

# Homework 3: Multi-Agent Search

## Part I. Implementation (5%):

- Part 1 MiniMax

```
● ● ●

1  class MinimaxAgent(MultiAgentSearchAgent):
2      """
3          Your minimax agent (Part 1)
4      """
5
6      def getAction(self, gameState):
7          """
8              Returns the minimax action from the current gameState using self.depth
9              and self.evaluationFunction.
10             """
11            # Begin your code (Part 1)
12            # raise NotImplemented("To be implemented")
13            __, nextAction = self.performMinimax(0, self.index, gameState)
14            return nextAction
15
16      def performMinimax(self, depth, agentIndex, gameState):
17          """
18              # Return the score if the game is over or the depth is reached
19              if (gameState.isWin() or gameState.isLose() or depth >= self.depth):
20                  return self.evaluationFunction(gameState), None
21
22          # Init variables and lists
23          scores = []
24          isMinplayer = True if agentIndex != 0 else False
25          bestScore = float("inf") if isMinplayer else float("-inf")
26          legalMoves = gameState.getLegalActions(agentIndex)
27          nextIndex = (agentIndex + 1) % gameState.getNumAgents() # 0 -> 1 -> 2 -> 0 -> 1 -> 2 -> ...
28
29          # Increase depth if the next agent is pacman
30          if nextIndex == 0:
31              depth +=1
32
33          # Remove the stop action to avoid some extreme cases
34          if Directions.STOP in legalMoves:
35              legalMoves.remove(Directions.STOP)
36
37          # Perform minimax
38          for move in legalMoves:
39              nextState = gameState.getNextState(agentIndex, move)
40              nextValue, __ = self.performMinimax(depth, nextIndex, nextState)
41              bestScore = min(bestScore, nextValue) if isMinplayer else max(bestScore, nextValue)
42              scores.append(nextValue)
43
44          # Return the best score and the next action
45          if isMinplayer:
46              return bestScore, None
47          else:
48              bestIndices = [index for index in range(len(scores)) if scores[index] == bestScore]
49              choisenIndex = random.choice(bestIndices)
50              return max(scores), legalMoves[choisenIndex]
51
52          # End your code (Part 1)
```

## ● Part 2 Alpha-Beta

```
1  class AlphaBetaAgent(MultiAgentSearchAgent):
2      """
3          Your minimax agent with alpha-beta pruning (Part 2)
4      """
5
6      def getAction(self, gameState):
7          """
8              Returns the minimax action using self.depth and self.evaluationFunction
9          """
10         # Begin your code (Part 2)
11         # raise NotImplemented("To be implemented")
12         alpha, beta = float("-inf"), float("inf")
13         _, nextAction = self.performAlphaBeta(0, self.index, gameState, alpha, beta)
14         return nextAction
15
16     def performAlphaBeta(self, depth, agentIndex, gameState, alpha, beta):
17         # Return the score if the game is over or the depth is reached
18         if (gameState.isWin() or gameState.isLose() or depth >= self.depth):
19             return self.evaluationFunction(gameState), None
20
21         # Init variables and lists
22         scores = []
23         isMinplayer = True if agentIndex != 0 else False
24         bestScore = float("inf") if isMinplayer else float("-inf")
25         legalMoves = gameState.getLegalActions(agentIndex)
26         nextIndex = (agentIndex + 1) % gameState.getNumAgents() # 0 -> 1 -> 2 -> 0 -> 1 -> 2 -> ...
27
28         # Increase depth if the next agent is pacman
29         if nextIndex == 0:
30             depth +=1
31
32         # Remove the stop action to avoid some extreme cases
33         if Directions.STOP in legalMoves:
34             legalMoves.remove(Directions.STOP)
35
36         # Perform alpha-beta pruning
37         for move in legalMoves:
38             nextState = gameState.getNextState(agentIndex, move)
39             nextValue, __ = self.performAlphaBeta(depth, nextIndex, nextState, alpha, beta)
40             bestScore = min(bestScore, nextValue) if isMinplayer else max(bestScore, nextValue)
41
42             # Pruning
43             if isMinplayer :
44                 if nextValue < alpha:
45                     return nextValue, None
46                 beta = min(beta, bestScore)
47             else:
48                 if nextValue > beta:
49                     return nextValue, move
50                 alpha = max(alpha, nextValue)
51             scores.append(nextValue)
52
53         # Return the best score and the next action
54         if isMinplayer:
55             return bestScore, None
56         else:
57             bestIndices = [index for index in range(len(scores)) if scores[index] == bestScore]
58             choisenIndex = random.choice(bestIndices)
59             return max(scores), legalMoves[choisenIndex]
60         # End your code (Part 2)
```

## ● Part3 Expectimax

```
● ● ●
1 class ExpectimaxAgent(MultiAgentSearchAgent):
2     """
3         Your expectimax agent (Part 3)
4     """
5
6     def getAction(self, gameState):
7         """
8             Returns the expectimax action using self.depth and self.evaluationFunction
9
10            All ghosts should be modeled as choosing uniformly at random from their
11            legal moves.
12        """
13        # Begin your code (Part 3)
14        # raise NotImplemented("To be implemented")
15        __, nextAction = self.performExpectimax(0, self.index, gameState)
16        return nextAction
17
18    def performExpectimax(self, depth, agentIndex, gameState):
19        # Return the score if the game is over or the depth is reached
20        if (gameState.isWin() or gameState.isLose() or depth >= self.depth):
21            return self.evaluationFunction(gameState), None
22
23        # Init variables and lists
24        scores = []
25        isMinplayer = True if agentIndex != 0 else False
26        bestScore = float("inf") if isMinplayer else float("-inf")
27        legalMoves = gameState.getLegalActions(agentIndex)
28        nextIndex = (agentIndex + 1) % gameState.getNumAgents() # 0 -> 1 -> 2 -> 0 -> 1 -> 2 -> ...
29
30        # Increase depth if the next agent is pacman
31        if nextIndex == 0:
32            depth +=1
33
34        # Remove the stop action to avoid some extreme cases
35        if Directions.STOP in legalMoves:
36            legalMoves.remove(Directions.STOP)
37
38        # Perform expectimax
39        for move in legalMoves:
40            nextState = gameState.getNextState(agentIndex, move)
41            nextValue, __ = self.performExpectimax(depth, nextIndex, nextState)
42            bestScore = min(bestScore, nextValue) if isMinplayer else max(bestScore, nextValue)
43            scores.append(nextValue)
44
45        # Return the best score and the next action
46        if isMinplayer:
47            s = sum(scores)
48            l = len(scores)
49            return s/l, None
50        else:
51            bestIndices = [index for index in range(len(scores)) if scores[index] == bestScore]
52            choisenIndex = random.choice(bestIndices)
53            return max(scores), legalMoves[choisenIndex]
54
# End your code (Part 3)
```

## ● Part4 Better evaluation function

```
● ● ●
1 def betterEvaluationFunction(currentGameState):
2     """
3         Your extreme ghost-hunting, pellet-nabbing, food-gobbling, unstoppable
4         evaluation function (Part 4).
5     """
6     # Begin your code (Part 4)
7     # raise NotImplemented("To be implemented")
8     # Score
9     curScore = currentGameState.getScore()
10
11    # Ghosts
12    pacmanPos = currentGameState.getPacmanPosition()
13    ghostsState = [(manhattanDistance(pacmanPos, currentGameState.getGhostPosition(Id)), Id) \
14                    for Id in range(1, currentGameState.getNumAgents())]
15    minGhostDist, minGhostId = (0, 0) if len(ghostsState) == 0 else min(ghostsState)
16    isScared = currentGameState.data.agentStates[minGhostId].scaredTimer > 1
17
18    # Food
19    foodDist = [manhattanDistance(pacmanPos, food) for food in currentGameState.getFood().asList()]
20    numFood = currentGameState.getNumFood()
21    minFoodDist = 1 if numFood == 0 else min(foodDist)
22
23    # Capsules
24    numCapsules = len(currentGameState.getCapsules())
25    capsulesDist = [manhattanDistance(pacmanPos, capsule) for capsule in currentGameState.getCapsules()]
26    minCapsuleDist = float('9999999') if len(currentGameState.getCapsules()) == 0 else min(capsulesDist)
27
28    # Init variables and weight
29    variables = [curScore, minGhostDist, numCapsules, minCapsuleDist, minFoodDist, numFood]
30    weight = [
31        [11, -6, 0, 0, 0, 0], # isScared
32        [9, 0, -5, -5, -1, -1], # minCapsuleDist < 5
33        [10, 0, 0, 0, -4, -4], # numCapsules == ^
34        [9, 1, -5, -5, -4, -4] # default
35    ]
36
37    # Return the score
38    if isScared:
39        weightNum = 0
40    elif minCapsuleDist < 6 :
41        weightNum = 1
42    elif numCapsules == 0:
43        weightNum = 2
44    else :
45        weightNum = 3 # default
46    return sum([x*y for x, y in zip(variables, weight[weightNum]) ])
# End your code (Part 4)
```

## Part II. Results & Analysis (5%):

- The screen shot of the result.

```
alfonso@ubuntu: ~/Git_workspace/Intro-to-AI-2023/HW3/AI_HW3
***      >= 5: 1 points
***      >= 10: 2 points
***      10 wins (4 of 4 points)
***      Grading scheme:
***          < 1: fail
***          >= 1: 1 points
***          >= 4: 2 points
***          >= 7: 3 points
***          >= 10: 4 points

### Question part4: 10/10 ###

Finished at 12:44:21

Provisional grades
=====
Question part1: 20/20
Question part2: 25/25
Question part3: 25/25
Question part4: 10/10
-----
Total: 80/80

ALL HAIL GRANDPAC.
```

- Analysis
- The main goal of this evaluation function is to achieve the highest possible score. Therefore, Pac-Man is designed to chase the target with the highest score, which includes scared ghosts, capsules, and food in a priority sequence from strong to weak. By applying a negative parameter to the number of prey and adding it to the evaluation function's return score, Pac-Man will be more likely to chase the target. The reason for this is that if the number of preys decrease after Pac-Man's move, the negative parameter will result in a higher return score, making the move more likely to be chosen.
- If there are any scared ghosts, Pac-Man will desperately chase them while avoiding unscared ghosts. In the absence of scared ghosts, Pac-Man will continue eating dots, but if a capsule appears within six units of distance, Pac-Man will switch to eating the capsule. If there are no capsules left, Pac-Man will focus on eating dots.

- In a previous version, Pac-Man had a main problem of repeatedly turning right and left near the capsule, waiting for the ghost to come close enough to eat the capsule. This problem led to a lower score due to the time wasted waiting for the ghost. After debugging the if-else condition, I realized that the evaluation function evaluated the move's result, and the condition used had not yet occurred.
- The main reason for this problem was that the evaluation function told Pac-Man that the ghost was scared, which would happen after Pac-Man made the move. This caused Pac-Man to switch targets to the ghost. After the Pac-Man left for a small distance, the evaluation function returned a value indicating that there were capsules within six units of distance, causing Pac-Man to switch back to the capsule again. I fixed the bug by increasing the score under the condition of any capsule being closer than six units of distance. This make Pac-Man choose to eat the capsule rather than chase for the ghost before the capsule is eaten.

- The Pac-Man turn repeatedly.

