A1

In this assignment, your Pac-Man agent will find paths through his maze world, both to reach a particular location and to collect food efficiently. You will build general search algorithms and apply them to Pac-Man scenarios.

The code for this project consists of several Python files, some of which you will need to read and understand in order to complete the assignment, and some of which you can ignore. You can download all the code as a zip here *(https://coursys.sfu.ca/2019fa-cmpt-310-d1/pages/A1.zip/view)* .

Files you'll edit:

For Q1.1, Q1.2, and Q1.3:

```
search.py          Where all of your search algorithms will reside.
```

For Q2.1 and Q2.2:

```
searchAgents.py  Where all of your search-based agents will reside.
```

Files you might want to look at:

```
pacman.py       The main file that runs Pac-Man games.
 This file describes a Pac-Man GameState type, which you use in this project.
game.py    The logic behind how the Pac-Man world works.
 This file describes several supporting types like AgentState, Agent, Direction, and Grid.
util.py     Useful data structures for implementing search algorithms.
```

Supporting files you can ignore:

```
graphicsDisplay.py      Graphics for Pac-Man
graphicsUtils.py        Support for Pac-Man graphics
textDisplay.py      ASCII graphics for Pac-Man
ghostAgents.py     Agents to control ghosts
keyboardAgents.py       Keyboard interfaces to control Pac-Man
layout.py        Code for reading layout files and storing their contents
```

What to submit: You will fill in portions of search.py and searchAgents.py during the assignment. You should submit these two files (only) to CourSYS *(https://coursys.sfu.ca/2019fa-cmpt-310-d1/)* .

Evaluation: Your code will be *autograded* for technical correctness. Please do not change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder. However, the correctness of your implementation -- not the autograder's output -- will be the final judge of your score. If necessary, we will review and grade assignments individually to ensure that you receive due credit for your work.

We have included an autograder for you to grade your answers on your machine. This can be run with the command:

```
python autograder.py
```

Academic Dishonesty: We will be checking your code against other submissions in the class for logical redundancy. We trust you all to submit your own work only; please don't let us down. If you do, we will pursue the strongest consequences available to us.

Getting Help: You are not alone! If you find yourself stuck on something, contact the TAs for help. The TAs will have additional office hours before the assignment is due. Alternatively, you can post your questions to Piazza.

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.

One more piece of advice: if you don't know what a variable does or what kind of values it takes, print it out.

Part 0 Welcome to Pac-Man

After downloading the code (A1.zip *(https://coursys.sfu.ca/2019fa-cmpt-310-d1/pages/A1.zip/view)*), unzipping it and changing the current working directory to the A1 directory, you should be able to play a game of Pac-Man by typing the following at the command line:

```
python pacman.py
```

Note: Make sure you are running a recent version of Python (3.6 or later). Depend on your system configuration, You may need to use python3 instead of python to run this program.

Pac-Man lives in a shiny blue world of twisting corridors and tasty round treats. Navigating this world efficiently will be Pac-Man'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
```

Part 1 Finding a Fixed Food Dot using Search Algorithms

In searchAgents.py, you'll find a fully implemented *SearchAgent*, which plans out a path through Pac-Man's world and then executes that path step-by-step. The search algorithms for formulating a plan 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. Pac-Man should navigate the maze successfully.

Now it's time to write full-fledged generic search functions to help Pac-Man plan routes! Pseudocode for the search algorithms you'll write can be found in the lecture slides and textbook. Remember that a search node must contain not only a state but also the information necessary to reconstruct the path (plan) which gets to that state.

Important note: All of your search functions need to return a **list** of actions 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).

Hint: Make sure to check out the *Stack*, *Queue* and *PriorityQueue* types provided to you in util.py!

Question 1.1 (2 points)

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 (R&N 3ed Section 3.3, Figure 3.7). 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 Pac-Man board will show an overlay of the states explored, and the order in which they were explored (brighter red means earlier exploration). Is the exploration order what you would have expected? Does Pac-Man actually go to all the explored squares on his way to the goal?

Hint: If you use a Stack as your data structure, the solution found by your DFS algorithm for mediumMazeshould have a length of 130 (provided you push successors onto the fringe in the order provided by getSuccessors; you might get 244 if you push them in the reverse order). Is this a least cost solution? If not, think about what depth-first search is doing wrong.

Question 1.2 (2 points)

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. Test your code the same way you did for depth-first search.

```
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 Pac-Man 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 (R&N 3ed Section 3.2, Figure 3.4) without any changes.

```
python eightpuzzle.py
```

Question 1.3 (2 points)

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).

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

Part 2 Finding All the Corners

The real power of A* will only be apparent with a more challenging search problem. Now, it's time to formulate a new problem and design a heuristic for it.

In corner mazes, there are four dots, one in each corner. Our new search problem is to find the shortest path through the maze that touches all four corners (whether the maze actually has food there or not). Note that for some mazes like *tinyCorners*, the shortest path does not always go to the closest food first!

Hint: the shortest path through *tinyCorners* takes 28 steps.

Question 2.1 (3 points)

Implement the *CornersProblem* search problem in searchAgents.py. That is, complete the missing part in the class *CornersProblem* to correctly represent the problem. You will need to choose a state representation that encodes all the information necessary to detect whether all four corners have been reached. Now, your search agent should solve:

```
python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
```

To receive full credit, you need to define an abstract state representation that does not encode irrelevant information (like the position of ghosts, where extra food is, etc.). In particular, do not use a Pac-Man GameState as a search state. Your code will be very, very slow if you do (and also wrong).

Hint: The only parts of the game state you need to reference in your implementation are the starting Pac-Man position and the location of the four corners.

Our implementation of *breadthFirstSearch* expands just under 2000 search nodes on *mediumCorners*. However, heuristics (used with A* search) can reduce the amount of searching required.

Question 2.2 (3 points)

Implement a heuristic for the CornersProblem in *cornersHeuristic*.

Grading: inadmissible heuristics will get no credit. 1 point for any admissible heuristic. 1 point for expanding fewer than 1600 nodes. 1 point for expanding fewer than 1200 nodes. Expand fewer than 800, and you're doing great!

```
python pacman.py -l mediumCorners -p SearchAgent -a fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeur
```

Admissibility vs. Consistency: Remember, heuristics are just functions that take search states and return numbers that estimate the cost to the nearest goal. More effective heuristics will return values closer to the actual goal costs. To be admissible, the heuristic values must be lower bounds on the actual shortest path cost to the nearest goal (and non-negative). To be consistent, it must additionally hold that if an action has cost c, then taking that action can only cause a drop in heuristic of at most c.

Remember that admissibility isn't enough to guarantee correctness in graph search -- you need the stronger condition of consistency. However, admissible heuristics are usually also consistent, especially if they are derived from problem relaxations. Therefore it is usually easiest to start out by brainstorming admissible heuristics. Once you have an admissible heuristic that works well, you can check whether it is indeed consistent, too. The only way to guarantee consistency is with a proof. However, inconsistency can often be detected by verifying that for each node you expand, its successor nodes are equal or higher in f-value. Moreover, if BFS and A* ever return paths of different lengths, your heuristic is inconsistent. This stuff is tricky!

Complexity: heuristic must be sub-exponential in the size of the maze. Most reasonable heuristics are O(1) in the size of the maze. For example, a constant number of Manhattan distance calculation is O(1). Calling BFS to find the true cost to goal is exponential.

You want a heuristic which reduces total compute time, though for this assignment the autograder will only check node counts (aside from enforcing a reasonable time limit).

Part 3 Feedback

Give one short piece of feedback about the course so far. What have you found most interesting? Is there a topic that you had trouble understanding? Are there any changes that could improve the value of the course to you?

How many hours did you spend on this assignment?

Please provide your answers in search.py.

This assignment is adapted from the Pacman AI projects developed at UC Berkeley.
http://ai.berkeley.edu/project_overview.html *(http://ai.berkeley.edu/project_overview.html)*

Updated Mon Sept. 16 2019, 13:04 by liuhengl.