IA EM ROBÓTICA MÓVEL

Definição do problema:

O problema descrito nesse trabalho envolvia criar um ambiente onde um robô deveria chegar até um destino sem colidir com um obstáculo móvel na menor quantidade de passos possíveis. Deverá ser implementado um algoritmo de aprendizado por reforço, que proporciona ao robô conseguir por conta própria encontrar o melhor caminho a ser seguido que se encaixe nos requisitos desejados. A IDE utilizada para esse trabalho foi o Visual Studio Code, e a linguagem de programação escolhida foi o python.

O agente nesse caso é um robô móvel. O ambiente é um grid de tamanho 9x14 (9 linhas e 14 colunas), que inclui um obstáculo que se move para baixo entre 1 a 3 passos de forma aleatória, e caso chegue ao final da grid, ele volta na mesma coluna que estava, mas na primeira linha. As ações possíveis que o robô pode tomar incluem se movimentar livremente em todas as direções possíveis, apenas um passo de cada vez, ou ficar parado. As variáveis de estado são definidas pela posição do robô e posição do obstáculo. Nesse caso, o robô está localizado em (3,1) e o obstáculo em (1,7). Esses valores são contados a partir do canto superior esquerdo da grid, onde x é a linha e y a coluna. Eles também são definidos dentro do código, nas variáveis “pos_inicial_robo” e “pos_inicial_obstaculo”.

Definindo o algoritmo e as constantes:

Tendo como base essas informações, vamos definir um algoritmo para resolução do problema. O algoritmo escolhido foi o Q-Learning. Ele irá fazer com que nosso agente aprenda a tomar decisões melhores dentro de nosso ambiente, de uma forma que ele maximize a recompensa acumulada ao longo do tempo. Para isso, teremos que utilizar uma “Q-Table”, uma matriz que irá registrar os estados e ações para os valores de Q, que representam a possível recompensa de um estado ao executar uma determinada ação.

Vamos então definir as constantes de nosso algoritmo. Nesse caso, α (alpha), representa a taxa de aprendizado que determina o quanto valores novos de Q substituem os antigos valores. Já γ (gamma) representa a taxa de desconto que determina a importância de recompensas futuras. Também usaremos para recompensa os valores 100 quando ele chega no destino, -100 por cada colisão com o obstáculo, e -1 por cada passo.

Implementação do algoritmo:

Agora, tendo todas essas definições, vamos implementar nosso algoritmo. A função responsável pelo treinamento do algoritmo foi a seguinte:

```python
def treinar_q_learning(num_teste):
    global q_table
    pontuacoes = []
    for teste in range(num_teste):
        pos_robo = pos_inicial_robo
        pos_obstaculo = pos_inicial_obstaculo
        pontuacao = 0  # pontuacao inicial em cada teste

        while pos_robo != pos_destino:
            acao = selecionar_acao(pos_robo, pos_obstaculo)
            nova_pos_robo = obter_nova_posicao(pos_robo, acao)
            pos_obstaculo = mover_obstaculo(pos_obstaculo)
            recompensa = calcular_recompensa(nova_pos_robo, pos_obstaculo)
            pontuacao += recompensa  # Acumula a pontuação

            # Atualiza a q-table
            q_atual = q_table[pos_robo[0], pos_robo[1], pos_obstaculo[0], pos_obstaculo[1], acao]
            max_q_futuro = np.max(q_table[nova_pos_robo[0], nova_pos_robo[1], pos_obstaculo[0], pos_obstaculo[1]])
            q_table[pos_robo[0], pos_robo[1], pos_obstaculo[0], pos_obstaculo[1], acao] = \
                q_atual + alpha * (recompensa + gamma * max_q_futuro - q_atual)

            pos_robo = nova_pos_robo

        pontuacoes.append(pontuacao)
        print(f"Teste {teste + 1}: Pontuação {pontuacao}")

    return pontuacoes
```

Nessa função, nós primeiro percorremos um vetor que é igual ao número de treinamentos a serem feitos. No início de cada teste, resetamos as posições iniciais do robô e obstáculo, bem como zeramos a pontuação. Depois, entramos num laço que, enquanto o robô não chegar ao destino, ele irá executar os seguintes passos:

- Determinar uma ação a ser tomada
- Obter sua posição após a ação
- Definir a posição do obstáculo
- Calcular a recompensa da ação tomada
- Somar na pontuação final do teste
- Atualizar a matriz Q com as novas informações obtidas
- Calcula o valor máximo de Q para o próximo estado para todas as possíveis posições
- Atualiza o valor Qb atual usando a formula do Q-learning, que é a seguinte:

$$Q(s,a) = Q(s,a) + \alpha \left(r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right)$$

- Por fim, atualiza a posição do robô.

Temos também a função que testar o agente após o treinamento ser concluído. Essa função usa a matriz Q treinada para determinar as ações do agente e então exibir a trajetória do robô em uma animação. Essa função ficou da seguinte forma:

```python
def testar_q_learning():
    pos_robo = pos_inicial_robo
    pos_obstaculo = pos_inicial_obstaculo

    caminho_robo = [pos_robo]
    caminho_obstaculo = [pos_obstaculo]

    while pos_robo != pos_destino:
        acao = selecionar_acao(pos_robo, pos_obstaculo)
        pos_robo = obter_nova_posicao(pos_robo, acao)
        pos_obstaculo = mover_obstaculo(pos_obstaculo)

        caminho_robo.append(pos_robo)
        caminho_obstaculo.append(pos_obstaculo)

    # Visualização do caminho
    fig, ax = plt.subplots()
```

```python
def update(num):
    (variable) ax: Any
                                colunas)
    ax.set_ylim(-1, num_linhas)

    # Desenha o robô
    robo = patches.Circle((caminho_robo[num][1], num_linhas - 1 - caminho_robo[num][0]), 0.3, color='blue')
    ax.add_patch(robo)

    # Desenha o obstáculo
    obstaculo = patches.Rectangle((caminho_obstaculo[num][1] - 0.5, num_linhas - 1 - caminho_obstaculo[num][0] - 0.5), 1, 1,
    ax.add_patch(obstaculo)

    # Desenha o destino
    ax.plot(pos_destino[1], num_linhas - 1 - pos_destino[0], 'kx', markersize=15)

# Cria a animação
ani = FuncAnimation(fig, update, frames=len(caminho_robo), interval=500, repeat=False)
plt.show()
```

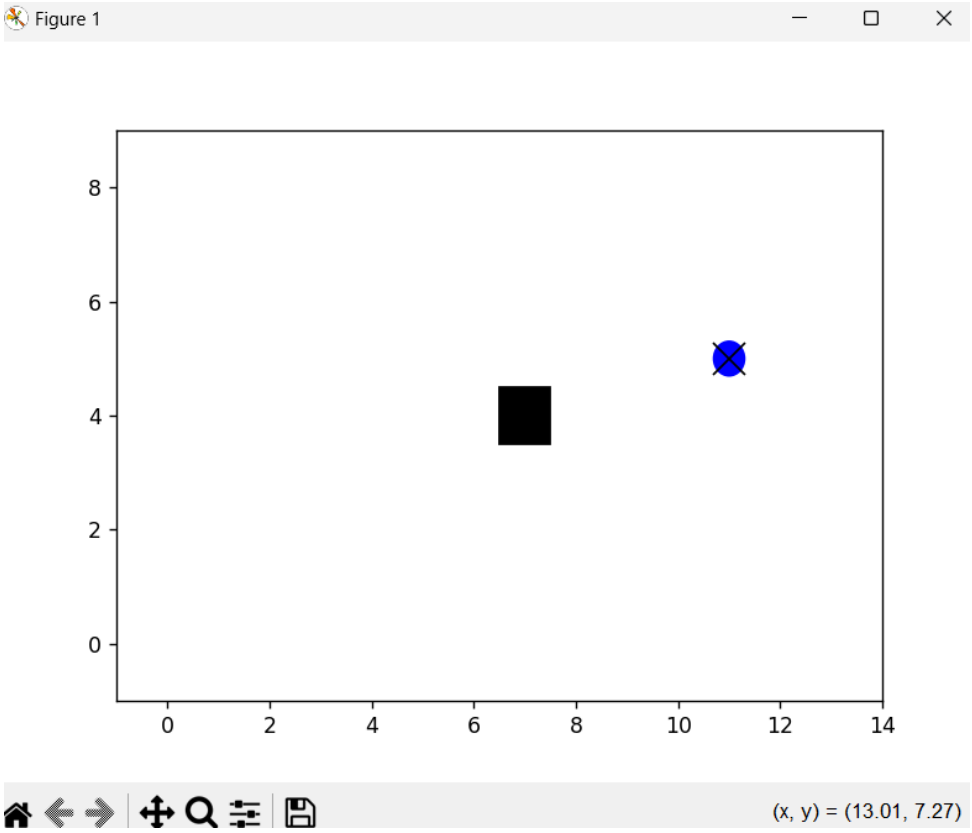
A primeira imagem inicializa as variáveis de posição e inicia um loop que dura todo o teste final. Temos então a definição do ambiente onde poderemos ver o teste ser executado. Utilizamos a biblioteca matplotlib para isso. Na segunda imagem, estamos utilizando a função update para atualizar a posição de cada elemento da imagem, neste caso, o robô representado por um círculo azul, o obstáculo por um quadrado preto e o destino por um X preto. Logo após a função update, temos a definição da nossa animação, que sera atualizada a cada meio segundo.

Essas são as maiores e mais importantes funções em nosso código, mas vamos passar rapidamente pelas outras funções presentes:

- Obter_nova_posicao: função responsável por obter a nova posição do robô dentro do grid
- Mover_obstaculo: função responsável por mover o obstáculo entre 1 a 3 passos, e retornar o obstáculo para a linha do topo caso ele ultrapasse a última linha
- Selecionar_acao: função responsável por selecionar a ação a ser tomada pelo robô
- Calcular_recompensa: função que define o valor de cada estado possível do robô
- Main: executa o programa

Testes com diversos valores das constantes:

Vamos agora iniciar a etapa pratica e ver como o algoritmo se sai para resolver o problema. O algoritmo primeiramente executa os testes, mostrando no terminal o número do teste em que se encontra, bem como a pontuação obtida no final daquele teste. Após todos o treinamento ser concluído, ele executara uma simulação utilizando o matplotlib, onde o usuário poderá ver o robô se movendo pelo ambiente, da seguinte forma:

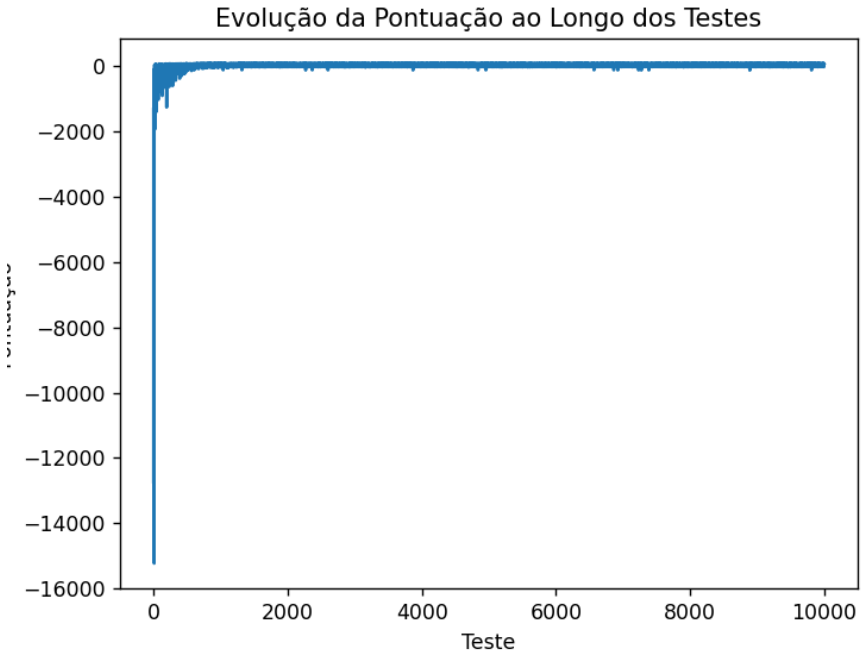


Neste caso, o algoritmo já encontrou o caminho e finalizou seus testes com sucesso.

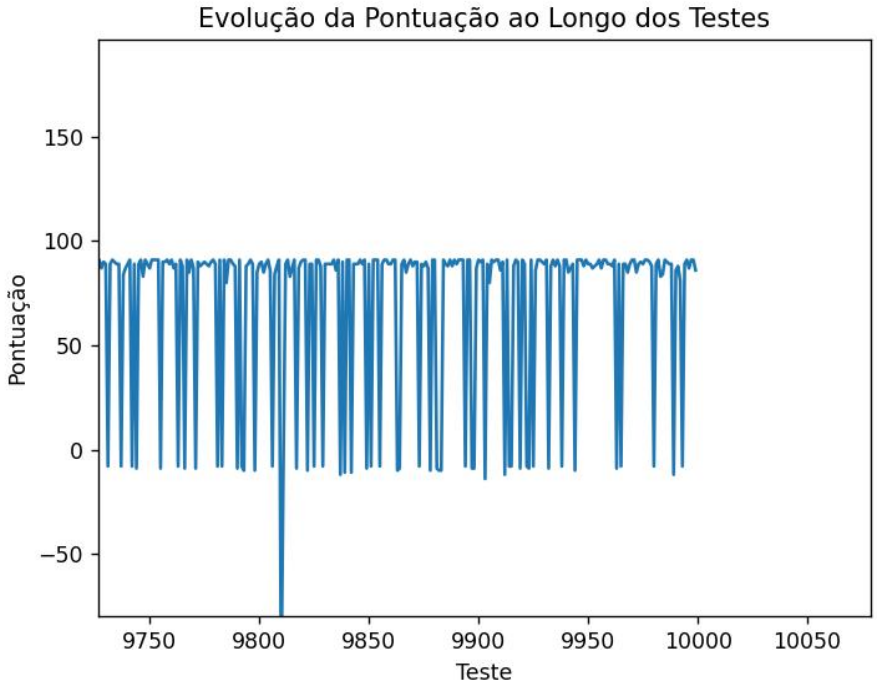
Após algumas pesquisas, foi definido que os melhores valores para começar os testes seriam com:

- Alpha = 0.1
- Gamma = 0.9

Dessa forma, o resultado obtido, após 10.000 testes nessas constantes, foi o seguinte:

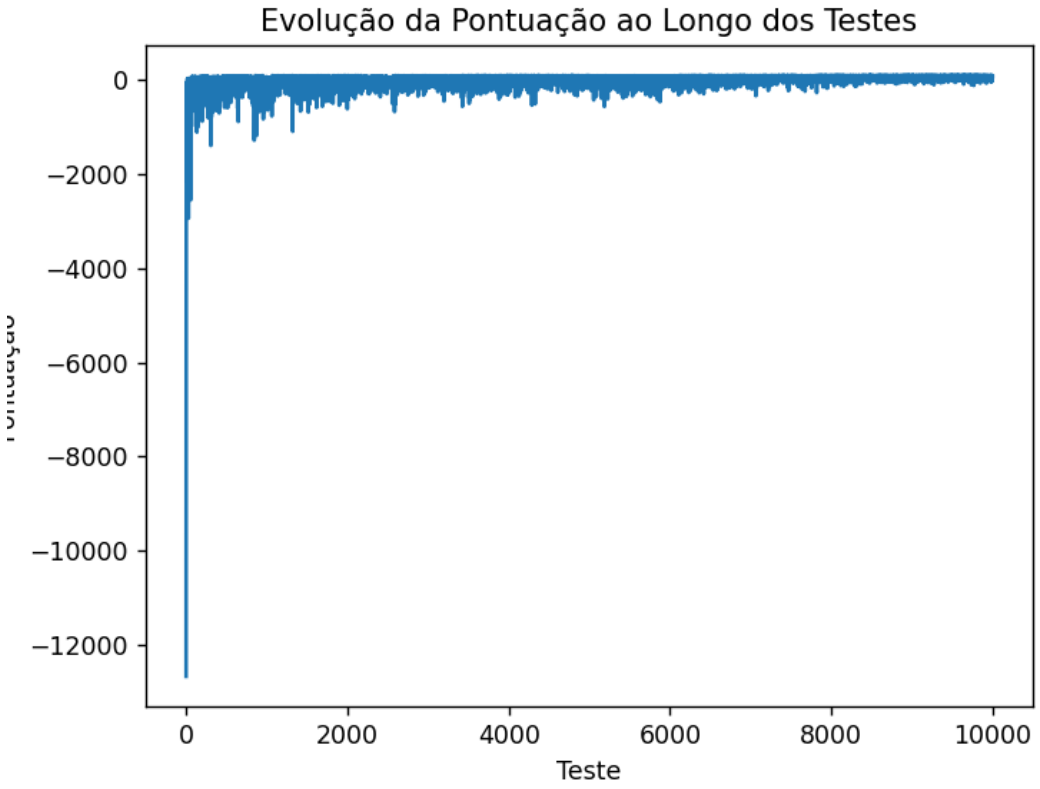


Assim, observamos que o algoritmo rapidamente consegue aprender a obter caminhos melhores que proporcionam uma recompensa maior, consequentemente, mais eficiente. Se aproximarmos o gráfico para os últimos testes feitos, encontraremos o seguinte resultado:

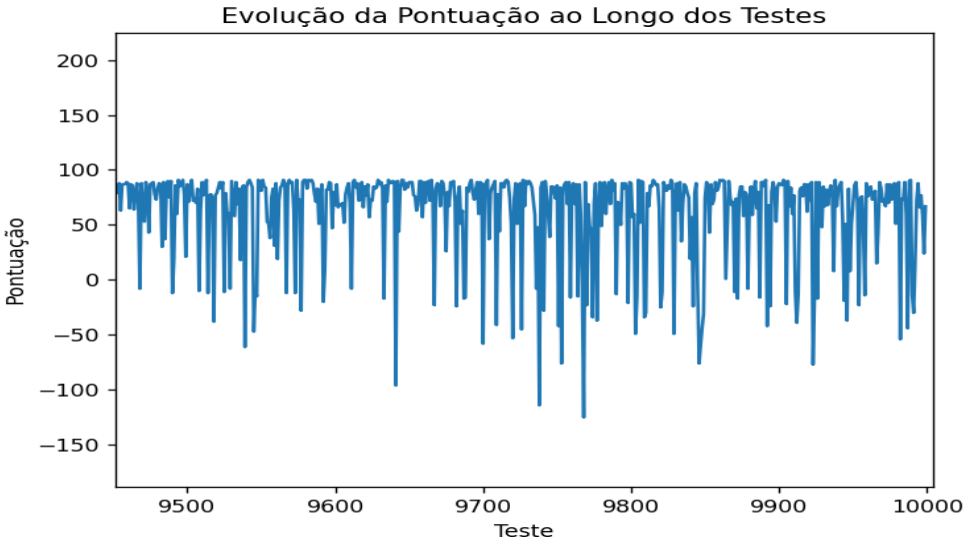


Como podemos observar, na grande maioria dos casos, o robô consegue recompensas maiores do que zero, ou seja, ele faz todo o trajeto sem colidir com o obstáculo, numa quantidade de passos razoável.

Agora, vamos alterar o valor de alpha para 0.01, e manter o valor de gamma como 0.9. Vamos ver o que acontece no gráfico:

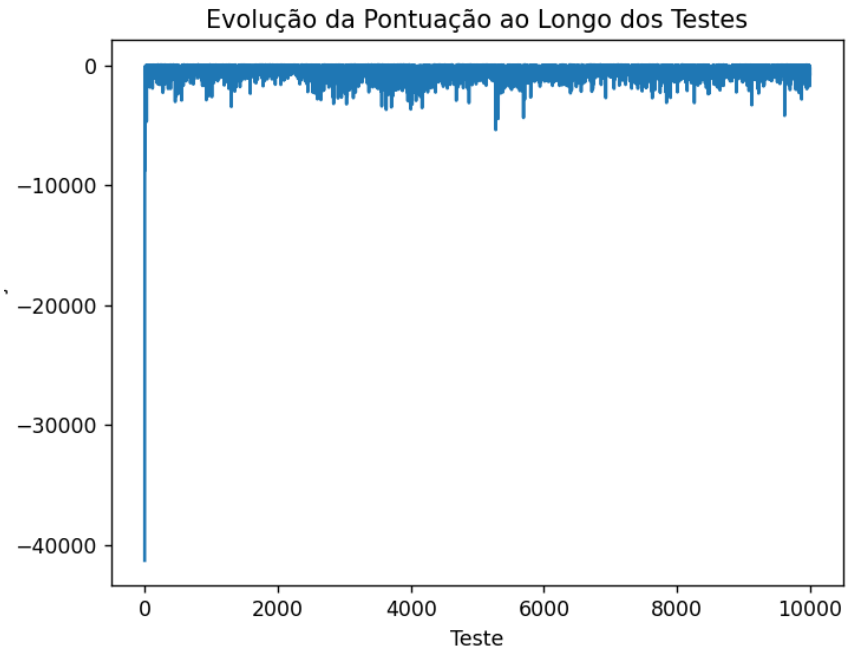


Como podemos observar, esse gráfico apresentou resultados mais inconstantes utilizando alpha como 0.01 do que alpha 0.1, o que mostra que o valor de alpha é importante para garantir uma boa taxa de aprendizado no algoritmo.

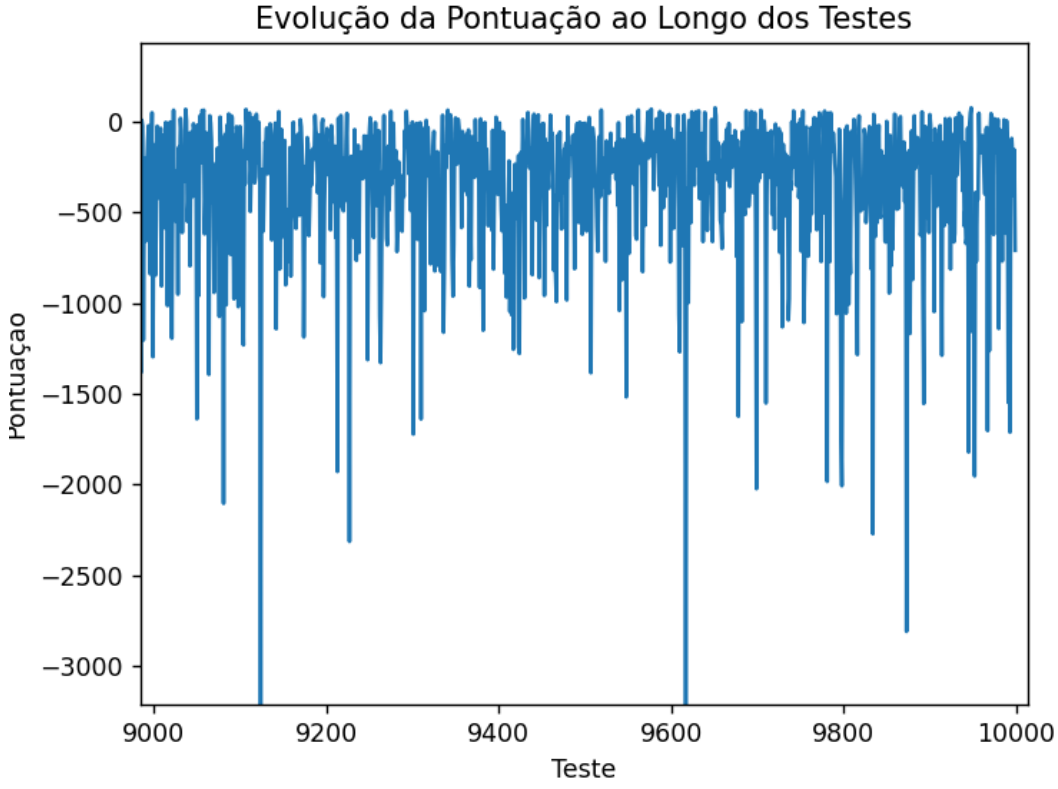


Podemos ver que mesmo nos últimos casos teste, a pontuação obtida ainda era inconstante, e apresentava vários valores negativos muito abaixo do desejado.

Agora, vamos voltar com o valor de alpha para 0.1 e mudar gamma para 0 e ver os resultados dessa mudança:



Como o valor de gamma é zero, o aprendizado será mais lento pois o futuro deixa de ter alguma importância para o nosso algoritmo.



Como podemos ver, por mais que exista alguns poucos testes com pontuação positiva, em sua grande maioria os testes apresentam falhas muito altas, nos mostrando o valor dos retornos futuros no aprendizado presente.

Dessa forma, vimos que o melhor caso foi quando equilibramos o valor de alpha(0.1) e gamma(0.9). E dessa forma, encontramos um algoritmo que consegue encontrar uma baixa margem de erro para o robô, com uma alta pontuação sendo obtida ao longo dos casos, de uma forma rápida e eficiente.