Util.py	Useful data structures for implementing search algorithms.

Introduction:
After downloading the code (search.zip), unzipping it, and changing to the directory, you should be able to play a game of Pacman by typing the following at the command line:

python pacman.py

Pacman lives in a shiny blue world of twisting corridors and tasty round treats. Navigating this world efficiently will be Pacman's first step in mastering his domain. The simplest agent in searchAgents.py is called the GoWestAgent, which always goes West (a trivial reflex agent). This agent can occasionally win:

python pacman.py --layout testMaze --pacman GoWestAgent

But, things get ugly for this agent when turning is required:

python pacman.py --layout tinyMaze --pacman GoWestAgent

If Pacman gets stuck, you can exit the game by typing CTRL+C into your terminal. Soon, your agent will solve not only tinyMaze, but any maze you want. Note that *pacman.py* supports a number of options that can each be expressed in a long way (e.g., --layout) or a short way (e.g., -l). You can see the list of all options and their default values via:

python pacman.py -h

Also, all of the commands that appear in this project also appear in commands.txt, for easy copying and pasting. The commands for **autograder** are also in commands.txt.

Questions:

1. Let's start with finding a Fixed Food Dot using Depth First Search:

   In searchAgents.py, you'll find a fully implemented SearchAgent, which plans out a path through Pacman's world and then executes that path step-by-step. The search algorithms for formulating a plan in search.py are not implemented – that's your job. First, test that the SearchAgent is working correctly by running:

   python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch

   The command above tells the SearchAgent to use tinyMazeSearch as its search algorithm, which is implemented in *search.py*. Pacman should navigate the maze successfully.

   Now it's time to write full-fledged generic search functions to help Pacman plan routes! Pseudocode for the search algorithms you'll write can be found in the tutorial slides. You can use them.

   Important note 1: Remember that a search node can contain not only a state (the position of the Pacman) but also the information necessary to reconstruct the path (plan) which gets to that state.

Important note 3: Make sure to use the Stack, Queue and PriorityQueue data structures provided to you in util.py! These data structure implementations have particular properties which are required for compatibility with the autograder.

Hint: Algorithms are very similar. Algorithms for DFS, BFS, UCS and A* differ only in the details of how the fringe is managed. So, please concentrate on getting iterative DFS right and the rest should be relatively straightforward. See the comments in the *search.py* for more hints.

What you need to do in this question is to implement the depth-first search (DFS) algorithm in the *depthFirstSearch* **function in** *search.py.* To make your algorithm complete, write the graph search version of DFS, which avoids expanding any already visited states.

Specifically, you need to provide **two depth-first search algorithms** (recursive and iterative) and **an iterative deepening search algorithm(**IDDFS**)**. You can write them in the same function (*depthFirstSearch*) with two of them commented out. For the recursive DFS and IDDFS, you can create a new function inside depthFirstSearch. Your code should quickly find a solution for:

python pacman.py -l tinyMaze -p SearchAgent

python pacman.py -l mediumMaze -p SearchAgent

python pacman.py -l bigMaze -z .5 -p SearchAgent

The Pacman board will show an overlay of the states explored, and the order in which they were explored (brighter red means earlier exploration). **Your two depth-first search algorithms (recursive and iterative) should also pass the autograder.**

Hint: The path found by your iterative DFS algorithm for mediumMaze should have a length of 130 (You will get 246 if you use recursive DFS. 76 or 70 if you use IDDFS).

(55%)

Solution:

Iterative function: Pass all commands (10%) Pass autograder(10%)
Recursive function: Pass all commands (10%) Pass autograder(10%)
IDDFS: Pass all commands (15%)

2. **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. Your code should **pass the autograder** and **quickly find a solution for**:

python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs

python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5

Does BFS find a least-cost solution? If not, check your implementation.
Hint: If Pacman moves too slowly for you, try the option --frameTime 0.
Note: If you've written your search code generically, your code should work equally well for the eight-puzzle search problem without any changes.

python eightpuzzle.py

(15%)

Solution:

BFS: Pass all commands (5%) Pass autograder(10%)

3. While BFS will find a fewest-actions path to the goal, we might want to find paths that are "best" in other senses. Consider mediumDottedMaze and mediumScaryMaze.

   By changing the cost function, we can encourage Pacman to find different paths. For example, we can charge more for dangerous steps in ghost-ridden areas or less for steps in food-rich areas, and a rational Pacman agent should adjust its behavior in response.

   Implement the uniform-cost graph search algorithm in the uniformCostSearch function in *search.py*. We encourage you to look through *util.py* for some data structures that may be useful in your implementation. You should now observe successful behavior in all three of the following layouts, where the agents below are all UCS agents that differ only in the cost function they use (the agents and cost functions are written for you). **Your code should pass the autograder** and **quickly find a solution for**:

   python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs

   python pacman.py -l mediumDottedMaze -p StayEastSearchAgent

   python pacman.py -l mediumScaryMaze -p StayWestSearchAgent

   Note: You should get very low and very high path costs for the StayEastSearchAgent and StayWestSearchAgent respectively, due to their exponential cost functions (see searchAgents.py for details)

   (15%)

   Solution:

   UCS: Pass all commands (5%) Pass autograder(10%)

4. **Implement A* graph search in the empty function aStarSearch in** *search.py*. 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.

   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 (implemented already as manhattanHeuristic in searchAgents.py). **Your code should pass the autograder and quickly find a solution for:**

   python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic

   You should see that A* finds the optimal solution slightly faster than uniform cost search (about 549 vs. 620 search nodes expanded in our implementation, but ties in priority may make your numbers differ slightly).

   (15%)

   Solution:

   A*: Pass all commands (5%) Pass autograder(10%)