Search in Pac-Man

All those colored walls,
Mazes give Pac-Man the blues,
So teach him to search.

Project designed by: John DeNero & Dan Klein and modified by Luke Zettlemoyer.

Due: Jan 30, 11:59pm
Introduction

In this project, 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 and supporting files (including this description) as a zip archive.

Files you'll edit:

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

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) along with a README file containing any additional comments that you may have.

Where to submit: Submit the two (or three if you have a README) files to `/afs/cs.unc.edu/project/courses/comp590-133-s14/students/<your-login>/HW1` directory by logging into classroom/snapper nodes. Where `<your-login>` should be replaced by your CS login.

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.

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 course staff 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. One more piece of advice: if you don't know what a variable does or what kind of values it takes, print it out. Also, we encourage you to work in a team of size 2.

Welcome to Pac-Man

After downloading the code (search.zip), unzipping it and changing to the search 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 specific version of Python (2.7 or earlier). If you get error messages regarding python-tk, use your package manager to install python-tk, or see this page for more detailed instructions.

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
```

Also, all of the commands that appear in this project also appear in *commands.txt*, for easy copying and pasting. In UNIX/Mac OS X, you can even run all these commands in order with bash *commands.txt*.

---

**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. As you work through the following questions, you might need to refer to this glossary of objects in the code. First, test that the SearchAgent is working correctly by running:

```python
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*: Each algorithm is very similar. Algorithms for DFS, BFS, UCS, and A* differ only in the details of how the fringe is managed. So, concentrate on getting DFS right and the rest should be relatively straightforward. Indeed, one possible implementation requires only a single generic search method which is configured with an algorithm-specific queuing strategy. (Your implementation need *not* be of this form to receive full credit).

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

**Question 1 (3 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:
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 mediumMaze should 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 2 (2 point)** 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
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
python eightpuzzle.py
```

**Varying the Cost Function**

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 Pac-Man 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 Pac-Man agent should adjust its behavior in response.

**Question 3 (3 points)** 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):

```python
python pacman.py -l tinyMaze -p SearchAgent
python pacman.py -l mediumMaze -p SearchAgent
python pacman.py -l bigMaze -z .5 -p SearchAgent
```
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).

A* search

Question 4 (4 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 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

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). What happens on openMaze for the various search strategies?

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 5 (3 points) Implement the CornersProblem search problem in searchAgents.py. 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
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. Also, look at how *FoodSearchProblem* class (in *searchAgent.py*) is implemented.

Our implementation of breadthFirstSearch expands just under 3000 search nodes on mediumCorners. However, heuristics (used with A* search) can reduce the amount of searching required.

**Question 6 (5 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!

We will time your agent using the no graphics option -q, and it must complete in under 30 seconds on our grading machines. Please describe what your agent is doing in a comment!

**Mini Contest (2 points extra credit)** Implement an ApproximateSearchAgent in *searchAgents.py* that finds a short path through the bigSearch layout. The three teams that find the shortest path using no more than 30 seconds of computation will receive 2 extra credit points and an in-class demonstration of their brilliant Pacman agents.

We reserve the right to give additional extra credit to creative solutions, even if they don't work that well. Don't hard-code the path, of course.

**Object Glossary**

Here's a glossary of the key objects in the code base related to search problems, for your reference:
**SearchProblem (search.py)**
A *SearchProblem* is an abstract object that represents the state space, successor function, costs, and goal state of a problem. You will interact with any SearchProblem only through the methods defined at the top of *search.py*.

**PositionSearchProblem (searchAgents.py)**
A specific type of SearchProblem that you will be working with --- it corresponds to searching for a single pellet in a maze.

**CornersProblem (searchAgents.py)**
A specific type of SearchProblem that you will define --- it corresponds to searching for a path through all four corners of a maze.

**FoodSearchProblem (searchAgents.py)**
A specific type of SearchProblem that you will be working with --- it corresponds to searching for a way to eat all the pellets in a maze.

**Search Function**
A search function is a function which takes an instance of SearchProblem as a parameter, runs some algorithm, and returns a sequence of actions that lead to a goal. Example of search functions are depthFirstSearch and breadthFirstSearch, which you have to write. You are provided tinyMazeSearch which is a very bad search function that only works correctly on tinyMaze.

**SearchAgent**
*SearchAgent* is a class which implements an Agent (an object that interacts with the world) and does its planning through a search function. The SearchAgent first uses the search function provided to make a plan of actions to take to reach the goal state, and then executes the actions one at a time.