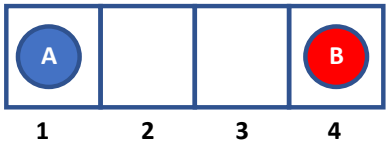
Artificial Intelligence	**Assignment 2**
comp440/comp557 Fall 2018	**4 September 2018**

This assignment is due September 17 at 8 pm on Owlspace. Download `assignment2.zip` *from Canvas. There are four problems worth 140 points (and a possible 15 extra credit points) for comp440, and worth 155 points for comp557. 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 your names and problem numbers clearly identified. Run the autograder using* `python grader.py` *and report results in your writeup. Upload two files separately onto Canvas: your* `writeup.pdf`*, your Python code in file* `multiAgents.py` *by the due date and time.*

1 Game trees with loops (20 points)

Consider the two-player game whose starting configuration is shown in the figure below. Player A moves first. The two players take turns moving, and each player must move his token to an open adjacent space in either direction. If the opponent occupies an adjacent space, then a player may jump over the opponent to the next open space if any. For example, if A is on square 3 and B is on square 2, then A may move back to square 1. The game ends when one player reaches the opposite end of the board. If player A reaches square 4 first, the value of the game to A is +1; if player B reaches square 1 first, the value of the game to A is -1.



- (5 points) Draw the complete game tree using the following conventions:
  - Write each state as (s_A, s_B) where s_A and s_B are token locations.
  - Put each terminal state in a square box and write its game value in a circle.
  - Put loop states (states that already appear on the path to the root) in double square boxes. Since their value is unclear, annotate each with a "?" in a circle.
- (5 points) Now mark each node with its backed up minimax value (also in a circle). Explain how you handled the "?" values and why.
- (5 points) Explain why the standard minimax algorithm would fail on this game tree and briefly sketch how you would fix it, drawing on your answer to the previous part. Does your modified algorithm give optimal decisions for all games with loops?
- (5 points) This 4-square game can be generalized into n squares for any $n \geq 2$. Prove that A wins if n is even and loses if n is odd.

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 ((75 points + 15 EC) for comp440, 90 points for comp557)

Pacman (the yellow guy in 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. The code base has not changed much from Assignment 1, but please use the files included in `assignment2.zip` , rather than intermingling files from Assignment 1. **You will only modify** `multiAgents.py`. You will only upload `multiAgents.py`.

As in Assignment 1, this project includes an autograder for you to grade your answers on your machine. This can be run on all questions with the command:

```
python autograder.py
```

Name	Read?	Modify?	Description
`multiAgents.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
`autograder.py`	No	No	Project autograder
`testParser.py`	No	No	Parses autograding test and solution files
`testClasses.py`	No	No	General autograding test classes
`test_cases/`	No	No	Directory containing test cases for each question
`multiagentTestClasses.py`	No	No	Assignment 2 specific autograding test classes.

Table 1: Code base for multi-agent pacman

It can be run for one particular question, such as q2, by:

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

It can be run for one particular test by commands of the form:

```
python autograder.py -t test_cases/q2/0-small-tree
```

By default, the autograder displays graphics with the `-t` option, but doesn't with the `-q` option. You can force graphics by using the `--graphics` flag, or force no graphics by using the `--no-graphics` flag.

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

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.

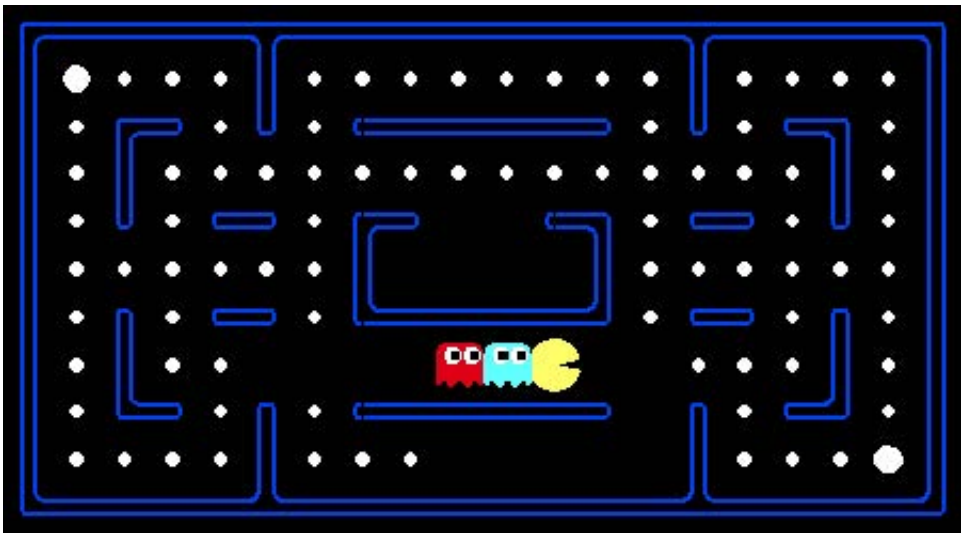


Figure 1: The standard pacman grid

Academic Dishonesty: We will be checking your code against other submissions in the class 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.

Getting Help: You are not alone! If you find yourself stuck on something, contact the course staff for help. Office hours, section, and the discussion forum 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 these projects to be rewarding and instructional, not frustrating and demoralizing. But, we don't know when or how to help unless you ask.

3.1 Problem 1: Reflex Agent (10 points)

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
```

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 with animation off, as follows.

```
python pacman.py --frameTime 0 -p ReflexAgent -k 1
python pacman.py --frameTime 0 -p ReflexAgent -k 2
```

How does your agent fare? It will likely often die with 2 ghosts on the default board, unless your evaluation function is quite good.

Now read the `ReflexAgent` code carefully (in `multiAgents.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. Now try adding as features, the reciprocal of important values (such as distance to food) rather than just the values themselves. Note: The evaluation function you're writing is evaluating state-action pairs; in later parts of the project, you'll be evaluating states.

Options: 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. If the randomness is preventing you from telling whether your agent is improving, you can use `-f` to run with a fixed random seed (same random choices every game).

Now read the `ReflexAgent` code carefully (in `multiAgents.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.

Grading: we will run your agent on the openClassic layout 10 times. You will receive 0 points if your agent times out, or never wins. You will receive 1 point if your agent wins at least 5 times, or 2 points if your agent wins all 10 games. You will receive an addition 1 point if your agent's average score is greater than 500, or 2 points if it is greater than 1000. You can try your agent out under these conditions with

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

To run it without graphics, use:

```
python autograder.py -q q1 --no-graphics
```

Don't spend too much time on this question, though, as the meat of the project lies ahead.

3.2 Problem 2: 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 single search ply is considered

to be one Pacman move and all the ghosts' responses, so depth 2 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 `multiAgents.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.

Grading: We will be checking your code to determine whether it explores the correct number of game states. This is the only way reliable way to detect some very subtle bugs in implementations of minimax. As a result, the autograder will be very picky about how many times you call `GameState.generateSuccessor`. If you call it any more or less than necessary, the autograder will complain. To test and debug your code, run

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

This will show what your algorithm does on a number of small trees, as well as a pacman game. To run it without graphics, use:

```
python autograder.py -q q2 --no-graphics
```

3.2.1 Hints and Observations for Problem 2

- The correct implementation of minimax will lead to Pacman losing the game in some tests. This is not a problem: as it is correct behavior, it will pass the tests.
- 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. **Have your program print these minimax values for the stated depths.** Note that your minimax agent will often win (665/1000 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
  ```

- 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 because he doesn't know where he would o after eating that dot. Do not worry if you see this behavior; question 5 will clean up these issues.

- Consider the following run:

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

  When Pacman believes that his death is unavoidable, he will try to end the game as soon as possible because of the constant penalty for living. Sometimes, this is the wrong thing to do with random ghosts, but minimax agents always assume the worst. Make sure you understand why Pacman rushes the closest ghost in this case.

3.3 Problem 3: 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 `smallClassic` should run in just a few seconds per move or faster.

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

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.

Grading: Because we check your code to determine whether it explores the correct number of states, it is important that you perform alpha-beta pruning without reordering children. In other words, successor states should always be processed in the order returned by GameState.getLegalActions. Again, do not call GameState.generateSuccessor more than necessary.

You must not prune on equality in order to match the set of states explored by our autograder. (Indeed, alternatively, but incompatible with our autograder, would be to also allow for pruning on equality and invoke alpha-beta once on each child of the root node, but this will not match the autograder.)

To test and debug your code, run

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

This will show what your algorithm does on a number of small trees, as well as a pacman game. To run it without graphics, use:

```
python autograder.py -q q3 --no-graphics
```

The correct implementation of alpha-beta pruning will lead to Pacman losing some of the tests. This is not a problem: as it is correct behavior, it will pass the tests.

3.4 Problem 4: 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`.

- (5 points) As with the search problems covered so far in this class, the beauty of the algorithms is their general applicability. To expedite your own development, we've supplied some test cases based on generic trees. You can debug your implementation on small game trees using the command:

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

  Debugging on these small and manageable test cases is recommended and will help you to find bugs quickly. Make sure when you compute your averages that you use floats. Integer division in Python truncates, so that $1/2 = 0$, unlike the case with floats where $1.0/2.0 = 0.5$.

  Once your algorithm is working on small trees, you can observe its success in Pacman. ExpectimaxAgent, will no longer take the min over all ghost actions, but the expectation according to your agent's model of how the ghosts act. To simplify your code, assume you will only be running against an adversary which chooses amongst their getLegalActions uniformly at random.

- (20 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.

  To see how the ExpectimaxAgent behaves in Pacman, run:

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

  You should now observe a more cavalier approach 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'll at least try. Investigate the results of these two scenarios:

  ```
  python pacman.py -p AlphaBetaAgent -l trappedClassic -a depth=3 -q -n 10
  python pacman.py -p ExpectimaxAgent -l trappedClassic -a depth=3 -q -n 10
  ```

  You should find that your ExpectimaxAgent wins about half the time, while your AlphaBetaAgent always loses. Make sure you understand why the behavior here differs from the minimax case.

  The correct implementation of expectimax will lead to Pacman losing some of the tests. This is not a problem: as it is correct behaviour, it will pass the tests.

  The final score would be -502 if pacman loses and 531 if it wins (**you can use these numbers to validate your implementation**).

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 and A* search from the previous assignment. With depth 2 search, your evaluation function should clear the smallClassic layout with one random ghost more than half the time and still run at a reasonable rate (to get full credit, Pacman should be averaging around 1000 points when he's winning).

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

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

Grading: the autograder will run your agent on the smallClassic layout 10 times. We will assign points to your evaluation function in the following way:

- If you win at least once without timing out the autograder, you receive 1 points. Any agent not satisfying these criteria will receive 0 points.
- +1 for winning at least 5 times, +2 for winning all 10 times

- +1 for an average score of at least 500, +2 for an average score of at least 1000 (including scores on lost games)
- +1 if your games take on average less than 30 seconds on the autograder machine. The autograder is run on EC2, so this machine will have a fair amount of resources, but your personal computer could be far less performant (netbooks) or far more performant (gaming rigs).
- The additional points for average score and computation time will only be awarded if you win at least 5 times.

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 Pruning in game trees with chance nodes (20 points)

Figure 5.19 in your textbook shows the complete game tree for a simple game. Assume that the leaf nodes are evaluated in left to right order and that before a leaf node is evaluated, we know nothing about its value, i.e. the range of possible values is $-\infty$ to $+\infty$.

- (5 points) Copy the game tree, mark the value of all internal nodes, and indicate the best move at the root with an arrow.
- (5 points) Given the values of the first six leaves, do we need to evaluate the seventh and eighth leaves? Explain your answers.
- (5 points) Suppose the leaf nodes are known to lie on the interval $[-2, 2]$. After the first two leaves are evaluated, what is the value range of the left hand chance node?
- (5 points) Circle all leaves that need not be evaluated under the assumption that the leaf values are all in $[-2, 2]$.