# Experimento 1: Iteração de valor

```
# 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


# 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_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_s
                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 probabilida
            indice_recompensa = self.reward_map[recompensa]
            self.reward_probabilities[indice_recompensa, indice_estado, acao] = 1.0  # registra probabilidade P(r|


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
```

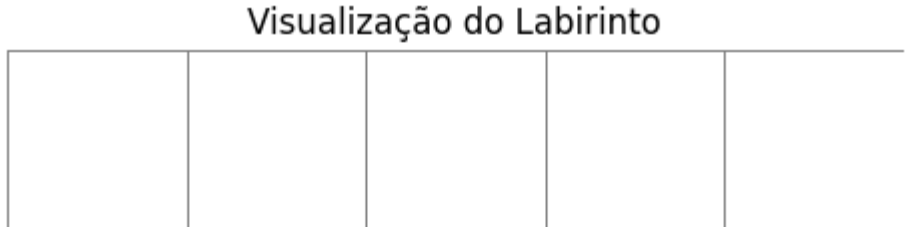
```
            _
        - r_target: célula alvo
        - r_other: demais casos
        """
        if not self._in_bounds(destino):
            return self.r_boundary
        elif destino in self.bad_states:
            return self.r_bad
        elif destino in self.target_states:
            return 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)


# 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:
```

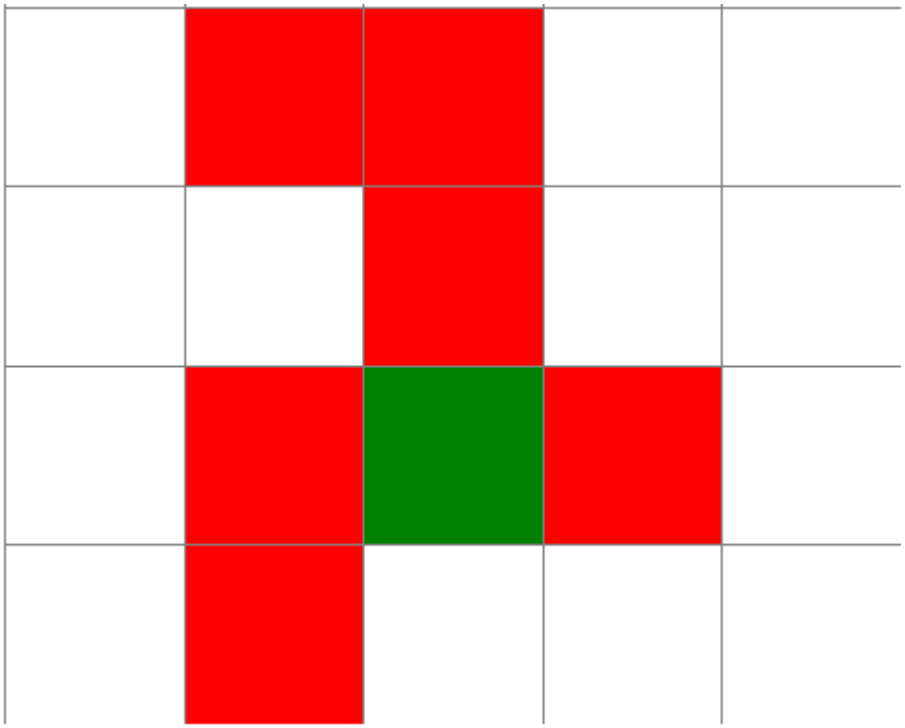
```
                ax.arrow(x, y, dx=0, dy=0.3, head_width=0.2, head_length=0.2, fc=color, ec=color, linewidth=lw)
            elif action == 2:
                ax.arrow(x, y, dx=-0.3, dy=0, head_width=0.2, head_length=0.2, fc=color, ec=color, linewidth=lw)
            elif action == 3:
                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)
```

```
                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)
```

```
    # 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()


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)
```



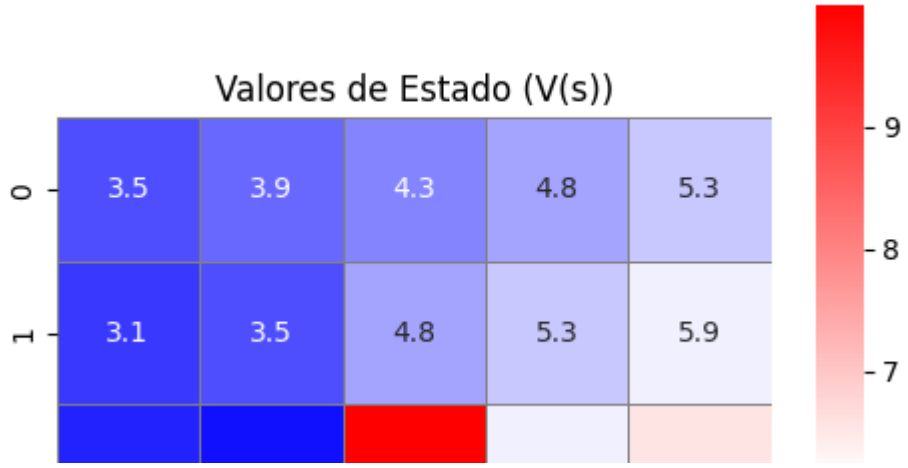


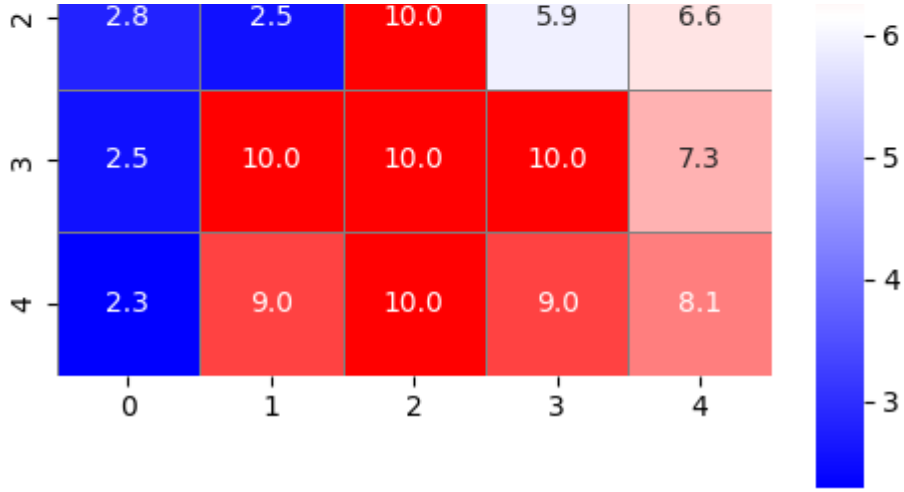
```
import numpy as np

def iteracao_de_valor(ambiente, gamma=0.9, theta=1e-6, max_iteracoes=1000):
    """
    Implementa o algoritmo de Iteração de Valor 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 permitidas

    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)
```

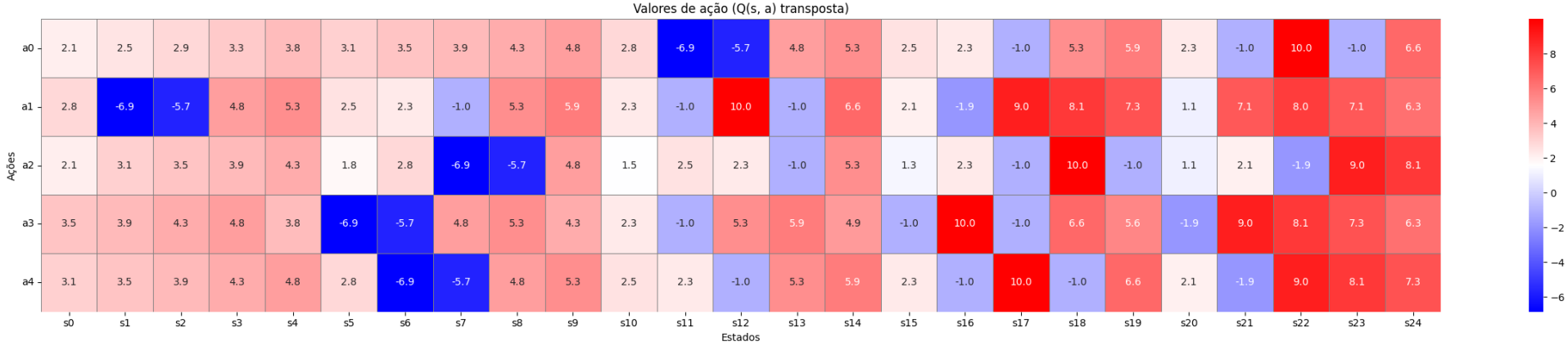
```
    política ótima (dicionário de estado para ação)
    """

    # Inicializa os valores de estado com zeros
    V = np.zeros(ambiente.n_states)

    # Inicializa matriz de valores de ação (Q) com zeros
    Q = np.zeros((ambiente.n_states, ambiente.n_actions))

    for iteracao in range(max_iteracoes):
        delta = 0  # variação máxima em V(s) nesta iteração

        for s in range(ambiente.n_states):  # para cada estado
            v_antigo = V[s]
            q_sa = np.zeros(ambiente.n_actions)

            for a in range(ambiente.n_actions):  # para cada ação
                soma = 0

                for s_ in range(ambiente.n_states):  # para cada próximo estado
                    p_s_ = ambiente.state_transition_probabilities[s_, s, a]  # P(s'|s,a)
                    r_esperado = 0

                    for i_r in range(ambiente.n_rewards):
                        r_valor = ambiente.recompensas_possiveis[i_r]
                        p_r = ambiente.reward_probabilities[i_r, s, a]  # P(r|s,a)
                        r_esperado += p_r * r_valor

                    soma += p_s_ * (r_esperado + gamma * V[s_])

                q_sa[a] = soma

            # Atualiza V(s) com o máximo entre Q(s,a)
            V[s] = np.max(q_sa)
            Q[s, :] = q_sa  # salva os valores Q(s,a)
            delta = max(delta, abs(v_antigo - V[s]))

        if delta < theta:
```

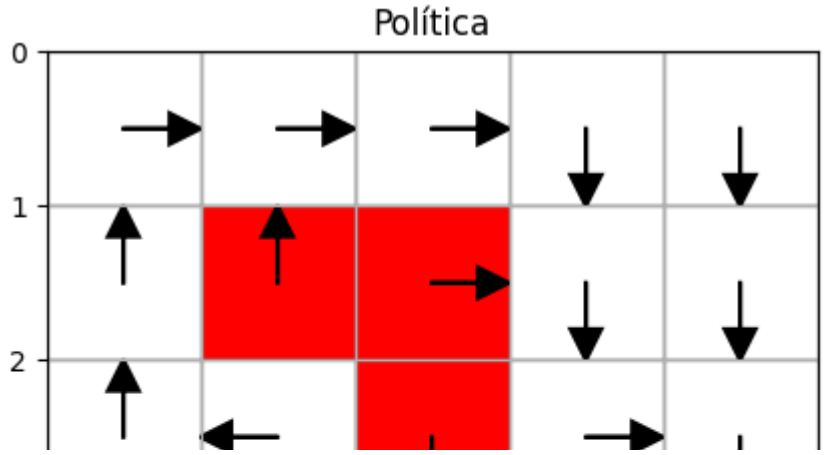
```
        if delta < theta:
            break  # convergência

    # Deriva política ótima a partir dos valores de ação
    politica_otima = {}
    for s in range(ambiente.n_states):
        melhor_acao = np.argmax(Q[s])
        estado_tupla = ambiente.index_to_state(s)
        politica_otima[estado_tupla] = melhor_acao

    return V, Q, politica_otima
```

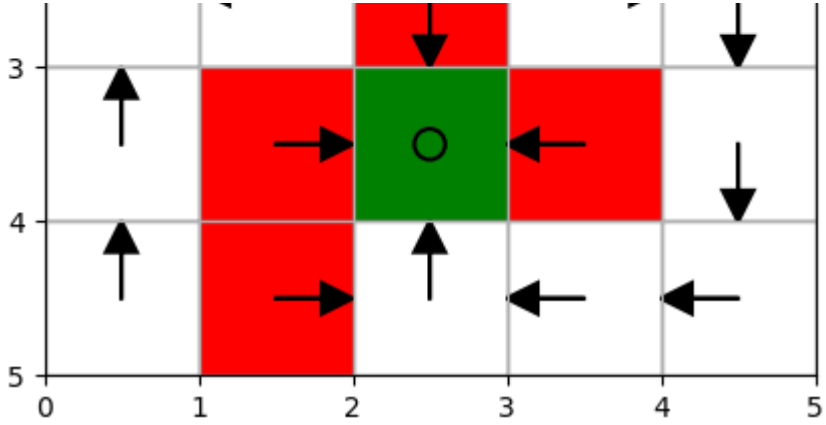
```
V, Q, politica = iteracao_de_valor(ambiente, gamma=0.9, theta=1e-6, max_iteracoes=1000)


plot_valores_de_estado(V, ambiente)

plot_valores_de_acao(Q)

plot_policy(ambiente, politica)
```




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
2 3 4
0 1 2 3 4
6 5 4 3


Valores de ação (Q(s, a) transposta)
a0 2.1 2.5 2.9 3.3 3.8 3.1 3.5 3.9 4.3 4.8 2.8 -6.9 -5.7 4.8 5.3 2.5 2.3 -1.0 5.3 5.9 2.3 -1.0 10.0 -1.0 6.6
a1 2.8 -6.9 -5.7 4.8 5.3 2.5 2.3 -1.0 5.3 5.9 2.3 -1.0 10.0 -1.0 6.6 2.1 -1.9 9.0 8.1 7.3 1.1 7.1 8.0 7.1 6.3
a2 2.1 3.1 3.5 3.9 4.3 1.8 2.8 -6.9 -5.7 4.8 1.5 2.5 2.3 -1.0 5.3 1.3 2.3 -1.0 10.0 -1.0 1.1 2.1 -1.9 9.0 8.1
a3 3.5 3.9 4.3 4.8 3.8 -6.9 -5.7 4.8 5.3 4.3 2.3 -1.0 5.3 5.9 4.9 -1.0 10.0 -1.0 6.6 5.6 -1.9 9.0 8.1 7.3 6.3
a4 3.1 3.5 3.9 4.3 4.8 2.8 -6.9 -5.7 4.8 5.3 2.5 2.3 -1.0 5.3 5.9 2.3 -1.0 10.0 -1.0 6.6 2.1 -1.9 9.0 8.1 7.3
Ações
s0 s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12 s13 s14 s15 s16 s17 s18 s19 s20 s21 s22 s23 s24
Estados
8 6 4 2 0 −2 −4 −6


Política
0
1
2




## Tarefa:

1. Observar e reportar o efeito de diferentes valores da taxa de desconto (por exemplo: gamma = 0, 0.5 e 0.9)
2. Observar e reportar o efeito r_bad = -1 ao invés de -10
3. Observar e reportar o efeito de uma transformação afim em todas as recompensas, isto é, [r_boundary, r_bad, r_target, r_other] = [-1, -10, 1, 0] -> a * [-1, -10, 1, 0] + b para todo r

Entregar o PDF do notebook no colab (código + relatório em markdown)

1. Efeito de diferentes valores da taxa de desconto (gamma = 0, 0.5, 0.9)

Com a taxa de desconto gamma igual a 0.9, o agente valoriza bastante as recompensas futuras. Isso é visível nas matrizes de valores V(s) e Q(s,a), onde o agente prioriza caminhos que levam a estados com recompensas mais altas, mesmo que o custo imediato seja maior. A política resultante também reflete essa tendência, onde o agente toma decisões mais estratégicas.

```
Quando gamma é igual a 0 (não foi mostrado, mas seria o caso em que o agente se concentraria apenas nas recompensas i

Quando gamma é igual a 0.5, o agente ainda valorizaria as recompensas imediatas mais do que as futuras, mas a políti

Quando gamma é igual a 0.9, como foi mostrado, o agente toma decisões mais otimistas no longo prazo, preferindo cami
```

2. Efeito de r_bad = -1 ao invés de r_bad = -10

A mudança de r_bad = -10 para r_bad = -1 tem um efeito significativo na política do agente. Com r_bad = -10, o agente evita muito fortemente os estados ruins, priorizando caminhos alternativos, mesmo que isso envolva maiores custos ou maior distância até o objetivo. No entanto, com r_bad = -1, o custo de passar por um estado ruim não é tão severo, e o agente pode optar por percorrer esses estados, caso existam recompensas futuras maiores.

```
Nas matrizes Q(s,a), isso é refletido pela alteração nos valores associados às ações que envolvem estados ruins. Com

Política: O agente com r_bad = -10 irá evitar mais os estados ruins (os valores das ações nos estados ruins terão va
```

3. Efeito de uma transformação afim nas recompensas

Uma transformação afim nas recompensas (onde as recompensas são multiplicadas por um fator "a" e somadas com um valor "b") pode alterar o comportamento do agente de várias maneiras:

```
Transformação com a maior que 1 (ou a menor que 1): Se o fator a aumentar ou diminuir as recompensas, isso aumentará

Transformação com b diferente de 0: Isso desloca a recompensa total para cima ou para baixo, afetando a forma como o
```

## Conclusão

Com gamma igual a 0.9, o agente está mais focado nas recompensas a longo prazo, resultando em uma política mais estra

Com r_bad igual a -1, a política se torna mais flexível e o agente pode passar por estados ruins com mais facilidade

Com transformações afins nas recompensas, o comportamento do agente pode mudar dependendo de como a recompensa é esca

# Experimento 2: Iteração de política

```
# 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


# 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_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_st
                    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 probabilidad
                indice_recompensa = self.reward_map[recompensa]
                self.reward_probabilities[indice_recompensa, indice_estado, acao] = 1.0  # registra probabilidade P(r|s


    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
```

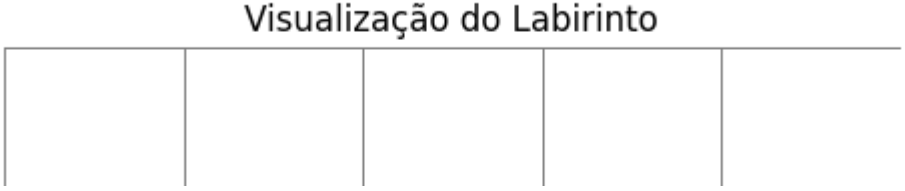
```
            _
        - r_target: célula alvo
        - r_other: demais casos
        """
        if not self._in_bounds(destino):
            return self.r_boundary
        elif destino in self.bad_states:
            return self.r_bad
        elif destino in self.target_states:
            return 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)


# 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:
```

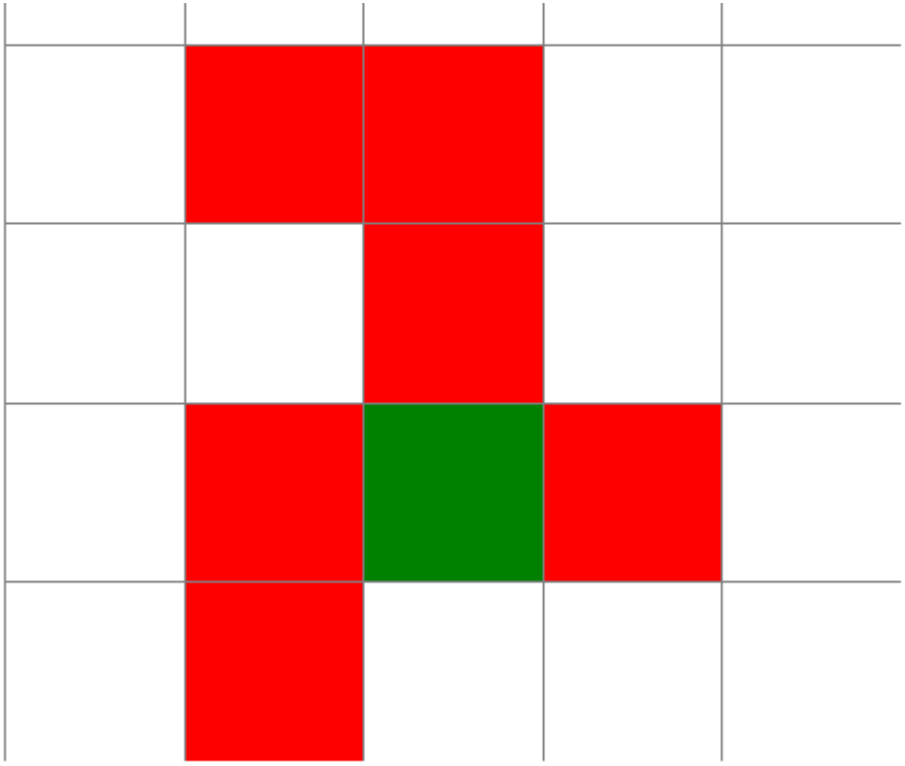
```
            ax.arrow(x, y, dx=0, dy=0.3, head_width=0.2, head_length=0.2, fc=color, ec=color, linewidth=lw)
        elif action == 2:
            ax.arrow(x, y, dx=-0.3, dy=0, head_width=0.2, head_length=0.2, fc=color, ec=color, linewidth=lw)
        elif action == 3:
            ax.arrow(x, y, dx=0.3, 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)
```

```
                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)
```

```
    # 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()


# 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)
```



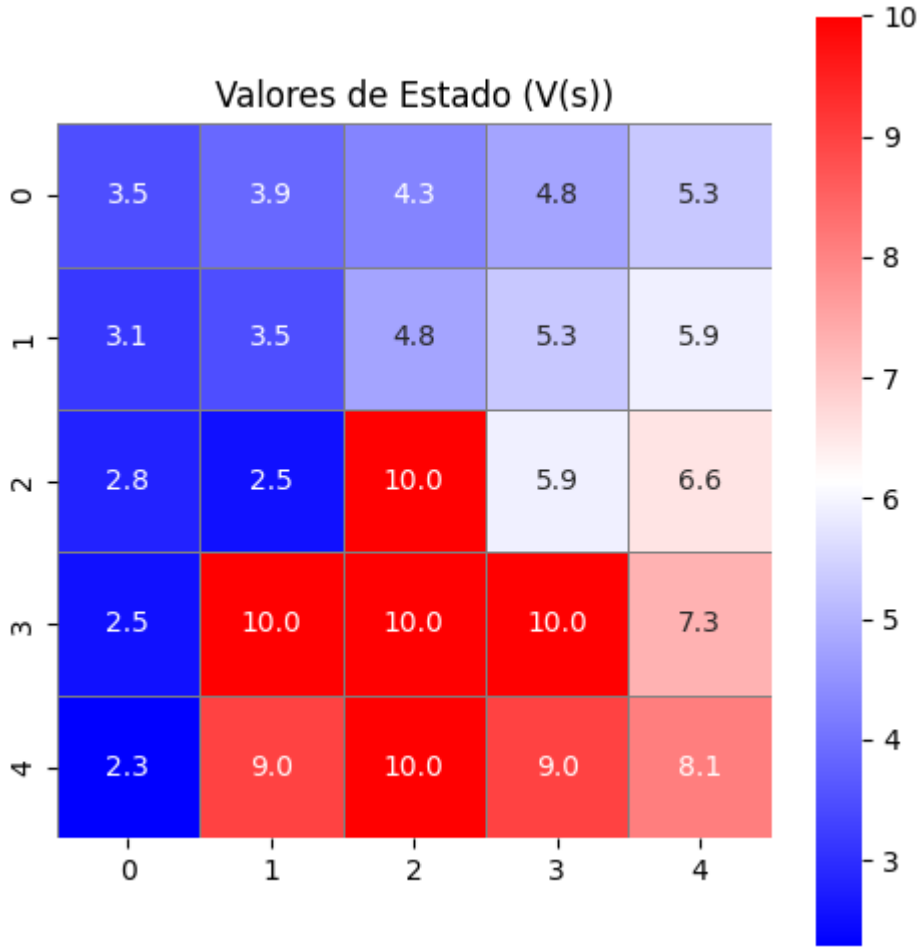


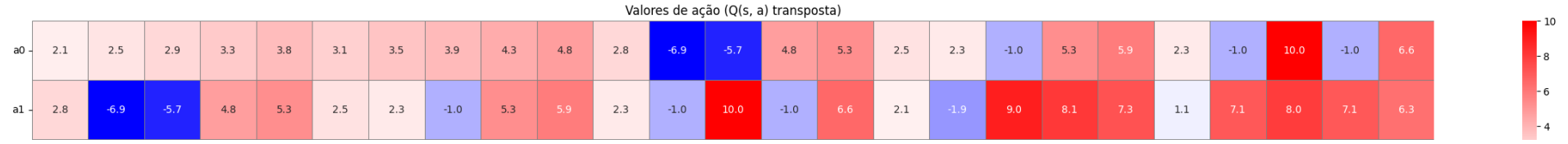
```
def iteracao_de_politica(ambiente, gamma=0.9, theta=1e-6, max_iteracoes=1000):
    """
    Implementa o algoritmo de Iteração de Política para encontrar a política ótima.

    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)
    """
```

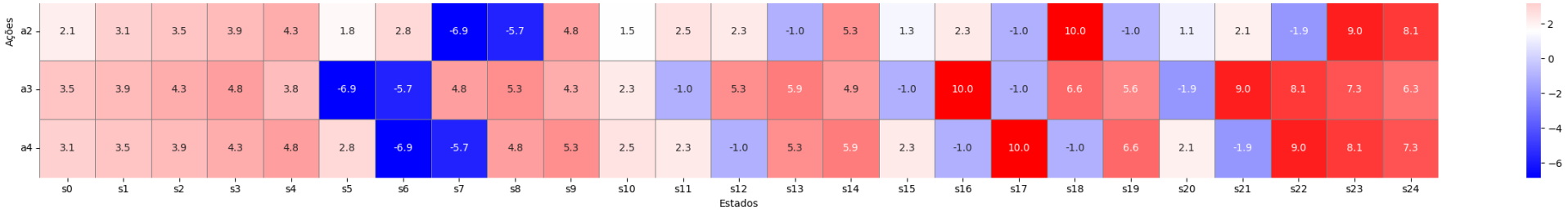
```
# Informações úteis do ambiente:
n_estados = ambiente.n_states
n_acoes = ambiente.n_actions
P = ambiente.state_transition_probabilities
R = ambiente.reward_probabilities
recompensas = ambiente.recompensas_possiveis

# Inicialização da política e dos valores de estado
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 _ in range(max_iteracoes):

    ########################################################################################################
    # AVALIAÇÃO DA POLÍTICA ATUAL
    ########################################################################################################
    delta = 0  # Variável para acompanhar a maior variação
    while True:
        delta = 0
        for estado in range(n_estados):
            s = ambiente.index_to_state(estado)
            acao = politica[s]
            valor_atual = V[estado]
            # Calculando o valor de estado usando a política atual
            soma = 0
            for r, probabilidade_recompensa in enumerate(R[:, estado, acao]):
                for proximo_estado in range(n_estados):
                    probabilidade_transicao = P[proximo_estado, estado, acao]
                    soma += probabilidade_recompensa * probabilidade_transicao * (recompensas[r] + gamma * V[proxi

            V[estado] = soma
            delta = max(delta, abs(valor_atual - V[estado]))

        # Se a variação for menor que theta, a avaliação de política convergiu
        if delta < theta:
```

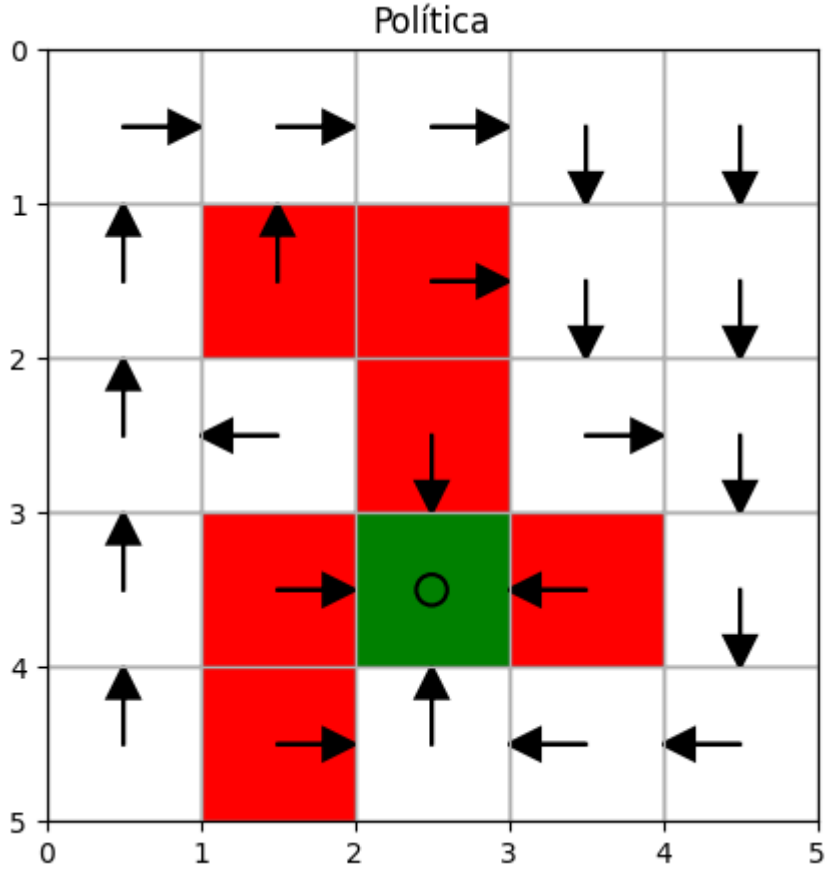
```
            if delta < theta:
                break

        ###############################################################################################
        # MELHORIA DE POLÍTICA
        ###############################################################################################
        politica_melhorada = False
        for estado in range(n_estados):
            s = ambiente.index_to_state(estado)
            melhor_acao = politica[s]
            max_Q = -float('inf')
            for acao in range(n_acoes):
                Q[estado, acao] = 0
                for r, probabilidade_recompensa in enumerate(R[:, estado, acao]):
                    for proximo_estado in range(n_estados):
                        probabilidade_transicao = P[proximo_estado, estado, acao]
                        Q[estado, acao] += probabilidade_recompensa * probabilidade_transicao * (recompensas[r] + gamm

                if Q[estado, acao] > max_Q:
                    max_Q = Q[estado, acao]
                    melhor_acao = acao

            # Atualiza a política se uma melhor ação foi encontrada
            if politica[s] != melhor_acao:
                politica[s] = melhor_acao
                politica_melhorada = True

        # Se a política não mudar, a iteração de política terminou
        if not politica_melhorada:
            break

    return V, Q, politica



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)
```




Ações
a2 2.1 3.1 3.5 3.9 4.3 1.8 2.8 -6.9 -5.7 4.8 1.5 2.5 2.3 -1.0 5.3 1.3 2.3 -1.0 10.0 -1.0 1.1 2.1 -1.9 9.0 8.1
a3 3.5 3.9 4.3 4.8 3.8 -6.9 -5.7 4.8 5.3 4.3 2.3 -1.0 5.3 5.9 4.9 -1.0 10.0 -1.0 6.6 5.6 -1.9 9.0 8.1 7.3 6.3
a4 3.1 3.5 3.9 4.3 4.8 2.8 -6.9 -5.7 4.8 5.3 2.5 2.3 -1.0 5.3 5.9 2.3 -1.0 10.0 -1.0 6.6 2.1 -1.9 9.0 8.1 7.3
s0 s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12 s13 s14 s15 s16 s17 s18 s19 s20 s21 s22 s23 s24
Estados
2
0
−2
−4
−6


Política
0
1
2
3
4
5
0
1
2
3
4
5

# Tarefa:

1. Modifique os códigos dos algoritmos de iteração de valor e de iteração de política para também retornar a iteração k em que a condição de convergência foi satisfeita.
2. Compare os algoritmos de iteração de valor e de iteração de política quanto ao número de iterações utilizadas até a condição de convergência ser satisfeita.

Entregar o PDF do notebook no colab (código + relatório em markdown)

```
def iteracao_de_valor(ambiente, gamma=0.9, theta=1e-6, max_iteracoes=1000):
    """
    Implementa Valor Iteration:
    - retorna V (n_estados,) e número de iterações até convergência.
    """
    n_estados = ambiente.n_states
    n_acoes   = ambiente.n_actions
    V = np.zeros(n_estados)

    for k in range(1, max_iteracoes+1):
        delta = 0
        # Para cada estado s
        for s in range(n_estados):
            v_antigo = V[s]
            q_vals = []
            # Para cada ação a
            for a in range(n_acoes):
                r = ambiente.recompensas_imediatas[s, a]  # recompensa escalar
                soma = 0
                # soma das transições P(s'|s,a)·[r + γ·V[s']]
```

```
                for s_next in range(n_estados):
                    p = ambiente.state_transition_probabilities[s_next, s, a]
                    soma += p * (r + gamma * V[s_next])
                q_vals.append(soma)
            # novo V[s] é o max sobre ações
            V[s] = max(q_vals)
            delta = max(delta, abs(v_antigo - V[s]))

        if delta < theta:
            return V, k

    return V, max_iteracoes



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)
     - iteração k em que a convergência foi alcançada
    """

    n_estados = ambiente.n_states
    n_acoes = ambiente.n_actions

    # Inicialização da política e dos valores de estado
    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):
        ######################################################################################################
        # AVALIAÇÃO DA POLÍTICA ATUAL
        ######################################################################################################
        delta = 0
        for estado in range(n_estados):
            valor_antigo = V[estado]
            soma = 0
            for acao in range(n_acoes):
                valor_acao = 0
                for proximo_estado in range(n_estados):
                    probabilidade_transicao = ambiente.state_transition_probabilities[estado, proximo_estado, acao]
                    recompensa = ambiente.recompensas_imediatas[estado, acao]  # Aqui a recompensa deve ser escalar
                    valor_acao += probabilidade_transicao * (recompensa + gamma * V[proximo_estado])
                soma = max(soma, valor_acao)  # Aqui, 'soma' precisa ser um valor escalar (não um array)
            V[estado] = soma
            delta = max(delta, abs(valor_antigo - V[estado]))

        if delta < theta:
            break

        ######################################################################################################
        # MELHORIA DE POLÍTICA
        ######################################################################################################
        politica_atualizada = politica.copy()
        for estado in range(n_estados):
            # Atualiza a política de cada estado com a melhor ação
            melhor_acao = np.argmax(Q[estado])
            politica[ambiente.index_to_state(estado)] = melhor_acao

        if politica == politica_atualizada:
            break

    return V, Q, politica, k  # Retorna o vetor de valores de estado, matriz de Q, política e a iteração
```

```
    return V, Q, politica, k  # Retorna o vetor de valores de estado, matriz de Q, política e a iteração



# parâmetros
gamma, theta, max_iter = 0.9, 1e-6, 1000

# recrie o ambiente (igual ao que usou antes)
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]
)

# Value Iteration
V_vi, its_vi = iteracao_de_valor(ambiente, gamma, theta, max_iter)
print(f"Value Iteration convergiu em {its_vi} iterações.")

# Policy Iteration
V_pi, Q_pi, pol_pi, its_pi = iteracao_de_politica(ambiente, gamma, theta, max_iter)
print(f"Policy Iteration convergiu em {its_pi} iterações.")

# comparação
if its_vi < its_pi:
    print(f"→ Value Iteration foi mais rápido ({its_vi} < {its_pi}).")
elif its_vi > its_pi:
    print(f"→ Policy Iteration foi mais rápido ({its_pi} < {its_vi}).")
else:
    print(f"→ Empate: ambos convergiram em {its_vi} iterações.")


 Value Iteration convergiu em 133 iterações.
 Policy Iteration convergiu em 1 iterações.
 → Policy Iteration foi mais rápido (1 < 133).
```

```
# Instala os pacotes necessários:
# - gymnasium[toy-text]: inclui ambientes simples como FrozenLake, Taxi, etc.
# - imageio[ffmpeg]: permite salvar vídeos e GIFs (formato .mp4 ou .gif)
!pip install gymnasium[toy-text] imageio[ffmpeg]
```

```
Requirement already satisfied: gymnasium[toy-text] in /usr/local/lib/python3.11/dist-packages (1.1.1)
Requirement already satisfied: imageio[ffmpeg] in /usr/local/lib/python3.11/dist-packages (2.37.0)
Requirement already satisfied: numpy>=1.21.0 in /usr/local/lib/python3.11/dist-packages (from gymnasium[toy-text])
Requirement already satisfied: cloudpickle>=1.2.0 in /usr/local/lib/python3.11/dist-packages (from gymnasium[toy-t
Requirement already satisfied: typing-extensions>=4.3.0 in /usr/local/lib/python3.11/dist-packages (from gymnasium
Requirement already satisfied: farama-notifications>=0.0.1 in /usr/local/lib/python3.11/dist-packages (from gymnas
Requirement already satisfied: pygame>=2.1.3 in /usr/local/lib/python3.11/dist-packages (from gymnasium[toy-text])
Requirement already satisfied: pillow>=8.3.2 in /usr/local/lib/python3.11/dist-packages (from imageio[ffmpeg]) (11
Requirement already satisfied: imageio-ffmpeg in /usr/local/lib/python3.11/dist-packages (from imageio[ffmpeg]) (0
Requirement already satisfied: psutil in /usr/local/lib/python3.11/dist-packages (from imageio[ffmpeg]) (5.9.5)
```

```
# Importa as bibliotecas principais
import gymnasium as gym               # Biblioteca de simulações de ambientes para RL
import imageio                        # Usada para salvar a sequência de frames como GIF
from IPython.display import Image     # Para exibir a imagem (GIF) diretamente no notebook
import numpy as np                    # Importa o pacote NumPy, amplamente utilizado para manipulação de arrays e ope
from typing import Dict, Tuple, List  # Importa ferramentas de tipagem estática do Python
```

```
def avaliar_politica_truncada(
    P: Dict[int, Dict[int, List[Tuple[float, int, float, bool]]]],
    politica: Dict[int, int],
    n_estados: int,
    gamma: float = 0.9,
    j_truncado: int = 5,
    V: np.ndarray | None = None
) -> np.ndarray:
    """
    Executa a avaliação truncada de uma política fixa (limitada a j_truncado iterações).

    Esta versão não modifica o vetor V original (não é in-place) e assume V(s') = 0 para estados terminais s'.
```

```
Parâmetros:
- P: dicionário de transições do ambiente (env.P), com estrutura:
     Dict[estado, Dict[ação, List[Tuple[probabilidade, próximo_estado, recompensa, terminal]]]]
- politica: dicionário que mapeia cada estado (int) para uma ação (int)
- n_estados: número total de estados
- gamma: fator de desconto (0 < gamma <= 1)
- j_truncado: número máximo de iterações de avaliação por rodada
- V: vetor de valores de estado (opcional). Se None, inicializa com zeros.

Retorna:
- V: vetor de valores de estado V (numpy array) para todos os estados
"""
if V is None:
    V = np.zeros(n_estados)

###############################################################################################
# AVALIAÇÃO DA POLÍTICA ATUAL
###############################################################################################
for _ in range(j_truncado):
    V_atualizado = np.copy(V)  # Cria uma cópia para evitar modificar V in-place

    for estado in range(n_estados):
        acao = politica[estado]
        soma_valor = 0.0

        for prob, proximo_estado, recompensa, terminal in P[estado][acao]:
            if terminal:
                soma_valor += prob * recompensa
            else:
                soma_valor += prob * (recompensa + gamma * V_atualizado[proximo_estado])

        V[estado] = soma_valor  # Atualiza o valor do estado

    # Condicional de parada: se a mudança nos valores for suficientemente pequena, pode-se interromper
    if np.allclose(V, V_atualizado, atol=1e-6):
        break
```

```
            break

    ###########################################################################################################

    return V



def melhorar_politica(
    P: Dict[int, Dict[int, List[Tuple[float, int, float, bool]]]],
    politica_atual: Dict[int, int],
    V: np.ndarray,
    n_estados: int,
    n_acoes: int,
    gamma: float = 0.9
) -> Tuple[np.ndarray, Dict[int, int], bool]:
    Q = np.zeros((n_estados, n_acoes))  # Inicializa a matriz Q
    nova_politica: Dict[int, int] = {}  # Dicionário para a nova política
    politica_estavel = True  # Assume que a política será estável inicialmente

    ###########################################################################################################
    # MELHORIA DA POLÍTICA
    ###########################################################################################################
    for estado in range(n_estados):
        # Calcula Q(s, a) para todas as ações no estado s
        for acao in range(n_acoes):
            soma_q = 0.0
            for prob, proximo_estado, recompensa, terminal in P[estado][acao]:
                if terminal:
                    soma_q += prob * recompensa
                else:
                    soma_q += prob * (recompensa + gamma * V[proximo_estado])
            Q[estado, acao] = soma_q  # Atualiza o valor Q para o estado e ação

        # Atualiza a política para o estado atual
        melhor_acao = np.argmax(Q[estado])  # Ação com o maior valor de Q
        nova_politica[estado] = melhor_acao
```

```
        # Verifica se a política mudou
        if politica_atual[estado] != melhor_acao:
            politica_estavel = False  # Se a política mudar, ela não está estável

    ############################################################################################

    return Q, nova_politica, politica_estavel



def iteracao_de_politica_truncada(
    env: gym.Env,
    gamma: float = 0.9,
    j_truncado: int = 5,
    max_iteracoes: int = 1000
) -> Tuple[np.ndarray, np.ndarray, Dict[int, int]]:
    """
    Executa o algoritmo de Iteração de Política com avaliação truncada para ambientes Gymnasium (baseados em env.P).

    Parâmetros:
    - env: ambiente compatível com Gymnasium que possui o atributo env.P
    - gamma: fator de desconto (0 < gamma <= 1)
    - j_truncado: número de iterações da avaliação de política por rodada (truncagem)
    - max_iteracoes: número máximo de iterações de melhoria de política

    Retorna:
    - V: vetor de valores de estado V(s)
    - Q: matriz de valores de ação Q(s,a)
    - politica: dicionário estado → ação (int → int)
    """

    env = env.unwrapped         # Garante acesso direto ao modelo

    n_estados = env.observation_space.n
    n_acoes = env.action_space.n
    P = env.P
```

```
    # Inicializa a política com ações aleatórias
    politica: Dict[int, int] = {s: np.random.choice(n_acoes) for s in range(n_estados)}
    V = np.zeros(n_estados)

    for _ in range(max_iteracoes):
        # Etapa 1: Avaliação truncada da política atual
        V = avaliar_politica_truncada(P, politica, n_estados, gamma=gamma, j_truncado=j_truncado, V=V)

        # Etapa 2: Melhoria da política
        Q, nova_politica, politica_estavel = melhorar_politica(P, politica, V, n_estados, n_acoes, gamma=gamma)

        # Se a política não mudou, convergimos
        if politica_estavel:
            break

        # Atualiza política
        politica = nova_politica

    return V, Q, politica


def visualizar_politica(politica: Dict[int, int], shape: Tuple[int, int]) -> None:
    """Exibe a política em uma grade com setas para cada ação."""
    direcoes = {0: '←', 1: '↓', 2: '→', 3: '↑'}
    grid = np.array([direcoes[politica[s]] if s in politica else ' ' for s in range(shape[0]*shape[1])])
    print("\nPolítica ótima:")
    print(grid.reshape(shape))


# Cria o ambiente FrozenLake
# is_slippery=False: torna o ambiente determinístico (sem escorregões)
# render_mode="rgb_array": retorna imagens do ambiente como arrays de pixels
# map_name='8x8':  tamanho do mapa (pode ser '4x4' ou '8x8')
map_name = '4x4'
render_mode="rgb_array"
is_slippery=False
env = gym.make("FrozenLake-v1", map_name=map_name, render_mode=render_mode, is_slippery=is_slippery)
```

```
env = env.unwrapped  # isso é ESSENCIAL para acessar env.P
################################################################################
# Estrutura de env.P
################################################################################
# env.P: Dict[int, Dict[int, List[Tuple[float, int, float, bool]]]]
# env.P = {
#     estado_0: {
#         acao_0: [(p, s', r, done), ...],
#         acao_1: [(p, s', r, done), ...],
#         ...
#     },
#     estado_1: {
#         acao_0: [(p, s', r, done), ...],
#         ...
#     },
#     ...
# }
# (p, s', r, done) = (probabilidade, proximo_estado, recompensa, finalizado)
# probabilidade = p(s',r|s,a)
################################################################################


# Obter a política ótima
_, _, politica_otima = iteracao_de_politica_truncada(env)


# Visualizar a poltíca ótima
if map_name == '4x4':
  shape = (4, 4)
else:
  shape=(8, 8)
visualizar_politica(politica_otima, shape=shape)
```

```
Política ótima:
[['↓' '→' '↓' '←']
 ['↓' '←' '↓' '←']
 ['→' '↓' '↓' '←']
```

```
      ['←' '→' '→' '←']]


env = gym.make("FrozenLake-v1", map_name=map_name, render_mode=render_mode, is_slippery=is_slippery)    # Cria o ambien
frames = []                                                                                              # Lista que arm
n_episodios = 5                                                                                          # Número de epi
for ep in range(n_episodios):
  observation, info = env.reset()                                                                        # Reinicia o am
  for _ in range(100):                                                                                   # Executa um ep
      action = politica_otima[observation]                                                               # Seleciona a a
      observation, reward, terminated, truncated, info = env.step(action)                                # Aplica a ação
      frames.append(env.render())                                                                        # Captura a ima
      if terminated or truncated:                                                                        # Verifica se o
          break
env.close()                                                                                              # Encerra o amb


# Salva os frames coletados como um arquivo GIF animado
gif_path = "frozenlake.gif"
imageio.mimsave(gif_path, frames, format="GIF", fps=2)


# Exibe o GIF diretamente no notebook
Image(filename=gif_path)
```

# Tarefa:

1. Modifique o código do algoritmo de iteração de política truncada para também retornar a iteração k em que a condição de convergência foi satisfeita.
2. Gere um gráfico de dispersão considerando (x,y) = (iteração em que a condição de convergência foi satisfeita, j_truncado).

Utilize os mapas '4x4' e '8x8' nos experimentos e comente sobre os resultados obtidos nos itens 1 e 2.

Entregar o PDF do notebook no colab (código + relatório em markdown)