CPSC 470/570 PS1: Pacman

Due: 2019 Feb 1st, 11:59 pm

Some reminders:

- **Grading contact:** Ngan Vu (ngan.vu@yale.edu) is the point of contact for initial questions about grading for this problem set.
- **Late assignments** are not accepted without a Dean's excuse.
- **Collaboration policy:** You are encouraged to discuss assignments with the course staff and with other students. However, you are required to implement and write any assignment on your own. This includes both pencil-and-paper and coding exercises. You are not permitted to copy, in whole or in part, any written assignment or program as part of this course. You are not to take code from any online repository or web source. You will not allow your own work to be copied. Homework assignments are your individual responsibility, and plagiarism will not be tolerated.

This homework is adapted from Dan Klein and John DeNero's Intro to AI class at UC Berkeley .

Introduction

In this project, your Pacman 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 Pacman 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 from canvas.

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, but should not edit:

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.

util.py: Useful data structures for implementing search algorithms.

Supporting files you can ignore:

graphicsDisplay.py: Graphics for Pacman

graphicsUtils.py: Support for Pacman graphics

textDisplay.py: ASCII graphics for Pacman

ghostAgents.py: Agents to control ghosts

keyboardAgents.py: Keyboard interfaces to control Pacman

layout.py: Code for reading layout files and storing their contents

autograder.py: Project autograder

testParser.py: Parses autograder test and solution files

testClasses.py: General autograding test classes

test_cases/: Directory containing the test cases for each question

searchTestClasses.py: Project 1 specific autograding test classes

Files to Edit: You will fill in portions of search.py and searchAgents.py during the assignment. You *may not* need to fill in all the "*** YOUR CODE HERE ***". Please ***do not*** change the other files in this distribution or submit any of our original files other than these files.

Welcome to Pacman

After copying the code to your directory, you should be able to play a game of Pacman by typing the following at the command line:

```
python3 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:

```
python3 pacman.py --layout testMaze --pacman GoWestAgent
```

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

```
python3 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:

```
python3 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.

Question 1 (3 points): 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 are not implemented -- that's your job.

First, test that the SearchAgent is working correctly by running:

```
python3 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! 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).

Important note: 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: 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).

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.

Your code should quickly find a solution for:

```
python3 pacman.py -l tinyMaze -p SearchAgent
```

```
python3 pacman.py -l mediumMaze -p SearchAgent
```

```
python3 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). Is the exploration order what you would have expected? Does Pacman 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 246 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. You don't need to submit your answer to this question though.

Question 2 (3 points): Breadth First Search

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.

```
python3 pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
```

```
python3 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.

```
python3 eightpuzzle.py
```

Question 3 (3 points): 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 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):

```
python3 pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
```

```
python3 pacman.py -l mediumDottedMaze -p StayEastSearchAgent
```

```
python3 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).

Question 4 (3 points): A* search

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).

```
python3 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).

Question 5 (4 points): Eating All The Dots

Now we'll solve a hard search problem: eating all the Pacman food in as few steps as possible. For this, we'll need a new search problem definition which formalizes the food-clearing problem: FoodSearchProblem in searchAgents.py (implemented for you). A solution is defined to be a path that collects all of the food in the Pacman world. For the present project, solutions do not take into account any ghosts or power pellets; solutions only depend on the placement of walls, regular food and Pacman. (Of course ghosts can ruin the execution of a solution! We'll get to that in the next project.) If you have written your general search methods correctly, A* with a null heuristic (equivalent to uniform-cost search) should quickly find an optimal solution to testSearch with no code change on your part (total cost of 7).

```
python3 pacman.py -l testSearch -p AStarFoodSearchAgent
```

Note: AStarFoodSearchAgent is a shortcut for -p SearchAgent -a fn=astar,prob=FoodSearchProblem,heuristic=foodHeuristic.

You should find that UCS starts to slow down even for the seemingly simple tinySearch. As a reference, our implementation takes 2.5 seconds to find a path of length 27 after expanding 5057 search nodes.

Note: **Make sure to complete Question 4 before working on Question 5**, because Question 5 builds upon your answer for Question 4.

Fill in **foodHeuristic** in **searchAgents.py** with a consistent heuristic for the **FoodSearchProblem**. Try your agent on the trickySearch board:

```
python3 pacman.py -l trickySearch -p AStarFoodSearchAgent
```

Our UCS agent finds the optimal solution in about 13 seconds, exploring over 16,000 nodes. So don't worry if your code may take some time to run. But it should not take several minutes.

Any non-trivial non-negative consistent heuristic will receive 1 point. Make sure that your heuristic returns 0 at every goal state and never returns a negative value. Depending on how few nodes your heuristic expands, you'll get additional points:

Number of nodes expanded	Grade
More than 15000	1/4
At most 15000	2/4
At most 12000	3/4
At most 9000	4/4
At most 7000	5/4 (extra credit)

Remember: If your heuristic is inconsistent, you will receive no credit, so be careful! Can you solve mediumSearch in a short time? If so, we're either very, very impressed, or your heuristic is inconsistent.

(For CPSC570 only) Question 6 (3 points): Suboptimal Search

This problem is required only for students enrolled in the graduate course (CPSC 570). Those in the undergraduate course (CPSC 470) are welcome to try this out, but will not receive any credit for their work.

Sometimes, even with A* and a good heuristic, finding the optimal path through all the dots is hard. In these cases, we'd still like to find a reasonably good path, quickly. In this section, you'll write an agent that always greedily eats the closest dot. ClosestDotSearchAgent is implemented for you in searchAgents.py, but it's missing a key function that finds a path to the closest dot.

Implement **the function findPathToClosestDot** in **searchAgents.py**. Our agent solves this maze (suboptimally!) in under a second with a path cost of 350:

```
python3 pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5
```

Hint: The quickest way to complete findPathToClosestDot is to fill in the AnyFoodSearchProblem, which is missing its goal test. Then, solve that problem with an appropriate search function. The solution should be very short!

Your ClosestDotSearchAgent won't always find the shortest possible path through the maze. Make sure you understand why and try to come up with a small example where repeatedly going to the closest dot does not result in finding the shortest path for eating all the dots. You don't need to submit the answer though.

Submission

Please upload the following to canvas:

- search.py
- searchAgents.py