Artificial Intelligence	**Assignment 2**
comp440/comp557 Fall 2016	**3 September 2016**

This assignment is due September 15 at 8 pm on Owlspace. Download `assignment2.zip` *from Canvas. There are four problems worth a total of 135 points and an optional extra credit problem worth 15 points for comp440 students. The extra problem is required for comp557 students. Problems 1, 2 and 4 require written work only, Problem 3 requires Python code and a writeup. All written work should be placed in a file called* `writeup.pdf` *with problem numbers clearly identified. Run the autograder using* `python grader.py` *and report results in your writeup. Caution: Our autograder only checks for running time and not correctness. Zip up the entire* `assignment2` *directory which should include your* `writeup.pdf` *and all of the Python code, and submit it as an attachment on Canvas by the due date/time.*

1 Pursuit evasion games (20 points)

(Slightly modified Problem 5.3 in your textbook) Consider a two-player pursuit-evasion game on a map where the players take turns moving. The game ends only when the players are on the same node. The terminal payoff to the pursuer P is minus the total time taken for capture. The evader E wins by never losing, i.e., never getting caught. An example map and a partial game tree for the map is shown in Figure 5.16 on page 196.

a. (2 points) Copy the game tree in Figure 5.16 and mark the values of the terminal nodes (there are three of them, labeled dd, dd and cc).

b. (2 points) Next to each named internal node in the partial game tree, write the strongest fact you can infer about its value (a number, one or more inequalities or a "?").

c. (1 points) Beneath each question mark in the partial game tree, write the name of the node reached by that branch.

d. (5 points) Explain how a bound on the value of the nodes in (b) can be derived from consideration of shortest-path lengths in the map, and derive such bounds for these nodes.

e. (5 points) Now suppose that the game tree as given, with the leaf value bounds from (d) is evaluated from left to right. Circle those "?" nodes that would **not** need to be *expanded* further, given the bounds from part (d), and cross out those that *need not be considered at all*.

f. (5 points) Can you prove anything in general about who wins the game on a map that is a tree?

2 Minimax and expectimax (25 points)

In this problem, you will investigate the relationship between expectimax trees and minimax trees for zero-sum two player games. Imagine you have a game which alternates between player 1 (max)

and player 2 (min). The game begins in state s_0, with player 1 to move. Player 1 can either choose a move using minimax search, or expectimax search, where player 2's nodes are chance rather than min nodes.

- (5 points) Draw a (small) game tree in which the root node has a larger value if expectimax search is used than if minimax is used, or argue why it is not possible.
- (5 points) Draw a (small) game tree in which the root node has a larger value if minimax search is used than if expectimax is used, or argue why it is not possible.
- (5 points) Under what assumptions about player 2 should player 1 use minimax search rather than expectimax search to select a move?
- (5 points) Under what assumptions about player 2 should player 1 use expectimax search rather than minimax search?
- (5 points) Imagine that player 1 wishes to act optimally (rationally), and player 1 knows that player 2 also intends to act optimally. However, player 1 also knows that player 2 (mistakenly) believes that player 1 is moving uniformly at random rather than optimally. Explain how player 1 should use this knowledge to select a move. Your answer should be a precise algorithm involving a game tree search.

3 Multi-agent pacman ((65 points + 15 EC) for comp440, 80 points for comp557)

Pacman (the yellow guy in the Figure 1) moves around in a maze and tries to eat as many food pellets (the small white dots) as possible, while avoiding the ghosts (the other two guys with eyes in Figure 1). If pacman eats all the food in a maze, it wins. The big white dots at the top-left and bottom right corner are capsules, which give pacman power to eat ghosts for a limited time (but you won't be worrying about them for the required part of the assignment). You can get familiar with the setting by playing a few games of classic pacman. In this problem, you will design and implement agents for the classic version of pacman. Your agents will use minimax and expectimax search to choose actions. The base code for this project contains files described in Table 1. You will only modify `submission.py`. A basic `grader.py` has been provided. However, it only checks for timing and not correctness. Passing this autograder does not necessarily mean you have a correct implementation.

3.1 Warmup

First, play a game of classic pacman to get a feel for the assignment by running the following on a Terminal command line in the `assignment2` folder.

```
python pacman.py
```

Name	Read?	Modify?	Description
`submission.py`	Yes	Yes	Where all of your multi-agent search code resides and the only file you need to modify for this assignment.
`pacman.py`	Yes	No	The main file that runs pacman games. This file also describes a pacman `GameState` type, which you will use extensively in this problem
`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
`graphicsUtils.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

Table 1: Code base for multi-agent pacman

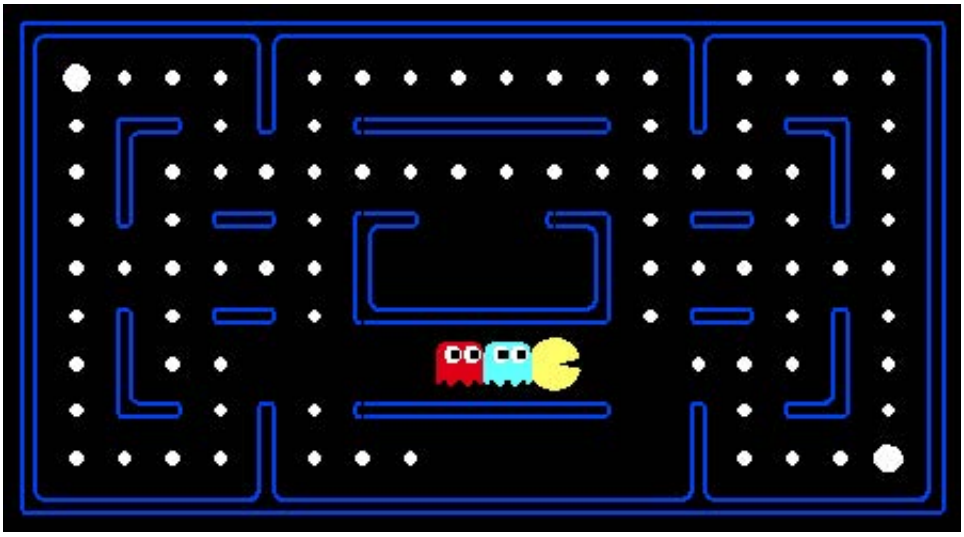

Figure 1: The standard pacman grid

The command `python pacman.py --frameTime -1` will cause the game to pause after every frame. We have provided a reflex agent for you in `multiAgents.py`. You can run it with the following command line.

```
python pacman.py -p ReflexAgent
```

The agent plays poorly, even on simple layouts.

```
python pacman.py -p ReflexAgent -l testClassic
```

You can also try out the reflex agent on the default `mediumClassic` layout with one ghost or two, as follows.

```
python pacman.py -p ReflexAgent -k 1
python pacman.py -p ReflexAgent -k 2
```

Default ghosts are random; you can also play for fun with slightly smarter directional ghosts using the command line option `-g DirectionalGhost`. You can also play multiple games in a row with the option `-n`. Turn off graphics with option `-q` to run lots of games quickly. Now read the `ReflexAgent` code carefully (in `submission.py`) and make sure you understand how it works. The reflex agent code provides some helpful examples of methods that query the `GameState` (a `GameState` specifies the full game state, including the food, capsules, agent configurations and score changes) for information. You will use these method calls in your agent code.

3.2 Problem 1: Minimax (25 points)

- (5 points) Pacman has multiple ghosts as adversaries. So we will extend the minimax algorithm from class (which had only one min stage for a single adversary) to the more general case of multiple adversaries. In particular, your minimax tree will have multiple min layers (one for each ghost) for every max layer.

  Consider depth limited minimax search with evaluation functions covered in class. Suppose there are $n+1$ agents on the board, $a_0, a_1, \ldots, a_n$, where a_0 is pacman and the rest are ghosts. Pacman is the max agent, and the ghosts are min agents. A depth one search consists of all agents making a move, and a depth two search will involve pacman and each ghost moving two times. In other words, a search depth of two generates a minimax game tree of height $2(n+1)$.

  Write the recurrence for $V_{opt}(s)$, which is the minimax value with search stopping at depth d_{max}. You should express your answer in terms of the following functions: `isEnd(s)`, which tells you if `s` is an end state; `Evaluation(s)`, an evaluation function for the state `s`; `Player(s)`, which returns the player whose turn it is in state `s`; and `Actions(s)`, which returns the possible actions in state `s`. Place your answer in `writeup.pdf`

- (20 points) Now fill out the `MinimaxAgent` class in `submission.py` using the above recurrence. Remember that your minimax agent should work with any number of ghosts, and your minimax tree should have multiple min layers (one for each ghost) for every max layer.

Your code should also be able to expand the game tree to an arbitrary depth. Score the leaves of your minimax tree with the supplied `self.evaluationFunction`, which defaults to `scoreEvaluationFunction`. The class `MinimaxAgent` extends `MultiAgentSearchAgent`, which gives access to `self.depth` and `self.evaluationFunction`. Make sure your minimax code makes reference to these two variables where appropriate as these variables are instantiated from the command line options. Other functions that you might use in the code: `GameState.getLegalActions()` which returns all the possible legal moves, where each move is `Directions.X` for some `X` in the set {`North, South, West, East, Stop`}. Read the `ReflexAgent` code to see how the above are used and also for other important methods like `GameState.getPacmanState()`, `GameState.getGhostStates()` etc. These are further documented inside the `MinimaxAgent` class.

3.2.1 Hints and Observations for Problem 1

- The evaluation function for this part is `self.evaluationFunction`. You should not change this function, but recognize that now we are evaluating states rather than actions. Look-ahead agents evaluate future states whereas reflex agents evaluate actions from the current state.
- The minimax values of the initial state in the `minimaxClassic` layout are 9, 8, 7, -492 for depths 1, 2, 3 and 4 respectively. You can use these numbers to verify if your implementation is correct. Note that your minimax agent will often win (15/20 games for us) despite the dire prediction of depth four minimax. The command line below will run your `Minimax` agent for depth 4.

  ```
  python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4
  ```

- To increase the search depth achievable by your agent, remove the `Directions.STOP` action from pacman's list of possible actions. Depth 2 should be pretty quick, but depth 3 or 4 will be slow. In the next problem, we will try to speed up search using alpha-beta pruning.
- Pacman is always agent 0, and the agents move in order of increasing agent index.
- Functions are provided to get legal moves for pacman or the ghosts and to execute a move by any agent. See `GameState` in `pacman.py` for details.
- All states in minimax should be `GameStates`, either passed in to `getAction` or generated via `GameState.generateSuccessor`. In this problem, you will not be abstracting to simplified states.
- On larger boards such as `openClassic` and `mediumClassic` (the default), you will find pacman to be good at not dying, but quite bad at winning. He'll often thrash around without making progress. He might even thrash around right next to a dot without eating it. Do not worry if you see this behavior. Why does pacman thrash around right next to a dot? Place your explanation in `writeup.pdf`.
- Consider the following run:

```
python pacman.py -p MinimaxAgent -l trappedClassic -a depth=3
```

Why does pacman rush to the closest ghost in minimax search on `trappedClassic`? Place your explanation in `writeup.pdf`.

3.3 Problem 2: Alpha-beta pruning (10 points)

Make an agent that uses alpha-beta pruning to more efficiently explore the minimax tree, in `AlphaBetaAgent` in `submission.py`. Again, your algorithm will be slightly more general than the pseudo-code in the slides, so part of the challenge is to extend the alpha-beta pruning logic appropriately to multiple minimizer agents. You should see a speed-up (depth 3 alpha-beta will run as fast as depth 2 minimax). Ideally, depth 3 on `mediumClassic` should run in just a few seconds per move or faster.

```
python pacman.py -p AlphaBetaAgent -a depth=3
```

The `AlphaBetaAgent` minimax values should be identical to the `MinimaxAgent` minimax values, although the actions it selects can vary because of different tie-breaking behavior. Again, the minimax values of the initial state in the `minimaxClassic` layout are 9, 8, 7 and -492 for depths 1, 2, 3 and 4 respectively.

3.4 Problem 3: Expectimax (30 points)

- (5 points) Random ghosts are of course not optimal minimax agents, so modeling them with minimax search is not optimal. Instead, write down the recurrence for $V_{opt,\pi}(s)$, which is the maximum expected utility for pacman in state `s` against ghosts that follow the policy π of choosing a legal move uniformly at random. Your recurrence should resemble that of Problem 3.2. Place your recurrence in `writeup.pdf`.

- (25 points) Fill in `ExpectimaxAgent`, where your agent will no longer take the min over all ghost actions, but the expectation according to your agent's model of how the ghosts act. Assume pacman is playing against `RandomGhosts`, each of which chooses from its legal actions (`getLegalActions`) uniformly at random. You should now observe a braver pacman when in close quarters with ghosts. In particular, if pacman perceives that he could be trapped but might escape to grab a few more pieces of food, he will at least try.

  ```
  python pacman.py -p ExpectimaxAgent -l trappedClassic -a depth=3
  ```

  You may have to run this scenario a few times to see pacman's gamble pay off. Pacman will win half the time on average and for this particular command, the final score would be -502 if pacman loses and 532 if it wins (you can use these numbers to validate your implementation). Why does pacman's behavior in expectimax differ from the minimax case (i.e., why doesn't he head directly for the ghosts)? Place your answer in `writeup.pdf`.

3.5 Problem 4: Evaluation function (15 extra credit points for comp440; required for comp557)

Write a better evaluation function for pacman in the provided function `betterEvaluationFunction`. The evaluation function should evaluate states. You may use any tools at your disposal for evaluation, including uniform cost search from the previous assignment. With depth 2 search, your evaluation function should clear the `smallClassic` layout with two random ghosts more than half the time for full credit and still run at a reasonable rate.

```
python pacman.py -l smallClassic -p ExpectimaxAgent -a evalFn=better -q -n 10
```

Document your evaluation function! Points will be determined based on the average game score in 10 runs.

3.5.1 Hints and Observations for Problem 4

You may want to use the reciprocal of important values (such as distance to food) rather than the values themselves. One way you might want to write your evaluation function is to use a linear combination of features. That is, compute values for features about the state that you think are important, and then combine those features by multiplying them by different values (weights) and adding the results together. You might decide what to multiply each feature by based on how important you think it is.

4 Expectimax search and pruning (25 points)

(Slightly modified Problem 5.20 in your textbook) In the following, a max tree consists only of max nodes, whereas an expectimax tree consists of a max node at the root with alternating layes of chance and max nodes. At chance nodes, all outcome probabilities are non-zero. The goal is to find the value of the root with a bounded-depth search. For each of the parts below, either give an example or explain why this is impossible.

- (5 points) Assume that leaf values are finite but unbounded, is pruning (as in alpha-beta) ever possible in a max tree?
- (5 points) Is pruning ever possible in an expectimax tree under the same conditions?
- (5 points) If leaf values are all in the range [0,1], is pruning ever possible in a max tree? Give an example, or explain why not.
- (5 points) If leaf values are all in the range [0,1], is pruning ever possible in a expectimaxmax tree? Give an example, or explain why not.
- (5 points) Consider the outcomes of a chance node in an expectimax tree. Which of the following evaluation orders is most likely to yield pruning opportunities: (a) lowest probability first, (b) highest probability first, or (c) doesn't make a difference.