Introduction

In this project, your Pacman agent will find more paths through the maze world, both to reach a particular location and to collect food efficiently.

As in homework 2, this project includes an autograder for you to grade your answers on your machine.

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 as a zip archive file: homework3.zip.

Note that you might also find it useful to refer to the object glossary.

Files you'll edit and submit:

search.py: where your A* search algorithms will reside
searchAgents.py: where all of your search-based agents and heuristics will reside.

You will fill in portions of search.py and searchAgents.py. You should submit these files with your code and comments. Please *do not* change the other files in this distribution or submit any of the original files other than these two files.

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 the test cases for each question

searchTestClasses.py: homework2 specific autograding test classes

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.

Academic Dishonesty: Your code will be checked against other submissions in the class for logical redundancy. If you copy someone else's code and submit it with minor changes, it will be easy to detect. These cheat detectors are quite hard to fool, so please don't try. I trust you all to submit your own work only; *please* don't let me down.

Getting Help: You are not alone! If you find yourself stuck on something, please ask for help. Office hours and the discussion forum are there for your support; please use them. I want these projects to be rewarding and instructional, not frustrating and demoralizing. But, I don't know when or how to help unless you ask.

Discussion: Please be careful not to post spoilers.

Question 1 (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 nullHeuristicheuristic 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 - implemented in homework 2 (about 549 vs. 620 search nodes expanded in our implementation, but ties in priority may make your numbers differ slightly).

Note that if you run A* with a null heuristic, you get ucs.

Question 2 (3 points): Corners Problem: Heuristic

The real power of A* will only be apparent with a more challenging search problem. In *corner mazes*, there are four dots, one in each corner. Our new search problem, CornersProblem, 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!

Implement a non-trivial, consistent heuristic for the CornersProblem in the function cornersHeuristic (in searchAgents.py.)

```
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.
```

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	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 3 (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).

```
python 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 0.7 second 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 with a total cost of 60 in about 3.4 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)

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

Question 4 (3 points): Suboptimal Search

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!) 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 (the function isGoalState). Then, solve that problem with an appropriate search function. (Remember that ucs is also A* with a null heuristic.) The solution should be very short!

Submission

Upload the files search.py and searchAgent.py to get credit on your project.