Artificial Intelligence
Report for Project 02: Multi-Agent Pac-Man

Question 1 (15 points): Reflex Agent

Improve the ReflexAgent in multiAgents.py to play respectably.

Analysis:

- Evaluation function made using three chief parameters:
  - Food Count difference between current state and previous state (0 or 1) .
  - Closest Food Distance.
  - Ghost Distance.

Reflex Agent			
Command	**Memory Usage (Complexity)**	**Running Time Complexity**	**Critical Analysis**
python pacman.py -p ReflexAgent -l testClassic	O(1)	O(n*m) where n and m are PacMan grid dimensions.	PacMan emerged victorious with a score of 564.
python pacman.py --frameTime 0 -p ReflexAgent -k 1			PacMan emerged victorious with a score of 1223.
python pacman.py --frameTime 0 -p ReflexAgent -k 2			PacMan emerged victorious 5/10 times.

Question 2 (25 points): 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 lecture. In particular, your minimax tree will have multiple min layers (one for each ghost) for every max layer.

Analysis:

- By running PacMan using minimax we come to know that it always makes decisions based on the fact that the opponent (ghost) always makes optimal decisions. This leads PacMan to die in some cases when it can be avoided.

<table>
<tr><th colspan="4">Minimax</th></tr>
<tr><th>Command</th><th>Memory Usage (Complexity)</th><th>Running Time Complexity</th><th>Critical Analysis</th></tr>
<tr><td>python autograder.py -q q2</td><td rowspan="3">O(bm)
where b= brancing factor of the tree and m is the depth of the tree.</td><td rowspan="3">$O(b^m)$
where b= brancing factor of the tree and m is the depth of the tree.</td><td>PacMan passed all autograder test-cases. In the last test case, Pacman died with an average score of 84 (which is the correct execution).</td></tr>
<tr><td>python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4</td><td>PacMan wins and loses. This is because it assumes optimal behavior from ghosts.</td></tr>
<tr><td>python pacman.py -p MinimaxAgent -l trappedClassic -a depth=3</td><td>PacMan rushes to the nearest ghost and dies to minimize the penalty of living and as his death is unavoidable.</td></tr>
</table>

Question 3 (25 points): Alpha-Beta Pruning

Make a new agent that uses alpha-beta pruning to more efficiently explore the minimax tree, in AlphaBetaAgent. Again, your algorithm will be slightly more general than the pseudocode from lecture, so part of the challenge is to extend the alpha-beta pruning logic appropriately to multiple minimizer agents.

Analysis:

- A modification of Minimax in which we prune off the unnecessary states that we know are not going to be chosen by PacMan or the ghosts, by keeping a track of alpha and beta variables. So, its running time is better than minimax.

<table>
<tr><th colspan="4">Alpha-Beta Pruning</th></tr>
<tr><th>Command</th><th>Memory Usage (Complexity)</th><th>Running Time Complexity</th><th>Critical Analysis</th></tr>
<tr><td>python pacman.py -p AlphaBetaAgent -a depth=3 -l smallClassic</td><td rowspan="2">With perfect ordering O(b*m/2)</td><td rowspan="2">With perfect ordering O($b^{m/2}$)</td><td>PacMan dies. But dies quicker compared to Minimax. This is because we are pruning some of the states that do not need expanding.</td></tr>
<tr><td>python autograder.py -q q3</td><td>PacMan passes all the autograder test cases. In the last test case, Pacman died with an average score of 84 (which is the correct execution).</td></tr>
</table>

Question 4 (25 points): 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 question you will implement the ExpectimaxAgent, which is useful for modeling probabilistic behavior of agents who may make suboptimal choices.

Analysis:

- In this we chose the maximum of the averages of all possible states of the min nodes.
- The runtime is same as minimax but it enables Pacman to survive longer than in minimax.

<table>
<tr><th colspan="4">Expectimax</th></tr>
<tr><th>Command</th><th>Memory Usage (Complexity)</th><th>Running Time Complexity</th><th>Critical Analysis</th></tr>
<tr><td>python pacman.py -p ExpectimaxAgent -l minimaxClassic -a depth=3</td><td rowspan="4">O(bnd)
where n is the number of chance nodes, d is the depth of the tree and b is the branching factor.</td><td rowspan="4">$O(bn)^d$
where b is the branching factor, n is the chance nodes and d is the depth of the tree</td><td>PacMan dies.</td></tr>
<tr><td>python autograder.py -q q4</td><td>PacMan passes all autograder test cases.</td></tr>
<tr><td>python pacman.py -p AlphaBetaAgent -l trappedClassic -a depth=3 -q -n 10</td><td>PacMan loses in all cases when using alpha-beta pruning.</td></tr>
<tr><td>python pacman.py -p ExpectimaxAgent -l trappedClassic -a depth=3 -q -n 10</td><td>PacMan wins on an average of 5/10 times.</td></tr>
</table>