Artificial Intelligence	**Assignment 1**
comp440/comp557 Fall 2018	**4 September 2018**

This assignment is due September 4 at 8 pm on Canvas. Download `assignment1.zip` *from Canvas. There are four problems worth a total of 153 regular points and 12 points of extra credit for comp440; and 165 points for comp557. Problems 1, 2 and 3 require written work only, Problem 4 requires Python code and a writeup. All written work should be placed in a file called* `writeup.pdf` *with problem numbers clearly identified. All code should be included in* `search.py` *and* `searchAgents.py` *at the labeled points. For Problem 4, please run the auto grader using the command line* `python autograder.py` *and append the results in your writeup. Upload three documents on Canvas when you are done:* `writeup.pdf`*,* `search.py` *and* `searchAgents.py`*. Your homework will not be graded if you do not follow our submission instructions.*

1 Search problems with negative path costs (25 points)

(From 3.8 in the textbook) In this problem we explore the class of deterministic, single-agent search problems with negative path costs.

- (5 points) Suppose that edges in the state space graph (i.e., actions) can have arbitrarily large negative costs; explain why this possibility would force any optimal shortest path finding algorithm to explore all paths in the state space.
- (5 points) Does the condition that all edge costs be greater than or equal to some fixed cost $c < 0$ allow an optimal algorithm to do better (i.e., not explore every path in the state space)? For instance, consider adding $-c$ to all edge costs in the graph to eliminate negative costs in the state space graph.
- (5 points) Suppose there is a sequence of edges (actions) that form a cycle, so that executing this sequence results in no net change to the state. If the sum of edge costs in the cycle is negative,what does this imply about the optimal behavior of an agent in such an environment?
- (5 points) One can easily imagine actions with high negative cost, even in domains such as route-finding. For example, some stretches of road might have such beautiful scenery as to far outweigh the normal costs of time and fuel. Explain, in precise terms, using the vocabulary of state-space search, why humans do not drive around scenic loops indefinitely, and explain how to define the state space and actions for route-finding so that artificial agents can also avoid looping.
- (5 points) Give a real-world example of a search problem in which there are cycles with negative cost.

2 Analyzing Search Algorithms: dynamic A* (20 points)

In this problem, you will analyze a variation of A*, called Dynamic A* or D* that has been successfully used in robot navigation in unknown terrains. In particular, it is now widely used in the DARPA Unmanned Ground Vehicle Program. It has been integrated into Mars Rover prototypes, and military robots for urban reconnaissance.

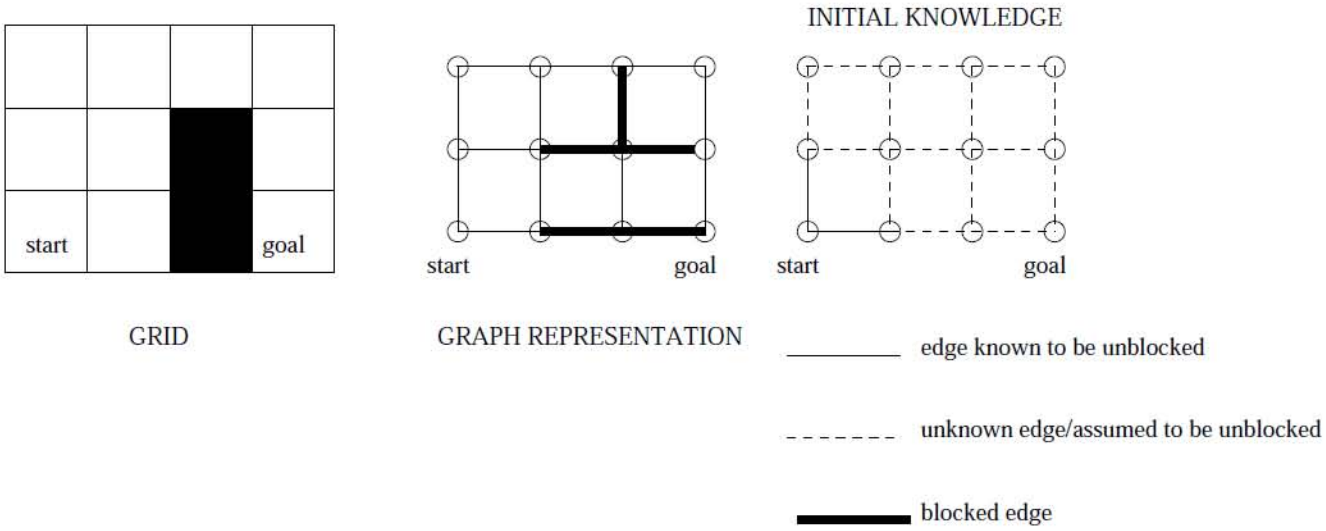

Figure 1: This is a simple grid world with the corresponding edge blocked graph shown, assuming that the robot knows the boundaries of its terrain and its start and goal location. It can observe walls or obstacles immediately surrounding it in the four main compass directions, and can move to each of the four adjacent cells/vertices unless an obstacle blocks its path.

Our context is mobile robot navigation where the robots are equipped with a radial short-distance sensor, and they are capable of error-free motion and sensing. Navigation in an unknown terrain is modeled as a search task over a finite undirected graph $G = (V, E)$ where a subset B of the edges E are blocked (due to obstacles) The robot knows V and E. It begins at a designated start vertex, and is given a goal vertex to get to. What it does not know is which subset of the edges are blocked. When the robot is at vertex v, it learns which edges incident to v are blocked. Dynamic A* uses A* to compute the shortest path to the goal vertex in G assuming that all edges it hasn't encountered are not blocked. It follows the computed path, until it meets a blocked edge, at which point it updates its map by adding the observed blocked edge to its list of blocked edges. It continues by finding another shortest path from the current vertex to the goal vertex in the updated map with the added blocked edge (and assuming unknown edges are traversable).

a. (5 points) Show that Dynamic A* is complete. That is, if there is a path from the start vertex to the goal vertex in the graph, Dynamic A* will find it, or tell us that no path exists.

b. (10 points) Dynamic A* is an instance of an online search algorithm. Unlike A*, which assumes all information about the graph is known at the start, Dynamic A* works with available information and augments it as it explores the graph. Define the travel cost of Dynamic A* to be the number of edges traversed before it terminates. Derive as tight a bound on the worst-case travel cost of Dynamic A* as you can. Explain your answer.

c. (5 points) Dynamic A* is widely used in practical mobile robotics. State two reasons why it may be preferred to other search algorithms.

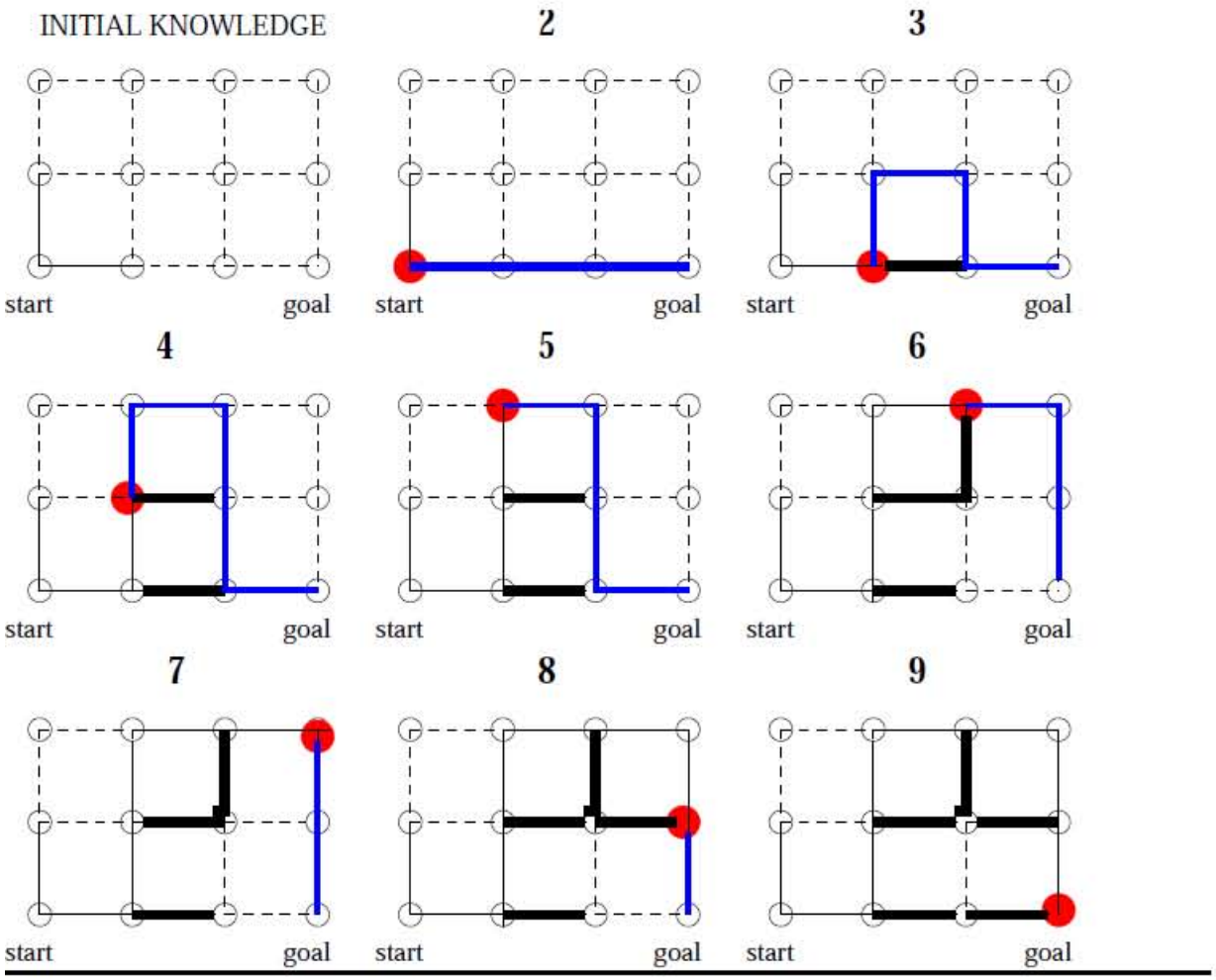

Figure 2: This figure shows a possible trajectory of Dynamic A* on the graph of Figure 2. Note that the cells in the grid-world correspond to the vertices of the graph. The robot is shown in red, and the calculated shortest paths are shown in blue. Thick black lines denote blocked edges.

3 Search space formulations and admissible heuristics (20 points)

n vehicles occupy squares $(1, 1)$ through $(1, n)$ (i.e., the bottom row of an $n \times n$ grid). The vehicles must be moved to the top row, but in reverse order; so that vehicle i that starts in $(i, 1)$ must end up at $(n, n - i + 1)$. On each time step, every one of the n vehicles can move one square up, down, left or right, or stay put. If a vehicle stays put, one other adjacent vehicle (but not more than one) can hop over it. Two vehicles cannot occupy the same square.

- (5 points) Calculate the size of the state space as a function of n.
- (5 points) Calculate the branching factor as a function of n.
- (7 points) Suppose that vehicle i is at (x_i, y_i). Write a nontrivial admissible heuristic h_i for the number of moves it will require to get to its goal location $(n-i+1, n)$, assuming no other vehicles are on the grid.
- (3 points) Which of the following heuristics are admissible for the problem of moving all n vehicles to their destinations? Explain your reasoning.
  - $\sum_{i=1}^{n} h_i$
  - $max(h_1, \ldots, h_n)$
  - $min(h_1, \ldots, h_n)$

4 Pacman Search (100 points)

In this problem, taken from the Berkeley AI Pacman 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. This project includes an autograder for you to grade your answers on your machine. This can be run with the command:

```
python autograder.py
```

If you pass a test case, the autograder prints out `*** PASS: ..` with the name of the test case that was used. It will run all tests on all problems and give you a composite score. However, if you want to run the autograder on a specific question, say Question 2, then use

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

The autograder's score will be multiplied by 4 to get your total score for this problem.. For every problem below we give you the autograder score only, your true score is four times that score.

The code for this problem 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 in the zip file `assignment1.zip` on Canvas.

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.

Academic Dishonesty: We will be checking your code against other submissions in the class as well as other solutions posted on github, for logical redundancy. If you copy someone else's code and submit it with minor changes, we will know. These cheat detectors are quite hard to fool, so please don't try. We trust you all to submit your own work only; please don't let us down. If you do, we will pursue the strongest consequences available to us. It is against the honor code to post your solutions on github. Warning: most of the solutions available on github are incorrect, so we advise you not to waste time on them.

Getting Help: You are not alone! If you find yourself stuck on something, contact the course staff for help. Office hours, section, and Piazza are there for your support; please use them. If you can't make our office hours, let us know and we will schedule more. We want this problem to be rewarding and instructional, not frustrating and demoralizing. But, we don't know when or how to help unless you ask.

Discussion: Please be careful not to post spoilers on Piazza.

File	Read?	Edit?	Description
search.py	Yes	Yes	Where all of your search algorithms will be.
searchAgents.py	yes	Yes	Where all of your search-based agents be.
pacman.py	Yes	No	The main file that runs Pacman games. This file describes a Pacman GameState type, which you use in this project.
game.py	Yes	No	The logic behind how the Pacman world works. This file describes several supporting types like AgentState, Agent, Direction, and Grid.
util.py	Yes	No	Useful data structures for implementing search algorithms.
graphicsDisplay.py	No	No	Graphics for Pacman.
garhicsUtils.py	No	No	Support for Pacman graphics.
textDisplay.py	No	No	ASCII graphics for Pacman.
ghostAgents.py	No	No	Agents to control ghosts.
keyboardAgents.py	No	No	Keyboard interfaces to control Pacman.
layout.py	No	No	Code for reading layout files and storing their contents.
autograder.py	No	No	Project autograder.
testParser.py	No	No	Parses autograder test and solution files.
testClasses.py	No	No	General autograding test classes.
test_cases/	No	No	Directory containing the test cases for each question.
searchTestClasses.py	No	No	Project 1 specific autograding test classes

Warming up

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 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`.

Note: if you get error messages regarding Tkinter, see
`http://tkinter.unpythonic.net/wiki/How_to_install_Tkinter`.

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:

```
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 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:

```
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). 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.

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.

```
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 (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):

```
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 (3 points): A* search

Implement A* graph search in the empty function
t 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). What happens on `openMaze` for the various search strategies? Write it up in `writeup.pdf`.

Question 5 (3 points): Finding All the Corners

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.

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

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.

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

Question 6 (3 points): Corners Problem: Heuristic

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

```
-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 7 (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 problem, 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 homework.) 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 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 7, because Question 7 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:

```
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. Report time for `mediumSearch` in `writeup.pdf`.

Question 8 (3 points): Suboptimal Search (extra credit for comp440 and required for comp557

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. Show this example in `writeup.pdf`.