Artificial Intelligence Homework 1

Search in Pacman

General Idea

Welcome to the world of pacman. This pacman program is based on the classic Berkeley CS188 course. We will use pacman as our HW1 and HW2. In HW1, you are required to implement a Depth First Search pacman to find a fixed food dot.

Detail Info:

Files you'll edit:

search.py	Where all of your search algorithms will reside. Please finish DFS, BFS, UCS, A Star!

Files you must look at:

util.py	Useful data structures for implementing search algorithms. Please use the data structures here only! Do not import any data structures from other python tools or package.

Files you might want to look at: (if you want to understand how pacman works)

pacman.py	The main file that runs Pacman games. This file describes a Pacman GameState type, which you use in this project.
game.py	The logic behind how the Pacman world works. This file describes several supporting types like AgentState, Agent, Direction, and Grid.

Important note:
Your search functions need to return a list of actions:

['North','West','East','South']

That is, a list of strings. That will lead the agent from the start to the goal. These actions all have to be legal moves (valid directions, no moving through walls, we don't want to see a super pacman).

Important note:
Make sure to use the Stack, Queue and PriorityQueue data structures provided to you in util.py only!

Problem 1: Depth First Search (DFS) 20pts

1. Download pacman.zip from the course website.
2. Please use python 2.7
3. Run a pacman game by typing python pacman.py to terminal. (remember to change directory to pacman folder). You will see two ghosts and one pacman. You can use arrow key to control the pacman. But this fancy game is just for fuun. Not related to our HW.
4. Run a search-based pacman game by typing
   ```
   python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch
   ```
   You will see pacman moving forward to the food dot. The function tinyMazeSearch is defined in search.py. However, there is no search algorithm inside tinyMazeSearch. All the action is simply written by us. That is, this function is only suitable for tinyMaze, not for mediumMaze or bigMaze.

5. Now your work is to finish the depthFirstSearch function inside the search.py. By calling the following three commands:
   ```
   python pacman.py -l tinyMaze -p SearchAgent -a fn=dfs
   python pacman.py -l mediumMaze -p SearchAgent -a fn=dfs
   python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=dfs
   ```
   The pacman will eat the food dot in tinyMaze, mediumMaze and bigMaze.

*** -l means which layout (maze) to use. -p means which pacman agent to run. -a means which algorithm to apply.*

6. If you are pretty much done with BFS, let's try other test cases by typing:
   ```
   python autograder.py -q q1
   ```

Problem 2: Breadth First Search (BFS) 20pts

Step by Step:

1. Implement the breadth-first search (BFS) algorithm in the breadthFirstSearch function in search.py. Again, write a graph search algorithm that avoids expanding any already visited states.

2. By calling the following three commands:
   ```
   python pacman.py -l tinyMaze -p SearchAgent -a fn=bfs
   python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
   python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=bfs
   ```
   The pacman will eat the food dot in tinyMaze, mediumMaze and bigMaze.

3. Compare with DFS, which algorithm find the better path? Is BFS optimal? Does BFS find a least cost solution? Why?

4. *Hint: From DFS to BFS, there is only a subtle change. You should finish BFS very quickly.*

5. If you are pretty much done with BFS, let's try other test cases by typing:
   ```
   python autograder.py -q q2
   ```

Problem 3: Uniform Cost Search (UCS) 30pts

Step by Step:

1. In the section of BFS and DFS, they do not care about the cost. Though BFS will find a fewest-actions path to the goal, we might want to find paths that are "best" in other senses.

2. Implement the uniform cost search (ucs) algorithm in the uniformCostSearch function in search.py. Is avoiding expanding any already visited states necessary?

3. *Hint: The cost function is handled automatically. Use problem.getSuccessors() to get the cost of each steps.*

4. Let's try on mediumMaze first. (the cost of mediumMaze is uniform)
   ```
   python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
   python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
   ```
   What's the difference (Search nodes expanded and Scores) between using bfs and ucs on mediumMaze? Why?

5. Let's vary the cost now.
   ```
   python pacman.py -l mediumDottedMaze -p StayEastSearchAgent
   python pacman.py -l mediumScaryMaze -p StayWestSearchAgent
   ```
   Both StayEastSearchAgent and StayWestSearchAgent will call the uniformCostSearch function in search.py. The only difference is the cost function. The agents and cost functions are written for you, you can take a look in searchAgents.py

6. If you are pretty much done with UCS, let's try other test cases by typing:
   ```
   python autograder.py -q q3
   ```

Problem 4: A star (A*) 30pts

Step by Step:

1. Implement A* graph search in the empty function aStarSearch in search.py.
2. A* takes a heuristic function as an argument. Heuristics take two arguments: a state in the search problem (the main argument), and the problem itself (for reference information). The nullHeuristic heuristic function in search.py is a trivial example.
3. Test your A* implementation on bigMaze using the Manhattan distance heuristic (implemented already as manhattanHeuristic in searchAgents.py).
   ```
   python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic
   ```
   Hint: If Pacman moves too slowly for you, try the option --frameTime 0.
4. Compare with UCS, you should see that A* finds the optimal solution slightly faster than uniform cost search. (Less nodes expanded than UCS). That's the power of heuristic function!
5. If you are pretty much done with A *, let's try other test cases by typing:
   ```
   python autograder.py -q q4
   ```

To Check if you Got All the Points:

```
python autograder.py
```

Submission Deadline and Method:

Deadline:
Look in the slides & submit to the course web.

Submission Method
Upload a zip file to the course website.

```
hw1_studentID.zip
   hw1_studentID/
        - search.py
```

Delay policy (may change):
25 points off each day