Homework 1: Search

Instructions

1. For this assignment you may form teams of up to two people.
2. This class is not coding-heavy but you must be familiar with Python and have it installed on your machine.
    a. If you are not, we recommend you to partner up with someone who is good in Python.
    b. If you cannot find such partner, obtain help from the TA
3. Each of the partners must submit the homework. We understand that they are identical, it's just the way Blackboard software works.
4. The final submission must be structured like this:
    a. FirstName_LastName_hw#.pdf - report, named after you - doing so helps us if we download all the submissions to a directory. Example: Dainis_Boumber_hw1.pdf
    b. FirstName_LastName_resources.zip - code goes here. Do not use other formats such as 7z or tar.xz for compatibility).
    Example: Dainis_Boumber_resources.zip
5. Your homework report must be formatted in a style not too different from a standard scientific conference or journal. The submission must be in single column format.
6. Use figures and tables when needed to explain your results.
7. Do **NOT** use screenshots where a figure is appropriate (which is essentially always). **DO** use screenshots when we ask you to (for example, as proof that certain code runs well).
8. Do **NOT** not just give a number or say yes or no. Show work. Explain why you think so.
9. Do **NOT** answer theory questions with short sentences. Provide enough detail.

Homework-specific requirements

1. For this particular homework you need python 2 (not python 3)
2. search.zip - skeleton code you will use for the assignment

Files to Edit and Submit: You will fill in portions of search.py and searchAgents.py during the assignment. You should submit these files with your code and comments. Please *do not* change the other files in this distribution or submit any of our original files other than these files.

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 judgements -- 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.

Autograder we have provided a local autograder and a set of test cases for you evaluate your code. The local autograder is a file called autograder.py. To run the autograder, run the command:

python autograder.py

You can also select individual questions for the autograder to run. For example, if you want to run question 2 on a project, you would run the command:

python autograder.py -q q2

The test cases are within the test_cases directory. For each test case, we provide the test suite along with the solution of the test case.

Setup

Download the code (search.zip), unzip it, and change to the search directory. You should be able to play a game of Pacman by typing the following at the command line:

python pacman.py

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.

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. Look for ## YOUR CODE GOES HERE ## comments. This homework may look intimidating, but in reality there is very little code for you to write. If lost, ask TA for help! If you don’t ask, we can’t know you need help.

Part I: Pacman (100 points, + up to 10 bonus points)

Question 1: Depth-First Search (DFS) (15 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.

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). Provide us with a screenshot of that overlay.

Question 2: Breadth First Search (15 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 Pacman moves too slowly for you, try the option --frameTime 0.

The Pacman board will show an overlay of the states explored, and the order in which they were explored (brighter red means earlier exploration). Provide us with a screenshot of that overlay.

Question 3: Uniform Cost Search (20 points)

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

The Pacman board will show an overlay of the states explored, and the order in which they were explored (brighter red means earlier exploration). Provide us with a screenshot of that overlay.

Question 4: A* search (30 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

The Pacman board will show an overlay of the states explored, and the order in which they were explored (brighter red means earlier exploration). Provide us with a screenshot of that overlay. **(15 points)**

Answer the following:

How many nodes did A* expand when doing bigMaze?**(5 points)** How does it compare to Uniform Cost Search in that regard (you need to run your UCS on wth -l bigMaze to answer this)?**(5 points)** What happens on openMaze for the various search strategies?**(5 points)**

Question 5: Finding All the Corners (20 points)

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.

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

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 Pacman 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 Pacman position and the location of the four corners.

Bonus Problem: (Question 6) Heuristic for corners (up to 10 points)

Implement a non-trivial, consistent heuristic for the CornersProblem in cornersHeuristic.

python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5

Note: AStarCornersAgent is a shortcut for

-p SearchAgent -a fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeuristic.

You may test your performance by typing:

python autograder.py -q

Admissibility vs. Consistency: Remember, heuristics are just functions that take search states and return numbers that estimate the cost to a 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 in f-value. Moreover, if UCS and A* ever return paths of different lengths, your heuristic is inconsistent. This stuff is tricky!

Non-Trivial Heuristics: The trivial heuristics are the ones that return zero everywhere (UCS) and the heuristic which computes the true completion cost. The former won't save you any time, while the latter will timeout the autograder. 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).

Grading: Your heuristic must be a non-trivial non-negative consistent heuristic to receive any points. 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 be graded:

Number of nodes expanded	Grade
more than 2000	2
at most 2000	6
at most 1600	8
at most 1200	10

Remember: You will still get 2 points even if the heuristic is inconsistent, just for attempting the problem!