lab_1_iteracao_de_valor

May 11, 2025

1 Experimento 1: Iteração de valor

[1]:
```
# 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
```

[2]:
```
# 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
        }

        # 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)
```

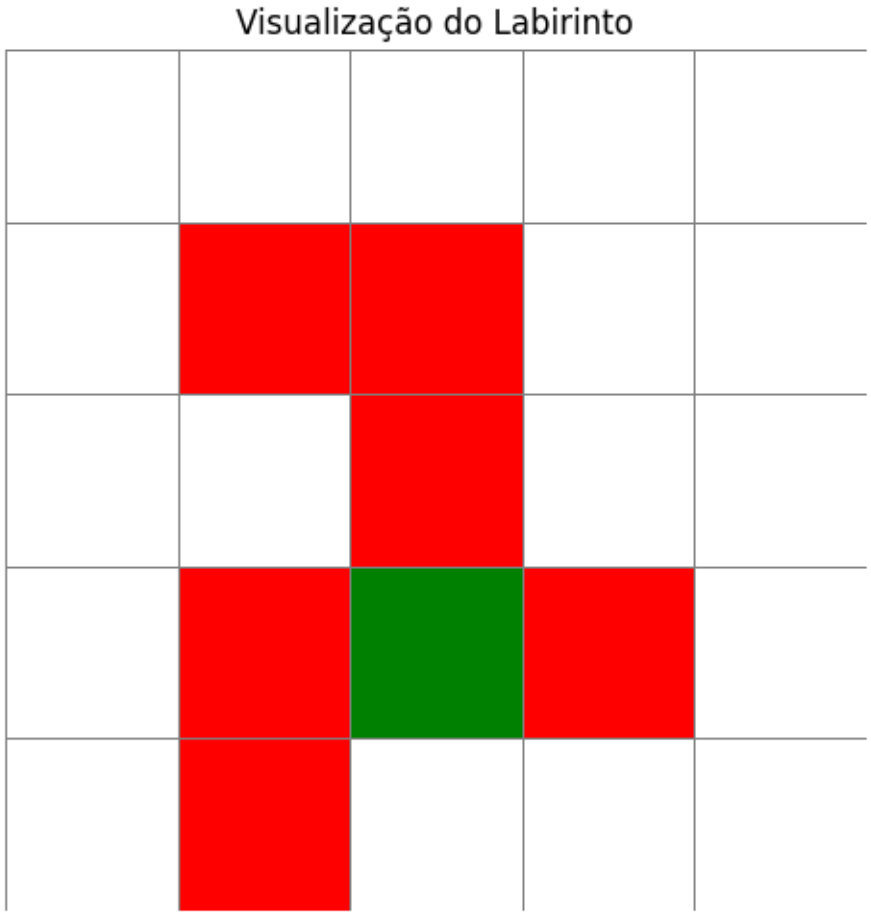
[3]:
```
# Funções auxiliares para visualização

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()
```

[4]:
```
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)
```



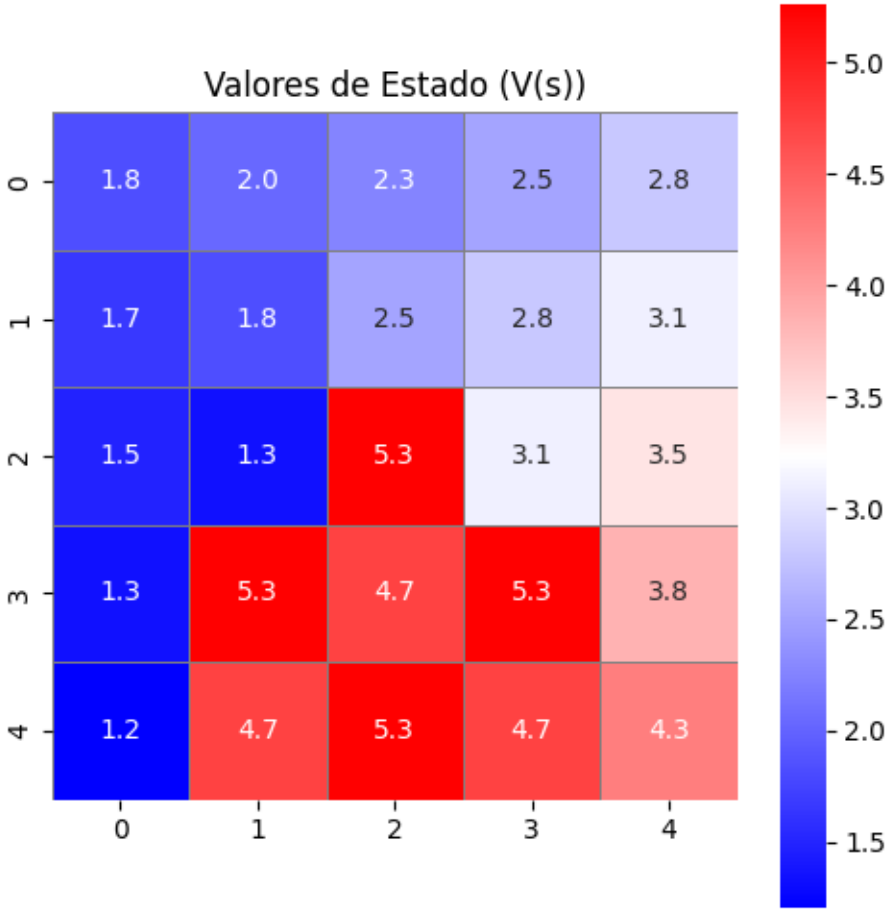
[5]:
```
def iteracao_de_valor(ambiente, gamma=0.9, theta=1e-6, max_iteracoes=1000):
    """
    Implementa o algoritmo de Iteração de Valor para encontrar a política ótima.

    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)
    """

    # 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)



    ##########################################################################################
    # Código da iteração de valor aqui

    # Deve calcular:
    # Valores de estado V (numpy array) para todos os estados (shape = (ambiente.n_states, ))
    # Valores de ação Q (numpy array) para todos os estados (shape = (ambiente.n_states, ambiente.n_actions))
    # política ótima como dicionário {estado - tupla: melhor_acao - int 0 a 4} (dica: usar ambiente.index_to_state(estado))
```

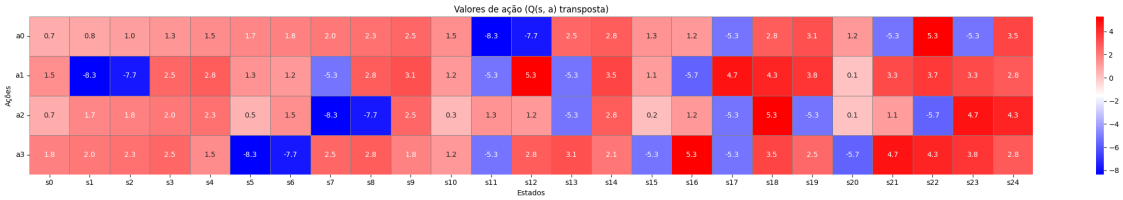
```
    # vetor com o valor de cada estado
    V = np.zeros(ambiente.n_states, dtype=float)

    # matriz para o valor de cada ação de um estado[[0,0,0,1,1],[...]...]
    Q = np.zeros(shape=(ambiente.n_states, ambiente.n_actions))

    # Loop principal
    for k in range(max_iteracoes):
        delta = 0.0
        V_novo = np.zeros_like(V)
        for estado in range(ambiente.n_states):

            for acao in range(ambiente.n_actions):
                recompensa_esperada = np.sum(
                    ambiente.reward_probabilities[:, estado, acao]
                    * ambiente.recompensas_possiveis
                )

                valor_futuro_esperado = np.sum(
                    ambiente.state_transition_probabilities[:, estado, acao] * V
                )

                Q[estado, acao] = recompensa_esperada + (gamma *␣
↪valor_futuro_esperado)

                # print(f"Estado: {estado}")
                # print(f"Ação  : {acao}")
                # print(f"RE    : {recompensa_esperada}")
                # print(f"Q({estado},{acao}): {Q[estado, acao]}")

            V_novo[estado] = np.max(Q[estado])
            delta = max(delta, abs(V_novo[estado] - V[estado]))

        V = V_novo

        if delta < theta:
            print(f"Convergiu com {k+1} interações.")
            break
    else:
        print(f"Convergiu com o máximo de interações")


    politica = {}
    for estado in range(ambiente.n_states):
        melhor_acao = int(np.argmax(Q[estado]))
```

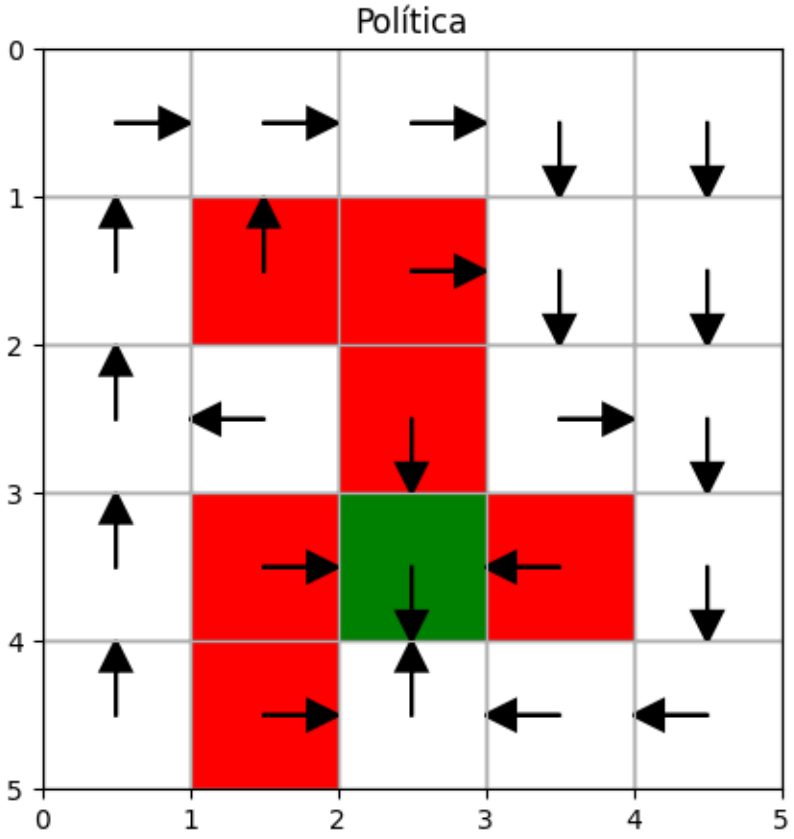
```
        coord_estado = ambiente.index_to_state(estado)
        politica[coord_estado] = melhor_acao

    # print(politica)
    return V, Q, politica
```

[6]:
```
# TESTADO ALGORITMO

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)
```

```
Convergiu com 133 interações.
```







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)

3 Influencia da taxa de desconto

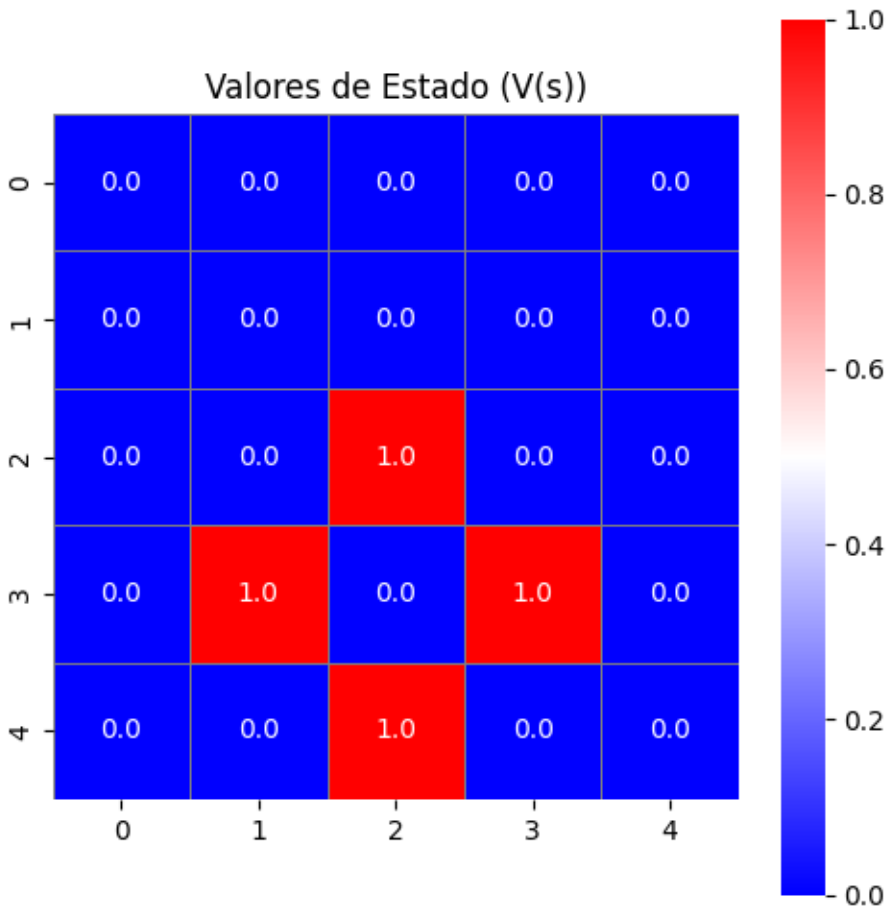
Nesse momento iremos testar o efeito da taxa de desconto ( ), onde para o mesmo ambiente utilizaremos diferentes valores de gamma (0.0, 0.5 e 0.9).

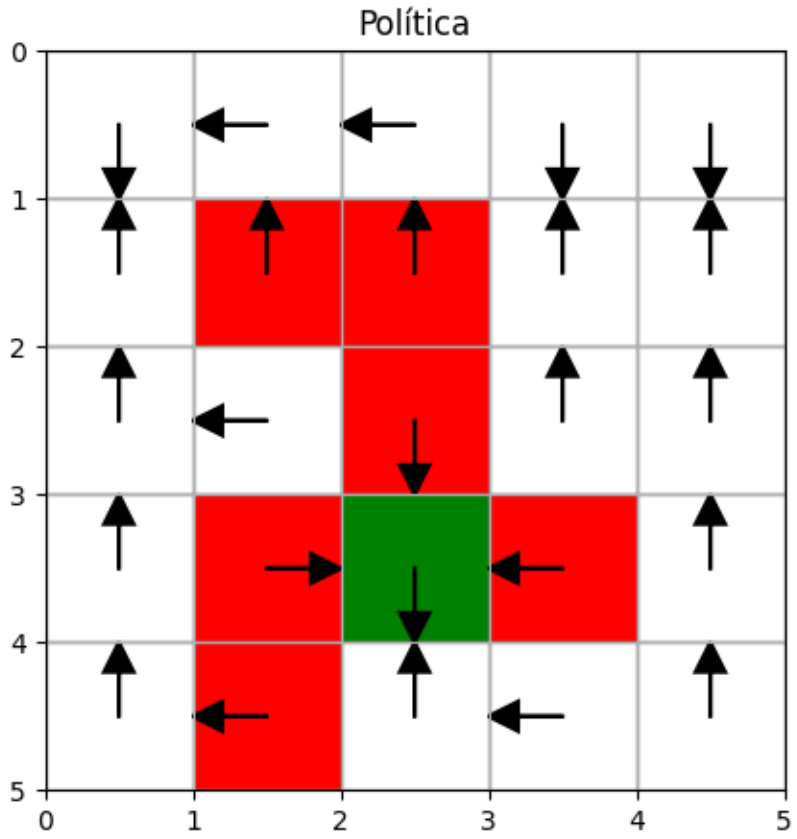
[7]:
```
ambiente = AmbienteNavegacaoLabirinto(
        world_size=(5, 5),
        bad_states=[(1, 1), (1, 2), (2, 2), (3, 1), (3, 3), (4, 1)],
        target_states=[(3, 2)],
        allow_bad_entry=True,
        rewards=[-1, -10, 1, 0]
    )
```

3.0.1 = 0

[8]:
```
V1, Q1, politica1 = iteracao_de_valor(ambiente, gamma=0, theta=1e-6,␣
 ↪max_iteracoes=1000)
plot_valores_de_estado(V1, ambiente)
plot_policy(ambiente, politica1)
```

```
Convergiu com 2 interações.
```



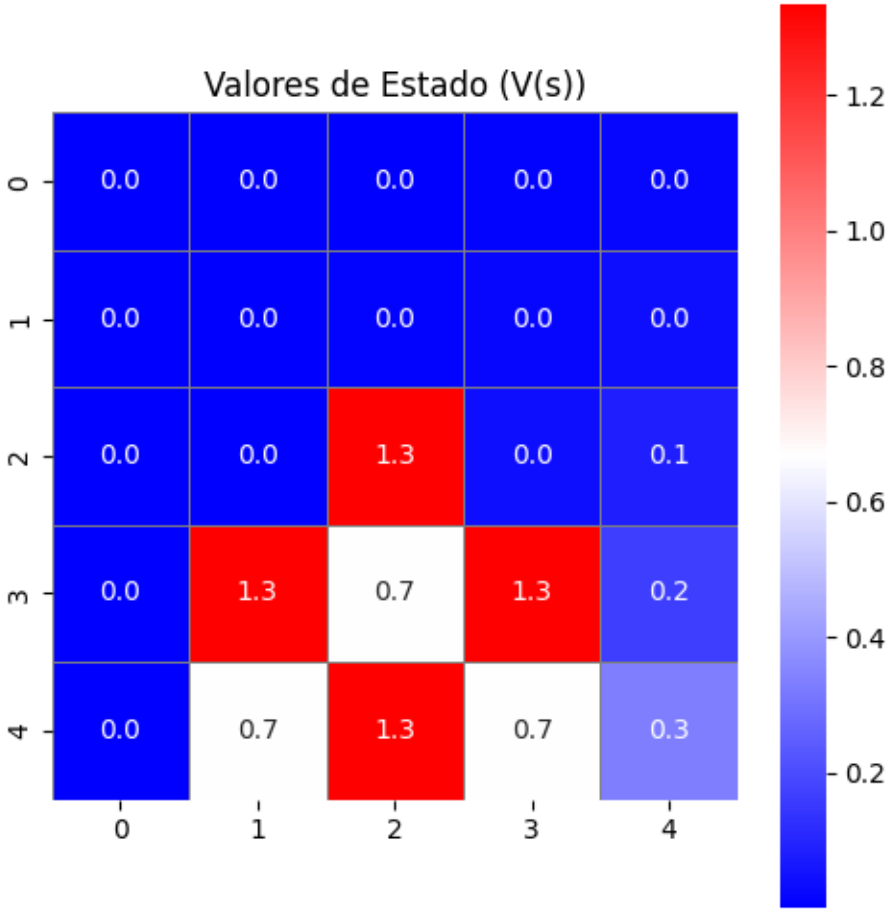


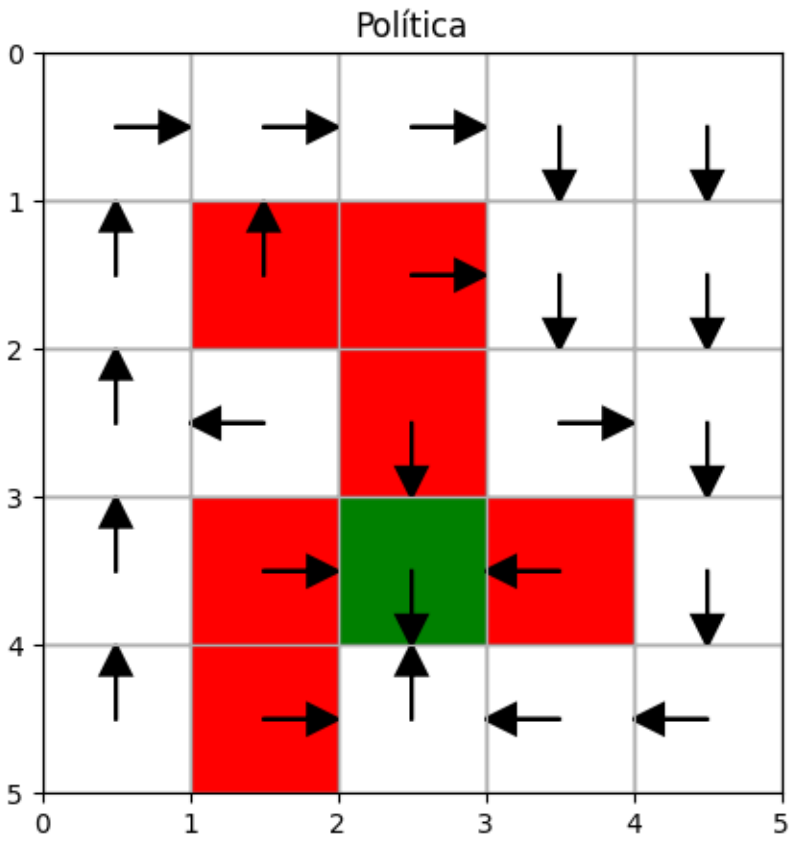
Nesse caso, é possível notar que **não temos uma política ótima**, pois o agente apenas procura a recompensa imediata, ignorando completamente recompensas futuras. Isso também explica o baixo valor de estado e o baixo número de iterações.

3.0.2 = 0.5

[9]:
```
V2, Q2, politica2 = iteracao_de_valor(ambiente, gamma=0.5, theta=1e-6,␣
↪max_iteracoes=1000)
plot_valores_de_estado(V2, ambiente)
plot_policy(ambiente, politica2)
```

```
Convergiu com 21 interações.
```


Valores de Estado (V(s))
0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0
0.0 0.0 1.3 0.0 0.1
0.0 1.3 0.7 1.3 0.2
0.0 0.7 1.3 0.7 0.3
0
1
2
3
4
1.2
1.0
0.8
0.6
0.4
0.2

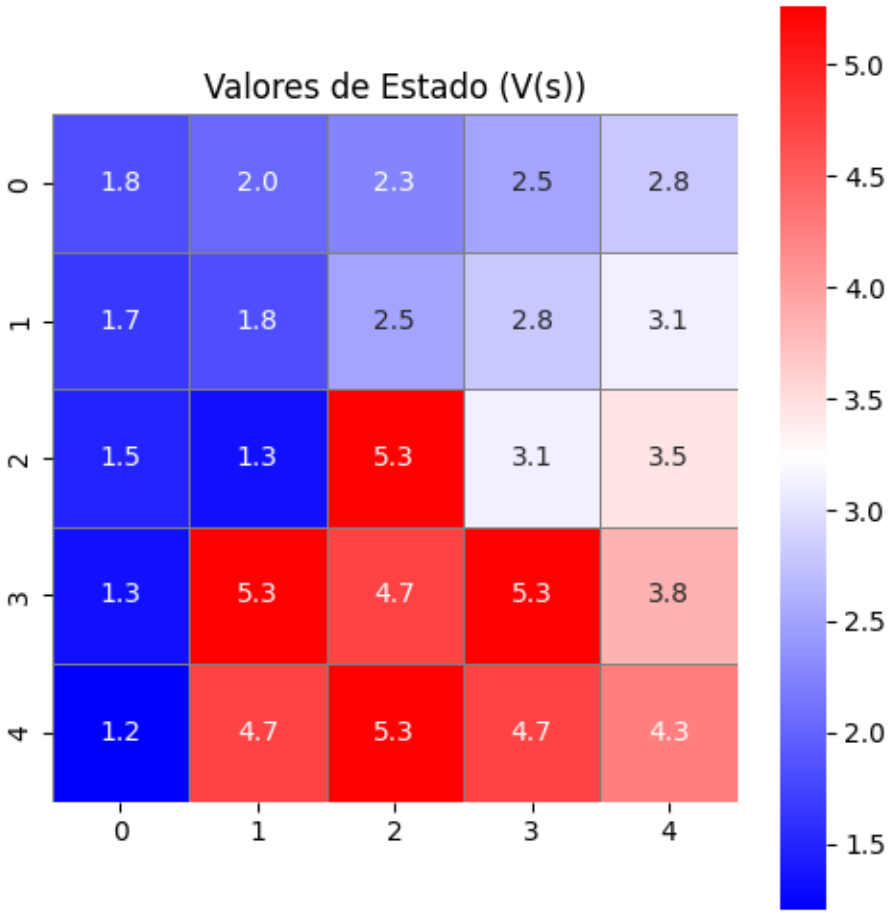


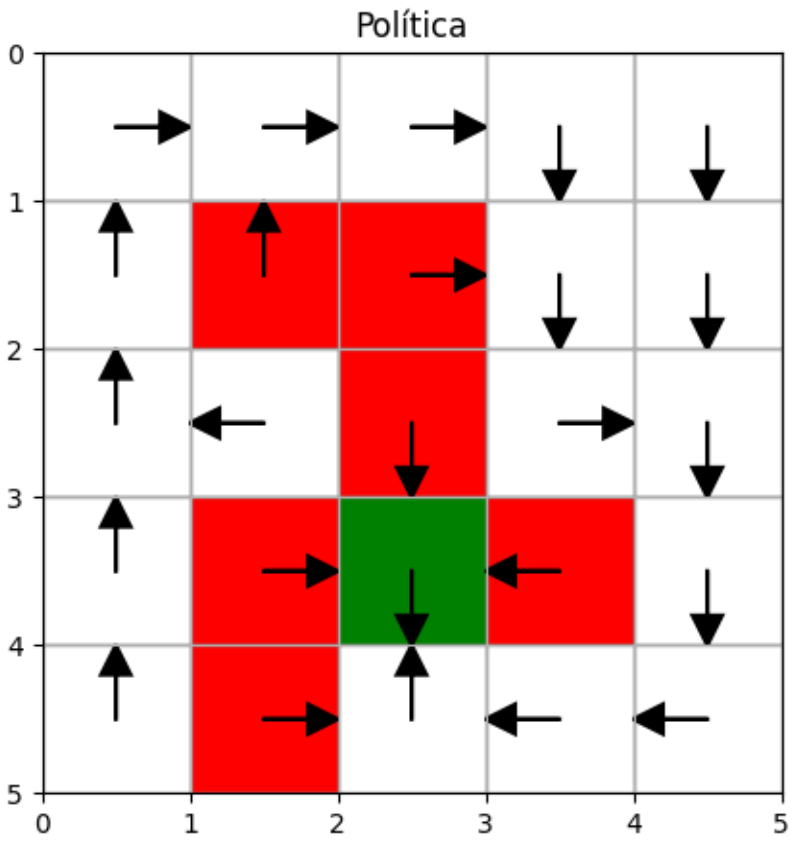
Nesse caso, o agente já considera recompensas futuras, e até conseguiu gerar uma política ótima, mas ainda assim o valor de estado é baixo, pois o agente não considera recompensas futuras com a mesma importância que as recompensas imediatas. Isso também explica o número de iterações, que foi maior do que no caso anterior, mas ainda assim baixo.

3.0.3 $\gamma = 0.9$

[10]:
```
V3, Q3, politica3 = iteracao_de_valor(ambiente, gamma=0.9, theta=1e-6,␣
↪max_iteracoes=1000)
plot_valores_de_estado(V3, ambiente)
plot_policy(ambiente, politica3)
```

```
Convergiu com 133 interações.
```


Valores de Estado (V(s))
1.8
2.0
2.3
2.5
2.8
1.7
1.8
2.5
2.8
3.1
1.5
1.3
5.3
3.1
3.5
1.3
5.3
4.7
5.3
3.8
1.2
4.7
5.3
4.7
4.3
0
1
2
3
4
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5



Agora, usando um valor de gamma mais alto, vemos que o agente, além de encontrar uma política ótima, também encontrou valores de estados mais altos. Isso ocorre devido o agente considerar agora as recompensas futuras e, com isso, o valor de cada estado aumenta. O número de iterações também aumentou, pois o agente agora tem que considerar mais estados e ações para encontrar a política ótima.

4 Influência de recompensas negativas

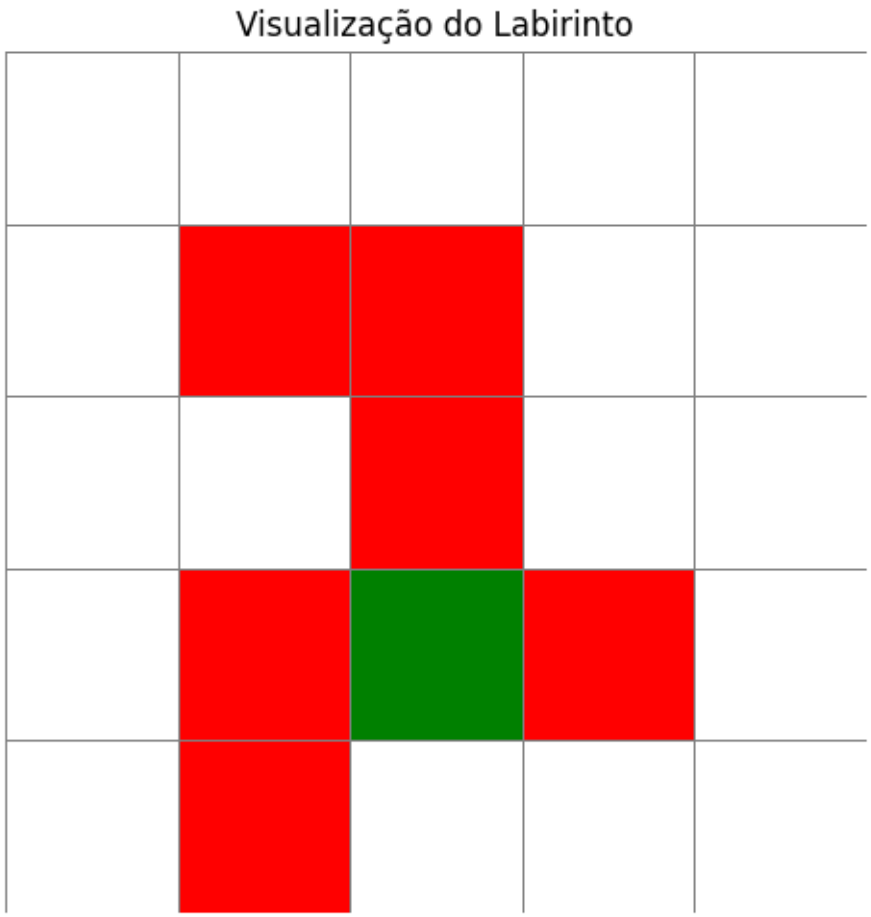
Nesta seção vamos observar como a política e os valores de estado mudam quando a penalidade para entrar em um estado ruim é menos severa, substituindo r_bad = -10 por r_bad = -1.

4.1 Configuração padrão: r_bad = -10

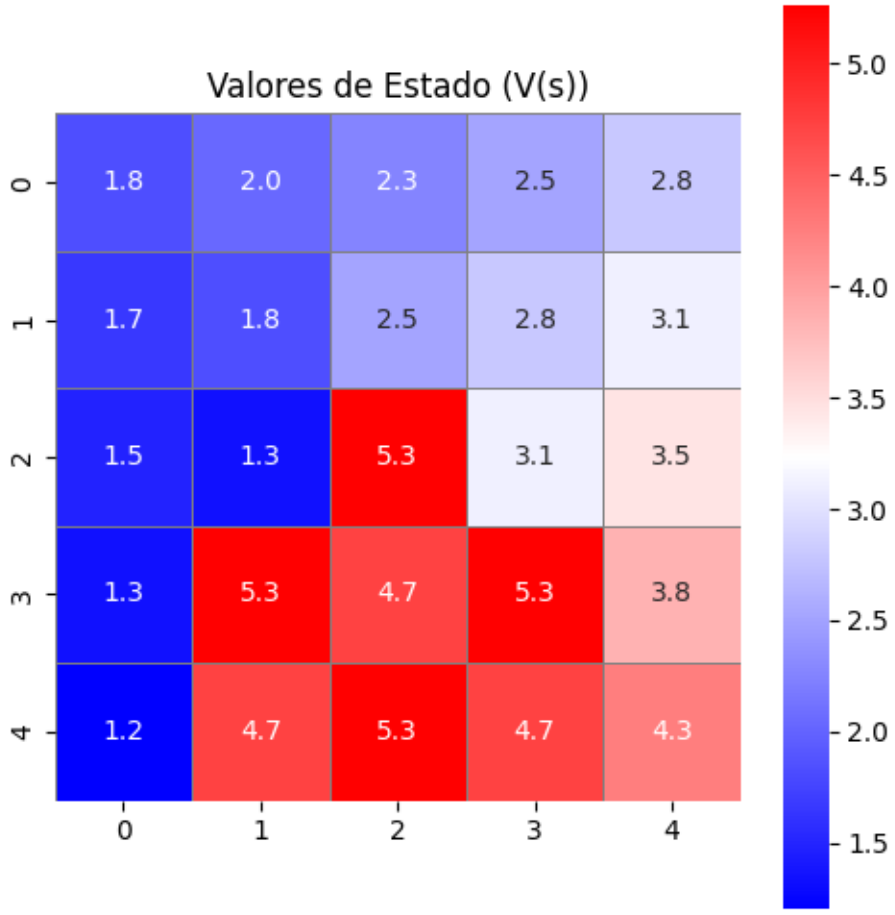
Vamos primeiro recordar a política original com a penalidade severa, para depois comparar com a nova configuração.

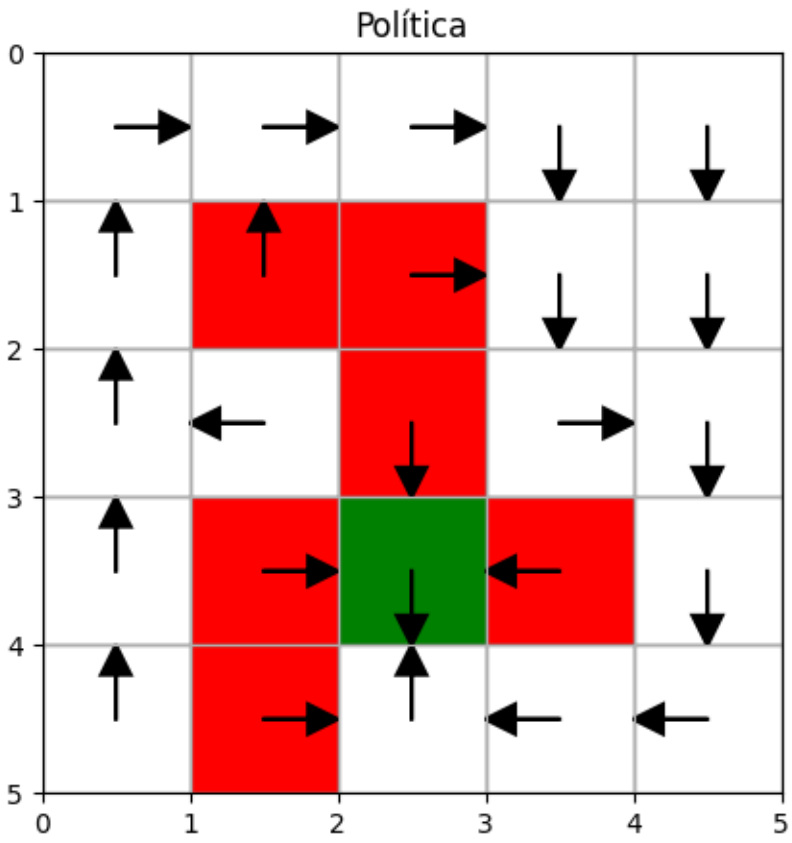
[11]:
```
ambiente_padrao = AmbienteNavegacaoLabirinto(
        world_size=(5, 5),  # Define o tamanho do ambiente como uma grade 5x5
        bad_states=[(1, 1), (1, 2), (2, 2), (3, 1), (3, 3), (4, 1)],  # Define␣
↪as posições dos estados ruins (células vermelhas)
```

```
        target_states=[(3, 2)],  # Define a posição do estado objetivo (célula␣
 ↪verde)
        allow_bad_entry=True,  # Permite que o agente entre em estados ruins
        rewards=[-1, -10, 1, 0]  # Define as recompensas: [r_boundary, r_bad,␣
 ↪r_target, r_other]
    )

plot_labirinto(ambiente_padrao)  # Visualiza o labirinto com cores␣
 ↪representando os tipos de estado

# Executa o algoritmo de iteração de valor com os parâmetros definidos
V_padrao, Q_padrao, politica_padrao = iteracao_de_valor(ambiente_padrao,␣
 ↪gamma=0.9, theta=1e-6, max_iteracoes=1000)
plot_valores_de_estado(V_padrao, ambiente_padrao)  # Visualiza os valores de␣
 ↪estado resultantes
plot_policy(ambiente_padrao, politica_padrao)  # Visualiza a política ótima␣
 ↪resultante
```



```
Convergiu com 133 interações.
```

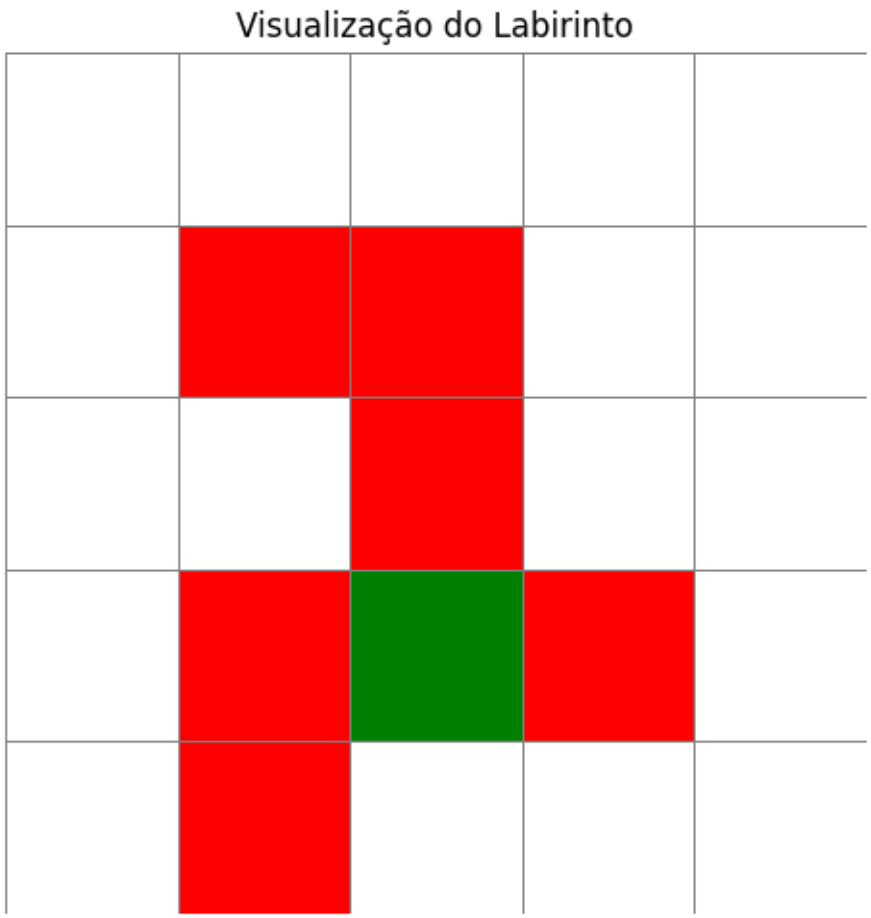




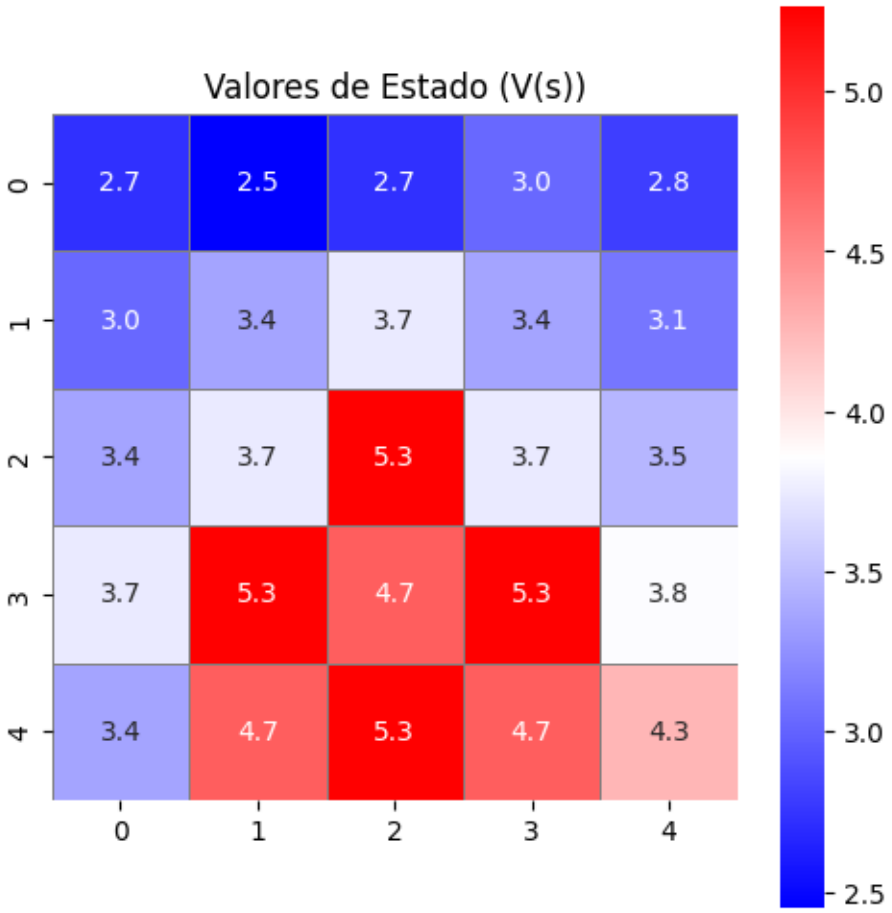
4.2 Nova configuração: r_bad = -1

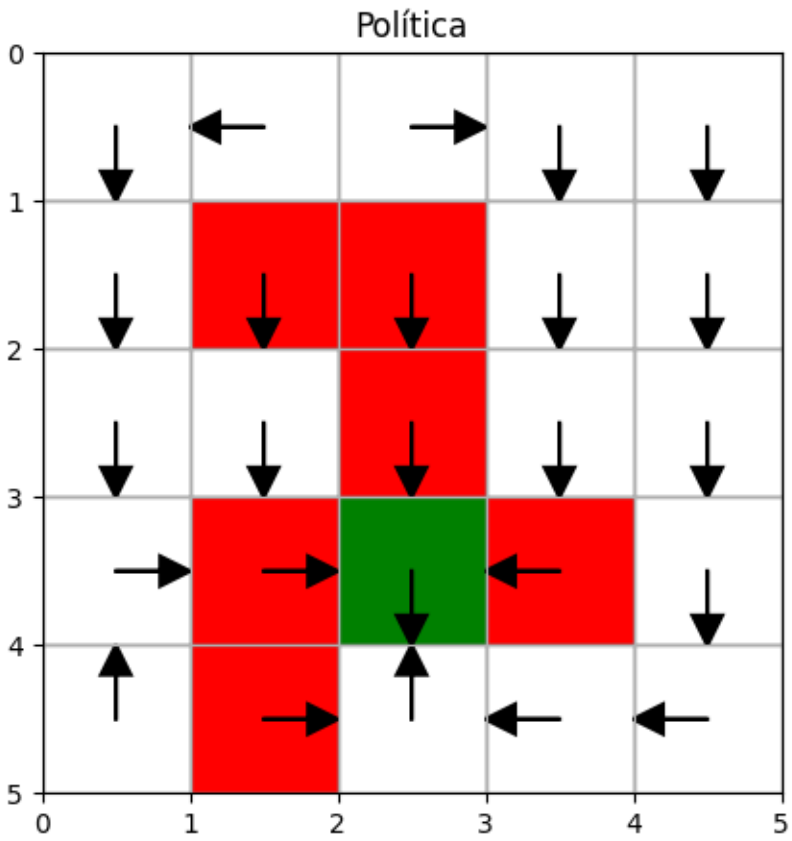
Agora, alteramos a recompensa negativa para estados ruins para -1, mantendo as demais recompensas iguais.

[12]:
```
ambiente_rbad1 = 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, -1, 1, 0]  # r_bad = -1 agora é igual a r_boundary
    )

plot_labirinto(ambiente_rbad1)

V_rbad1, Q_rbad1, politica_rbad1 = iteracao_de_valor(ambiente_rbad1, gamma=0.9,␣
 ↪theta=1e-6, max_iteracoes=1000)
plot_valores_de_estado(V_rbad1, ambiente_rbad1)
plot_policy(ambiente_rbad1, politica_rbad1)
```




```
Convergiu com 133 interações.
```


Valores de Estado (V(s))
2.7
2.5
2.7
3.0
2.8
3.0
3.4
3.7
3.4
3.1
3.4
3.7
5.3
3.7
3.5
3.7
5.3
4.7
5.3
3.8
3.4
4.7
5.3
4.7
4.3
0
1
2
3
4
5.0
4.5
4.0
3.5
3.0
2.5



4.2.1 Análise da Influência de r_bad = -1

Com r_bad = -1, a penalidade por entrar em estados ruins é muito menor do que antes (r_bad = -10). Isso provoca mudanças significativas na política e nos valores de estado:

1. **Valores de estado**: Os valores dos estados aumentam significativamente, já que a penalização reduzida faz com que os estados ruins tenham valores menos negativos.
2. **Comportamento da política**: A política se torna mais propensa a atravessar estados ruins quando isso representa um caminho mais curto para o objetivo, pois agora a penalidade de -1 é a mesma de bater em uma parede (r_boundary = -1).
3. **Rotas de navegação**: Podemos observar que o agente agora pode escolher caminhos que passam por células vermelhas para chegar mais rapidamente ao alvo, já que o custo de atravessar um estado ruim é relativamente baixo comparado ao benefício de chegar mais rapidamente ao objetivo.
4. **Convergência**: O algoritmo provavelmente converge mais rapidamente, já que a diferença entre valores de estados é menos drástica.

Esta mudança ilustra como as recompensas negativas atuam como barreiras no ambiente, e reduzir sua magnitude torna essas barreiras mais "permeáveis", alterando significativamente o comportamento do agente.

5 Influência de alterar as recompensas

Nesta seção vamos analisar como uma transformação afim nas recompensas afeta a política e os valores de estado. Uma transformação afim significa aplicar uma escala (`a`) e uma translação (`b`) em todas as recompensas:

`r' = a * r + b`

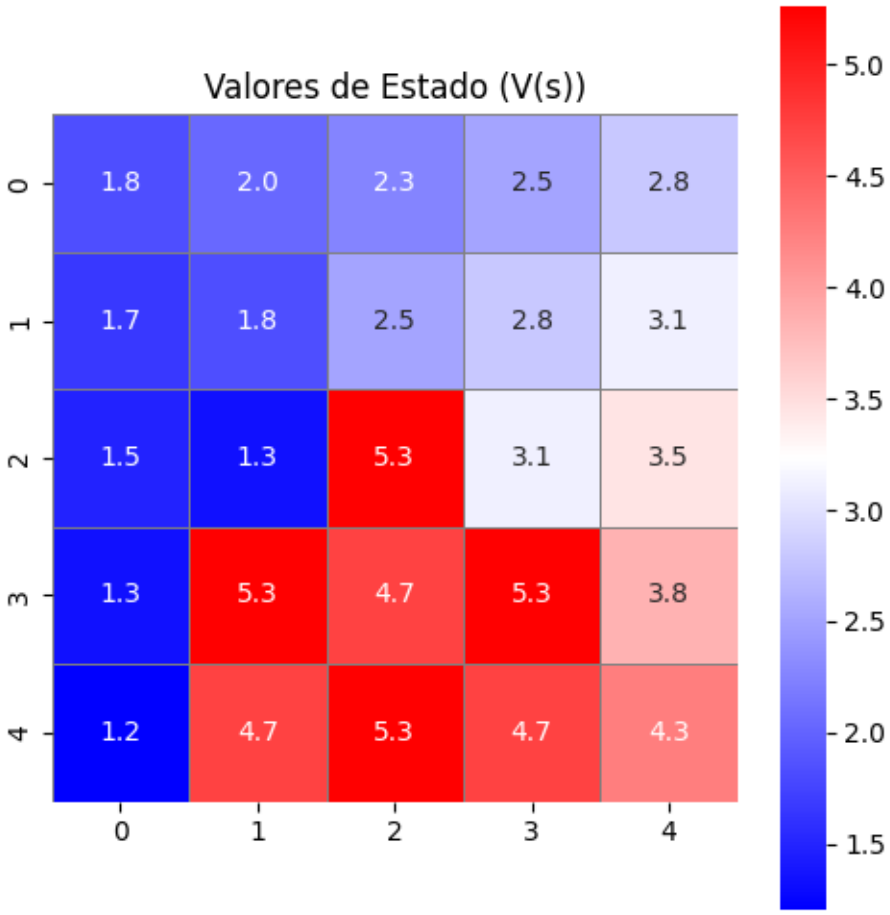
Vamos experimentar diferentes transformações afins e observar o impacto nos resultados do algoritmo de iteração de valor.

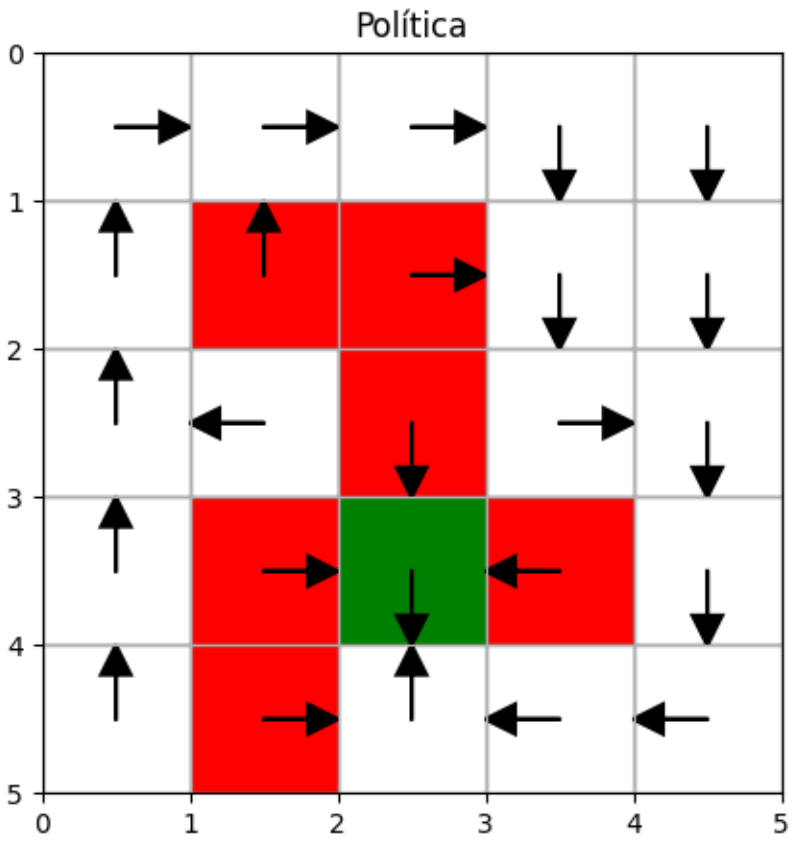
5.1 Caso base para comparação: Recompensas originais

Recordamos os valores originais de recompensa: - r_boundary = -1 - r_bad = -10 - r_target = 1 - r_other = 0

[13]:
```
# Recompensas originais
recompensas_originais = [-1, -10, 1, 0]

# Criando o ambiente com as recompensas originais
ambiente_original = 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=recompensas_originais
    )

# Executando o algoritmo
print("Política ótima com recompensas originais [-1, -10, 1, 0]:")
V_orig, Q_orig, politica_orig = iteracao_de_valor(ambiente_original, gamma=0.9,␣
 ↪theta=1e-6, max_iteracoes=1000)

# Visualizando os resultados
plot_valores_de_estado(V_orig, ambiente_original)
plot_policy(ambiente_original, politica_orig)
```

```
Política ótima com recompensas originais [-1, -10, 1, 0]:
Convergiu com 133 interações.
Convergiu com 133 interações.
```


Valores de Estado (V(s))
1.8
2.0
2.3
2.5
2.8
1.7
1.8
2.5
2.8
3.1
1.5
1.3
5.3
3.1
3.5
1.3
5.3
4.7
5.3
3.8
1.2
4.7
5.3
4.7
4.3
0
1
2
3
4
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5



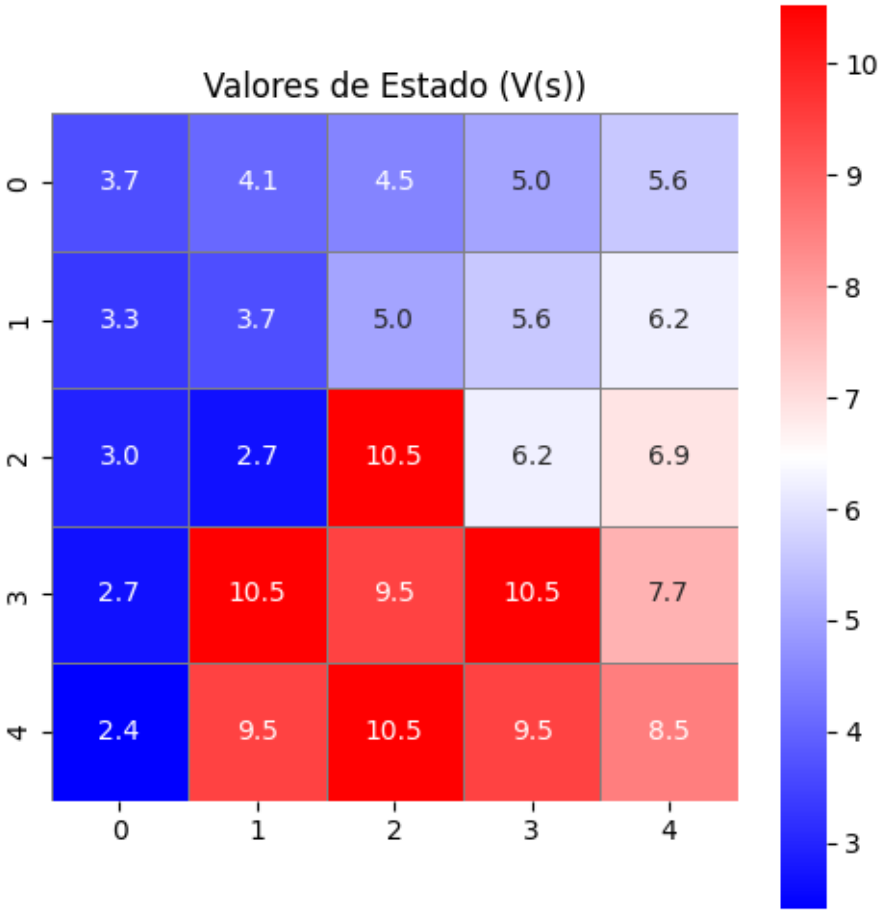
5.2 Experimento 1: Transformação afim - multiplicação (a=2, b=0)

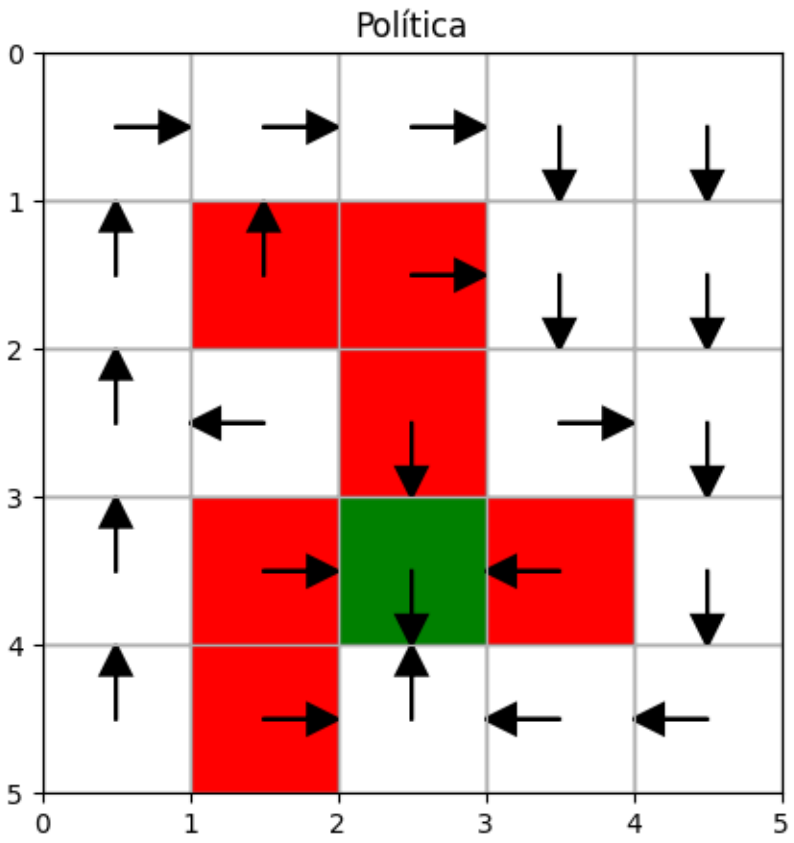
Neste caso, todas as recompensas são multiplicadas por 2, aumentando a magnitude tanto de recompensas positivas quanto negativas: - r_boundary = 2 * (-1) + 0 = -2 - r_bad = 2 * (-10) + 0 = -20 - r_target = 2 * 1 + 0 = 2 - r_other = 2 * 0 + 0 = 0

[14]:
```
# Aplicando transformação afim - multiplicação (a=2, b=0)
a, b = 2, 0
recompensas_transformadas_1 = [a * r + b for r in recompensas_originais]
print(f"Recompensas transformadas (a={a}, b={b}):␣
  ↪{recompensas_transformadas_1}")

# Criando o ambiente com as recompensas transformadas
ambiente_transformado_1 = 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=recompensas_transformadas_1
    )
```

```
# Executando o algoritmo
V_trans1, Q_trans1, politica_trans1 =␣
 ↪iteracao_de_valor(ambiente_transformado_1, gamma=0.9, theta=1e-6,␣
 ↪max_iteracoes=1000)

# Visualizando os resultados
plot_valores_de_estado(V_trans1, ambiente_transformado_1)
plot_policy(ambiente_transformado_1, politica_trans1)
```

```
Recompensas transformadas (a=2, b=0): [-2, -20, 2, 0]
Convergiu com 139 interações.
Convergiu com 139 interações.
```





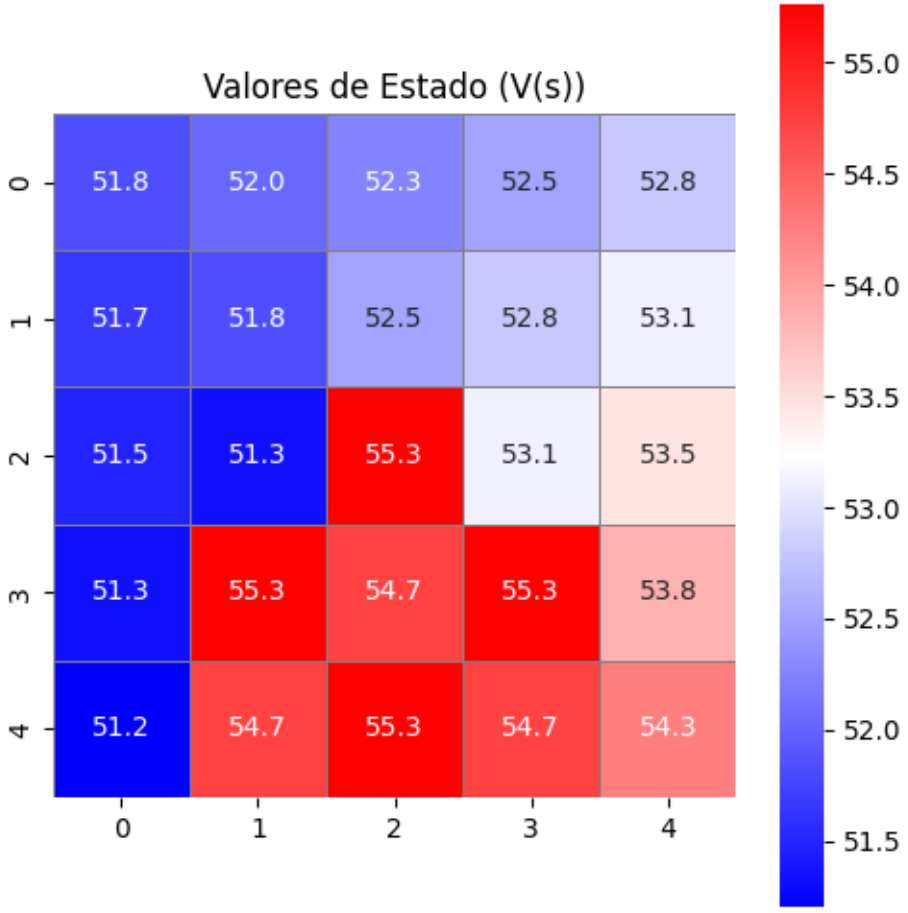
5.3 Experimento 2: Transformação afim - adição (a=1, b=5)

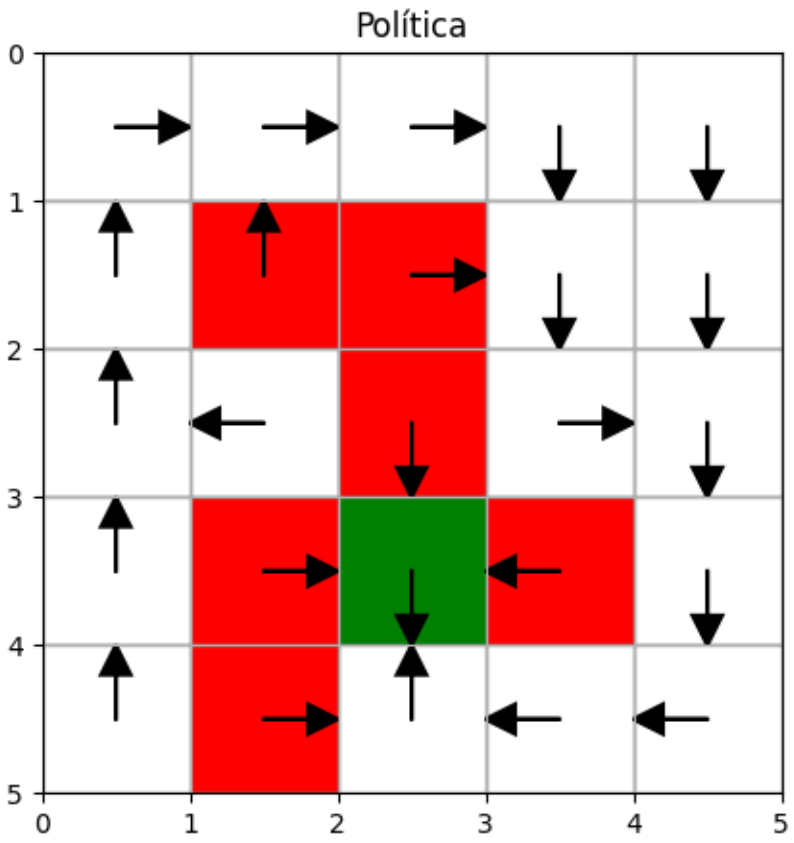
Neste caso, um valor constante de 5 é adicionado a todas as recompensas: - r_boundary = 1 * (-1) + 5 = 4 - r_bad = 1 * (-10) + 5 = -5 - r_target = 1 * 1 + 5 = 6 - r_other = 1 * 0 + 5 = 5

[15]:
```
# Aplicando transformação afim - adição (a=1, b=5)
a, b = 1, 5
recompensas_transformadas_2 = [a * r + b for r in recompensas_originais]
print(f"Recompensas transformadas (a={a}, b={b}):␣
 ↪{recompensas_transformadas_2}")

# Criando o ambiente com as recompensas transformadas
ambiente_transformado_2 = 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=recompensas_transformadas_2
    )
```

```
# Executando o algoritmo
V_trans2, Q_trans2, politica_trans2 =␣
 ↪iteracao_de_valor(ambiente_transformado_2, gamma=0.9, theta=1e-6,␣
 ↪max_iteracoes=1000)

# Visualizando os resultados
plot_valores_de_estado(V_trans2, ambiente_transformado_2)
plot_policy(ambiente_transformado_2, politica_trans2)
```

```
Recompensas transformadas (a=1, b=5): [4, -5, 6, 5]

Convergiu com 150 interações.
```





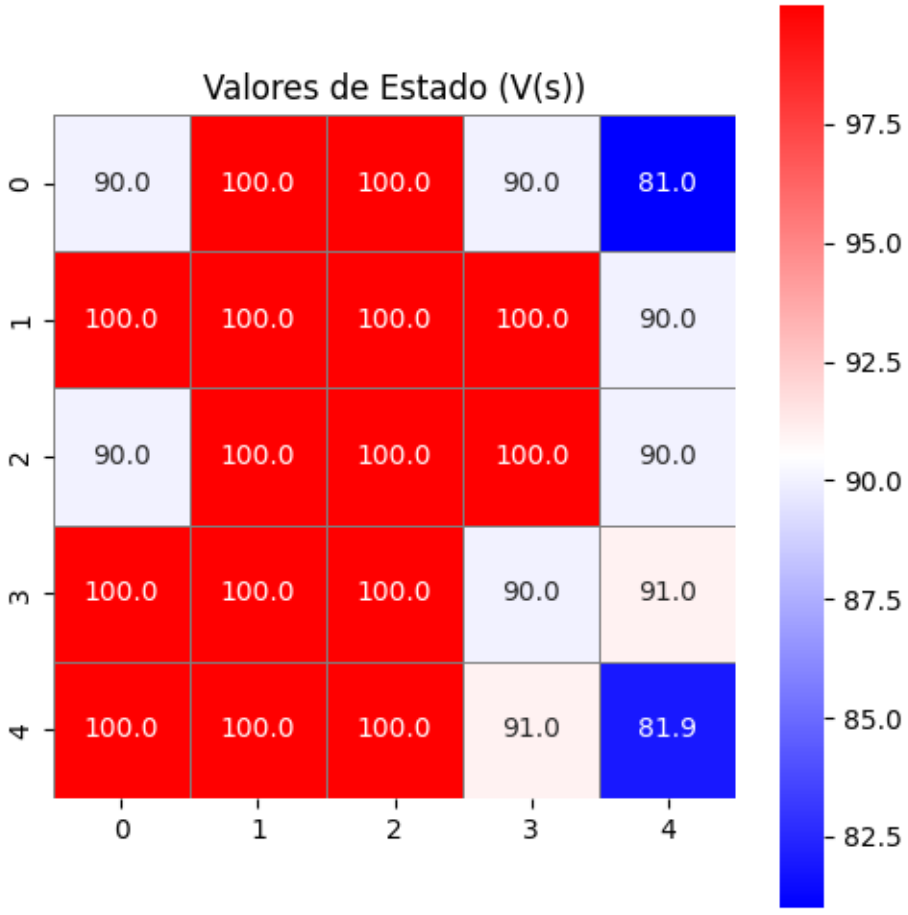
5.4 Experimento 3: Transformação afim - inversão de sinal (a=-1, b=0)

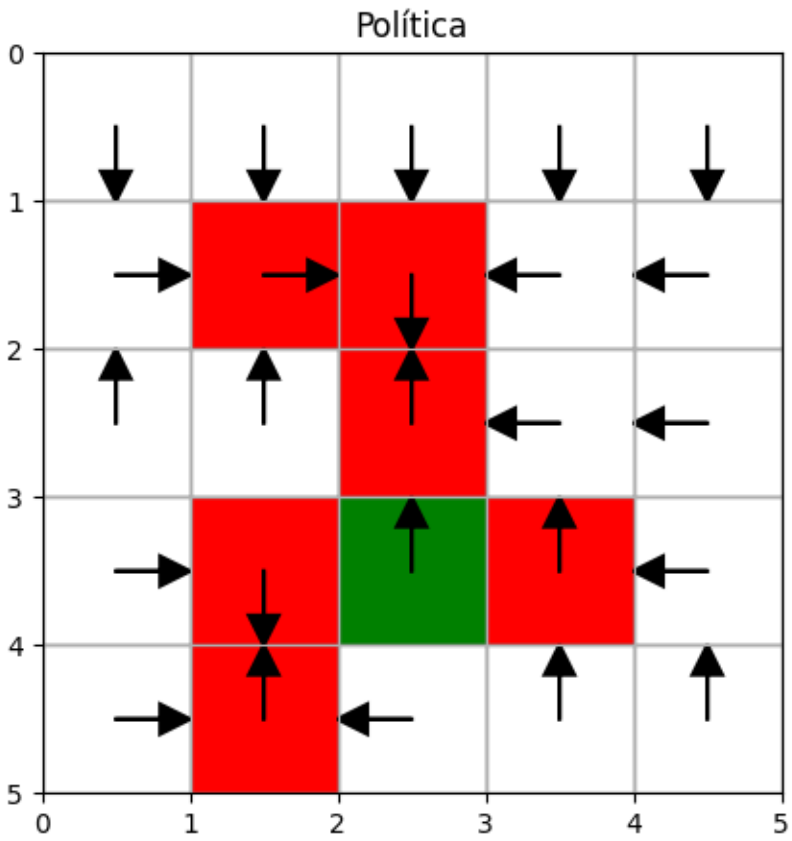
Neste caso, invertemos o sinal de todas as recompensas: - r_boundary = -1 * (-1) + 0 = 1 - r_bad = -1 * (-10) + 0 = 10 - r_target = -1 * 1 + 0 = -1 - r_other = -1 * 0 + 0 = 0

[16]:
```
# Aplicando transformação afim - inversão de sinal (a=-1, b=0)
a, b = -1, 0
recompensas_transformadas_3 = [a * r + b for r in recompensas_originais]
print(f"Recompensas transformadas (a={a}, b={b}):␣
 ↪{recompensas_transformadas_3}")

# Criando o ambiente com as recompensas transformadas
ambiente_transformado_3 = 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=recompensas_transformadas_3
    )
```

```
# Executando o algoritmo
V_trans3, Q_trans3, politica_trans3 =␣
 ↪iteracao_de_valor(ambiente_transformado_3, gamma=0.9, theta=1e-6,␣
 ↪max_iteracoes=1000)

# Visualizando os resultados
plot_valores_de_estado(V_trans3, ambiente_transformado_3)
plot_policy(ambiente_transformado_3, politica_trans3)
```

```
Recompensas transformadas (a=-1, b=0): [1, 10, -1, 0]
Convergiu com 154 interações.
Convergiu com 154 interações.
```





5.5 Análise das Transformações Afins

Após realizar estes experimentos com diferentes transformações afins nas recompensas, podemos observar os seguintes efeitos:

5.5.1 1. Transformação de escala (a=2, b=0)

- **Efeito nos valores de estado**: Os valores de estado são aproximadamente multiplicados pelo mesmo fator. A escala de valores muda, mas a relação entre os valores de estados diferentes se mantém proporcional.
- **Efeito na política**: A política ótima permanece inalterada. Isso ocorre porque a ordem de preferência entre as ações para cada estado não muda quando multiplicamos todas as recompensas por uma constante positiva.
- **Número de iterações até convergência**: Pode aumentar, pois a diferença entre valores de estado aumenta, tornando o critério de convergência mais difícil de ser satisfeito.

5.5.2 2. Transformação de translação (a=1, b=5)

- **Efeito nos valores de estado**: Todos os valores de estado aumentam, mas não uniformemente. A presença do fator de desconto faz com que estados mais distantes do objetivo recebam menos influência da constante aditiva.

- **Efeito na política**: Em geral, a política ótima tende a se manter inalterada para valores de mais próximos de 1. Para valores menores de , pode haver mudanças.
- **Aspecto teórico**: A adição de uma constante a todas as recompensas não deveria alterar a política ótima em MDPs de horizonte infinito com desconto, pois isso representa apenas uma mudança na "linha de base" das recompensas.

5.5.3 3. Inversão de sinal (a < 0)

- **Efeito dramático**: Inverte completamente o objetivo do agente. O que antes era para ser maximizado agora deve ser minimizado.
- **Política resultante**: É o oposto da política original - o agente agora tenta evitar o alvo original e busca ativamente os estados ruins.
- **Valores de estado**: São invertidos em relação aos valores originais.

5.5.4 Conclusão sobre transformações afins

As transformações afins das recompensas ilustram propriedades importantes do aprendizado por reforço:

1. **Escalamento positivo (a > 0, b = 0)**: Preserva a política ótima, modificando apenas a escala dos valores.
2. **Adição de constante (a = 1, b 0)**: Em teoria, não deveria alterar a política ótima para próximo de 1, mas na prática pode influenciar a velocidade de convergência e alterar ligeiramente a política para valores menores de .
3. **Inversão de sinal (a < 0)**: Inverte completamente o problema, fazendo o agente buscar o oposto do que era desejado originalmente.