lab_2_iteracao_de_politica

May 11, 2025

1 Experimento 2: Iteração de política

[9]:
```
# Importações
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from matplotlib.colors import ListedColormap
import numpy as np
import seaborn as sns
```

[10]:
```
# Ambiente: Navegação no Labirinto (gridworld)

class AmbienteNavegacaoLabirinto:
    def __init__(self, world_size, bad_states, target_states,␣
 ↪allow_bad_entry=False, rewards=[-1, -1, 1, 0]):
        """
        Inicializa o ambiente de navegação em labirinto.

        Parâmetros:
        - world_size: tupla (n_linhas, n_colunas)
        - bad_states: lista de tuplas com coordenadas de estados penalizados
        - target_states: lista de tuplas com coordenadas dos estados de objetivo
        - allow_bad_entry: bool, se False impede entrada em estados ruins␣
 ↪(rebote)
        - rewards: lista de recompensas com [r_boundary, r_bad, r_target,␣
 ↪r_other]
        """
        self.n_rows, self.n_cols = world_size   # dimensões da grade do␣
 ↪labirinto
        self.bad_states = set(bad_states)       # estados com penalidade alta
        self.target_states = set(target_states) # estados com recompensa alta
        self.allow_bad_entry = allow_bad_entry  # se o agente pode entrar em␣
 ↪estados ruins

        # Recompensas definidas para cada tipo de transição
        self.r_boundary = rewards[0]  # tentar sair da grade
        self.r_bad = rewards[1]       # transição para estado ruim
```

```
        self.r_target = rewards[2]     # transição para estado alvo
        self.r_other = rewards[3]      # demais transições

        # Espaço de ações: dicionário com deslocamentos (linha, coluna)
        self.action_space = {
            0: (-1, 0),  # cima
            1: (1, 0),   # baixo
            2: (0, -1),  # esquerda
            3: (0, 1),   # direita
            4: (0, 0)    # permanecer no mesmo estado
        }

        # Espaço de recompensas: lista de recompensas possíveis
        self.recompensas_possiveis = np.array(sorted(set(rewards)))
        self.reward_map = {r: i for i, r in enumerate(self.
↪recompensas_possiveis)}

        # número total de estados
        self.n_states = self.n_rows * self.n_cols

        # número total de ações
        self.n_actions = len(self.action_space)

        # número total de recompensas possíveis
        self.n_rewards = self.recompensas_possiveis.shape[0]

        # Tensor de probabilidades de transição: P(s'|s,a)
        self.state_transition_probabilities = np.zeros((self.n_states, self.
↪n_states, self.n_actions))

        # Tensor de probabilidade de recompensas: P(r|s,a)
        self.reward_probabilities = np.zeros((self.n_rewards, self.n_states,␣
↪self.n_actions))

        # Matriz de recompensa imediata
        self.recompensas_imediatas = np.zeros((self.n_states, self.n_actions))

        self.agent_pos = (0, 0)  # posição inicial do agente

        self._init_dynamics()  # inicializa as dinâmicas de transição e␣
↪recompensa


    def _init_dynamics(self):
        """
        Preenche as matrizes de transição e recompensa com base
        na estrutura do ambiente e regras de movimentação.
```

```
        """
        for indice_estado in range(self.n_states):
            estado_atual = self.index_to_state(indice_estado)

            for acao, (d_linha, d_coluna) in self.action_space.items():
                proxima_posicao = (estado_atual[0] + d_linha, estado_atual[1] +␣
↪d_coluna)

                # Verifica se o movimento é válido ou resulta em rebote
                if not self._in_bounds(proxima_posicao) or (not self.
↪allow_bad_entry and proxima_posicao in self.bad_states):
                    proximo_estado = estado_atual  # rebote: permanece no␣
↪estado atual
                else:
                    proximo_estado = proxima_posicao

                # Calcula a recompensa imediata da transição (s, a)
                recompensa = self._compute_reward(proxima_posicao)

                # Armazena a recompensa imediata na matriz
                self.recompensas_imediatas[indice_estado, acao] = recompensa

                # Ambiente determinístico
                indice_proximo = self.state_to_index(proximo_estado)
                self.state_transition_probabilities[indice_proximo,␣
↪indice_estado, acao] = 1.0  # registra probabilidade P(s'|s,a)
                indice_recompensa = self.reward_map[recompensa]
                self.reward_probabilities[indice_recompensa, indice_estado,␣
↪acao] = 1.0  # registra probabilidade P(r|s,a)


    def reset(self):
        """Reinicia a posição do agente para o estado inicial (0, 0)."""
        self.agent_pos = (0, 0)
        return self.agent_pos


    def step(self, acao):
        """
        Executa uma ação no ambiente e atualiza a posição do agente.

        Parâmetros:
        - acao: índice da ação a ser executada (0 a 4)

        Retorna:
        - nova posição do agente (linha, coluna)
        - recompensa recebida
```

```
        """
        d_linha, d_coluna = self.action_space[acao]
        linha_destino = self.agent_pos[0] + d_linha
        coluna_destino = self.agent_pos[1] + d_coluna
        destino = (linha_destino, coluna_destino)

        # Se movimento for inválido ou entrada proibida, permanece
        if not self._in_bounds(destino) or (not self.allow_bad_entry and␣
↪destino in self.bad_states):
            destino = self.agent_pos

        recompensa = self._compute_reward(destino)
        self.agent_pos = destino
        return self.agent_pos, recompensa


    def _in_bounds(self, posicao):
        """Verifica se uma posição está dentro dos limites do labirinto."""
        linha, coluna = posicao
        return 0 <= linha < self.n_rows and 0 <= coluna < self.n_cols


    def _compute_reward(self, destino):
        """
        Define a recompensa com base no destino proposto:
        - r_boundary: fora do grid
        - r_bad: célula ruim
        - r_target: célula alvo
        - r_other: demais casos
        """
        if not self._in_bounds(destino):
            return self.r_boundary
        elif destino in self.bad_states:
            return self.r_bad
        elif destino in self.target_states:
            return self.r_target
        else:
            return self.r_other


    def state_to_index(self, estado):
        """Converte coordenada (linha, coluna) para índice linear."""
        linha, coluna = estado
        return linha * self.n_cols + coluna


    def index_to_state(self, indice):
```

```
        """Converte índice linear para coordenada (linha, coluna)."""
        return divmod(indice, self.n_cols)  # (linha, coluna) = (indice // self.
↪n_cols, indice % self.n_cols)
```

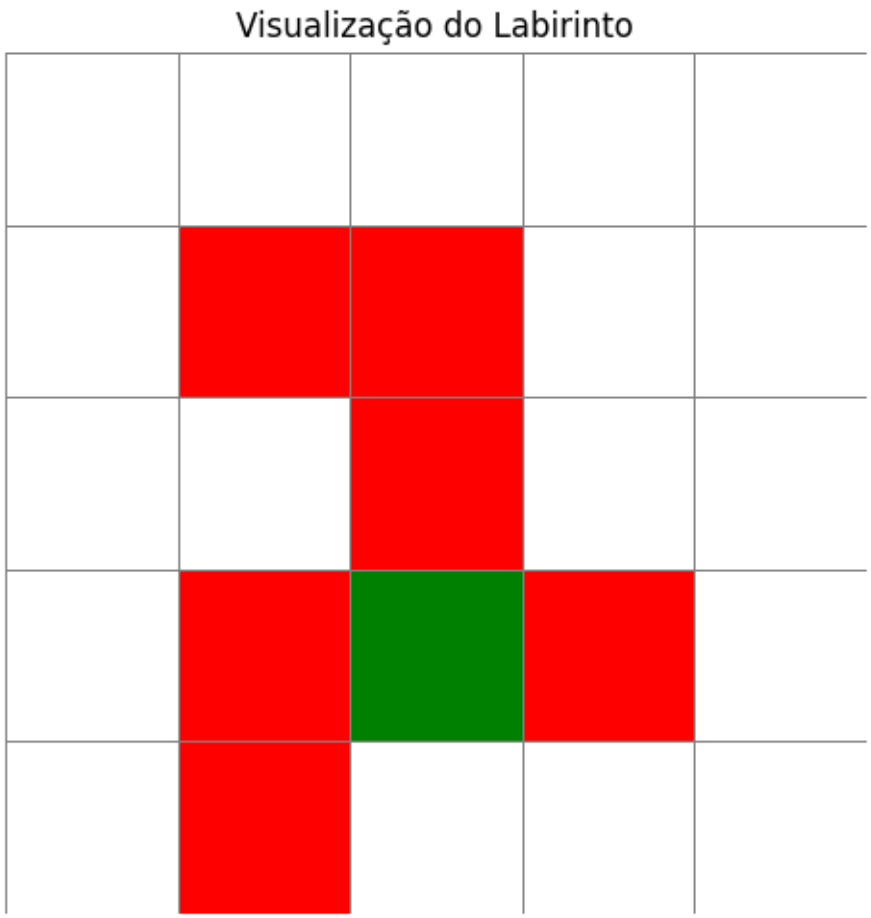
[11]:
```
# Funções auxiliares para visualização

def plot_policy(env, policy, ax=None):
    fig, ax = _prepare_grid(env, ax=ax)

    for (r, c), action in policy.items():
        x, y = c + 0.5, r + 0.5
        color = 'black'
        lw = 1.5

        if action == 0:
            ax.arrow(x, y, dx=0, dy=-0.3, head_width=0.2, head_length=0.2,␣
↪fc=color, ec=color, linewidth=lw)
        elif action == 1:
            ax.arrow(x, y, dx=0, dy=0.3, head_width=0.2, head_length=0.2,␣
↪fc=color, ec=color, linewidth=lw)
        elif action == 2:
            ax.arrow(x, y, dx=-0.3, dy=0, head_width=0.2, head_length=0.2,␣
↪fc=color, ec=color, linewidth=lw)
        elif action == 3:
            ax.arrow(x, y, dx=0.3, dy=0, head_width=0.2, head_length=0.2,␣
↪fc=color, ec=color, linewidth=lw)
        elif action == 4:
            circ = patches.Circle((x, y), 0.1, edgecolor=color,␣
↪facecolor='none', linewidth=lw)
            ax.add_patch(circ)

    ax.set_title("Política")
    plt.show()

    return

def _prepare_grid(env, ax=None, draw_cells=True):
    if ax is None:
        fig, ax = plt.subplots(figsize=(env.n_cols, env.n_rows))
    ax.set_xlim(0, env.n_cols)
    ax.set_ylim(0, env.n_rows)
    ax.set_xticks(np.arange(0, env.n_cols + 1, 1))
    ax.set_yticks(np.arange(0, env.n_rows + 1, 1))
    ax.grid(True)
    ax.set_aspect('equal')
    ax.invert_yaxis()
```

```
    if draw_cells:
        for r in range(env.n_rows):
            for c in range(env.n_cols):
                cell = (r, c)
                if cell in env.bad_states:
                    color = 'red'
                elif cell in env.target_states:
                    color = 'green'
                else:
                    color = 'white'
                rect = patches.Rectangle(xy=(c, r), width=1, height=1,␣
↪facecolor=color, edgecolor='gray')
                ax.add_patch(rect)

    return (None, ax) if ax else (fig, ax)


def plot_valores_de_estado(valores_estado, ambiente):
    plt.figure(figsize=(ambiente.n_rows, ambiente.n_cols))
    ax = sns.heatmap(
        data=valores_estado.reshape(ambiente.n_rows, ambiente.n_cols),
        annot=True,
        fmt='.1f',
        cmap='bwr',
        square=True,
        cbar=True,
        linewidths=0.5,
        linecolor='gray',
    )
    ax.set_title(r"Valores de Estado (V(s))")
    plt.tight_layout()
    plt.show()


def plot_valores_de_acao(valores_de_acao):
    Q_transposta = valores_de_acao.T
    n_acoes, n_estados = Q_transposta.shape

    plt.figure(figsize=(n_estados, n_acoes))
    ax = sns.heatmap(
        Q_transposta,
        annot=True,
        fmt='.1f',
        cmap='bwr',
        cbar=True,
        square=False,
        linewidths=0.5,
```

```
        linecolor='gray'
    )
    # Rótulos das colunas (estados)
    ax.set_xticks(np.arange(n_estados) + 0.5)
    ax.set_xticklabels([f"s{i}" for i in range(n_estados)], rotation=0)

    # Rótulos das linhas (ações)
    ax.set_yticks(np.arange(n_acoes) + 0.5)
    ax.set_yticklabels([f"a{i}" for i in range(n_acoes)], rotation=0)

    ax.set_xlabel(r"Estados")
    ax.set_ylabel(r"Ações")
    ax.set_title(r"Valores de ação (Q(s, a) transposta)")
    plt.tight_layout()
    plt.show()


def plot_labirinto(ambiente):
    """
    Visualiza o labirinto usando seaborn.heatmap sem ticks nos eixos.

    Representa:
    - Estado neutro: branco
    - Estado ruim: vermelho
    - Estado alvo: verde
    """
    # Cria matriz com valores padrão (0 = neutro)
    matriz = np.zeros((ambiente.n_rows, ambiente.n_cols), dtype=int)

    # Marca os estados ruins como 1
    for (r, c) in ambiente.bad_states:
        matriz[r, c] = 1

    # Marca os estados alvo como 2
    for (r, c) in ambiente.target_states:
        matriz[r, c] = 2

    # Mapa de cores: branco = neutro, vermelho = ruim, verde = alvo
    cmap = ListedColormap(["white", "red", "green"])

    plt.figure(figsize=(ambiente.n_cols, ambiente.n_rows))
    ax = sns.heatmap(
        matriz,
        cmap=cmap,
        cbar=False,
        linewidths=0.5,
        linecolor='gray',
```

```
        square=True
    )

    # Remove todos os ticks e labels
    ax.set_xticks([])
    ax.set_yticks([])
    ax.set_xticklabels([])
    ax.set_yticklabels([])

    ax.set_title("Visualização do Labirinto")
    plt.tight_layout()
    plt.show()
```

[12]:
```
# Instancia o ambiente
ambiente = AmbienteNavegacaoLabirinto(
        world_size=(5, 5),
        bad_states=[(1, 1), (1, 2), (2, 2), (3, 1), (3, 3), (4, 1)],
        target_states=[(3, 2)],
        allow_bad_entry=True,
        rewards=[-1, -10, 1, 0]
    )
plot_labirinto(ambiente)
```



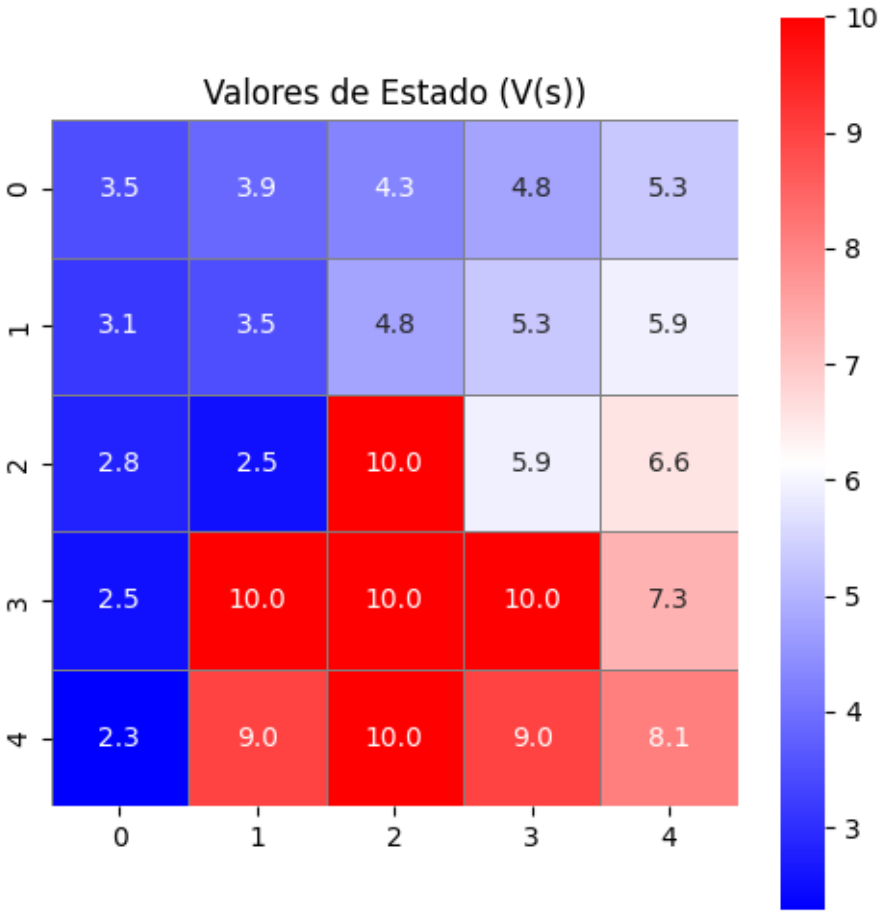
[13]:
```
def iteracao_de_politica(ambiente, gamma=0.9, theta=1e-6, max_iteracoes=1000):
    """
    Implementa o algoritmo de Iteração de Política para encontrar a política ótima.

    Parâmetros:
    - ambiente: instância da classe AmbienteNavegacaoLabirinto
    - gamma: fator de desconto (0 < gamma <= 1)
    - theta: limiar mínimo de variação para considerar convergência
    - max_iteracoes: número máximo de iterações de melhoria de política

    Retorna:
     - vetor de valores de estado V (numpy array) para todos os estados
     - matriz de valores de ação Q (numpy array) para todos os pares (estado, ação)
     - política ótima (dicionário de estado para ação)
    """
```

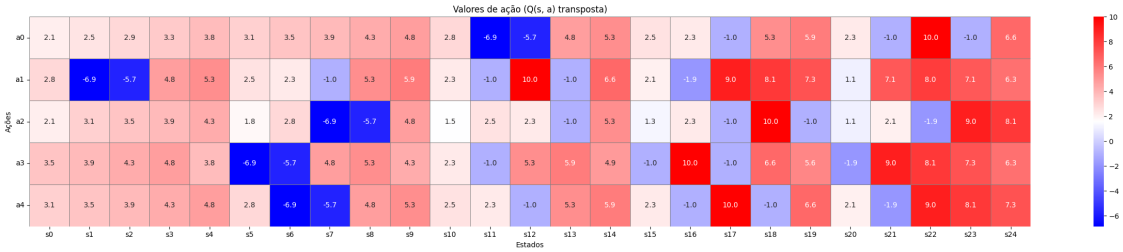
```
    # Informações úteis do ambiente:
    # número de estados: ambiente.n_states
    # número de ações: ambiente.n_actions
    # número de recompensas (únicas): ambiente.n_rewards
    # Tensor de probabilidade de recompensas: P(r|s,a): ambiente.
↪reward_probabilities shape=(ambiente.n_rewards, ambiente.n_states, ambiente.
↪n_actions)
    # recompensas: ambiente.recompensas_possiveis (ambiente.
↪recompensas_possiveis[i] com probabilidade ambiente.
↪reward_probabilities[i,s,a])
    # Tensor de probabilidades de transição: P(s'|s,a): ambiente.
↪state_transition_probabilities shape=(ambiente.n_states, ambiente.n_states,␣
↪ambiente.n_actions)
    # Converte coordenada (linha, coluna) para índice linear: ambiente.
↪state_to_index(estado)
    # Converte índice linear para coordenada (linha, coluna): ambiente.
↪index_to_state(indice)

    n_estados = ambiente.n_states
    n_acoes = ambiente.n_actions

    politica = {
        ambiente.index_to_state(s): np.random.choice(n_acoes)
        for s in range(n_estados)
    }
    V = np.zeros(n_estados)
    Q = np.zeros((n_estados, n_acoes))

    for K in range(max_iteracoes):
        while True:
            delta = 0.0
            for s_ind in range(n_estados):
                estado_tuple = ambiente.index_to_state(s_ind)
                a = politica[estado_tuple]
                r = ambiente.recompensas_imediatas[s_ind, a]
                P = ambiente.state_transition_probabilities[:, s_ind, a]
                v_novo = r + gamma * np.dot(P, V)
                delta = max(delta, abs(V[s_ind] - v_novo))
                V[s_ind] = v_novo
            if delta < theta:
                break

        politica_estavel = True

        for s_ind in range(n_estados):
            estado_tuple = ambiente.index_to_state(s_ind)
```

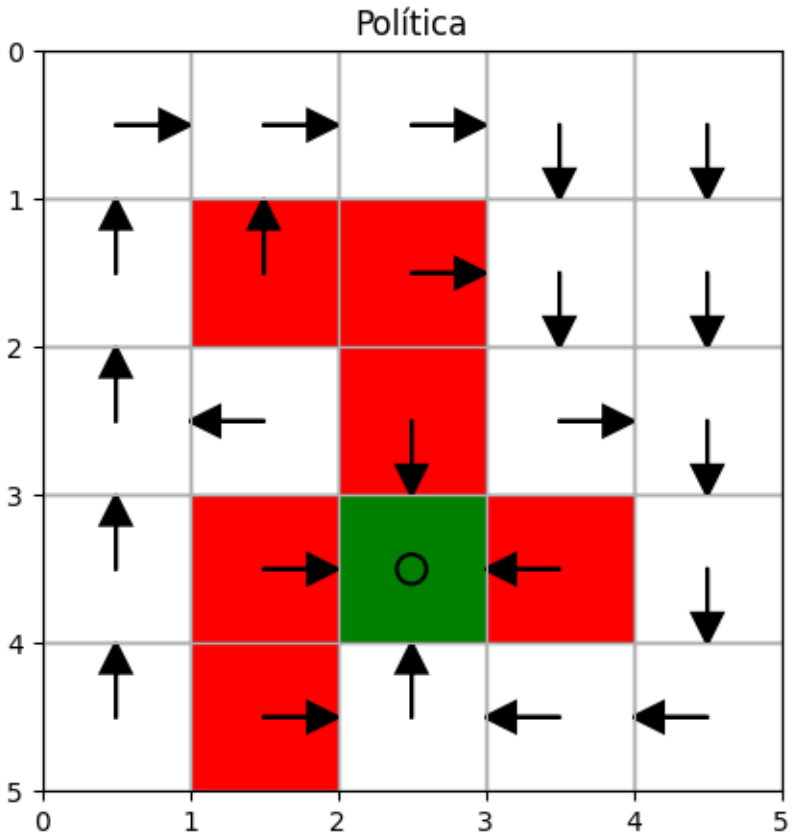
```
            antiga_acao = politica[estado_tuple]

            for a in range(n_acoes):
                r = ambiente.recompensas_imediatas[s_ind, a]
                P = ambiente.state_transition_probabilities[:, s_ind, a]
                Q[s_ind, a] = r + gamma * np.dot(P, V)

            melhor_acao = int(np.argmax(Q[s_ind, :]))
            politica[estado_tuple] = melhor_acao

            if melhor_acao != antiga_acao:
                politica_estavel = False

        if politica_estavel:
            print(f"Política ótima após {K} iterações.")
            break

    return V, Q, politica
```

[14]:
```
V, Q, politica = iteracao_de_politica(ambiente, gamma=0.9, theta=1e-6,␣
↪max_iteracoes=1000)

plot_valores_de_estado(V, ambiente)

plot_valores_de_acao(Q)

plot_policy(ambiente, politica)
```

```
Política ótima após 10 iterações.
```


Valores de Estado (V(s))
3.5 3.9 4.3 4.8 5.3
3.1 3.5 4.8 5.3 5.9
2.8 2.5 10.0 5.9 6.6
2.5 10.0 10.0 10.0 7.3
2.3 9.0 10.0 9.0 8.1
0 1 2 3 4
10 9 8 7 6 5 4 3


Valores de ação (Q(s, a) transposta)
Ações
Estados



2 Tarefa

2.1 1. Número de iterações: Valor x Política

Analisando o número de iterações, vemos que, para o mesmo ambiente, o algorimto de iteração de valor convergiu em 133 iterações, enquanto o algortitmo de iteração de política convergiu em 10. Isso se dá a natureza dos algoritmos, como estudado em sala. Todavia, é bom ressaltar que, mesmo com o menor número de iterações, isso não significa que ele seja mais eficiente. Para averiguar isso, é necessário analisar o tempo de execução dos algoritmos.