Inteligência Artificial
T2 - Relatório

Professor: Eduardo Bezerra
Aluno: Lucas Lima da Cruz

- # Q1 - ReflexAgent

A função *getActions* da classe *ReflexAgent* procura a melhor opção de movimento para o Pacman em seu estado atual. Este cálculo é feito usando uma função *evaluationFunction*, que é chamada iterativamente para cada movimento da lista de movimentos disponíveis (*legalMoves*), de forma a calcular e armazenar numa lista os valores (*scores*) associados à cada possível ação tomada pelo Pacman.

A escolha do melhor movimento é feita escolhendo de *legalMoves* um elemento que tenha como índice o mesmo índice de um dos melhores valores da lista de *scores* calculados pela função *evaluationFunction*.

```
   def getAction(self, gameState):
       # Armazena movimentos possíveis e estados sucessores
       legalMoves = gameState.getLegalActions()

       # Escolhe uma das melhores ações
       scores = [self.evaluationFunction(gameState, action) for action in
legalMoves]

       bestScore = max(scores)

       bestIndices = [index for index in range(len(scores)) if scores[index] ==
bestScore]

       chosenIndex = random.choice(bestIndices)

       return legalMoves[chosenIndex]
```

- Inserindo a função *evaluationFunction*:

```
def evaluationFunction(self, currentGameState, action):
       successorGameState = currentGameState.generatePacmanSuccessor(action)
       newPos = successorGameState.getPacmanPosition()
       newFood = successorGameState.getFood()
       newFoodList = newFood.asList()
       oldFood = currentGameState.getFood()
       ghostPositions = successorGameState.getGhostPositions()
       newGhostStates = successorGameState.getGhostStates()
       newScaredTimes = [ghostState.scaredTimer for ghostState in newGhostStates]

       # Computa distância para o fantasma mais próximo.
       minDistanceGhost = float("+inf")

       for ghostPos in ghostPositions:
           minDistanceGhost = min(minDistanceGhost,  util.manhattanDistance(newPos,
```

```
ghostPos))

        # Se a ação selecionada leva à colisão com o ghost, a pontuação é mínima.
        if minDistanceGhost == 0:
            return float("-inf")

        # Se a ação conduzir para a vitória, pontuação é máxima
        if successorGameState.isWin():
            return float("+inf")

        score = successorGameState.getScore()

        # Incentiva ação que conduz o agente para mais longe do fantasma mais próximo
        score += 2 * minDistanceGhost

        minDistanceFood = float("+inf")

        for foodPos in newFoodList:
            minDistanceFood = min(minDistanceFood, util.manhattanDistance(foodPos,
newPos))

        # Incentiva ação que conduz o agente para mais perto da comida mais próxima
        score -= 2 * minDistanceFood

        # Incentiva ação que leva a uma comida
        if(successorGameState.getNumFood() < currentGameState.getNumFood()):
            score += 5

        # Penaliza as ações de parada
        if action == Directions.STOP:
            score -= 10

        return score
```

Testando o agente reflexivo, percebe-se que ele tem bom desempenho nos layouts *mediumClassic* com 1 fantasma e *testClassic* fornecidos no enunciado. Já para mediumClassic com 2 fantasmas o desempenho piora consideravelmente.

Testando o agente 10 vezes no layout *mediumClassic* com 2 fantasmas:

- python3 pacman.py -p ReflexAgent -k 2 -q -n 10

```
Pacman died! Score: 67
Pacman died! Score: 15
Pacman emerges victorious! Score: 1397
Pacman died! Score: -401
Pacman died! Score: 159
Pacman died! Score: -284
Pacman emerges victorious! Score: 1154
Pacman died! Score: 168
Pacman emerges victorious! Score: 1258
```

```
Pacman died! Score: 325
Average Score: 385.8
Scores:       67.0, 15.0, 1397.0, -401.0, 159.0, -284.0, 1154.0, 168.0, 1258.0, 325.0
Win Rate:      3/10 (0.30)
Record:       Loss, Loss, Win, Loss, Loss, Loss, Win, Loss, Win, Loss
```

• Q2 - Minimax

O algoritmo Minimax opera através de uma chamada recursiva das funções *maxValue* e *minValue* da função driver *minimax:*

```
def minimax(self, gameState, agentIndex=0, depth='2', action=Directions.STOP):
       # Driver para as chamadas recursivas do algoritmo
       agentIndex = agentIndex % gameState.getNumAgents()
       if agentIndex == 0:
           depth = depth-1

       if gameState.isWin() or gameState.isLose() or depth == -1:
           return {'value':self.evaluationFunction(gameState), 'action':action}
       else:
           if agentIndex==0:
               return self.maxValue(gameState,agentIndex,depth)
           else:
               return self.minValue(gameState,agentIndex,depth)

   def maxValue(self, gameState, agentIndex, depth):
       # Escolhe o valor máximo entre estados sucessores de um fantasma
       v = {'value':float('-inf'), 'action':Directions.STOP}
       legalMoves = gameState.getLegalActions(agentIndex)

       for action in legalMoves:
           if action == Directions.STOP:
               continue
           successorGameState = gameState.generateSuccessor(agentIndex, action)
           successorMinMax = self.minimax(successorGameState, agentIndex+1, depth,
action)

           if v['value'] <= successorMinMax['value']:
               v['value'] = successorMinMax['value']
               v['action'] = action

       return v

   def minValue(self, gameState, agentIndex, depth):
       # Escolhe o valor mínimo entre estados sucessores do Pacman
       v = {'value': float('inf'), 'action': Directions.STOP}
       legalMoves = gameState.getLegalActions(agentIndex)

       for action in legalMoves:
```

```
            if action == Directions.STOP:
                continue
            successorGameState = gameState.generateSuccessor(agentIndex, action)
            successorMinMax = self.minimax(successorGameState, agentIndex+1, depth,
action)

            if v['value'] >= successorMinMax['value']:
                v['value'] = successorMinMax['value']
                v['action'] = action

        return v
```

● Q3 - Poda Alpha-Beta

Adicionando dois parâmetros *alpha* e *beta*, que correspondem respectivamente à melhor alternativa de caminho para o agente maximizador (Pacman) e a melhor alternativa de caminho para o minimizador (fantasmas). Esses parâmetros são passados nas chamadas recursivas das funções *minValue* e *maxValue*. Dentro dessas duas funções, é feita uma verificação e alteração dos valores dos parâmetros *alpha* e *beta*, a fim de evitar gastos computacionais explorando estados desnecessários.

Inicialmente os valores de alfa e beta são inicializados com respectivamente *'-inf'* e '*+inf'.*

- Função *minValue*:

```
def minValue(self, gameState, agentIndex, depth, alpha, beta):
        v = {'value':float('inf'), 'action':Directions.STOP}
        legalMoves = gameState.getLegalActions(agentIndex)

        for action in legalMoves:
            if action == Directions.STOP:
                continue

            successorGameState = gameState.generateSuccessor(agentIndex, action)
            successorMinMax = self.minimax(successorGameState, agentIndex+1, depth,
action, alpha, beta)

            if v['value'] >= successorMinMax['value']:
                v['value'] = successorMinMax['value']
                v['action'] = action

            if v['value'] < alpha:
            # Interrompe a exploração de estados desnecessários
                return v

            beta = min(beta, v['value'])
        return v
```

- Função *maxValue*:

```
def maxValue(self, gameState, agentIndex, depth, alpha, beta):
        v = {'value':float('-inf'), 'action':Directions.STOP}
        legalMoves = gameState.getLegalActions(agentIndex)

        for action in legalMoves:
            if action == Directions.STOP:
                continue
            successorGameState = gameState.generateSuccessor(agentIndex, action)
            successorMinMax = self.minimax(successorGameState, agentIndex+1, depth,
action, alpha, beta)

            if v['value'] <= successorMinMax['value']:
                v['value'] = successorMinMax['value']
                v['action'] = action

            if v['value'] > beta:
            # Interrompe a exploração de estados desnecessários
                return v

            alpha = max(alpha, v['value'])
        return v
```

- # Q4 - ExpectiMax

O algoritmo ExpectiMax difere do algoritmo Minimax ao assumir que o oponente nem sempre toma as decisões ideais. Para isso, a função que antes gerava o melhor valor para uma jogada do oponente (*minValue*) agora calcula o valor mais provável para uma jogada do mesmo.

- Função *expValue*:

```
  def expValue(self, gameState, agentIndex, depth):
       v = {'value': 0, 'action': Directions.STOP}
       legalMoves = gameState.getLegalActions(agentIndex)

       for action in legalMoves:
           if action == Directions.STOP:
               continue
           successorGameState = gameState.generateSuccessor(agentIndex, action)
           successorExpectiMax = self.expectimax(successorGameState, agentIndex+1,
depth, action)
           probability = successorExpectiMax['value']/len(legalMoves)
           v['value']+= probability
       return v
```

Comparando o desempenho entre os agentes Alfa-Beta e Expectimax no layout *trappedClassic*, através dos inputs abaixo, obtemos os seguintes resultados:

- python pacman.py -p AlphaBetaAgent -l trappedClassic -a depth=3 -q -n 10

```
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Average Score: -501.0
Scores:        -501.0, -501.0, -501.0, -501.0, -501.0, -501.0, -501.0, -501.0, -501.0, -501.0
Win Rate:      0/10 (0.00)
Record:        Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss
```

- python pacman.py -p ExpectimaxAgent -l trappedClassic -a depth=3 -q -n 10

```
Pacman emerges victorious! Score: 532
Pacman emerges victorious! Score: 532
Pacman emerges victorious! Score: 532
Pacman died! Score: -502
Pacman died! Score: -502
Pacman died! Score: -502
Pacman emerges victorious! Score: 532
Pacman died! Score: -502
Pacman died! Score: -502
Pacman died! Score: -502
Average Score: -88.4
Scores:        532.0, 532.0, 532.0, -502.0, -502.0, -502.0, 532.0, -502.0, -502.0, -502.0
Win Rate:      4/10 (0.40)
Record:        Win, Win, Win, Loss, Loss, Loss, Win, Loss, Loss, Loss
```

A disparidade entre o número de mortes do Pacman nos dois experimentos se dá pela movimentação ideal dos fantasmas no cenário com algoritmo Alfa-Beta, o que dificulta mais o jogo para o Pacman em relação ao cenário com algoritmo ExpectiMax.

● Q4 - Função de Avaliação

Inserindo o seguinte trecho de código na já fornecida função *betterEvaluationFunction*, foi possível melhorar o desempenho geral da função.

O trecho de código busca incentivar as ações do Pacman de forma com que este persiga fantasmas assustados no jogo e se afaste de fantasmas não assustados.

```
    # Recebe os estados dos fantasmas no jogo
```

```
    ghostStates = currentGameState.getGhostStates()

    # Inicializa uma lista para armazenar distâncias do Pacman para fantasmas
assustados
    scaredGhostDistanceList = []

    # Inicializa uma lista para armazenar distâncias do Pacman para fantasmas
não-assustados
    nonScaredGhostDistanceList = []

    for ghostState in ghostStates:
        if ghostState.scaredTimer > 0:
            # Calcula e armazena na lista a distância do Pacman para um fantasma
assustado
            scaredGhostDistanceList.append(util.manhattanDistance(newPos,
ghostState.getPosition()))
        else:
       # Calcula e armazena na lista a distância do Pacman para um fantasma não-
assustado
            nonScaredGhostDistanceList.append(util.manhattanDistance(newPos,
ghostState.getPosition()))

    minScaredGhostDistance = -1
    minNonScaredGhostDistance = -1

    if len(scaredGhostDistanceList) > 0:
        # Verifica se a lista é vazia
        minScaredGhostDistance = min(scaredGhostDistanceList)

    if len(nonScaredGhostDistanceList) > 0:
        # Verifica se a lista é vazia
        minNonScaredGhostDistance = min(nonScaredGhostDistanceList)

    # Incentiva o Pacman a perseguir fantasmas assustados e fugir de não-assustados
    score += 2 * minScaredGhostDistance
    score -= 4 * minNonScaredGhostDistance
```

Testando a nova função de avaliação com o código acima inserido, obtemos os seguintes resultados para *smallClassic*:

- python pacman.py -l smallClassic -p ExpectimaxAgent -a evalFn=better -q -n 10

```
Pacman emerges victorious! Score: 1350
Pacman emerges victorious! Score: 1360
Pacman emerges victorious! Score: 1364
Pacman emerges victorious! Score: 1326
Pacman emerges victorious! Score: 1357
Pacman died! Score: 339
Pacman emerges victorious! Score: 1375
Pacman died! Score: 117
Pacman emerges victorious! Score: 1366
```

```
Pacman emerges victorious! Score: 1112
Average Score: 1106.6
Scores:       1350.0, 1360.0, 1364.0, 1326.0, 1357.0, 339.0, 1375.0, 117.0,
1366.0, 1112.0
Win Rate:     8/10 (0.80)
Record:       Win, Win, Win, Win, Win, Loss, Win, Loss, Win, Win
```

Já para a sequência de testes realizadas pelo *autograder*, obtemos os seguintes resultados:

```
Pacman emerges victorious! Score: 1143
Pacman emerges victorious! Score: 956
Pacman emerges victorious! Score: 1134
Pacman emerges victorious! Score: 1173
Pacman emerges victorious! Score: 1214
Pacman emerges victorious! Score: 1360
Pacman emerges victorious! Score: 1260
Pacman emerges victorious! Score: 960
Pacman emerges victorious! Score: 1289
Pacman emerges victorious! Score: 1162
Average Score: 1165.1
Scores:       1143.0, 956.0, 1134.0, 1173.0, 1214.0, 1360.0, 1260.0, 960.0,
1289.0, 1162.0
Win Rate:     10/10 (1.00)
Record:       Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```