Department of Computer Science and Engineering

Islamic University of Technology (IUT)

A subsidiary organ of OIC

CSE 4618: Artificial Intelligence

Lab Report 04

Name	**:**	**M M Nazmul Hossain**
Student ID	**:**	**200042118**
Semester	**:**	**6th**
Academic Year	**:**	**2022-2023**
Date of Submission	**:**	**16.04.2024**

Problem-1 (Reflex Agent)

Analysis

The first task required the definition of a reflex agent that will be able to reliably detect, when provided a state and an action, from all the legal actions, whether that action is advised to be taken or not from that specific state. Essentially, it required the definition of an evaluation function of a state-action pair. The idea is to identify from any state, out of all the possible legal actions, the best move. This can be seen as the reflex action from a set state.

Solution

```
20  class ReflexAgent(Agent):
52
53      def evaluationFunction(self, currentGameState, action):
54          """
55          Design a better evaluation function here.
56
57          The evaluation function takes in the current and proposed successor
58          GameStates (pacman.py) and returns a number, where higher numbers are better.
59
60          The code below extracts some useful information from the state, like the
61          remaining food (newFood) and Pacman position after moving (newPos).
62          newScaredTimes holds the number of moves that each ghost will remain
63          scared because of Pacman having eaten a power pellet.
64
65          Print out these variables to see what you're getting, then combine them
66          to create a masterful evaluation function.
67          """
68          # Useful information you can extract from a GameState (pacman.py)
69          successorGameState = currentGameState.generatePacmanSuccessor(action)
70          newPosition = successorGameState.getPacmanPosition()
71          newFood = successorGameState.getFood()
72          newGhostStates = successorGameState.getGhostStates()
73          newScaredTimes = [ghostState.scaredTimer for ghostState in newGhostStates]
74
75          "*** YOUR CODE HERE ***"
76          # return successorGameState.getScore()
77          foods = newFood.asList()
78          foodDistances = []
79          evaluation =  0
80          capsules = successorGameState.getCapsules()
81          capsuleDistances = []
82
83          # print(str(newFood))
84          # print(str(foods))
85
86          if(newPosition == currentGameState.getPacmanPosition()):
87              return (-float("inf"))
88
89          if successorGameState.isLose():
90              return -float("inf")
91
92          if successorGameState.isWin():
93              return float ('inf')
94
95          for food in foods:
96              foodDistances.append(manhattanDistance(food,newPosition))
97
98          for capsule in capsules:
99              capsuleDistances.append(manhattanDistance(capsule,newPosition))
100
101         for ghost in newGhostStates:
102             if(manhattanDistance(newPosition,ghost.getPosition()) <=50):
103                 if (ghost.scaredTimer == 0 and manhattanDistance(newPosition,ghost.getPosition()) <=5):
104                     if (min(capsuleDistances, default=0) != 0):
105                         evaluation += 500 / (min(capsuleDistances, default=0))
106                     if manhattanDistance(newPosition,ghost.getPosition()) <=1:
107                         return -float("inf")
108                 # if (ghost.scaredTimer == 0 and manhattanDistance(newPosition,ghost.getPosition()) <=1):
109                 #     return -float('inf')
110                 elif ghost.scaredTimer != 0:
111                     evaluation += 2000 / (manhattanDistance(newPosition,ghost.getPosition()) +1)
112                     if(manhattanDistance(newPosition,ghost.getPosition()) == 0):
113                         return float('inf')
114
115         if(currentGameState.getNumFood() > successorGameState.getNumFood()):
116             evaluation += 1200
117         evaluation += 500/(min(foodDistances) + 1) + 600 / (sum(foodDistances) + 1)
118         return evaluation
119
```

Explanation

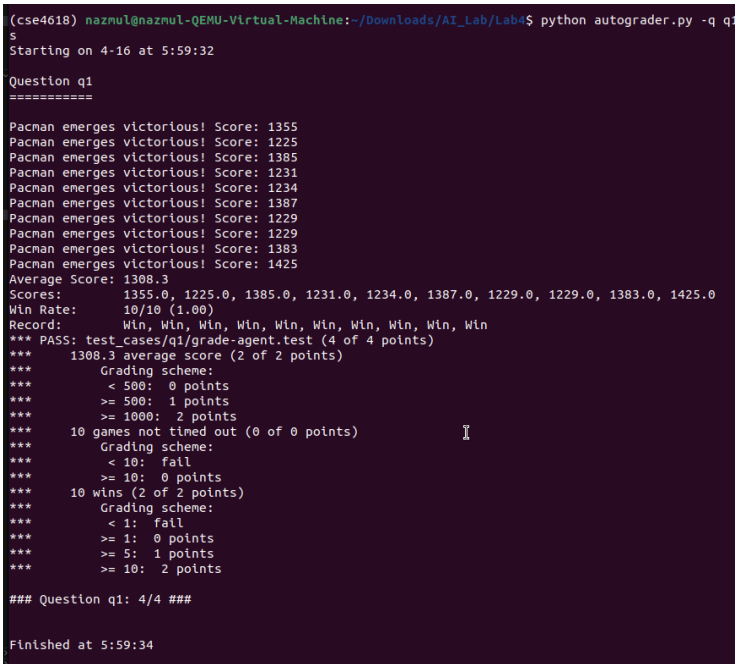
The solution approach that was taken evaluates the action poorly, punishing it heavily, in cases where Pacman's position doesn't change upon taking the new action, in case the chosen action causes pacman to lose the game, or if the distance between pacman and the ghost is 1 unit. It evaluates the action as the best action to take, rewarding it greatly if the chosen action will lead to winning the game and if the scaredTimer for a ghost is active and the chosen action will lead to eating the ghost.

After analyzing the ghost states, if the scaredTimer isn't active for a ghost, then the action taken to head towards a capsule is rewarded to encourage eating the capsules. Once the scaredTimer is active, then the action taken to head towards the scaredGhost is rewarded greatly so that pac man may eat the ghost.

In all cases, going closer to a food pellet is rewarded using the distance towards the closest food pellet. And reducing the number of the food pellets is encouraged using the sum of the distances between all the food pellets. Taking the advice provided in the question, the reciprocal values and rewarding with a flat weight. Between subsequent states, if a food pellet is eaten, it is rewarded with a flat amount, to further encourage eating food pellets. This ensures that the action that leads to eating the food pellets is rewarded.

Findings

After becoming familiar with the game of pacman, it was determined that the goals for the game was trying to eat all the food pellets, and along the way trying to eat as many capsules and then eating the scared ghost along the way. Eating the scared ghost was rewarded with a fairly good reward. So, the greatest score could be achieved by eating all the capsules, and eating all the ghosts for each capsule, and eating all the food pellets in the shortest amount of time while avoiding death from colliding with a ghost. The solution approach was designed in an attempt to meet those requirements.



Since eating a scared ghost rewards a fairly good amount, after considering it, the average score rose up a fair amount. The final average score of the solution designed for the appropriate test cases of this problem is found to be 1308.3. It tries to eat the capsule when the ghost isn't scared, and tries to eat the ghost once it is scared. When there is a capsule remaining on screen, for some reason, pacman decides to mimic the actions of the ghost. So, the requirement of trying to eat all the pellets in the shortest time possible wasn't met; however, the best possible solution after modifying all the possible hyperparameters was as such:

The agent doesn't time out and wins all 10 times, getting higher than 1000 in all 10 test cases, averaging around 1308.

In some previous cases, pacman would just stand still for no reason, which isn't very exciting or fun. For that reason, standing still between cases has been penalized in the evaluation metric heavily.

This solution fails in some very rare situations when there are walls. This can be found when running the reflex agent in the mediumClassic layout where there is a path that can block pacman's movement.

If the closest food pellet is on the other side of a wall, the fastest way to reach the food pellet needs pacman to go away from the food pellet and towards the food. Since we are using manhattanDistance, and not some other system that can detect the walls in the gameState, it fails to do so.

In our previous labs, there was a function called mazeDistance, that does detect the walls in the gameState, using a modification of bfs. Using this, the problem of pacman being trapped on the other side of the wall is avoided, however, this has its drawbacks. The function mazeDistance is very costly, which leads to a quite high average running time. Also, since the provided test cases do not have any walls, the score achieved by using mazeDistance and manhattanDistance to find the foodDistances turn out to be the same.

Initially, avoiding the non scared ghosts was a priority, so the minimum allowed distance was set to 5. However, the test classic layout is very congested. This makes it very difficult for pacman to maintain a distance of 5 units. It also brought on the realization that, coming closer to a ghost shouldn't be avoided. Just colliding with the ghost should have negative consequences. So, the minimum distance was set at 1. And that led to better performance in the testClassic layout. In the medium layout, this solution approach leads to a fairly good result, very rarely failing to win.

Challenges

There were a few self imposed challenges, since I was very excited about this lab task specifically. I wanted to implement logic which encouraged eating the scared ghosts. At first, I had to eat a capsule and then target the scared ghost. This felt like a daunting task at first. After reading the game files, I noticed that there were several helpful functions like capsule locations. Ghost states and positions etc. Using that I was able to implement the logic where if a ghost is not scared at first, then it rewards chasing the capsules, and once the scaredTimer is active, chasing the ghost is also evaluated highly.

I wanted to incorporate logic for maximizing the score by achieving more than 1000 in all the provided test cases. I was finally able to achieve that after implementing the logic for eating the ghosts properly.

I attempted to solve why pacman was mirroring the ghosts actions sometimes instead of just eating the pellets. I noticed that in my evaluation function, I provided a reward for going towards a pellet if pacman was near a ghost. This led to pacman wanting to stay near a ghost which makes it seem like pacman is mirroring the ghosts actions. I wanted to keep the capsule eating logic, so I simply disabled that increment in the evaluation once there were no more capsules left on screen.

There were several pieces of logic that had to be added to make sure that the evaluation function performed well in the medium and test classic layout like, not standing still, using mazeDistance, reducing lowest allowed ghost distance etc.

Additional Thoughts

Finding evaluation functions that get the job done is easy. But optimizing it to ensure it acts a certain way is quite difficult. Even if the evaluation works really well in some cases, it is never guaranteed to lead to a win. It is difficult to consider all the different variables in the gameState so implementing the perfect evaluation function is quite difficult.

Problem-2 (Minimax)

Analysis

The second task required us to define a general adversarial search agent that can handle any number of ghosts to allow pacman to calculate the evaluation of state. There will be a min layer to predict the scores ghosts are trying to achieve and pac man will have the max layer to maximize his possible score in each layer. The layers should be called recursively to evaluate a state

Solution

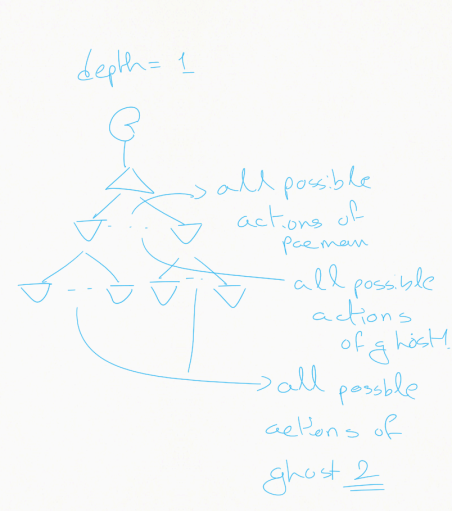
```
182  class MinimaxAgent(MultiAgentSearchAgent):
187      def getAction(self, gameState):
211          # util.raiseNotDefined()
212          _, action = self.value(gameState, 0, self.depth)
213          return action
214
215      def value(self, gameState, agentIndex, depth):
216          if gameState.isWin() or gameState.isLose() or depth == 0:
217              return self.evaluationFunction(gameState), Directions.STOP
218          elif agentIndex == 0:
219              return self.maxValue(gameState, agentIndex, depth)
220          elif agentIndex > 0:
221              return self.minValue(gameState, agentIndex, depth)
222
223      def minValue(self, gameState, agentIndex, depth):
224          currValue, currAction = 1e9, None
225
226          nextAgent = (agentIndex + 1) % gameState.getNumAgents()
227          if nextAgent == 0:
228              nextDepth = depth - 1
229          else:
230              nextDepth = depth
231
232          legalActions = gameState.getLegalActions(agentIndex)
233
234          for action in legalActions:
235              successorGameState = gameState.generateSuccessor(agentIndex, action)
236              successorScore, _ = self.value(successorGameState, nextAgent, nextDepth)
237
238              # currValue = min(currValue, successorScore)
239              if successorScore < currValue:
240                  currValue = successorScore
241                  currAction = action
242
243          return currValue, currAction
244
245      def maxValue(self, gameState, agentIndex, depth):
246          currValue, currAction = -1e9, None
247
248          nextAgent = (agentIndex + 1) % gameState.getNumAgents()
249          if nextAgent == 0:
250              nextDepth = depth - 1
251          else:
252              nextDepth = depth
253
254          legalActions = gameState.getLegalActions(agentIndex)
255
256          for action in legalActions:
257              successorGameState = gameState.generateSuccessor(agentIndex, action)
258
259              successorScore, _ = self.value(successorGameState, nextAgent, nextDepth)
260
261              # currValue = min(currValue, successorScore)
262              if successorScore > currValue:
263                  currValue = successorScore
264                  currAction = action
265
266          return currValue, currAction
```

Explanation

The code was provided by sir, so it shall not be discussed in depth. The minimizer and maximizer layers are used by pacman to determine the optimal move from his position. The ghosts are attached with the minimizer layers so, it is assumed that they'll make their optimal moves and Pacman attaches with himself the maximizer layer to obtain the highest score. This continues recursively until a goal is reached or the depth reaches its limit. If it wins, loses or if a depth limit is reached, its value is returned along the recursive chain to pacman. Pacman then uses the evaluation value to determine his own move. The minimizer and maximizer layers are declared in general so it can be applied by ghosts or pacman as needed. After calculating the recursive layers, the ghosts' minimizer layer always attempts to select the score which will be lower, and the maximizer in pacman selects the highest evaluation score out of the predicted values for each state.

Findings

This approach assumes that the adversaries will always make the optimal move, without any randomness. Also a simulation is run according to the set depth each time pac man makes his decision. The ghosts may or may not be moving optimally, so assuming they will move optimally each time may seem like a way to always remain safe, but that is seen to be otherwise.



The simulation for each step includes all possible legalActions of all the possible agents at each depth. This leads to a very computationally expensive process. When I attempted to set the depth for more than 6, my laptop froze. Even setting the depth to 4 makes pacman take its decision quite slowly.

The effects of depth can be seen in some cases. When pacman has cleared all nearby food pellets and there are no ghosts around him, he sometimes decides to stop moving and just wait around. Because he is getting rewarded by the score as he is, and in the nearby depth predicted future, there aren't any food pellets. So, he doesn't need to move around. But once a ghost comes close to him, he starts attempting to avoid the ghost and thus, moves since it sees a loss in its depth predicted future.

In case of the trappedClassic. The orange ghost is next to pacman which is bad. There is a blue ghost 3 unit in front of pacman. And there is a negative living reward. In this case, if we applied a depth of 3 for the minimax agent, pacman kills himself because that is the way it predicts will get the maximum possible rewards. It assumes the ghosts will act optimally.

However, if we were to use a depth 1 or depth 2, pacman will only see the orange ghost in its path, and try to move away from it. The blue ghost will be too far away for pacman. After the first turn, the ghosts move, and in trappedClassic, the ghosts are moving randomly. If the ghost moves up, pacman once again is trapped and decides to kill himself. If the ghost moves down however, Then pacman once again moves down and closer to the pellets. In cases of using depth 1 and depth 2, if the blue ghost moves down, we can see pacman win in some cases, whereas in case of depth 3 or higher, it always assumes the ghosts will make the optimal move and kills itself.

So, It is noticed that this doesn't lead to pacman always winning, the leading cause is the limited assigned depth, and in situations where pacman is trapped, it will try to lose with the highest amount of points.

Challenges

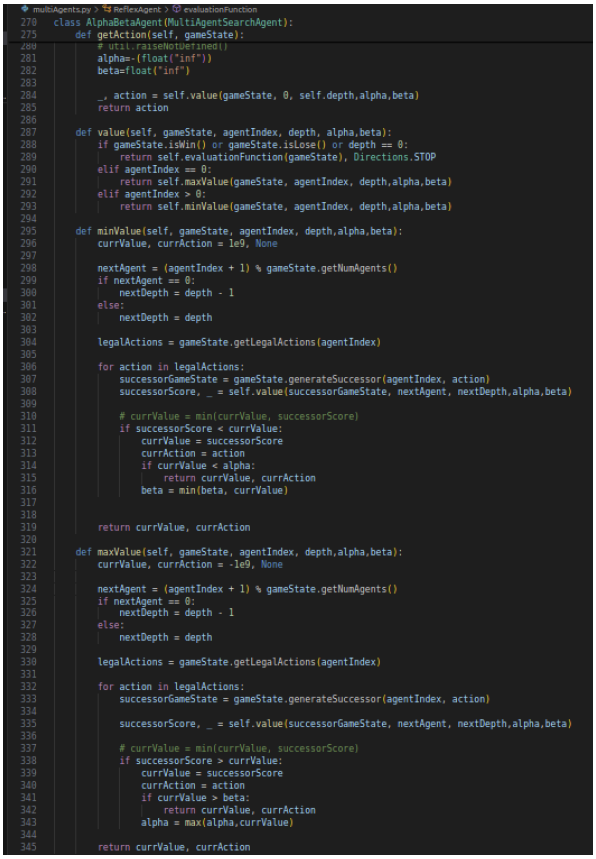
I had faced no challenges while attempting this problem, however, I am writing this quite late which is why organizing my thoughts about this is a bit challenging.

Problem-3 (Alpha-Beta Pruning)

Analysis

The third problem requires us to optimize the minimax agent by alpha beta pruning to efficiently explore the minimax tree and allow pacman to calculate the evaluation of state efficiently.

Solution



```
multiAgents.py > ReflexAgent > evaluationFunction
class AlphaBetaAgent(MultiAgentSearchAgent):
    def getAction(self, gameState):
        # util.raiseNotDefined()
        alpha=-(float("inf"))
        beta=float("inf")

        _, action = self.value(gameState, 0, self.depth,alpha,beta)
        return action

    def value(self, gameState, agentIndex, depth, alpha,beta):
        if gameState.isWin() or gameState.isLose() or depth == 0:
            return self.evaluationFunction(gameState), Directions.STOP
        elif agentIndex == 0:
            return self.maxValue(gameState, agentIndex, depth,alpha,beta)
        elif agentIndex > 0:
            return self.minValue(gameState, agentIndex, depth,alpha,beta)

    def minValue(self, gameState, agentIndex, depth,alpha,beta):
        currValue, currAction = 1e9, None

        nextAgent = (agentIndex + 1) % gameState.getNumAgents()
        if nextAgent == 0:
            nextDepth = depth - 1
        else:
            nextDepth = depth

        legalActions = gameState.getLegalActions(agentIndex)

        for action in legalActions:
            successorGameState = gameState.generateSuccessor(agentIndex, action)
            successorScore, _ = self.value(successorGameState, nextAgent, nextDepth,alpha,beta)

            # currValue = min(currValue, successorScore)
            if successorScore < currValue:
                currValue = successorScore
                currAction = action
                if currValue < alpha:
                    return currValue, currAction
                beta = min(beta, currValue)


        return currValue, currAction

    def maxValue(self, gameState, agentIndex, depth,alpha,beta):
        currValue, currAction = -1e9, None

        nextAgent = (agentIndex + 1) % gameState.getNumAgents()
        if nextAgent == 0:
            nextDepth = depth - 1
        else:
            nextDepth = depth

        legalActions = gameState.getLegalActions(agentIndex)

        for action in legalActions:
            successorGameState = gameState.generateSuccessor(agentIndex, action)

            successorScore, _ = self.value(successorGameState, nextAgent, nextDepth,alpha,beta)

            # currValue = min(currValue, successorScore)
            if successorScore > currValue:
                currValue = successorScore
                currAction = action
                if currValue > beta:
                    return currValue, currAction
                alpha = max(alpha,currValue)

        return currValue, currAction
```

Explanation

The pseudo code was provided in the task book, the goal was to just generalize it, so that we may be able to incorporate as my agents to each layer as we want.

As stated, pruning wasn't done on equality. Following the provided pseudo code the solution was written after modifying it slightly for generalization the inspiration for which was taken from task 2. After calculating the recursive layers, the ghosts' minimizer layer always

attempts to select the score which will be lower, and the maximizer in pacman selects the highest evaluation score out of the predicted values for each state.

Findings

It is found that alpha beta prunings result exactly the same as minimax, but this time is much more computationally less expensive. It runs comparatively smoother than the minimax agent even at depth 5, since this time around a lot less calculation is needed to be carried out.

The effects of depth can be seen in some cases. When pacman has cleared all nearby food pellets and there are no ghosts around him, he sometimes decides to stop moving and just wait around. Because he is getting rewarded by the score as he is, and in the nearby depth predicted future, there aren't any food pellets. So, he doesn't need to move around. But once a ghost comes close to him, he starts attempting to avoid the ghost and thus, moves since it sees a loss in its depth predicted future.

In case of the trappedClassic. The orange ghost is next to pacman which is bad. There is a blue ghost 3 unit in front of pacman. And there is a negative living reward. In this case, if we applied a depth of 3 even in the alpha-beta pruning agent, pacman kills himself because that is the way it predicts will get the maximum possible rewards. It assumes the ghosts will act optimally.

In cases of using depth 1 and depth 2, if the blue ghost moves down, we can see pacman win in some cases, whereas in case of depth 3 or higher, it always assumes the ghosts will make the optimal move and kills itself which is exactly the same we saw in minimax agent.

So, It is noticed that this doesn't lead to pacman always winning, the leading cause is the limited assigned depth, and in situations where pacman is trapped, it will try to lose with the highest amount of points.

Challenges

I had faced no challenges while attempting this problem since the code was provided by the taskbook, however, I am writing this quite late which is why organizing my thoughts about this is a bit challenging.

Problem-4 (Expectimax)

Analysis

The fourth problem required the definition of an Expectimax agent, where pac man himself dawns a maximizer layer, while predicting the moves of the adversarial ghosts with an expectimax layer, which predicts the move of the adversarial ghosts to calculate the evaluation of the state. An expectimax layer will replace each minimizer layer for each ghost in this general equation to allow pacman to calculate the evaluation of state.

Solution

```
348  class ExpectimaxAgent(MultiAgentSearchAgent):
353      def getAction(self, gameState):
354          """
355          Returns the expectimax action using self.depth and self.evaluationFunction
356
357          All ghosts should be modeled as choosing uniformly at random from their
358          legal moves.
359          """
360          "*** YOUR CODE HERE ***"
361          # util.raiseNotDefined()
362          _, action = self.value(gameState, 0, self.depth)
363          return action
364
365      def value(self, gameState, agentIndex, depth):
366          if gameState.isWin() or gameState.isLose() or depth == 0:
367              return self.evaluationFunction(gameState), Directions.STOP
368          elif agentIndex == 0:
369              return self.maxValue(gameState, agentIndex, depth)
370          elif agentIndex > 0:
371              return self.expValue(gameState, agentIndex, depth)
372
373      def expValue(self, gameState, agentIndex, depth):
374          currValue, currAction = 0, None
375          prob =  1/len(gameState.getLegalActions(agentIndex))
376          nextAgent = (agentIndex + 1) % gameState.getNumAgents()
377          if nextAgent == 0:
378              nextDepth = depth - 1
379          else:
380              nextDepth = depth
381
382          legalActions = gameState.getLegalActions(agentIndex)
383
384          for action in legalActions:
385              successorGameState = gameState.generateSuccessor(agentIndex, action)
386              successorScore, _ = self.value(successorGameState, nextAgent, nextDepth)
387
388              # currValue = min(currValue, successorScore)
389              # if successorScore < currValue:
390              currValue += prob * successorScore
391              currAction = action
392
393          return currValue, currAction
394
395      def maxValue(self, gameState, agentIndex, depth):
396          currValue, currAction = -1e9, None
397
398          nextAgent = (agentIndex + 1) % gameState.getNumAgents()
399          if nextAgent == 0:
400              nextDepth = depth - 1
401          else:
402              nextDepth = depth
403
404          legalActions = gameState.getLegalActions(agentIndex)
405
406          for action in legalActions:
407              successorGameState = gameState.generateSuccessor(agentIndex, action)
408
409              successorScore, _ = self.value(successorGameState, nextAgent, nextDepth)
410
411              # currValue = min(currValue, successorScore)
412              if successorScore > currValue:
413                  currValue = successorScore
414                  currAction = action
415
416          return currValue, currAction
417
```

Explanation

Once again, the pseudocode for this problem was provided in the taskbook, so it will not be discussed in depth. The probability of each action being taken was taken to be 1/(number of possible legal moves). The valuation for a state is multiplied by the probability of it taking place. Pacman, the agent index 0 once again uses a maximizer layer to maximize its scores. After calculating the recursive layers, the maximizer in pacman selects the highest evaluation score out of the predicted expected values for each state.

Findings

Unlike minimax and alpha beta pruning, in case of the trappedClassic, expectimax doesn't cause pacman to kill himself because of the depth being equal to or higher than 3. It considers the possibilities of the ghost taking random or noisy actions, instead of always being optimal, and it takes its actions based on that. It is nowhere near as safe as minimax, however, it tries to get as many pellets as possible and considers the suboptimal actions of the ghost as well. And the ghosts in the trappedClassic are moving randomly, not taking the optimal moves at all times. That's why Expectimax wins sometimes but Alpha beta pruning always looses at depth = 3.

Challenges

I had faced no challenges while attempting this problem since the code was provided by the taskbook, however, I am writing this quite late which is why organizing my thoughts about this is a bit challenging.

Problem-5 (Evaluation Function)

Analysis

The final task required the definition of an evaluation function that will be evaluating the states themselves instead of the state-action pair as in the reflex agent. Essentially, the idea is that using the evaluation function in expectimax, the expected values of each state will be calculated for the possible outcomes until a certain depth, and the course of action that may lead to the best possible outcome will be chosen.

Solution

```
419 def betterEvaluationFunction(currentGameState):
420     """
421     Your extreme ghost-hunting, pellet-nabbing, food-gobbling, unstoppable
422     evaluation function (question 5).
423 
424     DESCRIPTION: <write something here so we know what you did>
425     """
426     "*** YOUR CODE HERE ***"
427     # util.raiseNotDefined()
428     currentPosition = currentGameState.getPacmanPosition()
429     score = currentGameState.getScore()
430     foodPositions = currentGameState.getFood().asList()
431     ghostStates = currentGameState.getGhostStates()
432     foodDistances = []
433     capsules = currentGameState.getCapsules()
434     capsuleDistances = []
435 
436     for capsule in capsules:
437         capsuleDistances.append(manhattanDistance(capsule,currentPosition))
438 
439     evaluation = score
440     for food in foodPositions:
441         # foodDistances.append(mazeDistance(food,currentPosition,currentGameState))
442         foodDistances.append(manhattanDistance(food,currentPosition))
443 
444     for ghost in ghostStates:
445         if(manhattanDistance(currentPosition, ghost.getPosition()) <= 50):
446             if (manhattanDistance(currentPosition, ghost.getPosition()) <= 10 and ghost.scaredTimer == 0):
447                 if (min(capsuleDistances, default=0) != 0):
448                     evaluation += 500 / (min(capsuleDistances, default=0))
449                 if manhattanDistance(currentPosition, ghost.getPosition()) <=1:
450                     return -float("inf")
451             elif (ghost.scaredTimer != 0):
452                 evaluation += 2000 / (manhattanDistance(currentPosition, ghost.getPosition()))
453 
454     evaluation -= -150/(min(foodDistances, default=0) + 1) + 60*(currentGameState.getNumFood())
455     return evaluation
456 
```

Explanation

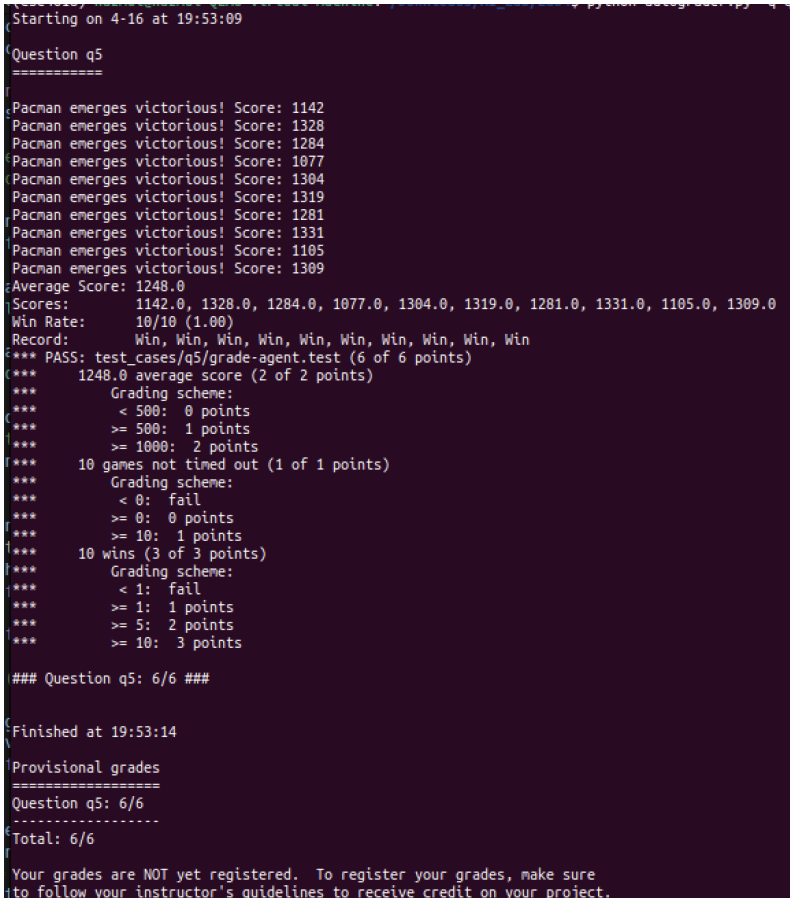
The approach is quite similar to the one taken to solve the first task. The logic for trying to eat the capsules when the ghosts aren't scared and trying to chase and finally eat the ghost remains almost unchanged with slight modifications. Here, eating the ghost doesn't need to be scored separately, since the score which is added as the baseline for evaluation handles that. Essentially, the logic for avoiding the ghost to survive when they're not scared and trying to maximize the score by eating them remains the same.

The differences between the two start showing up when considering the rest of the code. Since, the next move of pacman is unknown and won't be considered, pacman staying still can't be penalized. This leads to pacman standing still randomly sometimes which was unavoidable.

When considering reducing the number of food pellets, since the number of food in the successors couldn't be determined properly without the action, eating each pellet couldn't be appropriately rewarded. By penalizing the number of remaining pellets with a high flat weight, it was highly discouraged to keep a high number of pellets on screen. And in order to encourage going closer to the food pellets, the reciprocal weight of the minimum foodDistance was used.

Findings

Pacman is found to stay still in some situations. This approach ended up being quite heavily dependent on the position of the ghosts to encourage pacman to start moving. However, it works.



The solution agent doesn't time out in any case, and wins all 10 times, getting more than 1000 in all of them, averaging about 1248.

There are walls in this approach, so, applying mazeDistance would be recommended, but after using manhattanDistance and running the simulation quite a few times, it was determined that in the wall layout provided in the test cases, pacman doesn't get stuck anywhere. So, sticking with manhattanDistance to avoid the high computational complexity that comes with mazeDistance seemed like the more reasonable decision since both perform quite similarly.

Pacman does prioritize chasing the scaredGhost over eating the food pellets. Other than that however, pacman tries to reduce the number of food pellets. It isn't as fast as possible however, since, at times pacman does decide to just remain still. This action of staying still

couldn’t be properly discouraged as it requires a feature that hasn’t been considered or requires a modification of the weights.

In this approach, the win condition, or the loose condition, even though it wasn’t explicitly considered, since, once the game is won, it's won, and lost is lost. So, they didn’t have to be considered when evaluating just the state.

Challenges

This approach required significant sophistication for the logic in eating the food pellets. This approach isn’t perfect, and works well for the provided test cases. In some other random cases it might not hold up as well.

Similar to task 1, I had a few self imposed challenges like, it has to perform well in all the provided test cases, getting above a 1000 in all of them, and I wanted to implement the logic for chasing the scaredGhost and eating the capsules, which seemed daunting at the time. Both of these were achieved. However, the logic to eat the food pellets in the shortest possible amount of time remains unsatisfactory.

When designing the logic for reducing the food pellets, and trying to use the logic used in task 1, it was found that the ghost preferred to just eat as many pellets as it wanted and then just stand still close to the pellets without eating them. This has been changed after modifying some of the weights, significantly reducing the reward for going closer to a pellet, and penalizing highly for having a high number of pellets.

Additional Thoughts

After trying really hard to implement the perfect logic to make sure pacman doesn’t stay still, eats all the pellets in the shortest amount of time and doesn’t rely on the ghosts for its own movements, rather try to eat the pellet on its own during that time, I was unable to ensure all those requirements without the next actions. If we were using expectimax to determine the next moves, the states will also have an action assigned to them in each case. I am unsure why we

didn’t consider a state-action pair here as well, which would help me solve those requirements as well.