Project 1: Search Algorithms

Introduction:

For this project you will need to implement algorithms that can be used by your Malmo agent to find paths through its maze world. The goal of these paths is to reach a destination efficiently. This project will provide experience dealing with the logic behind how the Malmo agent moves to search for a particular location. You need to implement three algorithms for quickly finding the destination in a maze. The basic idea behind all three is the same: they all visit the node that is "next" in a data structure they maintain (e.g. a stack, queue or priority queue), mark it, and then add its unvisited neighbors to the data structure. These algorithms differ in which nodes they explore first and which they leave for later, you will need to complete the following methods.

Files you'll edit:

DFS.py

BFS.py

A_Star.py

Supporting Files:

search.py

smallMaze.lay

mediumMaze.lay

bigMaze.lay

Files to submit:

DFS.py

BFS.py

A_Star.py

Academic Dishonesty:
The University of Virginia's Honor Code: students pledge never to lie, cheat, or steal, and accept that the consequence for breaking this pledge is permanent dismissal from the University.

In general, we expect that you will be using code, examples, and ideas from many different websites and resources for your projects. This is allowed within reason. Wholesale copying of a project is not allowed and will result in an honor violation. In ALL cases, you need to cite all sources at the top of the file where the code or algorithm was used. Failure to properly attribute your sources will result in a 0 for the project at a minimum.

Getting Help:
You are not alone! If you find yourself stuck on something, contact the TAs for help. If you can't make our office hours, let us know and we will schedule more. We want these projects to be rewarding and instructional, not frustrating and demoralizing. But, we don't know when or how to help unless you ask.

Search function input:
The input of your search algorithms will be a maze in the form of a 2D array of tuples, as well as the start and end coordinates of the agent.

Question 1 (5 Points): Depth First Search

It is time to write your first search algorithm to help the Malmo agent navigate mazes! Pseudocode for the search algorithms you'll write can be found in the lecture slides. Remember that a search node must contain not only a state but also the information necessary to reconstruct the path which gets to that state.

Implement the depth-first search (DFS) algorithm in the DFS.py. To make your algorithm complete, write the graph search version of DFS, which avoids expanding any already visited states.

Your code should quickly find a solution for :

python search.py -m smallMaze

python search.py -m mediumMaze

python search.py -m bigMaze

Hint: All of your search functions need to return a list of coordinates that define the agent's path, as well as the number of nodes your algorithm searches through. Keep in mind that the path you return should be composed only of legal nodes (no moving through walls or moving the agent more than one step at a time). The search.py file will check if your path is continuous.

After getting the path, think about the following questions. Is the exploration order what you would have expected? Does the Malmo agent actually go to all the explored squares on his way to the goal? Is this a least cost solution? If not, think about what depth-first search is doing wrong.

Question 2 (5 Points): Breadth First Search

Implement the breadth-first search (BFS) algorithm in the BFS.py. Write a graph search algorithm that avoids expanding any already visited states. Test your code the same way you did for depth-first search.

Does BFS find a least cost solution? If not, check your implementation.

Question 3 (5 Points): A_Star Search

Implement A* graph search in the A_Star.py. A* takes a heuristic function as an argument. Heuristics take two arguments: the movement cost to move from the starting point to this point, and the estimated movement cost to move from this point to the destination point, this value is also called heuristic value.

You can test your A* implementation on the original problem of finding a path through a maze to a fixed position using the Manhattan distance heuristic. You should see that A* finds the optimal solution slightly faster than Breadth First Search

Grading:

We will run your code on a number of different terrains to ensure that your code can solve each maze correctly using three different algorithms. You can also test your code on different terrains through *python search.py -m LayoutFileName* before your submission. You will be awarded 5 points for the correct implementation of each search method (BFS, DFS, A*). The total grade of this project is 15 points.

Important: an implementation that only returns the correct path and does *not* return the number of nodes searched by the algorithm will receive a score of 0 for that question.