Steven Hyland
RBE 550 – Motion Planning
Homework 2

Dijkstra and A-Star Search Algorithms

A. Dijkstra Search Algorithm

The Dijkstra Algorithm is a search algorithm that finds the shortest path between a two nodes in a weighted graph (a graph with weighted edges). Dijkstra is similar to Breadth-First Search (BFS) in that it explores new nodes along a frontier in a 'wave' fashion. However, Dijkstra chooses the next node to explore based on the frontier node with the **lowest cost**. In this homework, the cost of each node was determined sequentially with edge weights (cost of traveling) = 1.

My own pseudocode explanation is below:

```
Define start, goal
Initialize wavefront[v ∈ G.v] = ∞, visited = start

wavefront[start] = 0

While wavefront not empty OR u != goal:
    u = min[wavefront]
    steps += 1
    for all directions (Right, Down, Left, Up):
        next_node = move(direction)

        if next_node == visited:
            continue

        new_weight = wavefront[u] + 1 ## 1 is cost of moving 1 step
        if new_weight < wavefront[next_node]:
            wavefront[next_node] = new_weight
```

As seen, this algorithm chooses the next node to visit based on the **minimum** cost of each node calculated in the previous step.

Note: My algorithm naturally searches in the order of (*row,column*) due to the way I set up the selection of the minimum next step in the while loop. It may not behave as well if, say, the *end* and the *goal* are swapped in the example graph we are given.

B. A-Star (A*) Search

A* search is similar to Dijkstra's Algorithm in that it can find the shortest path between two nodes of a graph by choosing the next best (minimum cost) of the next node. However, the approach is different, and more optimized. By nature, A* employs much of Dijkstra's Algorithm, and is built upon it. **The key difference** is that instead of blindly choosing the subsequent node to visit based on a pure minimum value, A* also uses a **heuristic.** The heuristic essentially adds an *additional* condition to Dijkstra, by choosing the next best node that is also closer to the *goal*. In essence, the next node to visit is determined based on the cost to get there, **plus** the future-cost of being at that node. For this homework, the heuristic was simply each node's *Manhattan Distance* (primary axes distance) to the goal.

My own pseudocode explanation is below:

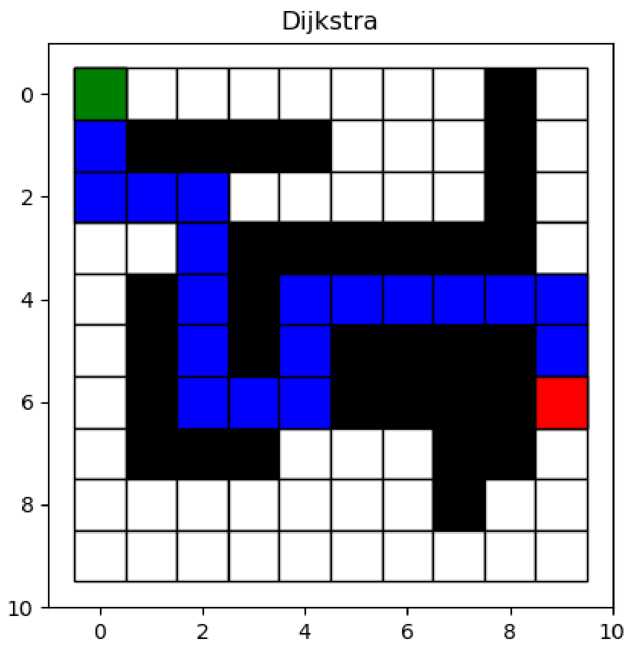
```
Define start, goal
Initialize wavefront[v Є G.v] = ∞, visited = start

Initialize heuristic[v Є G.v] = manhattan_distance
wavefront[start] = 0

While wavefront not empty OR u != goal:
    u = min[wavefront + heuristic]
    steps += 1
    for all directions (Right, Down, Left, Up):
        next_node = move(direction)

        if next_node == visited:
            continue

        new_weight = wavefront[u] + 1 ## 1 is cost of moving 1 step
        if new_weight < wavefront[next_node]:
            wavefront[next_node] = new_weight
```

This algorithm is so similar to Dijkstra that I highlighted the key changes between the two. As seen above, the heuristic is simply each node's manhattan distance to goal. This heuristic is added to the determination of *u* , the next node to visit.

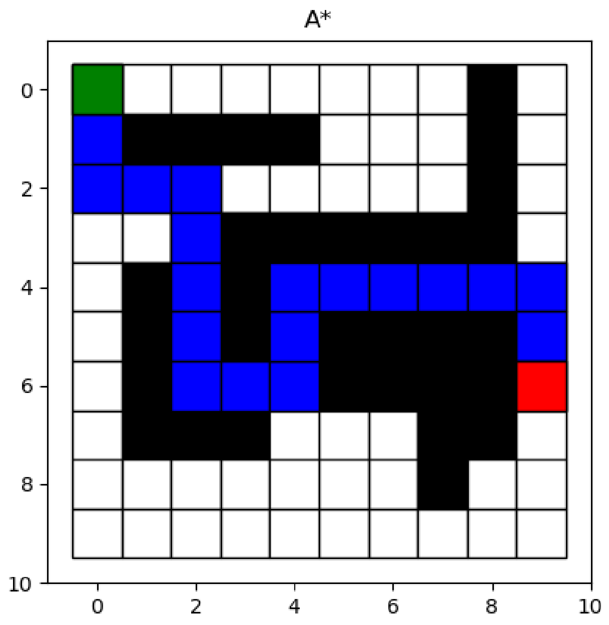
Note: My algorithm isn't the most efficient, since it calculates the heuristic of **each node** before the loop starts. This means it has to run through every single node. One optimization could be to instead only calculate the heuristic of potential candidates for nodes to visit. Regardless, my implementation works appropriately and isn't slow with the two example maps provided.

Furthermore, as with the previous homework, in my python scripts, both algorithms require a dictionary manipulation to correctly determine the path. This is done by storing the *next_node* as the **key** and *u (current vertex)* as the **value.** Afterwards, this dictionary can be turned into an ordered list by working backwards from the goal, moving from value to key, until the start is reached. See code for further documentation.

Results and Discussion:



Dijkstra:
It takes 64 steps to find a path using Dijkstra
Path:
[[0, 0], [1, 0], [2, 0], [2, 1], [2, 2], [3, 2], [4, 2], [5, 2], [6, 2], [6, 3], [6, 4], [5, 4], [4, 4], [4, 5], [4, 6], [4, 7], [4, 8], [4, 9], [5, 9], [6, 9]]



A*:
It takes 44 steps to find a path using DFS
Path:
[[0, 0], [1, 0], [2, 0], [2, 1], [2, 2], [3, 2], [4, 2], [5, 2], [6, 2], [6, 3], [6, 4], [5, 4], [4, 4], [4, 5], [4, 6], [4, 7], [4, 8], [4, 9], [5, 9], [6, 9]]

From this, we can see that both algorithms find the shortest path from *start* to *goal*. However, A* completes in only 44 steps, as opposed to the 64 steps of Dijkstra. This is due to the ‘smart’ selection of next nodes with *future costs* accounted for.

Note: Again, I wanted to mention that both algorithms search row-wise (begins at index 0 of each row) due to the way I initialized the graph/nodes. This works for this assignment and example, however, if the *start* and *goal* are swapped, the algorithm may take a few steps longer to complete. The shortest path, however, will be found. Also, if no path is available, my code will catch it.