, Euclidean distance, or beams. Depending on the specifications of the problems it may be best to implement either one of the aforementioend heuristics, however, given the specific requirements of this investigation, I found Manhattan distance with the added heuristic of weights to be the most optimal.

A* Search Heuristics

Manhattan Distance

When setting up the comparitor for the A* function that implemented Manhattan distance, I used the following coding segment:

```
            gn1 = self.path_cost
            gn2 = other.path_cost

            hn1 = ((width - self.state[0]) + (height - self.state[1]))
            hn2 = ((width - other.state[0]) + (height - other.state[1]))
            return gn1 + hn1 < gn2 + hn2
```

This code first obtains the self cost of the current node and compares it to another node in order to determine it's place on the priority queue. It then obtains the coordinates of the current node and other node and subtracts them from the width and height of the map respectively. Once obtained, it returns an inequality of the compared heuristics and adds them to the priority queue depending on the result.

Weighted Manhattea Distance

The weighted Manhattan distance does the same thing but adds a heuristic of weight which is multiplied by each value in order to influence which nodes should be evaluated first.

```
            return gn1 + hn1*weight < gn2 + hn2*weight
```

Results

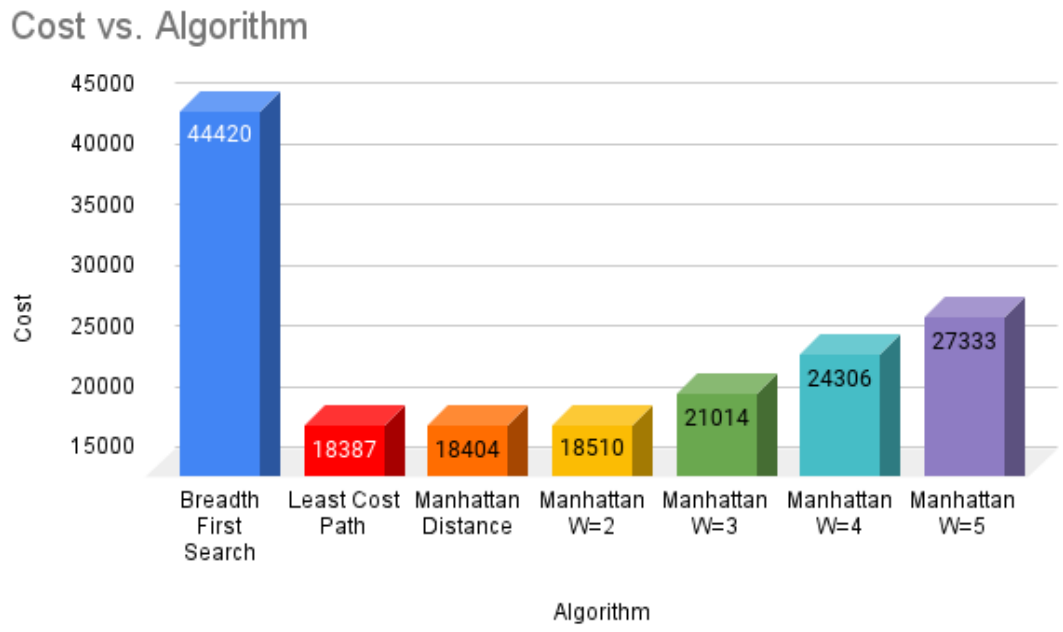

Figure 1:

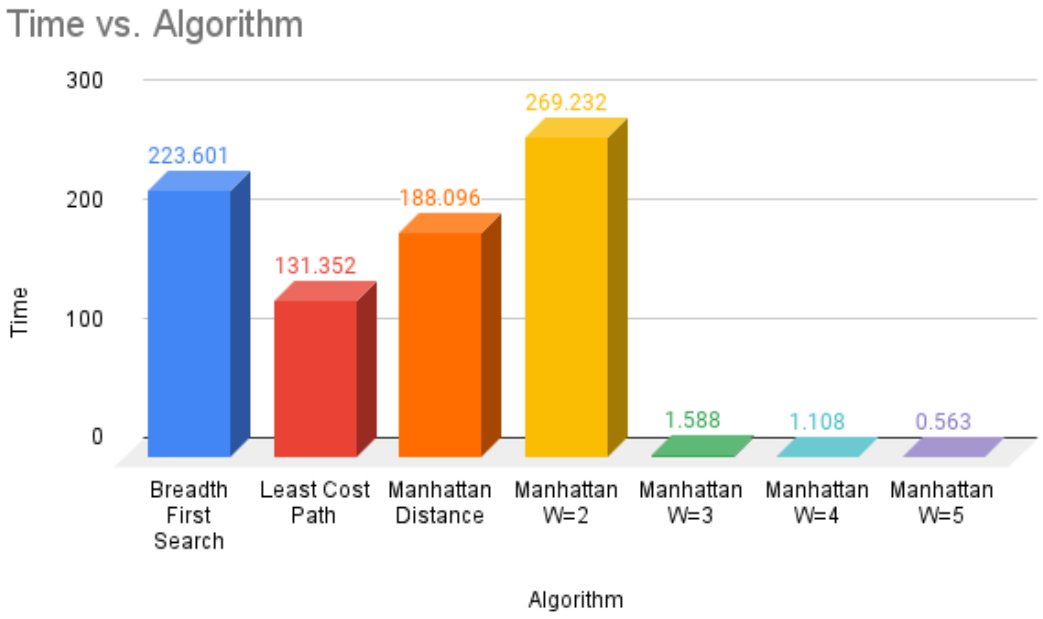

Figure 2:

The results of this research were calculated using a map that was 16777216 characters in size. Tests were performed on maps that were 19200, and 300 characters in size as well, however the results did not provide a good enough picture of the use of each algorithm so they were not included in this paper. I also did not include the path length as this figure is ultimately unimportant in this situation as path cost should be considered over the lenght of the path. As shown in the figures above, breadth first search proved to be one of the least efficient for finding the smallest path cost and was the second slowest in terms of finding a path. Dijkstra's algorithm proved to be quite effective, as it found the smallest path cost, and was in the middle compared to other algorithms in terms of execution time. The unweighted Manhattan distance found a path cost similar to Dijkstra's but took a bit longer to execute. Interestingly enough the weighted algorithm with a weight of two took longest to execute and found a path cost even worse than the breadth first search. However, once larger weights were used we were able to find a balance at a weight of 3, which found a relatively small path cost, as well as being one of the most time efficient implementations.

Conclusion

This investigation was aimed at understanding how a variety of pathfinding algorithms would perform on a weighted map. Through experimentation, it has been shown that the addition and optimization of heuristics makes a significant difference in the optimality of a uniform cost search. It was found that a breadth first search algorithm should be used in situations where you want to find the shortest path on an unweighted graph. Once the variable of movement cost it may be more effective to use dijkstra's algorithm to find the shortest path cost. If it is necessary to find a balance between performance and path cost, using an A star search that implements Manhattan distance and weight might be beneficial. The implementation of the A star algorithm worked best in this situation with a weight of three, however, this optimality of this heuristic will depend on the circumstances of the problem.

This investigation was limited quite limited in that it only took into account a small sample of variables that may be found in a pathfinding problem. If additional variables were to be added to this investigation such as diagonal movement at a higher movement cost or cells with varying degrees of direction, the results of this investigation may have differed. The A star algorithm may not have proved to be the most efficient as other algorithms may have to be implemented with a sense of intuition given the cirumstances.

Much like other problems there are a number of different solutions when it comes to pathfinding. The key to determining which algorithm to use is through observing the situation and determining what aspects should be considered most important. In this situation some aspects evaluated were path cost, execution time, path length, cells evaluated, time complexity, and

space complexity. With the addition of more variables one or two heuristics may prove to be unoptimal, but the choice of which ones to implement depends on the user.