CEFET/RJ

Trabalho 2 - Relatório

Inteligência Artificial

PACMAN

Aluno
Caio Seda Bittencourt

Período
2021.1

Questão 1 - Reflex Agent

Na primeira questão, a construção do Reflex Agent se baseia na análise do estado atual (e sua extensão com suas possíveis ações legais). Esse agente leva em consideração apenas o aumento do seu score com base em suas ações, sem definição de uma estratégia de busca através da análise da árvore de estados.

```
def getAction(self, gameState):
    """
    You do not need to change this method, but you're welcome to.

    getAction chooses among the best options according to the evaluation function.

    Just like in the previous project, getAction takes a GameState and returns
    some Directions.X for some X in the set {NORTH, SOUTH, WEST, EAST, STOP}
    """
    # Collect legal moves and successor states
    legalMoves = gameState.getLegalActions()

    # Choose one of the best actions
    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) # Pick randomly among the best

    "Add more of your code here if you want to"

    return legalMoves[chosenIndex]
```

Função getAction de ReflexAgent

A função *getAction()*, que realiza a escolha da próxima ação, apenas faz a chamada da função de avaliação para cada estado possível a partir do atual. Maximizando seu score localmente.

Já a função de Avaliação em si, resumidamente, utiliza 5 propriedades do jogo para realizar a avaliação da pontuação. São elas:

- **Distância do fantasma mais próximo** - *minDistanceGhost*
  Busca maximizar a distância do fantasma mais próximo, indicando derrota - pontuação mínima - caso ela chegue a 0. Pacman busca se afastar dos fantasmas.

- **Se o próximo estado do jogo é de vitória** - *successorGameState.isWin()*

Define pontuação máxima caso o estado de jogo indique vitória, ou seja, todas as comidas foram obtidas.

- **Distância da comida mais próxima** - *minDistanceFood*
  Busca minimizar a distância da comida mais próxima. Fazendo o Pacman se aproximar das mesmas.
- **Número de comidas do estado** - *.getNumFood()*
  Faz a comparação do número de comidas do estado sucessor com o do atual, incentivando a escolha deste estado caso o pacman tenha diminuído o número de comidas.
- **Se a Ação tomada foi a de ficar parado** - *Directions.STOP*
  Verifica se a ação tomada naquele estado foi a de ficar parado, penalizando o agente caso isso ocorra.

```
>python pacman.py --frameTime 0 -p ReflexAgent -k 2 -q -n100
```

Rodando o Pacman 100 vezes obtivemos uma porcentagem de vitória de 34% e uma média de pontuação de 531.

```
Win Rate:      34/100 (0.34)
Average Score: 531.16
```

Resultado obtido no Autograder

```
Question q1
===========

Pacman emerges victorious! Score: 1217
Pacman emerges victorious! Score: 1160
Pacman emerges victorious! Score: 892
Pacman emerges victorious! Score: 1131
Pacman emerges victorious! Score: 748
Pacman emerges victorious! Score: 1225
Pacman emerges victorious! Score: 1156
Pacman emerges victorious! Score: 1174
Pacman emerges victorious! Score: 1024
Pacman emerges victorious! Score: 1159
Average Score: 1088.6
Scores:        1217.0, 1160.0, 892.0, 1131.0, 748.0, 1225.0, 1156.0, 1174.0, 1024.0, 1159.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

```
### Question q1: 4/4 ###


Finished at 20:15:34

Provisional grades
==================
Question q1: 4/4
------------------
Total: 4/4
```

Questão 2 - MiniMax

Diferente do Agente anterior, o MiniMax Agent leva em consideração não só o estado atual, mas todas as ramificações de possíveis estados, até uma certa profundidade, presumindo que seus oponentes tentarão prejudicá-lo. Este tipo de Agente espera que seus oponentes tomarão a ação que minimize a sua probabilidade de vitória, minimizando sua pontuação.

```python
def minimax(self, gameState, agentIndex=0, depth='2', action=Directions.STOP):

    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)
```

Função minimax de MinimaxAgent

As funções *minimax()*, *minValue()* e *maxValue()*, juntas, exploram recursivamente a árvore de estados levando em consideração a intercalação de jogadas dos jogadores: Pacman e Fantasmas. Enquanto a função *minimax()*, realiza o papel de driver, apenas coordenando as jogadas levando em consideração o estado do jogo. A função *maxValue()* busca maximizar a pontuação do Pacman de acordo com os estados resultantes das ações minimizadoras dos seus adversários, enquanto a função *minValue()* busca fazer o oposto.

```python
def maxValue(self, gameState, agentIndex, depth):
    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
```

Função maxValue() de MinimaxAgent

```python
def minValue(self, gameState, agentIndex, depth):
    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
```

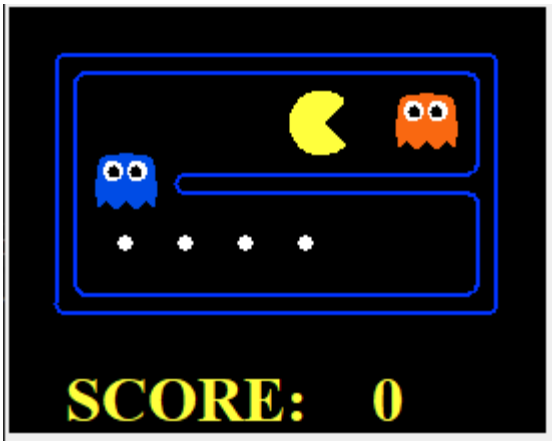
Função minValue() de MinimaxAgent

Contudo, ao realizar 10 jogadas no mapa *mediumClassic* e profundidade da árvore 4, percebemos que não houve muita melhora. Obtivemos 40% de vitória em 10 jogos com uma média de pontuação de 542.

```
>python pacman.py -p MinimaxAgent -l mediumClassic -a depth=4 -q -n10
```

```
Average Score: 542.7
Scores:        398.0, 229.0, 1869.0, -299.0, 1796.0, -253.0, 391.0, 1209.0, 83.0, 4.0
Win Rate:      4/10 (0.40)
Record:        Loss, Loss, Win, Loss, Win, Loss, Loss, Win, Loss, Win
```

Esse resultado se deu por conta dos agentes adversários (fantasmas), estarem realizando ações aleatórias e não a ações que sejam as mais eficientes no sentido de minimizar a pontuação do pacman. Como esse tipo de agente minimax sempre leva em consideração que o agente adversário é um agente perfeito, então o pacman acaba desconsiderando caminhos que levariam a uma pontuação maior - e até mesmo a vitória - pois não cogitam que seu adversário sequer as escolheria.



Isso, inclusive, explica o fato do pacman se matar ao perceber que todos os estados possíveis resultam em derrota. O pacman, como não cogita que seu adversário irá recuar, vai em direção a morte a fim de não minimizar ainda mais seu score com a penalidade de apenas estar vivo.

Resultado obtido no Autograder

```
Pacman died! Score: 84
Average Score: 84.0
Scores:        84.0
Win Rate:      0/1 (0.00)
Record:        Loss
```

```
### Question q2: 5/5 ###


Finished at 20:17:32

Provisional grades
==================
Question q2: 5/5
------------------
Total: 5/5
```

Questão 3 - Alpha Beta Pruning

Como a expansão de toda a árvore de ações é custosa, o método de poda Alpha beta evitará expansões desnecessárias uma vez que considera que seu adversário possui opções melhores e jamais optaria pelas outras opções.

```python
def minimax(self, gameState, agentIndex, depth, alpha, beta):
    agentIndex = agentIndex % gameState.getNumAgents()

    if agentIndex==0: depth = depth-1

    if gameState.isWin() or gameState.isLose() or depth == -1:
        return self.evaluationFunction(gameState), None

    if agentIndex == 0:
        bestVal, bestAction = self.maxValue(gameState, agentIndex, depth, alpha, beta)
    else:
        bestVal, bestAction = self.minValue(gameState, agentIndex, depth, alpha, beta)

    return bestVal, bestAction
```

Mais uma vez a função *minimax()* representará a função driver, sendo a única diferença os parâmetros passados para as funções *maxValue()* e *minValue()*. Os parâmetros Alpha e Beta representam os melhores valores obtidos até então pelas funções maximizadoras e minimizadoras respectivamente.

Quando percebe-se, ao expandir uma dessas funções, que o melhor valor obtido até então não será superado pelo agente subsequente, a expansão dessa subárvore é descartada.

Vale ressaltar que a poda AlphaBeta não melhora o desempenho de vitórias do agente, apenas em uma melhor otimização em relação ao tempo de execução do mesmo. A melhora no tempo obtida pela poda está diretamente correlacionada com a quantidade de estados nó ordenados. Quanto mais os estados estão ordenados, mais a poda conseguirá identificar subárvores que não precisarão ser expandidas.

Resultado obtido no Autograder

```
Pacman died! Score: 84
Average Score: 84.0
Scores:        84.0
Win Rate:      0/1 (0.00)
Record:        Loss
```

```
### Question q3: 5/5 ###


Finished at 20:18:49

Provisional grades
==================
Question q3: 5/5
------------------
Total: 5/5
```

Questão 4 - Expectimax

Se tratando de agentes adversários aleatórios, esperar que eles sempre tomem a ação que irá minimizar a sua pontuação não é o ideal. Com isso, o algoritmo Expectimax realiza uma média ponderada das pontuações obtidas do agente seguinte para tomar suas decisões.

Nesse sentido, a implementação desse agente só necessitou da implementação de uma média ponderada na função *minValue()* considerando que todas as ações são equiprováveis.

```
def minValue(self, gameState, agentIndex, depth):
    v = {'value': float('-inf'), 'action': Directions.STOP}
    legalMoves = gameState.getLegalActions(agentIndex)
    probMove = 1/len(legalMoves)
    meanVal = 0
    for action in legalMoves:
        if action == Directions.STOP: continue
        successorGameState = gameState.generateSuccessor(agentIndex, action)
        successorExpectiMax = self.expectimax(successorGameState, agentIndex+1, depth, action)

        meanVal += probMove * successorExpectiMax['value']

    v['value'] = meanVal
    v['action'] = action

    return v
```

A partir dessa mudança, o agente deixa de ser “pessimista” e começa a considerar todas as possibilidades de escolha de seu adversário. Com isso, pode-se notar uma melhora significativa na performance obtida dentro do layout *trappedClassic* em relação ao agente implementado minimax.

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

```
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
```

```
Average Score: 325.2
Scores:        532.0, 532.0, 532.0, 532.0, 532.0, -502.0, 532.0, 532.0, 532.0, -502.0
Win Rate:      8/10 (0.80)
Record:        Win, Win, Win, Win, Win, Loss, Win, Win, Win, Loss
```

Resultado obtido no Autograder

```
Pacman died! Score: 84
Average Score: 84.0
Scores:        84.0
Win Rate:      0/1 (0.00)
Record:        Loss
```

```
### Question q4: 5/5 ###


Finished at 20:42:17

Provisional grades
==================
Question q4: 5/5
------------------
Total: 5/5
```

Questão 5

```
# calcula distância entre o agente e a pílula mais próxima
minDistanceFood = float("+inf")
for foodPos in newFoodList:
    minDistanceFood = min(minDistanceFood, util.manhattanDistance(foodPos, newPos))

# incentiva o agente a se aproximar mais da pílula mais próxima
score -= 2 * minDistanceFood

# incentiva o agente a comer pílulas
score -= 4 * len(newFoodList)
```

```
  capsulelocations = currentGameState.getCapsules()
  score -= 4 * len(capsulelocations)
```

```
isScared = [True for time in scaredTimes if time is not 0]
if any(isScared):
    score -= 2 * minDistanceGhost
else:
    score -= 4 * minDistanceGhost

return score
```

Para uma melhor função de avaliação utilizamos uma estratégia que se baseia tanto em valorizar ficar perto de comidas e valorizar mais ainda diminuir seu número, uma vez que elas são essenciais para ganhar o jogo. Outra estratégia utilizada foi incentivar o pacman a se aproximar dos fantasmas uma vez que os mesmos estão assustados. Isso tudo aliado ao incentivo na diminuição do número de cápsulas, deu como resultado uma média de pontuação de 1056, além do aproveitamento de 100% dos jogos.

Resultado obtido no Autograder

```
Question q5
===========

Pacman emerges victorious! Score: 1165
Pacman emerges victorious! Score: 960
Pacman emerges victorious! Score: 1136
Pacman emerges victorious! Score: 1062
Pacman emerges victorious! Score: 1190
Pacman emerges victorious! Score: 1304
Pacman emerges victorious! Score: 955
Pacman emerges victorious! Score: 696
Pacman emerges victorious! Score: 957
Pacman emerges victorious! Score: 1141
Average Score: 1056.6
Scores:        1165.0, 960.0, 1136.0, 1062.0, 1190.0, 1304.0, 955.0, 696.0, 957.0, 1141.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

```
### Question q5: 6/6 ###


Finished at 23:24:26

Provisional grades
==================
Question q5: 6/6
------------------
Total: 6/6
```