ISLAMIC UNIVERSITY OF TECHNOLOGY

Organization of Islamic Cooperation

Board Bazar, Gazipur

Lab Report 4

CSE 4712

Md. Bakhtiar Hasan

30th October, 2022

## Question 01

The first problem involved designing an evaluation metric. The metric is a simple one which only considers the distance from the current position to the closest food. The reciprocal of the minimum distance is taken so that a state which is closer to a food pellet gets a higher score than one which is further away.

foodList = currentGameState.getFood().asList()  
closestFood = min([util.manhattanDistance(newPos, food) for food in foodList])  
  
return 1/(closestFood + 1)

PYTHON

In addition to the basic metric as outlined above, there was one additional situation that required special attention. If any ghost is at the same position as Pacman and is also not in the scared state, then we need to assign a large negative score so that Pacman does not get eaten.

foodList = currentGameState.getFood().asList()  
closestFood = min([util.manhattanDistance(newPos, food) for food in foodList])  
  
for ghost in newGhostStates:  
 if ghost.scaredTimer == 0 and ghost.getPosition() == newPos:  
 return -9999999  
  
return 1/(closestFood + 1)

PYTHON

## Question 02

The second problem required an implementation for Minimax search. The solution for this problem was provided, but the code is still being examined here.

There are three parts to the solution, with each part being its own method. The initial function call takes us to a method, value, which takes three arguments, the current game state, the agent index and the depth. The agent index refers to the ID of the agent who is making the current move, which is 0 to start with. The depth is the maximum depth up to which the search should continue.

def getAction(*self*, gameState):  
 return *self*.value(gameState, 0, *self*.depth)[1]  
  
def value(*self*, gameState, agentIndex, depth):  
 if depth == 0 or gameState.isWin() or gameState.isLose():  
 return *self*.evaluationFunction(gameState), Directions.STOP  
 elif agentIndex == 0:  
 return *self*.maxValue(gameState, agentIndex, depth)  
 else:  
 return *self*.minValue(gameState, agentIndex, depth)

PYTHON

There are three possible values that can be returned. If the depth is 0, meaning we have reached the end of our search limit, or if we have reached a state that is a win state or a lose state, we return the output from the evaluation function. The evaluation function is defined separately and is not being examined here, but for non-terminal states, the function returns an estimate of the optimal value from that state.

The other two possible values are the maximum value that can be achieved from this state or the minimum value that can be achieved from this state. Which of these is used depends on the index of the agent. For an agentIndex of 0, which means it is currently Pacman’s turn, the maximum value is returned. Otherwise, it is a ghost’s turn, so the minimum value is returned.

def maxValue(*self*, gameState, agentIndex, depth):  
 nextAgent = (agentIndex + 1) % gameState.getNumAgents()  
 if nextAgent == 0:  
 nextDepth = depth - 1  
 else:  
 nextDepth = depth  
   
 selectedScore, selectedAction = -1e9, Directions.STOP  
 actionList = gameState.getLegalActions(agentIndex)  
 for action in actionList:  
 successorState = gameState.generateSuccessor(agentIndex, action)  
 successorScore = *self*.value(successorState, nextAgent, nextDepth)[0]  
 if selectedScore < successorScore:  
 selectedScore = successorScore  
 selectedAction = action  
 return selectedScore, selectedAction

PYTHON

To obtain the maximum value possible, we go over the list of possible actions and examine the score obtained by the successor state for each of those actions. This score is obtained via a recursive call to the value method. The action which gives us the highest score is returned by the method.

In addition, the method will also get the next agent’s ID (to pass to the value method) as well as decrement the depth if the next agent happens to be Pacman (with ID 0).

Obtaining the minimum value for a state follows a similar process, except that we return the successor state for the action which gives us the least possible value.

def minValue(*self*, gameState, agentIndex, depth):  
 nextAgent = (agentIndex + 1) % gameState.getNumAgents()  
 if nextAgent == 0:  
 nextDepth = depth - 1  
 else:  
 nextDepth = depth

selectedScore, selectedAction = 1e9, Directions.STOP  
 actionList = gameState.getLegalActions(agentIndex)  
 for action in actionList:  
 successorState = gameState.generateSuccessor(agentIndex, action)  
 successorScore = *self*.value(successorState, nextAgent, nextDepth)[0]  
 if selectedScore > successorScore:  
 selectedScore = successorScore  
 selectedAction = action  
 return selectedScore, selectedAction

PYTHON

## Question 03

The third problem modifies the code for Minimax search to include alpha-beta pruning. This requires a few minor changes to the above code. Thus, the code shown in this section is mostly greyed out, with only the changed portions highlighted.

The first change involves passing an and a value to the value method, which is in turn passed to the maxValue and minValue methods. These values are initialized to and respectively.

def getAction(*self*, gameState):  
 return *self*.value(gameState, 0, *self*.depth, -1e9, 1e9)[1]  
  
def value(*self*, gameState, agentIndex, depth, alpha, beta):  
 if depth == 0 or gameState.isWin() or gameState.isLose():  
 return *self*.evaluationFunction(gameState), Directions.STOP  
 elif agentIndex == 0:  
 return *self*.maxValue(gameState, agentIndex, depth, alpha, beta)  
 else:  
 return *self*.minValue(gameState, agentIndex, depth, alpha, beta)

PYTHON

While searching over the successors for the possible actions, the maxValue method passes the highest value it has found so far to the successor nodes as the value. The next node will be a minimizer, which will stop searching its list of successors if it encounters a value that is lower than the value. This is because if the current node has already found a lower value, then the maximiser above it will not use this node in any case. Searching for even lower values is pointless.

A similar situation occurs inside the maxValue method. The minValue method passes the lowest value it has found so far to its successors as the value. The next node, which is a maximiser, stops searching its list of successors if it encounters a value that is higher than the value, since the minimizer above it will not use this node anyways.

def minValue(*self*, gameState, agentIndex, depth, alpha, beta):  
 nextAgent = (agentIndex + 1) % gameState.getNumAgents()  
 if nextAgent == 0:  
 nextDepth = depth - 1  
 else:  
 nextDepth = depth  
   
 selectedScore, selectedAction = 1e9, Directions.STOP  
 actionList = gameState.getLegalActions(agentIndex)

for action in actionList:  
 successorState = gameState.generateSuccessor(agentIndex, action)  
 successorScore = *self*.value(successorState, nextAgent, nextDepth, alpha, beta)[0]  
 if selectedScore > successorScore:  
 selectedScore = successorScore  
 selectedAction = action  
 if selectedScore < alpha:  
 return selectedScore, selectedAction  
 beta = min(beta, selectedScore)  
 return selectedScore, selectedAction  
  
def maxValue(*self*, gameState, agentIndex, depth, alpha, beta):  
 nextAgent = (agentIndex + 1) % gameState.getNumAgents()  
 if nextAgent == 0:  
 nextDepth = depth - 1  
 else:  
 nextDepth = depth  
   
 selectedScore, selectedAction = -1e9, Directions.STOP  
 actionList = gameState.getLegalActions(agentIndex)  
 for action in actionList:  
 successorState = gameState.generateSuccessor(agentIndex, action)  
 successorScore = *self*.value(successorState, nextAgent, nextDepth, alpha, beta)[0]  
 if selectedScore < successorScore:  
 selectedScore = successorScore  
 selectedAction = action  
   
 if selectedScore > beta:  
 return selectedScore, selectedAction  
   
 alpha = max(alpha, selectedScore)  
 return selectedScore, selectedAction

PYTHON

## Question 04

The fourth problem is again, a modification of Minimax search. This time however, the only change is that instead of a minimizer node, we have a chance node. Thus, the minValue method is replaced with an expValue method.

def expValue(*self*, gameState, agentIndex, depth):  
 nextAgent = (agentIndex + 1) % gameState.getNumAgents()  
 if nextAgent == 0:  
 nextDepth = depth - 1  
 else:  
 nextDepth = depth  
   
 selectedScore, selectedAction = 0, Directions.STOP  
 actionList = gameState.getLegalActions(agentIndex)

for action in actionList:  
 successorState = gameState.generateSuccessor(agentIndex, action)  
 successorScore = *self*.value(successorState, nextAgent, nextDepth)[0]  
 p = 1/len(actionList)  
 selectedScore = selectedScore + (p \* successorScore)  
 return selectedScore, random.choice(actionList)

PYTHON

This method returns the weighted average of the scores of all the successors for each possible action. The weight is the probability of that action occurring. The probabilities were assumed to be uniformly distributed. The action that was selected is random.

## Question 05

The last problem was similar to the first one in that we needed to create an evaluation metric. However, the evaluation metric in this case was required to be far more robust. A large amount of trial and error was required to pass all the test cases.

The initial intuition was to take an approach similar to the one taken in the first solution, considering the distance to the closest food pellet and whether Pacman was at the same position as a non-scared ghost (meaning it would get eaten).

def betterEvaluationFunction(currentGameState):   
 position = currentGameState.getPacmanPosition()  
  
 ghosts = [ghost for ghost in currentGameState.getGhostStates() if ghost.scaredTimer != 0]  
 for ghost in ghosts:  
 if ghost.scaredTimer == 0 and ghost.getPosition() == position:  
 return -9999999

foodList = currentGameState.getFood().asList()  
 foodDists = [util.manhattanDistance(position, food) for food in foodList]  
 closestFood = min(foodDists) if len(foodDists) > 0 else 0  
 return 1/(closestFood + 1)

PYTHON

This solution however, proved to be extremely inefficient, failing to win the games at all.

The next solution also considered the scared and non-scared ghosts. Points were deducted for getting closer to non-scared ghosts and points were added for getting closer to scared ghosts.

def betterEvaluationFunction(currentGameState):  
 score = 0  
 position = currentGameState.getPacmanPosition()  
  
 foodList = currentGameState.getFood().asList()  
 foodDists = [util.manhattanDistance(position, food) for food in foodList]  
 closestFood = min(foodDists) if len(foodDists) > 0 else 0  
 score += 1/(closestFood + 1)  
  
 ghosts = [ghost for ghost in currentGameState.getGhostStates() if ghost.scaredTimer != 0]  
 ghostDists = [util.manhattanDistance(position, ghost.getPosition()) for ghost in ghosts]  
 closestGhost = min(ghostDists) if len(ghostDists) > 0 else 0  
 score -= 1/(closestGhost + 1)  
  
 ghosts = [ghost for ghost in currentGameState.getGhostStates() if ghost.scaredTimer == 0]  
 ghostDists = [util.manhattanDistance(position, ghost.getPosition()) for ghost in ghosts]  
 closestGhost = min(ghostDists) if len(ghostDists) > 0 else 0  
 score += 1/(closestGhost + 1)  
  
 ghosts = [ghost for ghost in currentGameState.getGhostStates()]  
 for ghost in ghosts:  
 if ghost.scaredTimer == 0 and ghost.getPosition() == position:  
 return -9999999  
  
 return score

PYTHON

Even this failed to achieve the desired results. After a significant amount of trial and error, it was found that setting the initial score to the default score for the state allowed this algorithm to work.

def betterEvaluationFunction(currentGameState):  
 score = currentGameState.getScore()  
 position = currentGameState.getPacmanPosition()  
  
 foodList = currentGameState.getFood().asList()  
 foodDists = [util.manhattanDistance(position, food) for food in foodList]  
 closestFood = min(foodDists) if len(foodDists) > 0 else 0  
 score += 1/(closestFood + 1)  
  
 ghosts = [ghost for ghost in currentGameState.getGhostStates() if ghost.scaredTimer != 0]  
 ghostDists = [util.manhattanDistance(position, ghost.getPosition()) for ghost in ghosts]  
 closestGhost = min(ghostDists) if len(ghostDists) > 0 else 0  
 score -= 1/(closestGhost + 1)  
  
 ghosts = [ghost for ghost in currentGameState.getGhostStates() if ghost.scaredTimer == 0]  
 ghostDists = [util.manhattanDistance(position, ghost.getPosition()) for ghost in ghosts]  
 closestGhost = min(ghostDists) if len(ghostDists) > 0 else 0  
 score += 1/(closestGhost + 1)  
  
 ghosts = [ghost for ghost in currentGameState.getGhostStates()]  
 for ghost in ghosts:  
 if ghost.scaredTimer == 0 and ghost.getPosition() == position:  
 return -9999999  
  
 return score

PYTHON