

Universidade Federal do Maranhão  
Engenharia da Computação  
Curso de Inteligência Artificial  
Prof. Dr.: Thales Levi Azevedo Valente

Alunos e matrículas:

Gabriel Felipe Carvalho Silva - 2023098664  
Judson Rodrigues Ciribelli Filho - 2019038973  
Giordano Bruno de Araujo Mochel - 2019004080

## **RELATÓRIO DO JOGO DE LABIRINTO EM PYTHON**

São Luís - MA

2024

Gabriel Felipe Carvalho Silva

Judson Rodrigues Ciribelli Filho

Giordano Bruno de Araujo Mochel

## **RELATÓRIO DO JOGO DE LABIRINTO EM PYTHON**

Este relatório tem como objetivo detalhar as implementações realizadas na implementação do jogo de labirinto em Python bem como os conceitos utilizados para definir o comportamento do agente.

São Luís – MA

2024

## **Sumário**

<b>1 Introdução</b>	<b>4</b>
<b>2 Desenvolvimento</b>	<b>5</b>
<b>3 Resultados e Discussão</b>	<b>12</b>
<b>4 Conclusão</b>	<b>14</b>

## 1 Introdução

O projeto consiste no desenvolvimento de um sistema inteligente para a resolução de labirintos utilizando Python como linguagem de programação. A proposta inicial é implementar um agente autônomo capaz de explorar e encontrar a saída de um labirinto de forma eficiente, armazenando informações sobre sua interação com o ambiente em uma memória persistente. Essa abordagem permite ao agente melhorar seu desempenho em tentativas subsequentes, evitando caminhos previamente identificados como becos sem saída.

O sistema é composto por diversos módulos que realizam tarefas como geração de labirintos, navegação do agente, armazenamento de dados em arquivos JSON e interação com o usuário por meio de uma interface gráfica desenvolvida com a biblioteca **tkinter**. A lógica do agente utiliza técnicas baseadas em pilhas para explorar o labirinto, retornando ao ponto de decisão mais recente quando encontra um beco sem saída. Além disso, o nosso projeto foi projetado para ser modular e escalável, facilitando sua manutenção e o desenvolvimento de futuras melhorias.

## 2 Desenvolvimento

No desenvolvimento, foi realizada a modulação das funções de criação do labirinto, agente, interface baseando-se na teoria de algoritmo de busca em profundidade (DFS) e seus estudos. O projeto está estruturado da seguinte forma:

- main.py
- maze\_generator.py
- maze\_interface.py
- maze\_solver.py
- learning\_agent.py
- labirinto\_utils.py
- amostragem\_utils.py
- requests.txt

O arquivo **main.py** é responsável por gerenciar a lógica do código, chamando os módulos e libs necessários. Como é mostrado no Algoritmo 1 abaixo, temos a função única e principal,

Algoritmo 1 - Função principal do código

---

**Algoritmo 1:** Main.py

---

```
1. from maze_generator import generate_maze
2. from maze_interface import MazeApp
3. from labirinto_utils import load_labirinto, save_labirinto
4. import tkinter as tk
5.
6. def main():
7.     """
8.     Função principal para executar o programa.
9.     """
10.    width, height = 21, 21 # Dimensões do labirinto
11.
12.    # Tenta carregar o labirinto salvo
13.    maze = load_labirinto()
14.    if maze is None:
15.        print("Gerando novo labirinto fixo...")
16.        maze = generate_maze(width, height)
17.        save_labirinto(maze) # Salva o labirinto para futuras execuções
18.
19.    # Inicializa a interface gráfica
20.    root = tk.Tk()
21.    root.title("Labirinto - Agente com Aprendizado")
22.    app = MazeApp(root, maze)
23.    root.mainloop()
24.
25. if __name__ == "__main__":
26.     main()
```

---

Nesse arquivo observa-se a importação de três módulos e uma lib (tkinter). O módulo **labirinto\_utils** possui duas funções responsáveis por salvar em json e restaurar (ler do json) os dados do labirinto gerado de forma randômica na primeira vez que o código é executado. O módulo **maze\_generator** e **maze\_interface** são responsáveis por criar e configurar (por cores e formas) o labirinto, respectivamente, e serão abordados mais à frente. A biblioteca, por sua vez, é responsável pela interface gráfica (GUI) baseada em janela.

Na função main, observa-se a definição do tamanho da tela padrão da interface gráfica (21x21) e em seguida é criado ou lido os dados para o labirinto. As últimas ações estão relacionadas com a criação da interface gráfica e a execução do labirinto com a movimentação do agente em loop (frame a frame).

Analisando o gerador do labirinto (**maze\_generator**) no Algoritmo 2, pode-se observar que a função **generate\_maze** cria uma matriz bidimensional de tamanho definido anteriormente e com valor fixo de 1 (parede). A função recursiva **carve\_passage** é responsável por gerar os caminhos livres para o agente ao mudar as células de 1 para 0. A direção do caminho é definida de forma aleatória (random.shuffle) a partir da posição relativa cx e cy, considerando os limites do labirinto. Ao encontrar uma célula válida para “cavar”, ele altera seu valor e chama recursivamente a próxima posição.

---

**Algoritmo 2:** Maze\_generator

---

```
1. import random
2.
3. def generate_maze(width, height):
4.     """
5.     Gera um labirinto usando busca em profundidade recursiva.
6.     """
7.     maze = [[1 for _ in range(width)] for _ in range(height)]
8.
9.     def carve_passages(cx, cy):
10.         directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
11.         random.shuffle(directions)
12.         for dx, dy in directions:
13.             nx, ny = cx + dx * 2, cy + dy * 2
14.             if 0 <= nx < width and 0 <= ny < height and maze[ny][nx] == 1:
15.                 maze[cy + dy][cx + dx] = 0
16.                 maze[ny][nx] = 0
17.                 carve_passages(nx, ny)
18.
19.     maze[1][1] = 0
20.     carve_passages(1, 1)
21.     return maze
```

```
22.  
23. # Gera aleatoriedade  
24. random.seed()
```

---

Fonte: Autores (2024)

Essa última função é iniciada sempre na posição [1][1] do labirinto.

No módulo **maze\_interface** é definida a classe que configura a parte visual do labirinto (Algoritmo 3), além de chamar o agente para realizar o percurso.

---

#### Algoritmo 3: Maze\_Interface

---

```
1. import tkinter as tk  
2. from learning_agent import LearningAgent  
3.  
4. class MazeApp:  
5.     def __init__(self, root, maze):  
6.         """  
7.         Inicializa a interface gráfica com o agente DFS.  
8.         """  
9.         self.root = root  
10.        self.maze = maze  
11.        self.cell_size = 20 # Tamanho de cada célula  
12.        self.canvas = tk.Canvas(  
13.            root,  
14.            width=len(maze[0]) * self.cell_size,  
15.            height=len(maze) * self.cell_size  
16.        )  
17.        self.canvas.pack()  
18.  
19.        # Inicializa o agente DFS  
20.        self.agent = LearningAgent(  
21.            canvas=self.canvas,  
22.            cell_size=self.cell_size,  
23.            start=(1, 1), # Posição inicial do agente  
24.            maze=self.maze  
25.        )  
26.  
27.        # Desenha o labirinto na interface gráfica  
28.        self.draw_maze()  
29.  
30.        # Inicia o movimento do agente
```

```
31. self.agent.dfs()
```

---

Fonte: Autores (2024)

Ainda no Algoritmo 3, o construtor (**init**) da classe **MazeApp** recebe a instância da lib **tkinter** e a configuração do labirinto. A partir desses dados são geradas as configurações e variáveis da classe referentes ao labirinto (**root**, **maze**, **canvas** e **pack**) e ao agente (**agent**). Por último, é chamado o método para desenhar o labirinto e o código de execução do agente por **DSF**.

No Algoritmo 4 é apresentado o código do labirinto em gráfico (cor e forma retangular), onde a parede é definida como preto, o caminho livre branco e a chegada (ou saída do labirinto) como vermelho - sendo por padrão a última célula.

---

**Algoritmo 4:** Desenho do labirinto.

---

```
33. def draw_maze(self):
34.     """
35.     Desenha o labirinto na interface gráfica.
36.     """
37.     for y, row in enumerate(self.maze):
38.         for x, cell in enumerate(row):
39.             if cell == 1:
40.                 color = "black" # Preto para paredes
41.             elif (x, y) == (len(self.maze[0]) - 2, len(self.maze) - 2):
42.                 color = "red" # Vermelho para linha de chegada
43.             else:
44.                 color = "white" # Branco para caminhos
45.             self.canvas.create_rectangle(
46.                 x * self.cell_size, y * self.cell_size,
47.                 (x + 1) * self.cell_size, (y + 1) * self.cell_size,
48.                 fill=color, outline="gray"
49.             )
```

---

Fonte: Autores (2024).

Por fim, o agente e seu movimento são definidos pela classe **LeaningAgent** instanciada no construtor da classe **MazeApp**. No Algoritmo 5 é apresentado o construtor da **LeaningAgent**. Pode-se observar que são criadas as variáveis necessárias para definir o posicionamento, o labirinto e as variáveis que irão orientar e gravar a memória do agente.



---

**Algoritmo 5:** Classe de movimento do agente

---

```
1. class LearningAgent:
2.     def __init__(self, canvas, cell_size, start, maze):
3.         """
4.         Inicializa o agente com lógica DFS e aprendizado usando amostragem.json.
5.         """
6.         self.canvas = canvas
7.         self.cell_size = cell_size:
8.         self.start = start # Posição inicial do agente
9.         self.maze = maze
10.        self.amostragem = load_amostragem() # Carrega a amostragem do arquivo
11.        self.visited = set() # Células visitadas nesta execução
12.        self.directions = [(0, 1), (1, 0), (0, -1), (-1, 0)] # Movimentos possíveis
13.        self.agent_visual = None # Representação gráfica do agente
```

---

Fonte: Autores.

O método **dfs** dessa classe é responsável pela lógica de DFS com o aprendizado baseado no parâmetro **self.amostragem** que carrega a memória do agente. Nos Algoritmos subsequentes serão apresentadas parte a parte a lógica do método.

No Algoritmo 6, é apresentado o início do método DFS que implementa a lógica principal do agente para navegar no labirinto. Primeiramente, é definida uma pilha (stack), que será utilizada para armazenar o estado atual (posição) do agente durante o percurso. Além disso, é inicializado o path, que funcionará como uma memória para registrar o caminho percorrido.

---

**Algoritmo 6:** Método DFS (Parte 1)

---

```
17. def dfs(self):
18.     """
19.     Implementa a lógica de DFS com aprendizado baseado na tabela amostragem.json.
20.     """
21.     stack = [self.start] # Pilha para DFS
22.     path = []
23.
24.     while stack:
25.         state = stack[-1] # Pega o estado atual no topo da pilha
26.
27.         # Verifica se o agente chegou ao objetivo
28.         if self.is_goal(state):
29.             print("Linha de chegada alcançada!")
30.             save_amostragem(self.amostragem)
```

```
31.         return True
32.
33.     # Marca o estado como visitado
34.     if state not in self.visited:
35.         self.visited.add(state) # Adiciona à lista de visitados
36.         path.append(state) # Salva no caminho atual
37.
38.     # Obtém as ações válidas a partir do estado atual
39.     valid_actions = self.get_valid_actions(state)
```

---

Fonte: Autores (2024)

Dentro do loop principal, o state (estado atual) é extraído do topo da pilha, representando a posição do agente no labirinto naquele momento. A primeira condição avaliada é se o agente alcançou o objetivo, utilizando o método **is\_goal()**. Esse método verifica se a posição atual corresponde à última célula do labirinto, indicando o fim do trajeto. Caso esteja nessa posição, o agente salva as informações de aprendizado na tabela de amostragem e encerra o método com sucesso. Caso contrário, o agente marca o estado como visitado e adiciona-o ao caminho percorrido (visited e path).

Em seguida, o agente obtém as ações válidas a partir do estado atual usando o método **get\_valid\_actions()** (Algoritmo 8), que verifica quais movimentos estão dentro dos limites do labirinto, não levam a paredes e não correspondem a células já visitadas ou marcadas como becos sem saída.

---

**Algoritmo 8:** Método para validar ações possíveis do agente.

---

```
66.     def get_valid_actions(self, state):
67.         """
68.         Retorna as ações válidas a partir do estado atual, considerando o labirinto e visitas.
69.         """
70.         return [
71.             action for action in self.directions
72.             if self.is_valid((state[0] + action[0], state[1] + action[1]))
73.         ]
```

---

Fonte: Autores

No Algoritmo 9 é apresentada a continuação da lógica DFS. Nessa etapa, se houver ações válidas, o método **choose\_action()** seleciona a melhor com base na tabela de amostragem (Algoritmo 10). Essa escolha prioriza ações com menor penalidade, registrando células marcadas como becos sem saída. O agente então se move para o próximo estado, atualiza sua posição graficamente por meio do método **move\_agent()** (Algoritmo 11) e adiciona o novo estado à pilha.

---

**Algoritmo 9:** Método DFS (Parte 2)

---

```
41. if valid_actions:
42.     # Escolhe a melhor ação com base na amostragem
43.     action = self.choose_action(state, valid_actions)
44.     next_state = (state[0] + action[0], state[1] + action[1])
45.
46.     # Move o agente graficamente
47.     self.move_agent(state, next_state)
48.
49.     # Adiciona o próximo estado à pilha
50.     stack.append(next_state)
51. else:
52.     print(f'Beco sem saída na célula {state}. Retornando para o estado anterior.')
53.     self.register_error(state)
54.     save_amostragem(self.amostragem)
55.     stack.pop()
56.     if stack:
57.         previous_state = stack[-1]
58.         # Anima o movimento de retorno
59.         self.move_agent(state, previous_state)
60.
61. print("Caminho sem solução!")
62. save_amostragem(self.amostragem)
63. return False
```

---

Fonte: Autores (2024)

Se o agente não encontrar ações válidas, ele detecta que está em um beco sem saída, registrado pelo método **register\_error()** na tabela de amostragem. O agente então retorna ao estado anterior na pilha, movendo-se graficamente para trás, **self.move\_agent(state, previous\_state)**. Em último caso é informado que o caminho não há solução e a amostragem é salva retornando falso no loop.

---

**Algoritmo 10:** Método choose\_action.

---

```
75. def choose_action(self, state, valid_actions):
76.     """
77.     Escolhe a melhor ação com base na tabela amostragem.json.
78.     """
79.     best_action = None
80.     best_score = float("inf") # Queremos o menor "peso" ou penalidade
```

```

81.
82.     for action in valid_actions:
83.         next_state = (state[0] + action[0], state[1] + action[1])
84.         score = self.amostragem["visited"].get(str(next_state), 0) # Menor penalidade é
melhor
85.
86.         # Penaliza células marcadas como becos sem saída
87.         if str(next_state) in self.amostragem["errors"]:
88.             score += 100 # Penalidade alta para erros
89.
90.         if score < best_score:
91.             best_score = score
92.             best_action = action

```

---

Fonte: Autores (2024)

---

**Algoritmo 11:** Método move\_agent.

---

```

121. def move_agent(self, current_state, next_state):
122.     """
123.     Move o agente graficamente de uma célula para outra.
124.     """
125.     cx, cy = current_state
126.     nx, ny = next_state
127.
128.     # Apaga o agente na célula atual
129.     if self.agent_visual:
130.         self.canvas.delete(self.agent_visual)
131.
132.     # Desenha o agente na nova célula
133.     self.agent_visual = self.canvas.create_rectangle(
134.         nx * self.cell_size, ny * self.cell_size,
135.         (nx + 1) * self.cell_size, (ny + 1) * self.cell_size,
136.         fill="blue", outline="gray"
137.     )
138.     self.canvas.update()
139.     self.canvas.after(100)

```

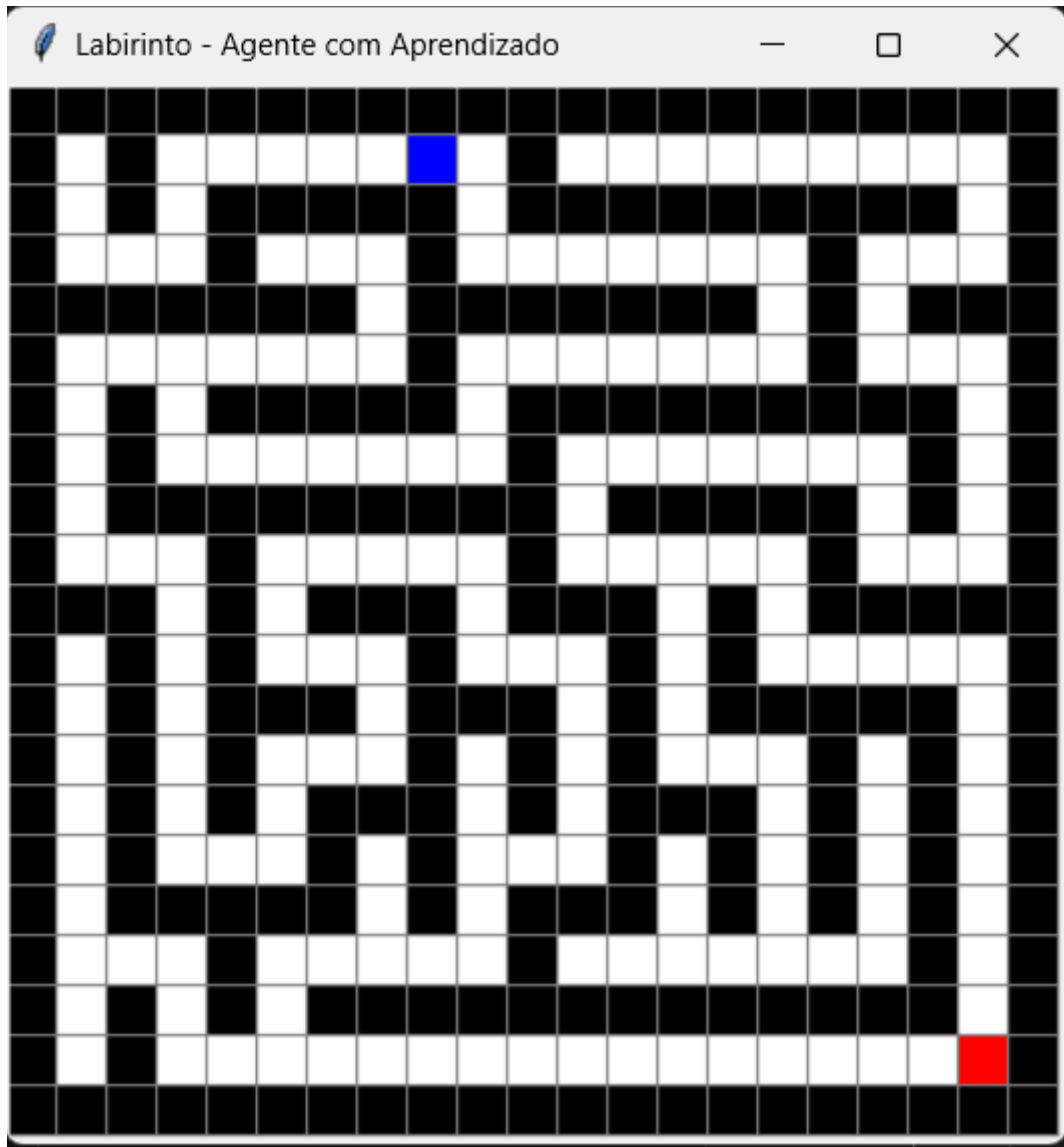
---

Fonte: Autores (2024)

### 3 Resultados e Discussão

O resultado final para o labirinto é apresentado na Figura 1 a seguir.

Figura 1 - Labirinto.



Fonte: Autores (2024)

Em azul é apresentado o agente que irá percorrer o caminho livre em branco até encontrar o ponto final do labirinto apresentado pela cor vermelha. As partes em preto são as paredes, ou seja, células em que o agente não pode transpor. O mapa é recriado ou então resgatado de um json dentro do repositório. Para a recriação do mapa, um labirinto diferente será construído, preservando as células de início e fim.

No Algoritmo 12, é apresentado o json criado a partir das posições sem saída encontradas pelo agente durante a exploração do labirinto. Esse arquivo é a memória do agente, do qual o impede de repetir o mesmo caminho que resultou em um ponto sem saída. Enquanto esse arquivo estiver com dados de memória, o agente não repetirá o mesmo

caminho que se mostrou errado. Do contrário, há a probabilidade do agente tentar o caminho e chegar nos mesmos resultados, repopulando (ou criando) esse json.

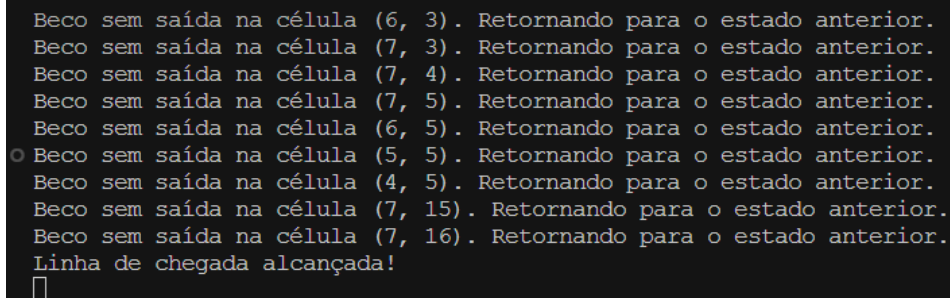
#### Algoritmo 12 - Amostragem.json

```
{  
  "errors": [  
    "(5, 3)",  
    "(6, 3)",  
    "(7, 3)",  
    "(7, 4)",  
    "(7, 5)",  
    "(6, 5)",  
    "(5, 5)",  
    "(4, 5)"  
  ]  
}
```

Fonte: Autores (2024)

Também é possível acompanhar o desenvolvimento do agente pelo console, com informativos das células consideradas becos sem saída, informando seu movimento de *backtracking* e com o momento de chegada (Figura 2).

Figura 2 - Visualização do console.



```
Beco sem saída na célula (6, 3). Retornando para o estado anterior.  
Beco sem saída na célula (7, 3). Retornando para o estado anterior.  
Beco sem saída na célula (7, 4). Retornando para o estado anterior.  
Beco sem saída na célula (7, 5). Retornando para o estado anterior.  
Beco sem saída na célula (6, 5). Retornando para o estado anterior.  
○ Beco sem saída na célula (5, 5). Retornando para o estado anterior.  
Beco sem saída na célula (4, 5). Retornando para o estado anterior.  
Beco sem saída na célula (7, 15). Retornando para o estado anterior.  
Beco sem saída na célula (7, 16). Retornando para o estado anterior.  
Linha de chegada alcançada!  
█
```

Fonte: Autores (2024)

## 4 Conclusão

Este trabalho desenvolveu um agente inteligente que utiliza a busca em profundidade (DFS) combinada com aprendizado incremental para resolver um labirinto. Com o uso de um arquivo de memória persistente (amostragem.json), o agente registra experiências anteriores, como becos sem saída, otimizando suas decisões em execuções futuras. A integração com Tkinter permitiu uma visualização interativa do processo, tornando a experiência mais acessível e intuitiva para o usuário.

A abordagem demonstrou como técnicas clássicas de busca podem ser aprimoradas por um aprendizado simples, criando um agente adaptativo e eficiente. Embora focado em um ambiente controlado, o sistema pode ser expandido para problemas mais complexos, como navegação robótica ou exploração de ambientes desconhecidos, incorporando heurísticas ou outros algoritmos híbridos.

No geral, o projeto mostrou de forma prática como combinar algoritmos fundamentais com aprendizado incremental. Essa solução não apenas reforça conceitos básicos de ciência da computação e inteligência artificial, mas também evidencia o potencial de melhorias futuras, como a otimização do caminho ou o uso de heurísticas para explorar ambientes dinâmicos.