Steven Hyland
RBE 550 – Motion Planning
Homework 1

BFS and DFS Search Algorithms

A. Breadth-First Search

As explained in class, breadth first search is a tree traversal or *node search* algorithm. It finds a path between a start and a goal in a network. The BFS in this homework was used to find the shortest path between a start and end space in a gridworld. It functions by searching neighboring nodes (also can be considered the subsequent graph layer) in a first-in first-out manner. Essentially, the algorithm sends out a "wave" from the starting node, in which nodes from *each layer* are explored before continuing to the following, deeper layers.

My own pseudocode explanation is below:

```
Define start, goal
Initialize wavefront, visited = start

While thisVertex != goal:
      thisVertex = wavefront[0]
      steps += 1
      for all directions (Right, Down, Left, Up):
            nextVertex = move(direction)
            if nextVertex == obstacle || nextVertex == wall:
                  do Nothing
            elif nextVertex NOT visited:
                  wavefront.append(nextVertex)
                  visited.append(nextVertex)
```

As seen, this algorithm moves to the next node from the **front** of the list, which ensures each graph level is explored in the order it is seen: top-down.

B. Depth-First Search

A depth first search is similar to BFS in that it can find a path between two nodes of a graph. However, the approach is different. DFS also searches neighbors for possible paths, but instead of completely visiting all the nodes of a particular layer, it explores in a *first-in last-out* manner, relying on a particular order of search (left-right or East-South-West-North). In essence, this means the node that was last added to the "*frontier*" list, is the next node to be explored. Therefore, this algorithm will explore a certain direction as *deep* as it will go – thus the title *Depth-First Search*.

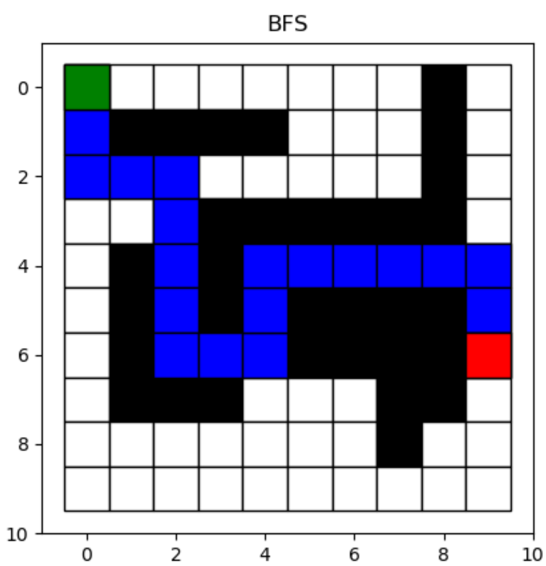
My own pseudocode explanation is below:

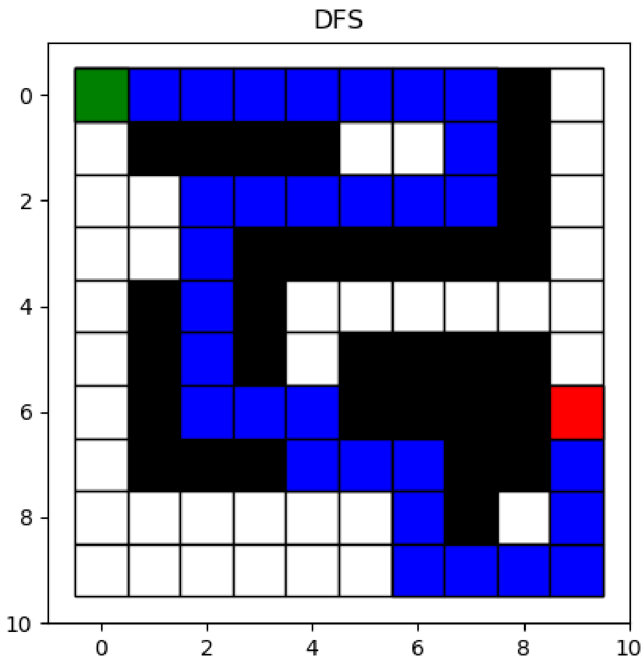
```
Define start, goal
Initialize wavefront, visited = start

While thisVertex != goal:
     thisVertex = wavefront.pop()
     visited.append(thisVertex)

     # NOTE: inverse order is required to allow for FILO
     for all directions (UP, LEFT, DOWN, RIGHT):
          nextVertex = move(direction)
          if nextVertex == obstacle || nextVertex == wall:
               do Nothing
          elif nextVertex NOT visited:
               wavefront.append(nextVertex)
```

Something to note is the difference in direction order for this algorithm. Since we are operating in a First-In, Last-Out configuration, the inverse order allows for the **right** direction to be the next direction we explore, if possible.

Furthermore, in my real python file, both algorithms require a dictionary manipulation to correctly determine the path. This is done by storing the *nextVertex* as the **key** and *thisVertex (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:





BFS:
It takes 64 steps to find a path using BFS
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]]

DFS:
It takes 32 steps to find a path using DFS
Path:
[[0, 0], [0, 1], [0, 2], [0, 3], [0, 4], [0, 5], [0, 6], [0, 7], [1, 7], [2, 7], [2, 6], [2, 5], [2, 4], [2, 3], [2, 2], [3, 2], [4, 2], [5, 2], [6, 2], [6, 3], [6, 4], [7, 4], [7, 5], [7, 6], [8, 6], [9, 6], [9, 7], [9, 8], [9, 9], [8, 9], [7, 9], [6, 9]]

From this, we can see that the BFS was much better at finding a shortest path, with almost half the number of path length. However, this comes at a cost of number of computation steps during calculation of the path, double the number of steps as DFS. This is due to the naiive nature of the DFS. It explores blindly using a *snake-like* or *branch-like* pattern, where it quickly jumps to the last acceptable node it has encountered. Whereas, BFS sends a theoretical “wave” from the start, exploring each neighboring node before moving “down” a level in the graph. This allows this algorithm to find the optimal path, if it exists at the cost of computation time.