Project 2

Submission requirements

- Submit one file to Carmen: `multiAgents.py`
- Expected time commitment: 3-6 hrs
- Due: Friday, Sept. 21, 5:00PM

Directions

1. Download and unzip this file with the Pacman source code: multiagent.zip. This code, and the idea for the assignment, comes from UC Berkeley.
2. Open up the Windows Command Line or Mac Terminal or Linux Terminal.
3. Change your directory to the folder with the pacman code.
4. Run `python pacman.py` to make sure your setup works.
5. As in project 1, you might need to look at these files:
    - `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.
6. Take a look at the minimax algorithm outlined on my website to complete the tasks.
7. Although we are only completing problems 2 and 4, you can run the autograder to check your work using the following command:
    - `python autograder.py`
8. Once completed, submit just the file `multiAgents.py` to carmen.

Task 1 (6 pts)

Implement minimax! 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 lecture. 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.

Important: 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.

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
```

Hints and 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 behaviour, it will pass the tests.
- The evaluation function for the pacman test in this part is already written (self.evaluationFunction). You shouldn't change this function, 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.
- The minimax values of the initial state in the minimaxClassic layout are 9, 8, 7, -492 for depths 1, 2, 3 and 4 respectively. Note that your minimax agent will often win (665/1000 games for us) despite the dire prediction of depth 4 minimax.

  ```
  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.
- All states in minimax should be `GameStates`, either passed in to `getAction` or generated via `GameState.generateSuccessor`. In this project, you will not be abstracting to simplified states.
- 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, you can solve it in the extra credit if you wish.
- 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:

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

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

Task 2 (4 points)

Minimax is great, but it assumes 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 question you will implement the `ExpectimaxAgent`, which is useful for modeling probabilistic behavior of agents who may make suboptimal choices.

As with the search and constraint satisfaction problems covered so far in this class, the beauty of these 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 the 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 2 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. Random ghosts are of course not optimal minimax agents, and so modeling them with minimax search may not be appropriate. `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.

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 `MinimaxAgent` 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.

Extra credit (+2 points)

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 last project. 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). Check your work with:

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

We will be holding a competition in the class to see who can come up with the best evaluation function for Minimax. We will award more extra credit to the winners of the competition: +3 points to the person who gets first place, +2, to the player in second, and +1 for third.

Hints and Observations

- 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 and adding the results together. You might decide what to multiply each feature by based on how important you think it is.
- You may want to use the reciprocal of important values (such as distance to food) rather than the values themselves.