Obtenção da melhor rota para problemas de navegação em grids quadradas: Implementação do algoritmo Q-learning.

Gustavo Caldas José Gabriel Lucas Feitosa Ricardo Nabuco Vinícius Vasconi

Departamento de Computação
Universidade Federal de Sergipe

23 de fevereiro de 2026

Resumo

Este projeto é uma demonstração interativa do algoritmo de Aprendizado por Reforço Q-Learning aplicado a um ambiente de mineração em grid 2D. O objetivo é treinar um agente autônomo para coletar todos os minérios valiosos espalhados pelo mapa da forma mais eficiente possível, evitando rotas vazias e obstáculos.

Sumário

---

1 Introdução

O jogo Minecraft possui como uma de suas principais mecânicas a mineração, atividade que envolve exploração do ambiente para obtenção de recursos. Quando realizada sem

planejamento, essa tarefa pode demandar tempo excessivo e resultar em trajetórias ineficientes.

Nesse contexto, utilizamos os métodos do Aprendizado por Reforço para modelar agentes capazes de aprender estratégias ótimas a partir da interação com o ambiente. Este trabalho propõe a aplicação do algoritmo *Q-learning* para criação de um agente que faça o caminho mais eficiente até um objetivo, inspirado no processo de mineração do jogo.

2 O problema e a modelagem

O problema escolhido foi a determinação da rota mais eficiente para mineiração com pedras representando obstáculos (superáveis) para o minerador.

O problema foi modelado com uma grade bidimensional quadrada (grid 2D), onde cada posição da grade corresponde a um estado possível do agente. O objetivo final é minerar todas as células de minério obtendo a maior pontuação ao final do percurso.

Para penalizar o tempo gasto na mina, atribuímos às pedras uma recompensa negativa, forçando que o agente encontre o percurso mais eficiente. Segue a tabela com os tipos de células definidas, sua representação na interface e as respectivas recompensas:

Tabela 1: Definição de recompensas do ambiente

Elemento	Recompensa
Agente (posição inicial)	0
Pedra	-1
Ferro	+10
Redstone	+20
Ouro	+30
Diamante	+60

Definimos também o conjunto de ações disponíveis ao agente como o conjunto discreto composto por quatro movimentos: direita, esquerda, cima e baixo, penalizando de forma mais severa caso o agente execute um movimento proibido. Temos portanto o que precisamos para definir a estrutura de um porcesso de decisão de markov: Um conjunto de estados (posições na grade); Um conjunto de ações possíveis em cada estado; Uma função de transição implícita, determinada pelos movimentos na grade; Uma função de recompensa.

3 Algoritmo Q-Learning como solução

Como falamos, a descrição do problema se adequa ao processo de decisão markoviana, de modo que podemos empregar o algoritmo Q-learning para treinar o agente. O algoritmo estima a chamada função Q, que associa a cada par estado-ação um valor esperado de recompensa futura acumulada. Esses valores são armazenados em uma estrutura conhecida como Q-table e atualizados iterativamente por meio da interação do agente com o ambiente de acordo com a fórmula:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right] \tag{1}$$

onde:

- s_t é o estado atual;
- a_t é a ação escolhida no instante t;
- r_{t+1} é a recompensa recebida após executar a ação;
- $\alpha \in (0, 1]$ é a taxa de aprendizado;
- $\gamma \in [0, 1]$ é o fator de desconto;
- $\max_a Q(s_{t+1}, a)$ representa a melhor recompensa futura estimada.

A Figura abaixo ilustra o ciclo fundamental do Aprendizado por Reforço. O agente observa o estado atual do ambiente e, com base em sua política, seleciona uma ação. Essa ação modifica o estado do ambiente, que por sua vez retorna uma recompensa e um novo estado ao agente.

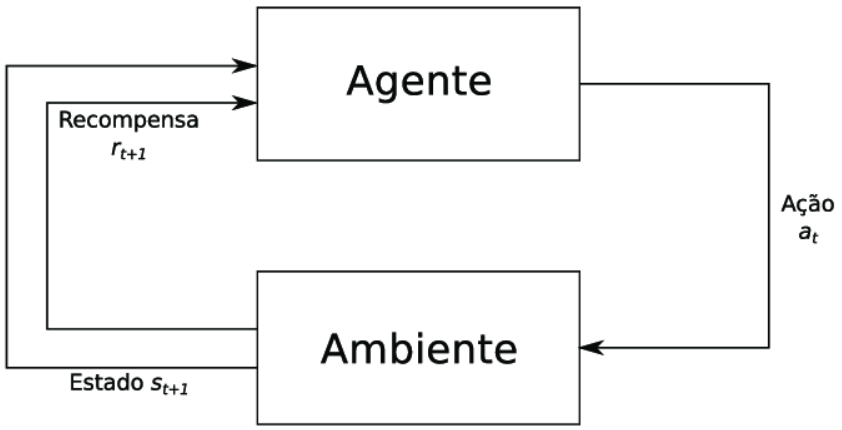

Figura 1: Modelo de interação agente-ambiente no Aprendizado por Reforço [1]

Após múltiplos episódios de treinamento, os valores convergem para uma aproximação da função ótima (ver [2]), permitindo a extração de uma política eficiente. Assim, o agente aprende automaticamente o melhor trajeto até o objetivo, otimizando o processo de mineração no cenário modelado.

4 Implementação

4.1 Inicialização dos estados

O primeiro passo foi definir a Q-Table, e a estrutura escolhida foi um dicionário. Nesse dicionário, as chaves serão os possíveis estados e os valores são outros dicinários que contém o par ação recompensa. Esse modelo facilita a consulta no momento do treinamento pois permite para cada estado consultar diretamente a recompensa de cada ação.

A função abaixo recebe a referência para a tabela, e um estado (coordenada da grid) e verifica se o estado já está contido. Em caso positivo, ela apenas ignora a inserção do estado preservando os valores atuais. Em caso negativo, ela instancia uma nova chave para o estado descoberto e atribui 0 como recompensa de cada ação, sinalizando que o estado ainda não foi devidamente explorado. O código referente à essa função segue:

Listing 1: Adicionar estados possíveis na Q-table

```
def inicializar_estado(tabela_q, estado):
    if estado not in tabela_q:
        tabela_q[estado] = {acao: 0.0 for acao in
           acoes_possiveis}
```

4.2 As ações do agente

A próxima função é responsável por coordenar a exploração do agente, essencial para adicionar os estados mediante exploração do ambiente, assim como apresentado na Figura 1.

A função recebe inicialmente o estado atual, a ação desejada, e o ambiente. Usa a ação desejada para alterar o estado atual e avaliar se o estado atual é válido, ou seja, uma posição válida na grid. Se a posição for invalida, ele marca aquela ação com uma penalidade alta para que o agente não saia dos limites novamente. Se a posição for válida ele minera a célula e atualiza as recompensas conforme a tabela 1 que apresentamos.

O retorno da função consiste o próximo estado, composto pela nova posição e o mapa atualizado, a recompensa, e se o agente minerou. Note que a estrutura escolhida para representar o mapa foi uma tupla de tuplas em razão do agente não conhecer previamente o ambiente.

Listing 2: A função de interação com o ambiente

```
def interagir_com_ambiente(estado, acao, grid_atual):
    x, y = estado[0], estado[1]

    if acao == 'cima': y -= 1
    elif acao == 'baixo': y += 1
    elif acao == 'esquerda': x -= 1
    elif acao == 'direita': x += 1

    if x < 0 or x >= len(grid_atual[0]) or y < 0 or y >= len(
       grid_atual):
        x, y = estado[0], estado[1]
        recompensa = -5
        minerou = False
    else:
        id_minerio = grid_atual[y][x]
        recompensa = RECOMPENSAS[id_minerio]
        minerou = (id_minerio != 0)
        if minerou:
            grid_atual[y][x] = 0

    mapa_estado = tuple(tuple(linha) for linha in grid_atual)
    return (x, y, mapa_estado), recompensa, minerou
```

4.3 A decisão

A próxima função implementa a estratégia de seleção de ações baseada no método ϵ-greedy. Inicializamos o estado atual caso não esteja devidamente inicializado na Q-table, e em

seguida decide entre exploração e explotação com base no parâmetro ϵ. Com probabilidade ϵ, o agente realiza exploraçãom, caso contrário o agente realiza explotação, escolhendo a ação que maximiza o valor da Q-table no estado atual.

Listing 3: A função de decisão

```
def escolher_acao(tabela_q, estado, epsilon):
    inicializar_estado(tabela_q, estado)
    if random.random() < epsilon:
        return random.choice(acoes_possiveis)
    else:
        return max(tabela_q[estado], key=tabela_q[estado].get)
```

4.4 A função de treinamento

Para treinar o agente, iniciamos a Q-table com um dicionário vazio representando que não houve exploração. Definimos a taxa de aprendizado α, o fator de desconto γ, o parâmetro de exploração ϵ e sua taxa de decaimento.

Cada episódio do treinamento é reinicializado a partir da representação atual do mapa, construído mediante exploração do agente. Durante a execução de cada episódio, o agente interage até coletar todos os minérios.

Chamamos as funções já discutidas anteriormente e atualizamos a Q-table até encerrar o número de episódios. Segue o código:

Listing 4: A função de treinamento

```
    def treinar_agente(grid_inicial, episodios, total_minerios)
        :
    tabela_q = {}
    alfa, gama, epsilon, decaimento = 0.1, 0.9, 1.0, 0.9999

    tamanho_grid = len(grid_inicial)

    for _ in range(episodios):
        grid_atual = [linha[:] for linha in grid_inicial]
        mapa_estado = tuple(tuple(linha) for linha in
            grid_atual)
        x_inicial = random.randint(0, tamanho_grid - 1)
        y_inicial = random.randint(0, tamanho_grid - 1)
        estado_atual = (x_inicial, y_inicial, mapa_estado)
        terminou = False
        minerios_coletados = 0

        while not terminou:
            acao = escolher_acao(tabela_q, estado_atual,
                epsilon)
            proximo_estado, recompensa, minerou =
                interagir_com_ambiente(estado_atual, acao,
                grid_atual)

            if minerou:
                minerios_coletados += 1
```

```
22              if total_minerios > 0 and minerios_coletados ==
                    total_minerios:
23                  terminou = True
24
25              inicializar_estado(tabela_q, estado_atual)
26              inicializar_estado(tabela_q, proximo_estado)
27
28              q_atual = tabela_q[estado_atual][acao]
29              max_q_futuro = max(tabela_q[proximo_estado].values
                    ())
30              tabela_q[estado_atual][acao] = q_atual + alfa * (
                    recompensa + gama * max_q_futuro - q_atual)
31
32              estado_atual = proximo_estado
33
34          epsilon *= decaimento
35      return tabela_q
```

5 Conclusão

O produto final obtido foi o algoritmo de obtenção da Q-Table que pode ser reaproveitado para outros problemas similares, com a vantagem de não depender de bibliotecas extras. Além disso, produzimos uma interface gráfica interativa disponibilizadda na plataforma hugging faces (ver [3] para visualização do agente.

Adições futuras desse projeto incluem se aproveitar da escolha das estruturas para explorar mapas não quadrados, adicionar diferentes tipos de elementos no mapa como água e lava, e adaptar o algoritmo para 3 dimensões. Mais ambiciosamente, expandir o algoritmo para rodar juntamente ao jogo, e fazer um comparativo dos resultados em um cenário PVE (*Player versus Enviroment*)

Referências

[1] Einar Cesar Santos, *Figura 3: Modelo de Aprendizado por Reforço*, https://www.researchgate.net/figure/Figura-33-Modelo-de-aprendizado-por-reforco-Extraido-traduzido-e-adaptado-de-89_fig7_335311074, Acesso em: 22 fev. 2026, 2019.

[2] C. J. C. H. Watkins e P. Dayan, "Technical note: Q-learning," *Machine Learning*, v. 8, pp. 279–292, 1992, Acesso em: 22 fev. 2026. endereço: http://www.gatsby.ucl.ac.uk/~dayan/papers/cjch.pdf.

[3] LucasFe0162, *Q-learning Miner - Hugging Face Space*, https://huggingface.co/spaces/LucasFe0162/Q-learning-mineracao-minecraft, Acesso em: 22 fev. 2026, 2025.