



Unicamp - Instituto de Computação

**MC886 B / MO416 A**

Introdução à Inteligência Artificial

Prof. Dr. Jacques Wainer

## Exercício 6

**Aluno:**

George Gigilas Junior - 216741

# Introdução

Este exercício corresponde ao exercício 17.11 do livro texto, cujo enunciado está disponível juntamente com a atividade. Este arquivo corresponde às minhas soluções para os itens a e b do problema.

## Solução

**Item a** - Implementar um simulador de ambiente para o ambiente da imagem, tal que a geografia específica do ambiente seja facilmente alterada. Um pedaço do código para fazer isso está no repositório online.

Encontrei esse pedaço código presente no repositório oficial do livro no seguinte link: [aima-lisp/4x3-mdp.lisp at master · aimacode/aima-lisp \(github.com\)](https://github.com/aimacode/aima-lisp/blob/master/4x3-mdp.lisp)

A partir desse código, escrevi meu próprio código em Python e implementei a parte que faltava. No meu código, criei a função `getDestination()` que, dados o estado de origem e a direção para qual o agente quer se mover, a função retorna o estado para o qual o agente vai e a recompensa que o agente ganhou com isso. Também criei a função `getReward()` (que retorna a recompensa de um estado, dado um estado) e a função `getTransitionProbabilities()` (que retorna o modelo de transição de um estado, dados o estado e a direção para qual se quer mover).

Vale ressaltar que o estado para o qual o agente vai tem uma probabilidade de não ser o estado para o qual ele queria ir. Esses cálculos são feitos internamente na função `getDestination()`, a partir da geração de números aleatórios.

Segue o código:

OBS: o código também pode ser encontrado no meu repositório no seguinte link: [MC886/Projeto 6 at main · GeorgeJuniorGG/MC886 \(github.com\)](https://github.com/GeorgeJuniorGG/MC886/blob/main/Projeto%206/Projeto%206.py)

```
import os

class mdp:

    def __init__(self, initialState) -> None:
        self.initialState = initialState
        self.terminalStates = ((4,2), (4,3))
        self.actions = {(1,1): {"left": ((1,1), 0.9), ((1,2), 0.1)},
                        "right": ((2,1), 0.8), ((1,2), 0.1),
                        ((1,1), 0.1)},
```

```

        "up": ((1,2), 0.8), ((2,1), 0.1),
((1,1), 0.1)),
        "down": ((1,1), 0.9), ((2,1), 0.1))},
(1,2): {"left": ((1,2), 0.8), ((1,1), 0.1),
((1,3), 0.1)),
        "right": ((1,2), 0.8), ((1,1), 0.1),
((1,3), 0.1)),
        "up": ((1,2), 0.2), ((1,3), 0.8)),
        "down": ((1,2), 0.2), ((1,1), 0.8))},
(1,3): {"left": ((1,3), 0.9), ((1,2), 0.1)),
        "right": ((1,3), 0.1), ((2,3), 0.8),
((1,2), 0.1)),
        "up": ((1,3), 0.9), ((2,3), 0.1)),
        "down": ((1,3), 0.1), ((2,3), 0.1),
((1,2), 0.8))},
(2,1): {"left": ((2,1), 0.2), ((1,1), 0.8)),
        "right": ((2,1), 0.2), ((3,1), 0.8)),
        "up": ((2,1), 0.8), ((1,1), 0.1),
((3,1), 0.1)),
        "down": ((2,1), 0.8), ((1,1), 0.1),
((3,1), 0.1))},
(2,2): {"left": 0,
        "right": 0,
        "up": 0,
        "down": 0},
(2,3): {"left": ((2,3), 0.2), ((1,3), 0.8)),
        "right": ((2,3), 0.2), ((3,3), 0.8)),
        "up": ((2,3), 0.8), ((1,3), 0.1),
((3,3), 0.1)),
        "down": ((2,3), 0.8), ((1,3), 0.1),
((3,3), 0.1))},
(3,1): {"left": ((3,1), 0.1), ((3,2), 0.1),
((2,1), 0.8)),
        "right": ((3,1), 0.1), ((3,2), 0.1),
((4,1), 0.8)),
        "up": ((3,2), 0.8), ((2,1), 0.1),
((4,1), 0.1)),
        "down": ((3,1), 0.8), ((2,1), 0.1),
((4,1), 0.1))},
(3,2): {"left": ((3,2), 0.8), ((3,1), 0.1),
((3,3), 0.1)),
        "right": ((4,2), 0.8), ((3,1), 0.1),
((3,3), 0.1)),

```

```

        "up": ((3,2), 0.1), ((4,2), 0.1),
((3,3), 0.8)),
        "down": ((3,2), 0.1), ((4,2), 0.1),
((3,1), 0.8))},
        (3,3): {"left": ((2,3), 0.8), ((3,3), 0.1),
((3,2), 0.1)),
        "right": ((3,2), 0.1), ((4,3), 0.8),
((3,3), 0.1)),
        "up": ((2,3), 0.1), ((4,3), 0.1),
((3,3), 0.8)),
        "down": ((3,2), 0.8), ((2,3), 0.1),
((4,3), 0.1))},
        (4,1): {"left": ((4,1), 0.1), ((3,1), 0.8),
((4,2), 0.1)),
        "right": ((4,1), 0.9), ((4,2), 0.1)),
        "up": ((4,2), 0.8), ((4,1), 0.1),
((3,2), 0.1)),
        "down": ((4,1), 0.9), ((3,1), 0.1))},
        (4,2): {"left": 0,
        "right": 0,
        "up": 0,
        "down": 0},
        (4,3): {"left": 0,
        "right": 0,
        "up": 0,
        "down": 0}}

    self.rewards = {(1,1): -0.04, (1,2): -0.04, (1,3): -0.04,
        (2,1): -0.04, (2,2): -0.04, (2,3): -0.04,
        (3,1): -0.04, (3,2): -0.04, (3,3): -0.04,
        (4,1): -0.04, (4,2): -1, (4,3): 1}

    # Dadas uma posicao e uma direcao, simula qual sera o proximo
    estado do agente, retornando tambem a recompensa desse estado
    def getDestination(self, position, direction):
        rand = int.from_bytes(os.urandom(8), byteorder="big") / ((1 <<
64) - 1)

        possibilities = []
        for i in self.actions[position][direction]:
            possibilities.append(i[1])

        previous = 0

```

```

        for i in range(len(possibilities)):
            if(previous + possibilities[i] >= rand):
                #
                # Determina qual sera o novo estado de acordo com o modelo de
                # transicao estabelecido
                pos = i
                break
            previous += possibilities[i]

        newpos = self.actions[position][direction][pos][0]
        #
        # Corresponde ao novo estado do agente
        return (newpos, self.getReward(newpos))

    # Dada uma posicao, retorna a recompensa de chegar ate ela
    def getReward(self, position):
        return self.rewards[position]

    # Dadas uma posicao e uma direcao, retorna o modelo de transicao
    def getTransitionProbabilities(self, position, direction):
        return self.actions[position][direction]

```

**Item b** - Crie um agente que utilize policy iteration, e meça seu desempenho no simulador de ambiente a partir de vários estados iniciais. Execute vários experimentos a partir de cada estado inicial, e compare a recompensa total média recebida por execução com a utilidade do estado inicial, determinada pelo seu algoritmo.

Para criar esse agente, implementei o algoritmo Policy Iteration, a partir do que foi visto em aula. É importante destacar que o vetor de utilidade foi calculado a partir do método do ponto fixo, que faz várias iterações até convergir para um certo valor. Fiz isso pois achei que era mais fácil de implementar, e optei por esse método ter 25 iterações, o que julguei empiricamente como sendo um bom número. Para o valor de  $\gamma$ , escolhi 1 com base no que foi dito em sala (e para que não haja um fator de desconto, o que iria interferir nos resultados).

Para obter a recompensa total média para cada estado inicial, fiz 10.000 simulações para cada estado (sem incluir os estados terminais e o muro). Então, acumulei em uma variável a recompensa total obtida naquele estado e, após as 10.000 iterações, dividi o valor acumulado por 10.000, obtendo a recompensa total média para cada estado.

Segue o código:

OBS: o código também pode ser encontrado no meu repositório no seguinte link:

[MC886/Projeto 6 at main · GeorgeJuniorGG/MC886 \(github.com\)](https://github.com/GeorgeJuniorGG/MC886)

```

from environment_simulator import mdp

def argmax(a0, a1, a2, a3):
    aux = max(max(a0, a1), max(a2, a3))
    if(aux == a0):
        return 0
    elif(aux == a1):
        return 1
    elif(aux == a2):
        return 2
    else:
        return 3

def policyIteration(states, actions, environment, utility, policy,
gamma, reverseStates):
    naoMudou = False

    # Quando nao mudar, eh porque chegamos na politica otima, ja que
estabilizou
    while(not naoMudou):

        naoMudou = True
        utility = policyEvaluation(policy, utility, gamma, environment)
# Calcula a utilidade para a politica em uso

        for i in range(9):

            if(policy[i] == 0):
# Decide a direcao para ir, de acordo com a politica
                direction = "left"
            elif(policy[i] == 1):
                direction = "right"
            elif(policy[i] == 3):
                direction = "down"
            else:
                direction = "up"

            # Calcula o valor da recompensa esperada utilizando a
politica
            anterior = environment.getReward(states[i])
            for (newS, prob) in
environment.getTransitionProbabilities(states[i], direction):

```

```

        anterior += gamma * prob *
utility[reverseStates[newS]]

        # Determina se existe alguma acao que aumenta o valor da
recompensa esperada
        a0 = environment.getReward(states[i])
        a1 = a0
        a2 = a0
        a3 = a0

        for (newS, prob) in
environment.getTransitionProbabilities(states[i], "left"):
            a0 += gamma * prob * utility[reverseStates[newS]]

        for (newS, prob) in
environment.getTransitionProbabilities(states[i], "right"):
            a1 += gamma * prob * utility[reverseStates[newS]]

        for (newS, prob) in
environment.getTransitionProbabilities(states[i], "up"):
            a2 += gamma * prob * utility[reverseStates[newS]]

        for (newS, prob) in
environment.getTransitionProbabilities(states[i], "down"):
            a3 += gamma * prob * utility[reverseStates[newS]]

        a = argmax(a0, a1, a2, a3)
        if(max(max(a0, a1), max(a2, a3)) > anterior):
            policy[i] = a
            naoMudou = False

    return policy

# Calcula pelo metodo do ponto fixo
def policyEvaluation(policy, utility, gamma, environment):
    n = 25 # Numero de iteracoes que
determinei empiricamente para a aproximacao
    for i in range (n):
        for j in range(9):
            if(policy[j] == 0):
                direction = "left"
            elif(policy[j] == 1):
                direction = "right"

```

```

        elif(policy[j] == 3):
            direction = "down"
        else:
            direction = "up"

        utilityAux = environment.getReward(states[j])
        for (newS, prob) in
environment.getTransitionProbabilities(states[j], direction):
            utilityAux += gamma * prob *
utility[reverseStates[newS]]
            utility[j] = utilityAux
        return utility

environment = mdp((1,1))

# Valor de gamma definido com base no que foi dito em sala
gamma = 1

# Mapeamento de identificadores de estados
# OBS: o estado (2,2) nao esta presente pois ele corresponde a uma
parede
states = {0: (1,1), 1: (1,2), 2: (1,3),
          3: (2,1), 4: (2,3),
          5: (3,1), 6: (3,2), 7: (3,3),
          8: (4,1), 9: (4,2), 10: (4,3)}
reverseStates = {(1,1): 0, (1,2): 1, (1,3): 2,
                 (2,1): 3, (2,3): 4,
                 (3,1): 5, (3,2): 6, (3,3): 7,
                 (4,1): 8, (4,2): 9, (4,3): 10}

# Vetor de acoes
# 0 corresponde a ir para a esquerda, 1 para a direita, 2 para cima e 3
para baixo
actions = [0, 1, 2, 3]

# Vetor inicial de política (quaisquer valores)
# Em cada posição, temos a politica para cada estado: 0 corresponde a
ir para a esquerda, 1 para a direita, 2 para cima e 3 para baixo
# OBS: Nao existe politica para os estados terminais porque eles ja sao
o final do problema
policy = [0, 1, 2,
          0, 1,
          0, 1, 2,

```



```

0]

# Vetor inicial de utilidade (inicializados com 0)
# OBS: A utilidade eh o proprio valor de recompensa para os estados
# terminais porque eles ja sao o final do problema
utility = [0, 0, 0,
           0, 0,
           0, 0, 0,
           0, -1, 1]

# Obter a politica otima
optimalPolicy = policyIteration(states, actions, environment, utility,
                                policy, gamma, reverseStates)

# Vetor inicial de valores esperados (inicializados com 0)
expectedValue = [0, 0, 0,
                 0, 0,
                 0, 0, 0,
                 0, -1, 1]

# Este loop calcula o valor esperado de cada estado, a partir de
# multiplas simulacoes
for i in range (9):
    # Nao estou incluindo os estados
    # terminais
    n = 10000
    accum = 0
    # Numero de simulacoes
    # Acumula as somas das
    # recompensas de todas as simulacoes

    for j in range (n):
        # Faz as n
        # simulacoes
        state = states[i]
        # Reseta o estado
        # inicial para cada simulacao
        accum2 = environment.getReward(state)
        # Reseta o
        # acumulador interno para cada simulacao
        while state not in environment.terminalStates:
            aux = optimalPolicy[reverseStates[state]]
            # Escolhe a
            # direcao para ir com base na politica otima
            if(aux == 0):
                direction = "left"
            elif (aux == 1):
                direction = "right"
            elif (aux == 3):
                direction = "down"

```

```

        else:
            direction = "up"

            (state, reward) = environment.getDestination(state,
direction) # Pega o proximo estado e a recompensa atrelada a ele
            accum2 += reward

            accum += accum2 # Atualiza o acumulador
externo

            expectedValue[i] = accum/n # Calcula a media de
valores esperados

# Saida
print("Politica Otima: ")
for i in range(3,0,-1):
    for j in range(1,5):
        if (i, j) == (2,2):
            char = '#'
        elif (i,j) == (3,4) or (i,j) == (2,4):
            char = "T"
        else:
            aux = optimalPolicy[reverseStates[(j, i)]]
            if(aux == 0):
                char = "L"
            elif (aux == 1):
                char = "R"
            elif (aux == 3):
                char = "D"
            else:
                char = "U"
        print(char, end=" ")
    print("")
print("L = Left, R = Right, U = Up, D = Down, # = Hole, T = Terminal
State")
print("#####")
print("Valor Esperado de Cada Estado:")
for i in range(3,0,-1):
    for j in range(1,5):
        if (i, j) == (2,2):
            print(" # ", end="")
        else:
            char = expectedValue[reverseStates[(j,i)]]

```

```

        print("%.2f" %char,end=" ")
    print("")
print("#####")
print("Utilidade de Cada Estado:")
for i in range(3,0,-1):
    for j in range(1,5):
        if (i, j) == (2,2):
            print("  #  ", end="")
        else:
            char = utility[reverseStates[(j,i)]]
            print("%.2f" %char,end=" ")
    print("")

```

A partir desse código, obtive:

- Política Ótima (L = Left, R = Right, U = Up, D = Down, # = Hole, T = Terminal State):  
R R R T  
U # U T  
U L L L
- Utilidade para cada estado:  
0.81 0.87 0.92 1.00  
0.76 # 0.66 -1.00  
0.71 0.66 0.61 0.39
- Valor esperado de cada estado:  
0.81 0.87 0.92 1.00  
0.76 # 0.65 -1.00  
0.71 0.66 0.61 0.39

Com base nos valores obtidos, podemos concluir que eles são muito parecidos. Isso acontece porque o valor de utilidade para um dado estado corresponde, justamente, à recompensa total que se espera receber ao iniciar uma rodada naquele estado. Sendo assim, se medirmos o valor total médio de recompensa para um determinado estado, esperamos obter um valor bem próximo do valor de utilidade. Caso o valor de  $\gamma$  fosse menor que 1, teríamos uma variação maior entre esses dois valores, por ser um fator que desconta a utilidade dos estados adjacentes (que é levada em consideração na geração do vetor de utilidade).