CSCI 3202 – Introduction to Artificial Intelligence
Instructor: Hoenigman
Assignment 2
Due: Wednesday, September 17 before 3pm

Search in Pacman

All files needed for this assignment are included in the `search.zip` file on Moodle. To submit your assignment, print the search.py and searchAgents.py files and the output of the autograder and turn these in at the beginning of class on September 17. Zip all files together, including the files you didn't modify, and submit them on Moodle.

There are four questions that require written answers where you're asked to explain what the code is doing. Type your answers to these questions as comments in your search.py file.

This assignment is reproduced from the Pacman Search assignment in the CS188x Berkeley edX Artificial Intelligence course.

The search.zip file should contain the following files:

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 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 and test cases for each question.
searchTestClasses.py	Assignment-specific autograding test classes.

Using the Autograder

Included in the search.zip archive is an autograder that includes graph based test cases for testing your code. We encourage you to master these test cases and debug on them before running your code with Pacman. Since the Pacman world is considerably more complex, and generally has much more state than the graph based test cases, debugging your code using Pacman will be a difficult and error prone process.

There are several ways to invoke the autograder, which you may find helpful during your development process. For example, to invoke the autograder for question 2 only, run

```
python autograder.py -q q2
```

Note that the extra credit is invoked using -q extra.

If you notice that you are failing a particular test within question 2, such as graph_infinite.test, you can specify that you would like the autograder to run only that test as follows

```
python autograder.py -t test_cases/q2/graph_infinite
```

Notice that the argument given is the actual path to the file specifying the test case itself, sans the .test extension. If you explore the test_cases directory, you will notice that there is a subdirectory corresponding to each question, and that there is a .solution file corresponding to each test.

Finally, if you would like the autograder to display both the test case and the solution for all tests it runs, you may add the flag -p as follows

```
python autograder.py -p -t test_cases/q2/graph_bfs_vs_dfs
```

Once you're passing the graph based test cases and have used those to debug your code, we encourage you to give Pacman a try and watch your code help him navigate his world.

Welcome to Pacman

After downloading the code (`search.zip`), 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
```

Note: if you get error messages regarding Tkinter, see

http://tkinter.unpythonic.net/wiki/How_to_install_Tkinter

Question 1: Depth First Search (2 points)

In this question, you will 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. As you work through the following questions, you might find it useful to refer to the object glossary at the end of this assignment writeup.

To help you get started, pseudocode for the search algorithms you'll write can be found in your textbook *(Chapter 3 in the third edition and Chapter 2 in the second*

edition). Remember that a search node must contain not only a state but also the information necessary to reconstruct the path (plan) that gets to that state.

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.

Important note: All of your search functions need to return a list of *actions* that will reach a goal state from a start state.

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).

As you complete your implementation, you may test it using the command

```
python autograder.py -q q1
```

Notice that the autograder includes a graph based test (test_cases/q1/graph_infinite) designed to make sure you have implemented the *graph* version of DFS as this will be a necessary improvement in the Pacman world.

Applying Your Search Implementations to Pacman

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 you implement in questions 1-4 will formulate the plan.

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.

Once your DFS implementation is complete, 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).

Written Questions to Answer: (1 point each)
1. Is the exploration order what you would have expected?
2. 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).
3. Is this a least cost solution? If not, what is depth-first search doing wrong?

Question 2: Breadth First Search (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 using the autograder the same way you did for depth-first search.

```
python autograder.py -q q2
```

Once your BFS implementation is complete, you may wish to view its performance with Pacman. To do this, run

```
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`.

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: Uniform Cost Search (2 points)

While BFS will find a fewest-actions path to the goal, we might want to find paths that are "best" in other senses. Uniform-cost graph search achieves this by allowing us to find optimal solutions while associating a distinct cost with each action.

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.

The test cases provided by the autograder invoked as

```
python autograder.py -q q3
```

include graph based examples which assign unique costs to each edge, helping you to debug your implementation.

Varying Costs in Pacman

To understand the motivation for varying costs in the context of Pacman, 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.

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: A* (3 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.

Notice that, unlike the generic search algorithms you have been implementing, heuristics are inherently problem specific. For the graph based search problems in the autograder we have supplied arbitrary heuristic functions, which will be given to A* search as arguments. Graph based tests (as well as Pacman tests) can be invoked using

```
python autograder.py -q q4
```

Unlike the arbitrary heuristics used in graph-based test cases, Pacman heuristics have significance in the context of Pacman. 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).

Written Question to Answer: (1 point)
4. What happens on `openMaze` for the various search strategies?

Question 5: Corners (2 points)

Note: Make sure to complete Question 2 before working on Question 5, because Question 5 builds upon your answer for Question 2.

The real power of A* will only be apparent with a more challenging search problem. Like heuristics, search problem implementations are not generic but are (or course) specific to the problem you are solving. 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.

Since search problem implementations are not generic, there will be no generic graph based tests to help you debug this implementation. However, you can still run the test suite for this problem by invoking

```
python autograder.py -q q5
```

Once your search problem is fully implemented, your 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 `PacmanGameState` 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.

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

Question 6: Heuristics (3 points)

Note: Make sure to complete Question 4 before working on Question 6, because Question 6 builds upon your answer for Question 4.

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

```
python pacman.py -l mediumCorners -p SearchAgent -a
fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeurist
ic -z 0.5.
```

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 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 that 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	0/3
At most 2000	1/3

At most 1600	2/3
At most 1200	3/3

Remember: If your heuristic is inconsistent, you will receive *no* credit, so be careful!

Question 7: Eating All The Dots (4 points)

Note: Make sure to complete Question 4 before working on Question 7, because Question 7 builds upon your answer for Question 4.

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).

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

Note: AStarFoodSearchAgent is a shortcut for

```
python pacman.py -l testSearch -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.

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

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

Our UCS agent finds the optimal solution in about 13 seconds, exploring over 16,000 nodes.

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 (full credit; medium)
At most 7000	5/4 (optional extra credit; hard)

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.

Question 8: Replanning (2 points)

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:

```
python 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.

Object Glossary

Here's a glossary of the key objects in the code base related to search problems, for your reference:

Search Problem (search.py)

A Search Problem is an abstract object that represents the state space, successor function, costs, and goal state of a problem. You will interact with any Search Problem only through the methods defined at the top of search.py.

PositionSearchProblem (searchAgents.py)

A specific type of Search Problem that you will be working with --- it corresponds to searching for a single pellet in a maze.

CornersProblem (searchAgents.py)

A specific type of Search Problem 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 Search Problem 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 that takes an instance of Search Problem as a parameter, runs some algorithm, and returns a sequence of actions that lead to a goal.

Examples 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.