Reinforcement learning – a1rl

André Filipe Frade Guerra

Outubro/2022

Conteúdo

Introdução

Neste relatório irá ser explicado num resumo o funcionamento do tabuleiro de jogo, as classes criadas, funcionalidade dos métodos criados e por fim a execução e discussão dos resultados dos exercícios.

Tabuleiro de jogo

O tabuleiro de jogo é constituído por uma matriz 12x12 que contem em cada posição o objeto “Position” com os seguintes atributos:

```
2   class Position:
3       state = 0
4       parede = False
5       reward = 0.0
```

state é o número identificador da posição.

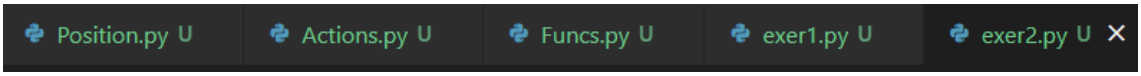
parede é um booleano que indica se a posição é uma parede.

reward é o valor da recompensa dessa posição.

O tabuleiro contém nas laterais e rodeado, posições com a característica parede = TRUE, fazendo com que seja impossível ir para essa posição.

Classes e métodos

Para desenvolver os exercícios da primeira prática da disciplina de Introdução à Aprendizagem Automática foi desenvolvido um código em python usando o vsCode para realizar a mesma.



As seguintes classes foram criadas, sendo as 3 primeiras auxiliares para as seguintes, que são os exercícios resolvidos.

Começando pela classe “Funcs.py”, contem métodos que são comuns nos exercícios:

```
6   def Variance(data, ddof=0):
7       n = len(data)
8       mean = sum(data) / n
9       return sum((x - mean) ** 2 for x in data) / (n - ddof)
10
11
12  def Stdev(data):
13      var = Variance(data)
14      std_dev = math.sqrt(var)
15      return std_dev
```

Métodos usados para calcular o desvio padrão.

```python
def BuildMatrix(matrix, cols, rows, reward):
    j = 1
    for i in range(rows):
        for w in range(cols):
            if i > 0 and i < 11 and w > 0 and w < 11:
                p = Position()
                p.state = j
                p.reward = 0.0
                p.parede = False
                matrix[i][w] = p
            else:
                p = Position()
                p.state = j
                p.reward = 0.0
                p.parede = True
                matrix[i][w] = p
            j += 1
    matrix[10][10].reward = reward
```

Método que cria uma matriz 12x12 em que as linhas e colunas das extremidades têm a propriedade parede = True.

Para cada posição é definido um número.

Por fim é atribuído à posição de destino o valor da recompensa.

```python
def RandomAction():
    r = randrange(1,5)
    if r == 1:
        return Action.UP
    elif r == 2:
        return Action.DOWN
    elif r == 3:
        return Action.LEFT
    elif r == 4:
        return Action.RIGHT
```

Método que defina aleatoriamente a próxima ação.

```python
def StateTransition(currentState, action):
    nextState = 0
    if action == Action.UP:
        nextState = currentState - 12
    elif action == Action.DOWN:
        nextState = currentState + 12
    elif action == Action.RIGHT:
        nextState = currentState + 1
    elif action == Action.LEFT:
        nextState = currentState - 1
    return nextState
```

Método que retorna a nova posição dependendo da ação realizada.

nota: tendo em conta que é uma matriz 12x12 as ações subir e descer subtraem e adicionam 12 à posição atual.

```python
def IsParede(matrix, nextState):
    return SearchStateInMatrix(matrix, nextState).parede

def SearchStateInMatrix(matrix, state):
    for row in matrix:
        for element in row:
            if element.state == state:
                return element
    return False
```

Método IsParede valida se a posição é uma parede.

Método SearchStateInMatrix retorna a posição através da pesquisa pelo numero do estado.

```python
def UpdateReward(matrix, currentState, nextState, alfaVar, gamaVar):
    currentStateValue = SearchStateInMatrix(matrix, currentState).reward
    nextStateValue = SearchStateInMatrix(matrix, nextState).reward
    bestValueNextStepCanDo = GetBestValueAction(matrix, nextState)
    if nextStateValue > 0.0:
        #SearchStateInMatrix(matrix, currentState).reward = ( (1 - alfaVar) * currentStateValue ) + alfaVar * ( nextStateValue + ( gamaVar * bestValueNextStepCanDo ) )
        #SearchStateInMatrix(matrix, currentState).reward = ( currentStateValue + (alfaVar * ( 1 + ( gamaVar * bestValueNextStepCanDo ) ) ) - currentStateValue )
        SearchStateInMatrix(matrix, currentState).reward = (1 - alfaVar) * currentStateValue + alfaVar * (nextStateValue + gamaVar * bestValueNextStepCanDo)
```

Método UpdateReward atualiza a recompensa da posição atual tendo em conta a posição para onde vai a seguir.

```python
def GetBestValueAction(matrix, currentState):
    upStateReward = SearchStateInMatrix(matrix, StateTransition(currentState, Action.UP)).reward
    downStateReward = SearchStateInMatrix(matrix, StateTransition(currentState, Action.DOWN)).reward
    leftStateReward = SearchStateInMatrix(matrix, StateTransition(currentState, Action.LEFT)).reward
    rightStateReward = SearchStateInMatrix(matrix, StateTransition(currentState, Action.RIGHT)).reward

    return max(upStateReward, downStateReward, leftStateReward, rightStateReward)

def GetBestStepAction(matrix, currentState):
    upState = SearchStateInMatrix(matrix, StateTransition(currentState, Action.UP))
    downState = SearchStateInMatrix(matrix, StateTransition(currentState, Action.DOWN))
    leftState = SearchStateInMatrix(matrix, StateTransition(currentState, Action.LEFT))
    rightState = SearchStateInMatrix(matrix, StateTransition(currentState, Action.RIGHT))

    if upState.reward == downState.reward == leftState.reward == rightState.reward:
        return StateTransition(currentState, RandomAction())

    aux = max(upState.reward, downState.reward, leftState.reward, rightState.reward)
    if upState.parede == False and upState.reward == aux:
        return upState.state
    elif downState.parede == False and downState.reward == aux:
        return downState.state
    elif leftState.parede == False and leftState.reward == aux:
        return leftState.state
    elif rightState.parede == False and rightState.reward == aux:
        return rightState.state
```

Método GetBestValueAction retorna a recompensa mais alta tendo em conta a atual posição.

Método GetBestStepAction retorna o estado para o qual a atual posição deve ir tendo em conta o valor da recompensa mais alta. Caso as recompensas em volta sejam todas iguais, é retornado um estado aleatório.

```python
def GetMatrixOnlyRewards(matrix,cols,rows):
    matrixAux = [[0 for _ in range(cols)] for _ in range(rows)]
    for i in range(rows):
        for w in range(cols):
            matrixAux[i][w] = matrix[i][w].reward
    return matrixAux
```

Método que retorna uma matriz apenas com os valores das recompensas para construir o heatmap.

```python
def Test1000(initState, finalState, steps, FAIL_FUN, matrix, mainTestStepsCount,reward):
    currentState = initState
    rewards = []
    countSteps = 0
    while currentState != finalState:
        if countSteps >= steps:
            countSteps = FAIL_FUN
            break

        nextState = GetBestStepAction(matrix, currentState)
        if IsParede(matrix, nextState) == False:
            reward = SearchStateInMatrix(matrix, currentState).reward
            try:
                rewards.append(steps / reward)
            except ZeroDivisionError:
                rewards.append(0)
            currentState = nextState
            countSteps += 1

    #print("Avarage reward per "+str(steps)+" steps at " +str(mainTestStepsCount)+ " => " + str( rewards ) )
    return rewards
```

Método que testa a alínea 2a.

Exercício 1

Presente no ficheiro exer1.py, este ficheiro começa por declarar vários valores iniciais que definem o jogo, como cálculos, runs, steps e o tamanho da matriz.

```
10  reward = 1000
11  FAIL_FUN = "FAIL RUN"
12  alfaVar = 0.7
13  lambdaVar = 0.99
14  #final state
15  finalState = 131
16  #initial state
17  initialState =40
18  currentState = initialState
19  #size matrix/gameboard
20  rows = 12
21  cols = 12
22
23  runs = 30
24  steps = 1000
25
26  matrix = [[0 for _ in range(cols)] for _ in range(rows)]
27
28  BuildMatrix(matrix,cols,rows, reward)
```

```
35  for r in range(runs):
36      currentState = initialState
37      countSteps = 0
38
39      st = time.time()
40      while currentState != finalState:
41          if countSteps >= steps:
42              countSteps = FAIL_FUN
43              break
44
45          nextState = StateTransition(currentState, RandomAction())
46          if IsParede(matrix, nextState) == False:
47              currentState = nextState
48              countSteps += 1
49
50      et = time.time()
51      runTimePerRun.append(et - st)
52      numStepsPerRun.append(countSteps)
```

Este pedaço de código faz correr a simulação do exercício 1 tendo em conta os runs e os steps por run.

Confirma se a próxima ação é uma parede, caso seja, volta a tentar outra ação aleatória.

Caso chega ao máximo de steps é declarada uma FAIL RUN.

```
57  numStepsPerRunClean = []
58  numRunsReachEnd = 0
59  somaTimeOfRun = 0
60  somaStepsOfRuns = 0
61  j = 0
62  for i in numStepsPerRun:
63      if i != FAIL_FUN:
64          numStepsPerRunClean.append(i)
65          numRunsReachEnd += 1
66          somaTimeOfRun += runTimePerRun[j]
67          somaStepsOfRuns += i
68      j += 1
69
70  numberStepsPerRunMRPT = []
71  for i in numStepsPerRunClean:
72      numberStepsPerRunMRPT.append(100/i)
```

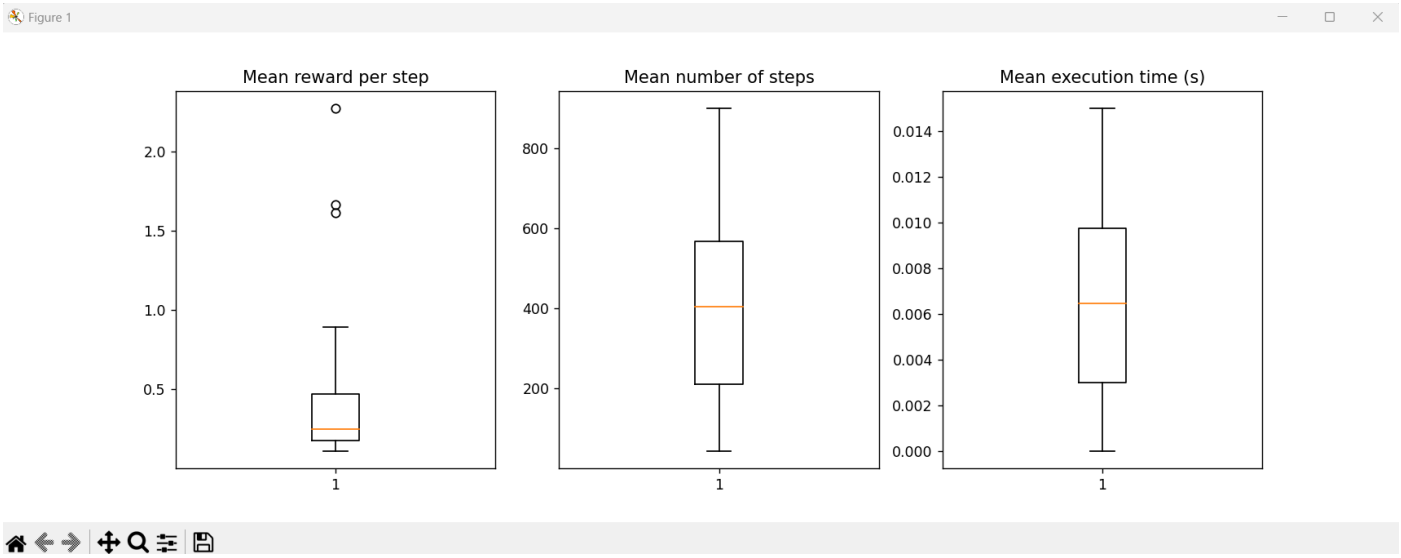
Código auxiliar para depois calcular as métricas pretendidas.

```
77   print("average reward per step in "+str(steps)+" steps => "+ str((numRunsReachEnd*100)/runs) + "%")
78   print("standard deviation steps => " + str(Stdev(numStepsPerRunClean)))
79
80   print("average time of runs "+str(steps)+" steps => "+ str(somaTimeOfRun/numRunsReachEnd))
81   print("standard deviation time => " + str(Stdev(runTimePerRun)))
82
83   print("Average number actions per run => " + str(somaStepsOfRuns/numRunsReachEnd))
84   print("standard deviation time => " + str(Stdev(runTimePerRun)))
85
86   fig, axs = plt.subplots(1,3)
87   axs[0].boxplot(numberStepsPerRunMRPT)
88   axs[0].set_title('Mean reward per step')
89   axs[1].boxplot(numStepsPerRunClean)
90   axs[1].set_title('Mean number of steps')
91   axs[2].boxplot(runTimePerRun)
92   axs[2].set_title('Mean execution time (s)')
93
94   plt.show()
```

Código que mostra os cálculos pretendidos no exercício, como também no final as boxplots pretendidas.

Resultados exercício 1

```
average reward per step in 1000 steps => 83.33333333333333%
standard deviation steps => 234.51210970864597
average time of runs 1000 steps => 0.0052825927734375
standard deviation time => 0.004025386729196857
Average number actions per run => 393.52
standard deviation time => 0.004025386729196857
```



Exercício 2

O exercício dois é uma cópia do primeiro exercício, com algumas alterações no pedaço de código que corre a simulação e os resultados. Os valores iniciados na classe também mudaram de maneira que agora os steps têm um valor de 20.000.

No código que corre a simulação foi adicionado o método UpdateReward que atualiza as recompensas a cada ação feita, e o Test1000 que corre a simulação a cada X steps, neste caso nos steps 100, 200, 500, 600, 700, 800, 900, 1000, 2500, 5000, 7500, 10000, 12500, 15000, 17500 e 20000.

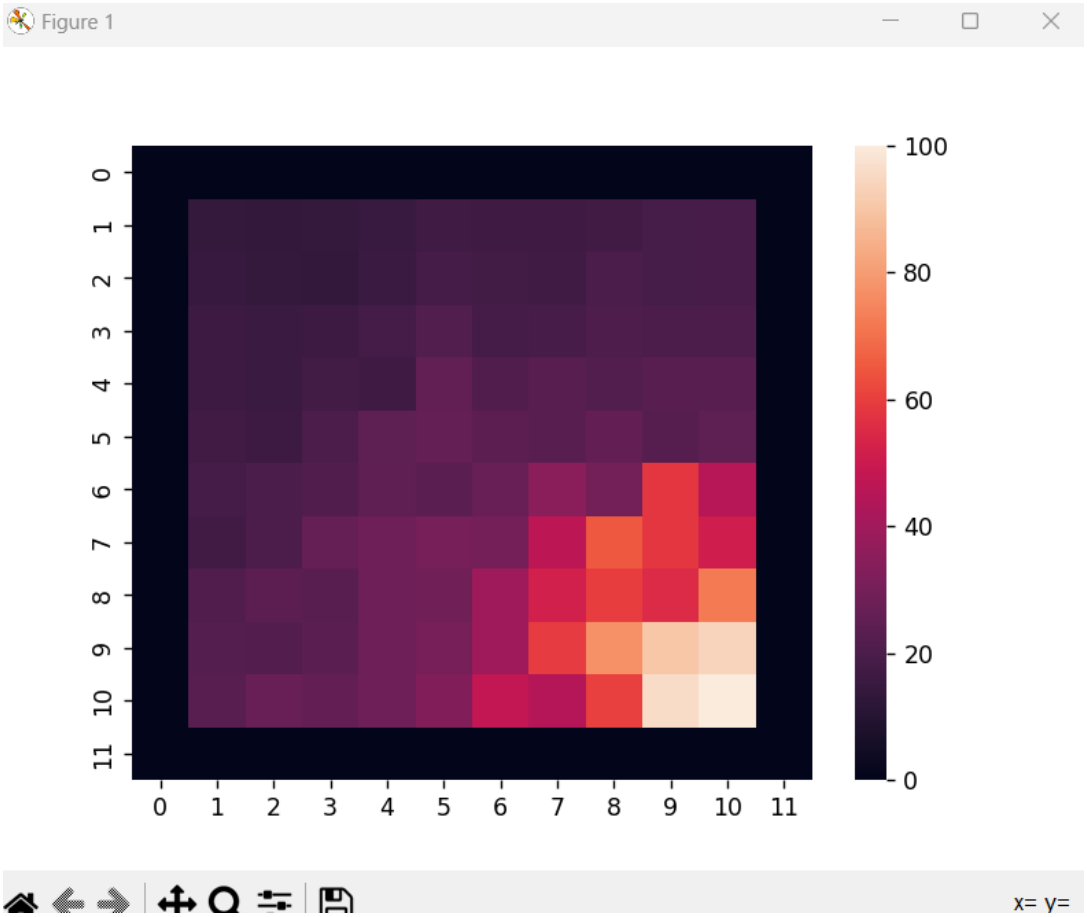
```
numStepsPerRun = []
runTimePerRun = []

for r in range(runs):
    currentState = initialState
    countSteps = 0

    st = time.time()
    while currentState != finalState:
        if countSteps >= steps:
            countSteps = FAIL_FUN
            break

        nextState = StateTransition(currentState, RandomAction())
        if IsParede(matrix, nextState) == False:
            UpdateReward(matrix, currentState, nextState, alfaVar, gamaVar)
            currentState = nextState
            countSteps += 1

        if stepsToStop.__contains__(countSteps):
                rewards = Test1000(initialState, finalState, 1000, FAIL_FUN, matrix, countSteps, reward)
                #plt.plot(rewards)
                #plt.show()

    #m = GetMatrixOnlyRewards(matrix, cols,rows)
    #ax = sns.heatmap(m, cbar=False)
    #plt.show()

    et = time.time()
    runTimePerRun.append(et - st)
    numStepsPerRun.append(countSteps)

```

```
numStepsPerRunClean = []
numRunsReachEnd = 0
somaTimeOfRun = 0
somaStepsOfRuns = 0
j = 0
for i in numStepsPerRun:
    if i != FAIL_FUN:
        numStepsPerRunClean.append(i)
        numRunsReachEnd += 1
        somaTimeOfRun += runTimePerRun[j]
        somaStepsOfRuns += i
    j += 1

print("average time of runs "+str(steps)+" steps => "+ str(somaTimeOfRun/numRunsReachEnd))
print("standard deviation time => " + str(Stdev(runTimePerRun)))

m = GetMatrixOnlyRewards(matrix, cols,rows)
ax = sns.heatmap(m)
plt.show()
```

Código auxiliar para calcular os valores pedidos e depois o gráfico do heatmap.

Resultados 2a

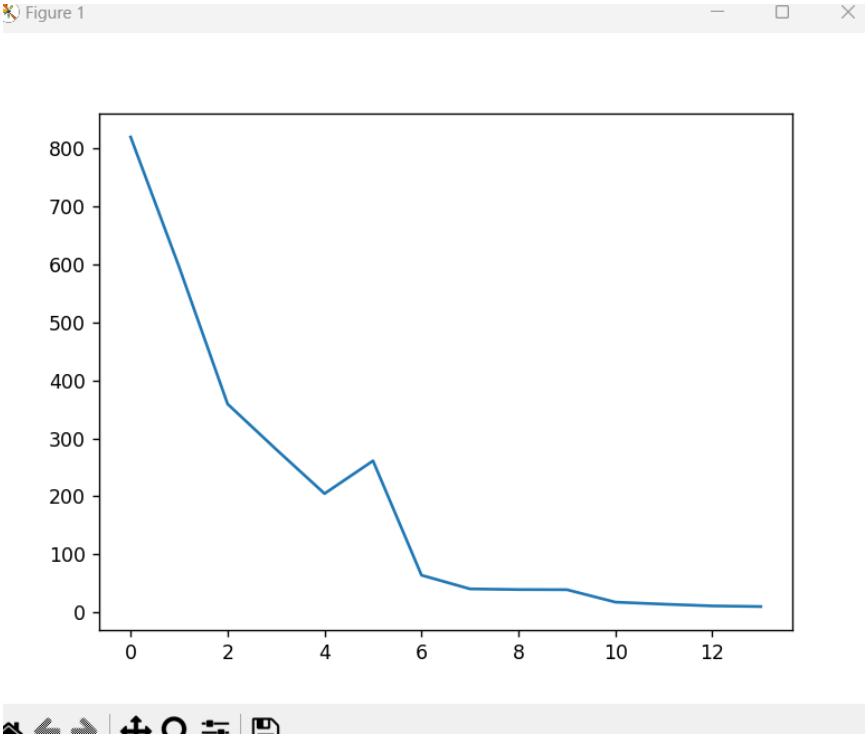
```
average time of runs 20000 steps => 0.1094772736231486
standard deviation time => 0.11016083910338276
```

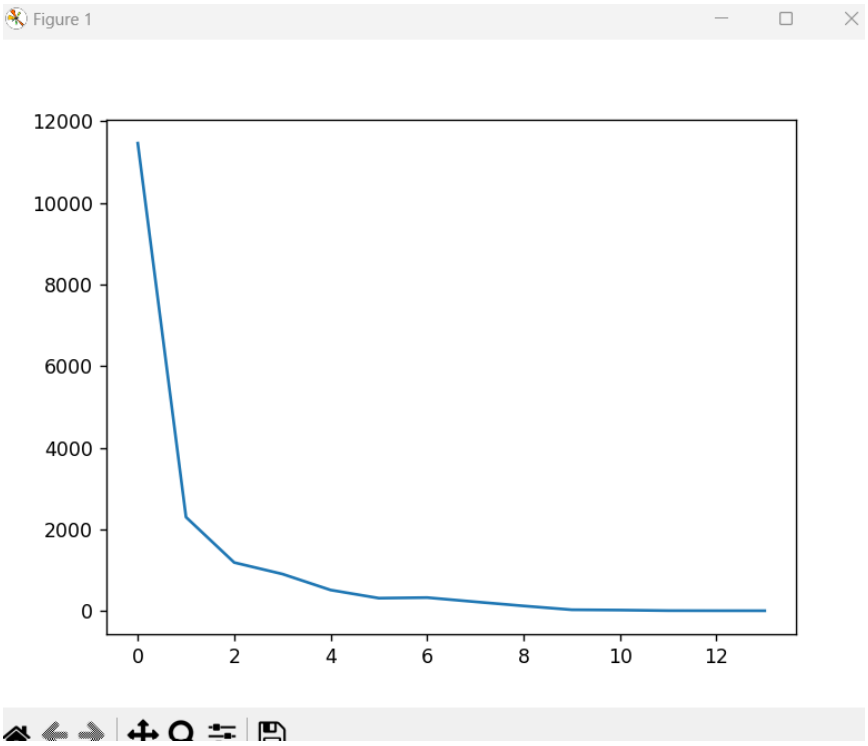


Resultado do heatmap com os valores das recompensas de cada posição.

Podemos observar este mapa a ser construído a cada run descometendo as linhas 57, 58 e 59.

Também podemos observar um gráfico para cada teste nos steps pretendidos descomentando as linhas 54 e 55, onde inicialmente vemos um gráfico que vai alterando ao longo de cada run:



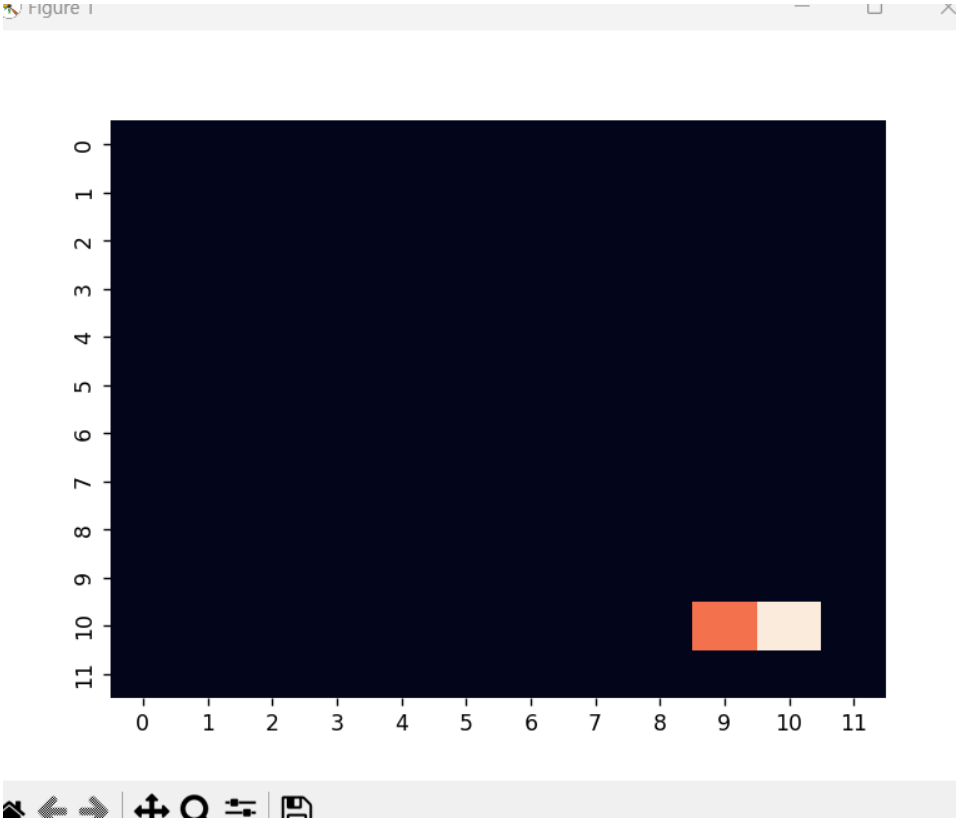


Resultados 2b

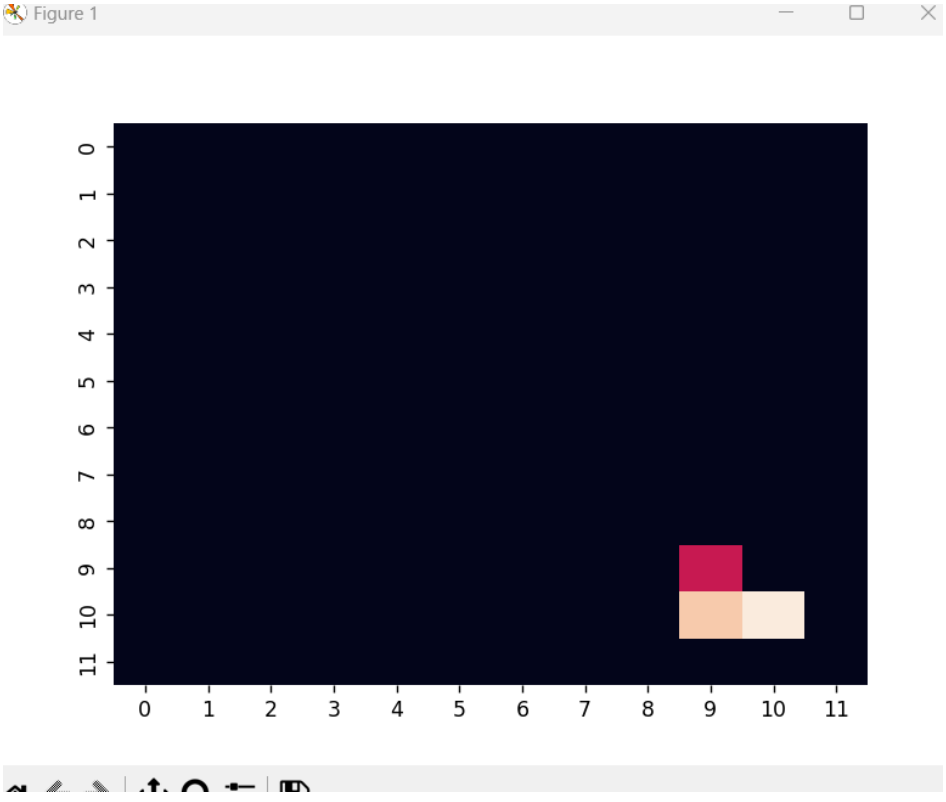
No exercício 2b a simulação foi alterada de maneira que escolha a melhor posição seguinte.

E para se obter resultados validos os runs foram alterados para 40 e os steps para 300.

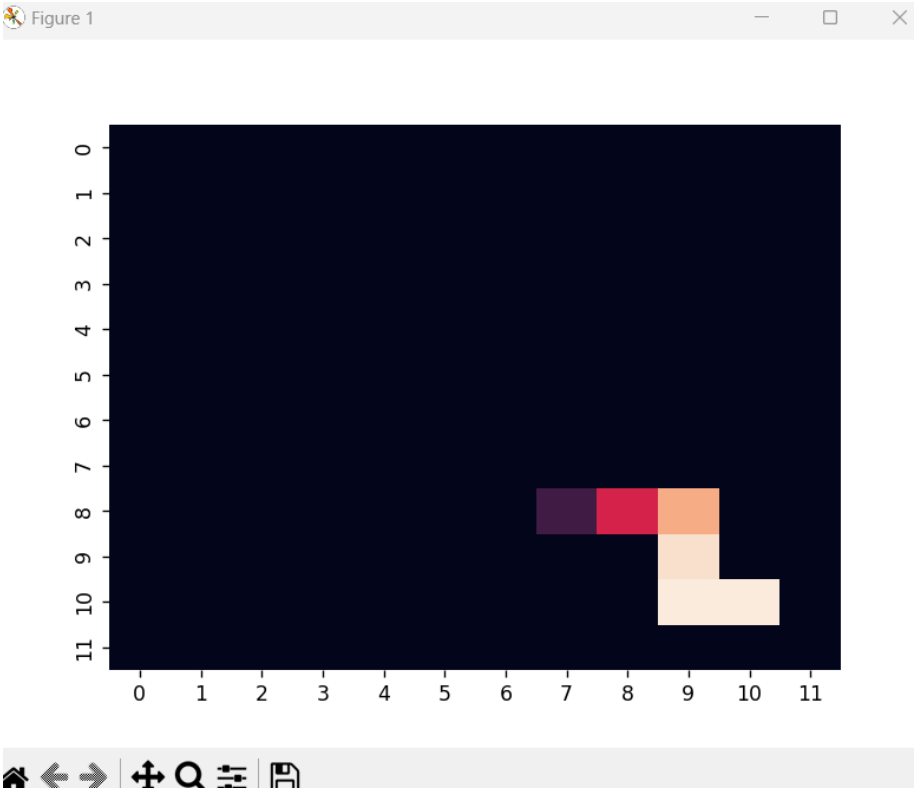
```
35  for r in range(runs):
36      currentState = initialState
37      countSteps = 0
38
39      st = time.time()
40      while currentState != finalState:
41          if countSteps >= steps:
42              countSteps = FAIL_FUN
43              break
44
45          #nextState = StateTransition(currentState, RandomAction()) #Exericio 2a
46          nextState = GetBestStepAction(matrix, currentState)#Exericio 2b
47          if IsParede(matrix, nextState) == False:
```



Resultado da primeira run.

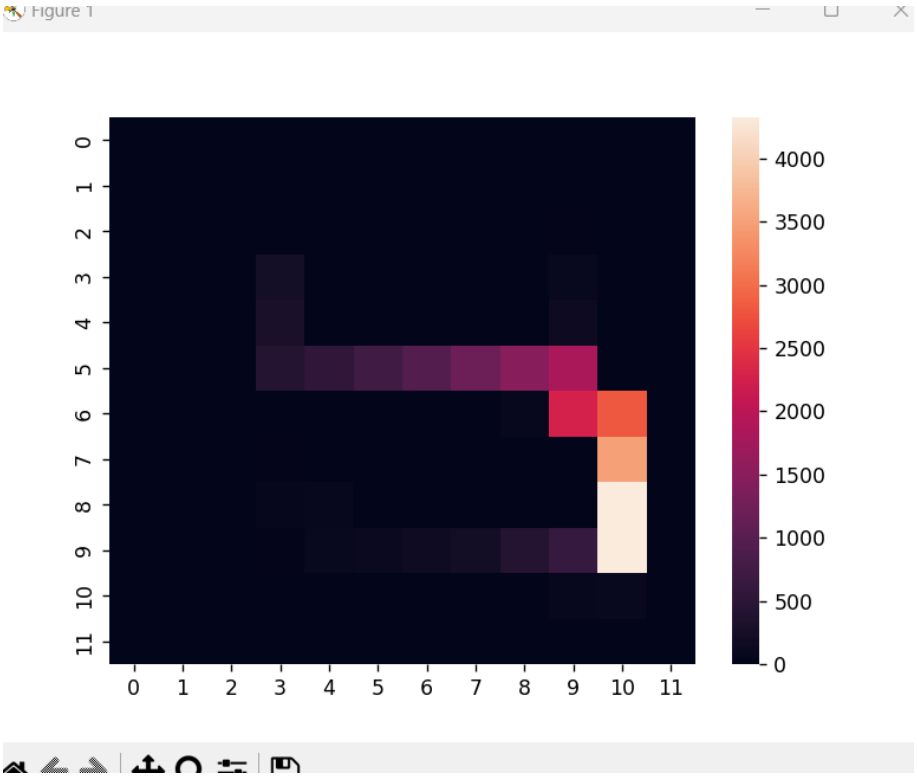


Resultado da segunda run.



Resultado quinta run.

Podemos concluir que comparativamente à simulação onde não se escolhe a melhor opção seguinte, o resultado do melhor caminho possível é mais rápido a ser calculado.

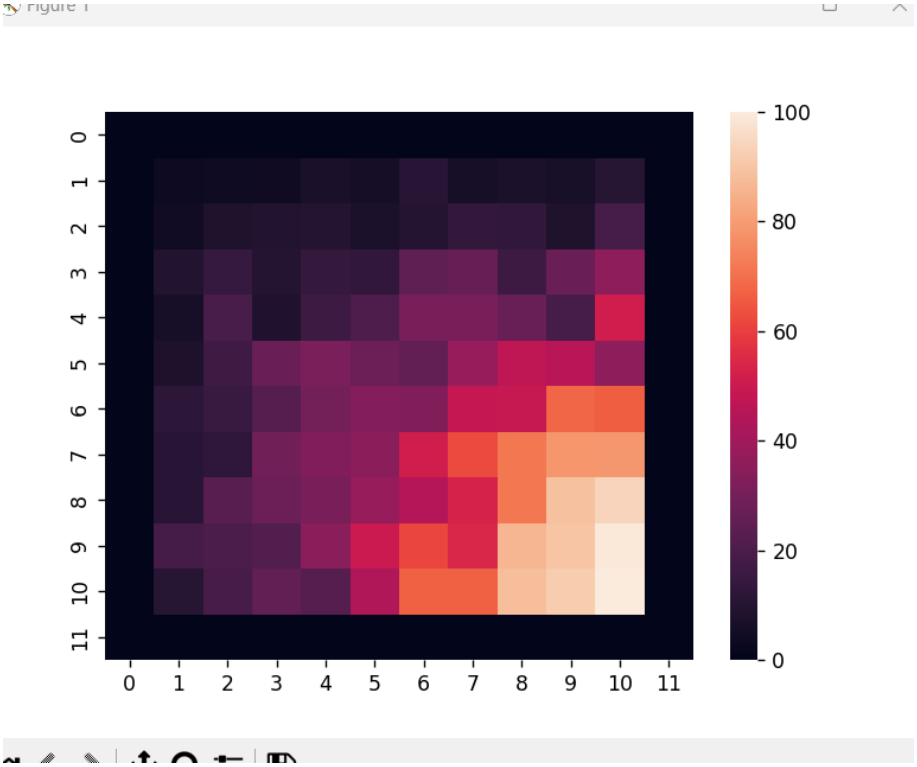


Resultado após os 40 runs com 300 steps cada uma.

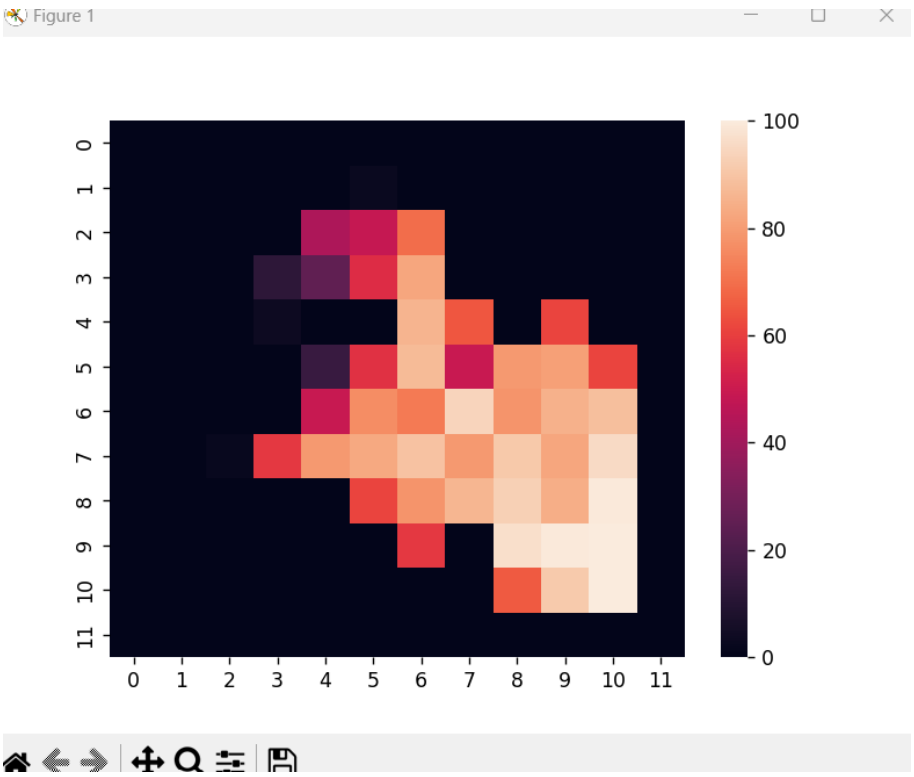
Exercício 3

No exercício 3 foi adicionada na simulação o cálculo probabilístico que permite escolher se a ação é aleatória ou não:

```
nextState = np.random.choice( [StateTransition(currentState, RandomAction()), GetBestStepAction(matrix, currentState)], p=[0.9, 0.1])
```



Neste gráfico a probabilidade de chamarmos o método StateTransition é de 90%, enquanto o método GetBestStepAction tem apenas 10%.



Ao alterarmos as probabilidades, tendo o StateTransition 10% e o GetBestStepAction de 90% de probabilidade, podemos observar diferenças.

Resultados exercício 3

Podemos concluir que quando o método que escolhe a melhor ação seguinte, tem mais probabilidade de acontecer, o caminho acaba por ser mais linear. Ao contrário, quando o método que escolhe uma ação aleatório, tem mais probabilidade de acontecer, o caminho fica menos linear dispersando-se.

Exercício 4

No exercício 4 foi criado um método para construir as paredes no tabuleiro e dar-lhes recompensas negativas:

```
148  def BuildInsideWalls(matrix, rows, cols):
149      for i in range(rows-2):
150          matrix[i][4].parede = True
151          matrix[i][4].reward = -0.0000005
152      aux = cols - 1
153      for i in range(rows-2):
154          matrix[aux][7].parede = True
155          matrix[aux][7].reward = -0.0000005
156          aux -= 1
```

Resultados exercício 4

Como podemos ver no heatmap as simulações chegaram ao caminho contornando as paredes criadas.