CS440 – Multi-Agent Search

Purpose: In this lab, you will design agents for the classic version of Pacman, including ghosts. Along the way, you will implement both minimax and expectimax search and try your hand at evaluation function design.

Getting Started: Download the multiagent.zip files and complete each of the following assignments to be submitted for grading. Each should be done individually but you can consult with a classmate to discuss your strategies or if you get an error message that you do not understand.

First play a game of classic Pacman by running `pacman.py` and using the arrow keys to move. Now run pacman with the provided `ReflexAgent` in `multiAgents.py` by using each of the command line options:

```
 -p ReflexAgent
 -p ReflexAgent -l testClassic
```

It plays quite poorly even on the simple layouts. Inspect the `ReflexAgent` code and make sure you understand what it is doing.

Assignment 1 – Reflex Agent:

Improve the `RefexAgent` to play respectably. The provided reflex agent code provides some helpful examples of methods that query the GameState for information. A capable reflex agent will have to consider both food locations and ghost locations to perform well. You may also want to try the reciprocal of important values (such as distance to objects) rather than just the values themselves.

Criteria for Success:

Your agent should easily and reliably clear the testClassic layout:
```
-p ReflexAgent -l testClassic
```

Your agent will likely often die with 2 ghosts on the default board, unless your evaluation function is quite good:

```
--frameTime 0 -p ReflexAgent -k 1
--frameTime 0 -p ReflexAgent -k 2
```

You will be graded by how well your agent performs on the OpenClassic layout on 10 trials:

You will receive 0 pts if your agent times out or never wins.
You will receive 1 pt if your agent wins at least 5 times.
You will receive 2 pts if your agent wins all 10 games.
You will receive an additional 1 pt if your agent's average score is greater than 500 or an additional 2 pts if it is greater than 1000.

You may use the autograder with q1 to verify everything is working.

Assignment 2 – Minimax:

Now you will write an adversarial search agent in the provided `MinimaxAgent` class stub in `multiAgents.py`. Your minimax agent should work with any number of ghosts, so you'll have to write an algorithm that is slightly more general than what you've previously seen in class. In particular, your minimax tree will have multiple min layers (one for each ghost) for every max layer.

Your code should also expand the game tree to an arbitrary depth. Score the leaves of your minimax tree with the supplied `self.evaluationFunction`, which defaults to `scoreEvaluationFunction`. `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 populated in response to command line options. Also note that Pacman is always agent 0, and the agents move in order of increasing agent index.

Criteria for Success:

Your code will be checked to determine whether it explores the correct number of game states. This is the only 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 the autograder with q2.

Observations: 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 the Pacman test in this part is already written and you shouldn't change it, but recognize that now we're evaluating states rather than actions, as we were for the reflex agent. Look-ahead agents evaluate future states whereas reflex agents evaluate actions from the current state.

On larger boards such as openClassic and mediumClassic (the default), you'll 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'd go after eating that dot. Don't worry if you see this behavior; assignment 5 will clean up all of these issues.

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. You can see this with:

```
 -p MinimaxAgent -l trappedClassic -a depth=3
```

Make sure you understand why Pacman rushes the closest ghost in this case.

Assignment 3 – Expectimax:

Minimax and alpha-beta are great, but they both assume that you are playing against an adversary who makes optimal decisions. As anyone who has ever won tic-tac-toe can tell you, this is not always the case. In this assignment you will implement the `ExpectimaxAgent`, which is useful for modeling probabilistic behavior of agents who may make suboptimal choices.

Criteria for Success:

As usual you can test your code with the autograder with q3. This tests on several small game trees.

Once your code is debugged you can see it in action with Pacman:

```
-p ExpectimaxAgent -l minimaxClassic -a depth=3
```

Assignment 4 – Evaluation Function:

Write a better evaluation function for pacman in the provided function `betterEvaluationFunction`. The evaluation function should evaluate states, rather than actions like your reflex agent evaluation function did. You may use any tools at your disposal for evaluation, including your search code from the first lab.

Criteria for Success:

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).

The autograder with q4 will run your agent on the smallClassic layout 10 times and award points 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.
You will get +1 points for winning at least 5 times, +2 points for winning all 10 times
You will get +1 points for an average score of at least 500, +2 points for an average score of at least 1000 (including scores on lost games)
You will get +1 points if your games take on average less than 30 seconds on phoenix when run with –no-graphics.
The additional points for average score and computation time will only be awarded if you win at least 5 times.

Submit your code in Canvas for grading.