Project 1 – Path-based Location Search (Chapter 3)

Due: See online for exact due date (11:55PM)

Details

Online Source: Many parts of this project are taken directly from UC Berkeley; this project has been trimmed and edited to meet our specific needs, but you can look at the online prompt for more background info and details about how to use Berkeley's auto-grader (which we will not use) to help you ensure you are doing the problems correctly.

Language Used: Python 2.7.X (NOTE: Python 3.0 will NOT work)

Development Environment Tutorial: Please watch the video for startup tutorial for this project.

Collaboration: You may work with 1 partner, or alone if you choose. Plagiarism and cheating will be detected algorithmically against your classmates' code and online sources. Furthermore, the instructor is extremely aware of your abilities and is very familiar with various solutions and will give you an automatic 0 on any projects where external code has been used.

Introduction

In this project, your Pacman agent will find paths through his maze world to reach a particular location. 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 on the Sakai assignment page for Project 1.

Files

Files you'll edit:

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

Files you may want to look at, but won't need to edit:

searchAgents.py	Where all of your search-based agents will reside…will edit in Project 2.
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 (Berkeley 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)

Welcome to Pacman

After downloading the code, 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 description appear in the zip file in the *commands.txt* file, for easy copying and pasting. In UNIX/Mac OS X, you can even run all these commands in order with bash *commands.txt*. You should also use the BashPacmanProject1Commands.txt file, found on the Sakai assignment page, which separates all the test cases you'll do per problem.

Problems

Question 1 (20 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:

```
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 lecture slides. 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 this Pacman project.

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:

```
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).

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).

Question 2 (20 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:

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

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
```

Question 3 (20 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):

```
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).

Question 4 (20 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`):

```
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).

Report (20 points)

You must turn in a **concise report** with the following:

a.) File should be named "P1_Lastname1_Lastname2_Report.pdf

b.) Report must have Title of project, course, your names, instructor, date, etc.

c.) For Questions 1-4, take a screenshot of your agent in progress, showing the red cells (so I can see the order in which your agent expanded nodes). For each of these questions, include the image of your agent in progress, the execution time (seconds), number of nodes expanded and Pacman score….for tinyMaze, mediumMaze AND bigMaze.

d.) A **clean** summary chart (or charts) for questions 1-4 (i.e., for DFS, BFS, UCS and A* w/ manhattan) for each of the 3 main mazes (tinyMaze, mediumMaz, bigMaze) showing:

- The execution time (seconds)
- Time complexity (# nodes expanded)
- Pacman score

e.) An explanation of your chart from "d.)". Basically, with "d.)" and "e.)", I want you to compare and contrast your results and show me what you learned about the different algorithms in practice. Please do not provide obvious observations without any intuition as to why something happened (i.e., "You can see that BFS expands more nodes than DFS, while BFS uses less nodes than DFS. I also noticed that A* is the fastest" is **very bad** and will get you 0 points.). **You must accompany observations with explanations as to why you think the results turned out the way they did.**

NOTE: Even if you can't get all the test cases to work, please report as much as you do get working. If you can't get a test case to work, you will not necessarily be penalized in the report, assuming you were able to make solid conclusions from the findings you do have (essentially, I will give partial credit).

Grading & Test Cases

You should use the *BashPacmanProject1Commands.txt* file, found on the Sakai assignment page, which provides a single file detailing all of the test cases I will be using to grade your algorithms.

The points breakdown for this project will be as follows:

<u>100 Total Points with breakdown as follows:</u>
20 pts for Question 1 (DFS)
20 pts for Question 2 (BFS)
20 pts for Question 3 (UCS)
20 pts for Question 4 (A*)
20 pts for Report (see details above)

I encourage you to use Berkeley's auto-grader to check correctness of your algorithms, but you should **focus primarily on coding solutions that complete the specified test cases in each question description** because these are where your grades will come from (not the auto-grader).

NOTE: You must write code that solves each problem using the requested algorithm/method (e.g., Question 1 MUST solve the problem using DFS and no other method). DO NOT hardcode solutions; the instructor will thoroughly examine your code to ensure you coded things up properly.

Submission Instructions

1.) <u>Templates:</u> Use the project template files for this lab found in the Sakai assignment page.
   a. Use the specified **search.py** and **<u>DO NOT change the file name</u>** (doing so will cause your code to work incorrectly)
      i. Make sure to include a header at the top of *search.py* with your name(s)
   b. Include a PDF report entitled *CS430_P1_Lastname1_Lastname2.pdf*
2.) <u>Sakai Submission</u>: **Make sure both names are on all files you turn in.**
   a. Upload your edited version of *search.py,* as well as your *CS430_P1_Lastname1_Lastname2.pdf* Sakai for this project (do NOT place them in a .zip file)
      i. Check to make sure all the files in the .zip file contain your latest work before submitting
   b. Only 1 partner should submit the **.zip** file
      i. The final grade will be returned to both students via the Sakai assignment section