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

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 na Figura 1 abaixo, temos a função principal,

Figura 1 - Função principal do código

```python
from maze_generator import generate_maze
from maze_interface import MazeApp
from labirinto_utils import load_labirinto, save_labirinto
import tkinter as tk

def main():
    """
    Função principal para executar o programa.
    """
    width, height = 21, 21  # Dimensões do labirinto

    # Tenta carregar o labirinto salvo
    maze = load_labirinto()
    if maze is None:
        print("Gerando novo labirinto fixo...")
        maze = generate_maze(width, height)
        save_labirinto(maze)  # Salva o labirinto para futuras execuções

    # Inicializa a interface gráfica
    root = tk.Tk()
    root.title("Labirinto - Agente com Aprendizado")
    app = MazeApp(root, maze)
    root.mainloop()

if __name__ == "__main__":
    main()
```

Fonte: Autores (2024)

Nesse arquivo é possível observar 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 gerando 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**) na Figura 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 **cave_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 (ramdom.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.

Figura 2 - Gerador do labirinto

```python
import random

def generate_maze(width, height):
    """
    Gera um labirinto usando busca em profundidade recursiva.
    """
    maze = [[1 for _ in range(width)] for _ in range(height)]

    def carve_passages(cx, cy):
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
        random.shuffle(directions)
        for dx, dy in directions:
            nx, ny = cx + dx * 2, cy + dy * 2
            if 0 <= nx < width and 0 <= ny < height and maze[ny][nx] == 1:
                maze[cy + dy][cx + dx] = 0
                maze[ny][nx] = 0
                carve_passages(nx, ny)

    maze[1][1] = 0
    carve_passages(1, 1)
    return maze

#Gera aleatoriedade
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 (Figura 3), além de chamar o agente para realizar o percurso.

Figura 3 - Interface do labirinto.

```
class MazeApp:
    def __init__(self, root, maze):
        """
        Inicializa a interface gráfica com o agente DFS.
        """
        self.root = root
        self.maze = maze
        self.cell_size = 20  # Tamanho de cada célula
        self.canvas = tk.Canvas(
            root,
            width=len(maze[0]) * self.cell_size,
            height=len(maze) * self.cell_size
        )
        self.canvas.pack()
        # Inicializa o agente DFS
        self.agent = LearningAgent(
            canvas=self.canvas,
            cell_size=self.cell_size,
            start=(1, 1),  # Posição inicial do agente
            maze=self.maze
        )
        # Desenha o labirinto na interface gráfica
        self.draw_maze()
        # Inicia o movimento do agente
        self.agent.dfs()
```

Fonte: Autores (2024)

A primeira função de init da classe 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.

Na Figura 5 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 como padrão a última célula.

Figura 5 - Desenho do labirinto.

```
def draw_maze(self):
    """
    Desenha o labirinto na interface gráfica.
    """
    for y, row in enumerate(self.maze):
        for x, cell in enumerate(row):
            if cell == 1:
                color = "black"  # Preto para paredes
            elif (x, y) == (len(self.maze[0]) - 2, len(self.maze) - 2):
                color = "red"  # Vermelho para linha de chegada
            else:
                color = "white"  # Branco para caminhos
            self.canvas.create_rectangle(
                x * self.cell_size, y * self.cell_size,
                (x + 1) * self.cell_size, (y + 1) * self.cell_size,
                fill=color, outline="gray"
            )
```

Fonte: Autores (2024).

Por fim, o agente e seu movimento são definidos pela classe **LeaningAgent** chamada na instância do módulo anteriormente explicado. Na instanciação (Figura 6) são criadas as variáveis necessárias para definir o posicionamento, o labirinto e as variáveis que irão orientar e gravar a memória do agente.

Figura 6 - Classe de movimento do agente

```
class LearningAgent:
    def __init__(self, canvas, cell_size, start, maze):
        """
        self.canvas = canvas
        self.cell_size = cell_size
        self.start = start  # Posição inicial do agente
        self.maze = maze
        self.amostragem = load_amostragem()  # Carrega a amostragem do arquivo
        self.visited = set()  # Células visitadas nesta execução
        self.directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # Movimentos possíveis
        self.agent_visual = None  # Representação gráfica do agente
```

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

Figura 7 - Método DFS (Parte 1)

```
def dfs(self):
    """
    Implementa a lógica de DFS com aprendizado baseado na tabela amostragem.json.
    """
    stack = [self.start]  # Pilha para DFS
    path = []

    while stack:
        state = stack[-1]  # Pega o estado atual no topo da pilha

        # Verifica se o agente chegou ao objetivo
        if self.is_goal(state):
            print("Linha de chegada alcançada!")
            save_amostragem(self.amostragem)
            return True

        # Marca o estado como visitado
        if state not in self.visited:
            self.visited.add(state)  # Adiciona à lista de visitados
            path.append(state)  # Salva no caminho atual

        # Obtém as ações válidas a partir do estado atual
        valid_actions = self.get_valid_actions(state)
```

Fonte: Autores (2024)

Figura 8 - Método DFS (Parte 2)

```
def dfs(self):
        if valid_actions:
            # Escolhe a melhor ação com base na amostragem
            action = self.choose_action(state, valid_actions)
            next_state = (state[0] + action[0], state[1] + action[1])

            # Move o agente graficamente
            self.move_agent(state, next_state)

            # Adiciona o próximo estado à pilha
            stack.append(next_state)
        else:
            print(f"Beco sem saída na célula {state}. Retornando para o estado anterior.")
            self.register_error(state)
            save_amostragem(self.amostragem)
            stack.pop()
            if stack:
                previous_state = stack[-1]
                # Anima o movimento de retorno
                self.move_agent(state, previous_state)

    print("Caminho sem solução!")
    save_amostragem(self.amostragem)
    return False
```

Fonte: Autores (2024)

Na Figura 7, é apresentado o início do método 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.

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()** (Figura 9), 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.

Figura 9 - Método para validar ações possíveis do agente.

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

Fonte: Autores

Na Figura 8 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 (Figura 10). Essa escolha prioriza ações com menor penalidade, penalizando fortemente 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()** (Figura 11) e adiciona o novo estado à pilha.

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.

Figura 10 - Método choose_action.

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

    for action in valid_actions:
        next_state = (state[0] + action[0], state[1] + action[1])
        score = self.amostragem["visited"].get(str(next_state), 0)  # Menor penalidade é melhor

        # Penaliza células marcadas como becos sem saída
        if str(next_state) in self.amostragem["errors"]:
            score += 100  # Penalidade alta para erros

        if score < best_score:
            best_score = score
            best_action = action
```

Fonte: Autores (2024)

Figura 11 - Método move_agent.

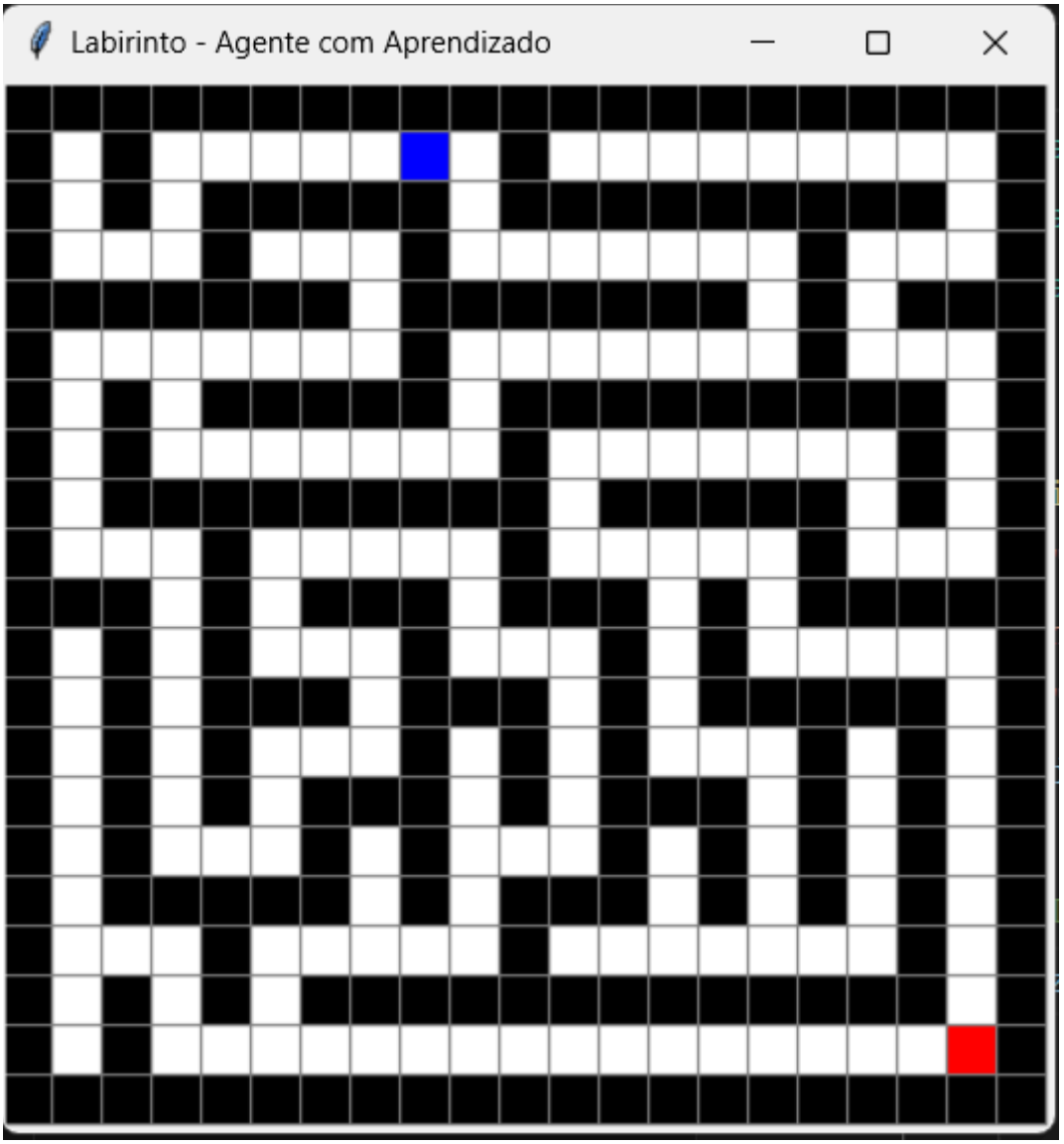
```python
def move_agent(self, current_state, next_state):
    """
    Move o agente graficamente de uma célula para outra.
    """
    cx, cy = current_state
    nx, ny = next_state

    # Apaga o agente na célula atual
    if self.agent_visual:
        self.canvas.delete(self.agent_visual)

    # Desenha o agente na nova célula
    self.agent_visual = self.canvas.create_rectangle(
        nx * self.cell_size, ny * self.cell_size,
        (nx + 1) * self.cell_size, (ny + 1) * self.cell_size,
        fill="blue", outline="gray"
    )
    self.canvas.update()
    self.canvas.after(100)
```

Fonte: Autores (2024)

3 Resultados e Discussão

O resultado final para o labirinto é apresentado na Figura 12 a seguir.

Figura 12 - Labirinto.



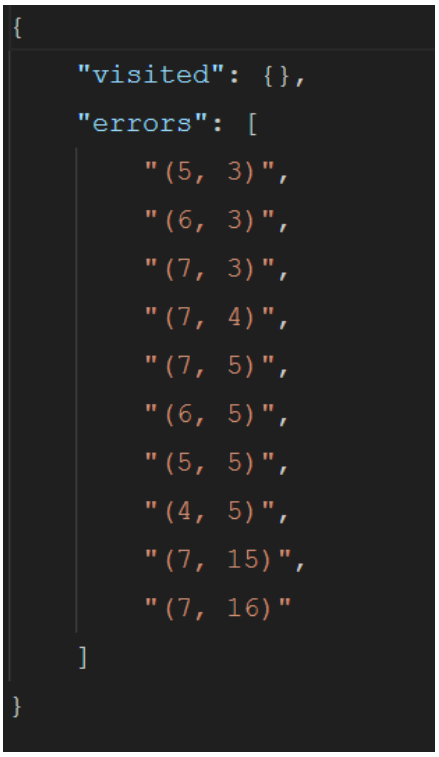
Fonte: Autores (2024)

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.

Na Figura 13, 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.

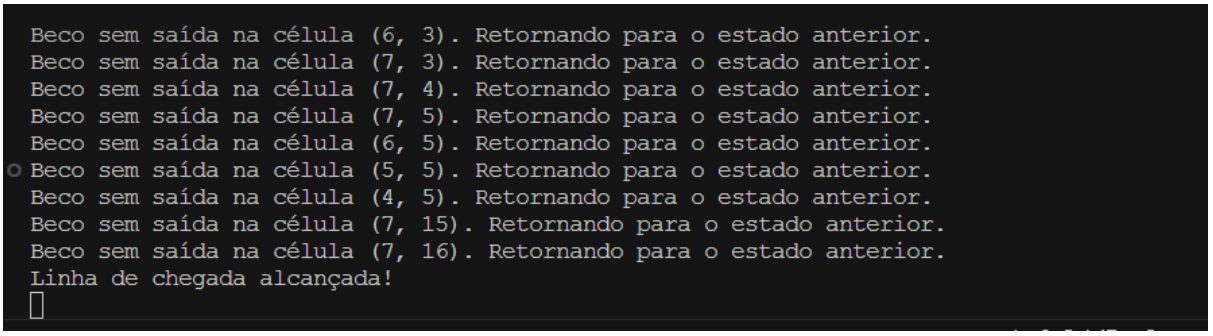
Figura 13 - Amostragem.json



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 14 - Visualização do console.



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.