**Project 2: Multi-Agent Pacman**

Parker Gray

CSC 665

**Overview**

Like project 1, the goal of this project is to create algorithms that help Pacman through a maze to a goal state. Except this time, there’s an addition of enemy agents which Pacman will have to dodge to avoid dying and losing the game. This project is about incorporating algorithms that allow Pacman to achieve the goal state without dying to the enemy agents (ghosts).

**Problem Statement**

*Question 1* is concerned with improving the ReflexAgent class, which is a class that takes in the game state every time Pacman moves and makes the best choice according to the move that has the highest utility. *Question 2* incorporates the Minimax algorithm, which looks into possible future moves that Pacman and opposing ghosts can perform and chooses following moves by choosing the best path (most utility) for itself and assuming the worst path for the ghosts. *Question 3* adds alpha-beta pruning into the mix. Alpha-beta pruning is a version of Minimax in which the process of determining the best course is made considerably shorter by pruning possibilities that have no chance of having greater utility than whatever the current highest utility path is.

*Question 4* is all about Expectimax, which is Minimax with added probabilities for the leaf nodes of the path tree. This is meant to represent the fact that choices that ghosts make are random- each possible movement in the north, south, east, or western direction have a 25% chance of being performed assuming random movement. Expectimax calculates the possible movements of ghosts per movement of Pacman by adding in this probability aspect. *Question 5* is about creating a better evaluation function than the one provided in the code. This evaluation function takes in a state variable and deduces the best possible movement from the possible choices.

**Solution Design**

Starting off with Q1, the code to be added to the ReflexAgent class is located in the evaluationFunction function, which takes parameters of the current game state and the action to be performed. To start, the successor game state for Pacman after the action is performed is created, and aspects of the successor game state like Pacman’s new position, the locations of food, and the ghost states (including amount of time left scared if a big pellet was consumed). Following this, the reflex agent needs to determine the best move with the most utility in this successor state. To do this, I created a list of the food positions and recorded the total number of food- then iterate through the list of food positions until one is found that is the closest to Pacman’s current position. Pacman is sent on his way to that closest food pellet. If there’s no food left to collect, return 0.

In addition to checking where the closest food position is, the positions of ghosts are also checked- an if statement is run that checks if the current distance from Pacman to a ghost is less than or equal to one (that is, the ghost is next to Pacman). If the ghost *is* next to Pacman, give that coordinate a score value of -1e6 and avoid that position. Following the calculation of closest food and closest ghost, the score value is returned.

Q2 is up next. Minimax as an algorithm is organized like a tree in which the top node is what is to be returned. Returning that top node requires exploring all of the children nodes first and determining the optimal actions for Pacman and the ghosts. The total number of agents is collected first and foremost, then the Minimax algorithm begins. The minimax algorithm first checks if the number of iterations of Minimax exceeds the limit number (# of agents\*current depth in the tree) or if the game state is a win or a loss. If any of those scenarios is true, then run the evaluationFunction on the current state and get the best action. Otherwise, first find the minimum utility move for the ghosts- do this by iterating through the list of possible actions for the ghost for the next state and compare the utility of those actions to the placeholder value of 1e10. Return the lowest value between 1e10 and the generated lowest utility action for the ghost.

For Pacman, the maximum utility is required. Pacman’s part is a repeat of the above part with two changes- the placeholder value is -1e10 and the max function is called when comparing that placeholder value and Pacman’s possible actions in the next game state. If this is the first time that \_miniMax() is being ran, then the best utility action that Pacman makes in the next game state is appended to a previously defined list ActionScore.

Following both calculations of the max for Pacman and the min for ghosts, the returned value of MinimaxAgent() is the index within ActionScore that has the highest total utility from the set of current legal actions.

Q3 is a repeat of Minimax with the addition of alpha beta pruning. This means that the AlphaBetaAgent() code is mostly a repeat of MinimaxAgent() with a few changes. In the calculation of the minimum utility choice for the ghosts, a value called ‘beta’ is updated to be the minimum between itself (initialized at -1e20) and the calculated lowest utility choice for the ghosts. And in the calculation of the maximum utility choice for Pacman, the value ‘alpha’ is modified such that it is the maximum value between itself (initialized at 1e20) and the highest utility choice for Pacman. Every time that beta is less than alpha within either calculation for ghosts or Pacman, the function is broken out of the loop of iterating actions and ‘prunes’ whatever iteration it was on by doing so. This function returns the same highest ActionScore index as before.

Q4 is Expectimax, which again means it’s mostly a repeat of Minimax. Gone is the alpha beta aspect of Q3- instead, a new list called successorScore is created that will be made to hold the utilities of all possible actions of ghosts. After calculation of the ghost min utilities, a value called averageScore is calculated that sums up the values x/(# of entries in successorScore) for all number of entries x in successorScore. The job of this line is to calculate the probabilities of each possible ghost move and store it in averageScore, that is, the average utility of the ghosts’ moves given random probability. The averageScore is returned and is used to get out of the if statement in which the ghost minimum is calculated. Pacman’s max calculation is unchanged- only the ghost’s moves have probabilities assigned to them. Pacman’s moves are always purposeful.

Q5 is the creation of a better evaluation function. The evaluation function I created takes into account three different scoring situations- the scoring created concerning ghosts in their natural or scared states, the scoring created when deciding which food to go after, and the scoring created for which big pellet to go after. In addition, there’s also a function to make Pacman suicide if necessary. The three different scoring situations outlined above all have a similar code implementation- all three get the states of ghosts, food, and big pellets and then calculate the closest ghost, food, or big pellet to Pacman’s current position. In the case of a big pellet being consumed, the ghost positions go from very negative scoring to very positive scoring- Pacman is incentivized to chase after the scared ghosts for big points. For food scoring and big pellet scoring, these are straightforward ‘aim for the closest of each’ functions. The suicide function targets the closest possible ghost and makes a beeline for it. The score from the three scoring functions is added to the current game state’s score and returned to finish off the evaluation function.

**Solution Result/Evaluation**

All autograder.py tests pass with zero issues.

**Conclusion**

It’s pretty fortunate that this project has three questions that use basically the same code more or less. The project was helpful and informative when the tests were finally ran- it was useful to see how the algorithms discussed in class as theory were used in practice.