**UNIVERSIDADE FEDERAL DO MARANHÃO**

**DEPARTAMENTO DE ENGENHARIA DA COMPUTAÇÃO**

| | |
|---|---|
| **Curso:** Engenharia da Computação | **Ano / Semestre:** 2024 / 2 |
| **Disciplina:** Inteligência Artificial | **Professor:** Thales Levi Azevedo Valente |
| **Tema:** Análise e Implementação do Algoritmo BFS para Labirintos em Python | |
| **Aluno:** Lucyene Pinheiro Neves | **Código:** 2020010394 |

**ATIVIDADE AVALIATIVA PARA A DISCIPLINA DE INTELIGÊNCIA ARTIFICIAL**

Análise e Implementação do Algoritmo BFS para Labirintos em Python

**São Luís/MA**

**2024.2**

## 1. INTRODUÇÃO

O projeto em questão apresenta a implementação de um algoritmo de Busca em Largura (BFS) aplicado em um labirinto, utilizando a biblioteca turtle do Python para a representação gráfica. O principal objetivo é demonstrar o funcionamento do BFS para encontrar o caminho mais curto entre o ponto inicial e o ponto final de um labirinto.

## 2. OBJETIVOS

- Implementar e visualizar o algoritmo de Busca em Largura (BFS) em um labirinto.
- Demonstrar a formação da solução e o traçado do caminho de volta.
- Utilizar elementos gráficos para facilitar a compreensão do funcionamento do algoritmo.

.

## 3. REFERENCIAL TEÓRICO

A busca em largura (BFS) é um método usado para explorar todos os pontos de um grafo. Começa por um ponto inicial e visita todos os pontos diretamente ligados a ele, depois passa para os pontos ligados a esses, e assim por diante. Isso é feito usando uma fila para controlar a ordem de visitação. A BFS é útil para encontrar o caminho mais curto em grafos onde todas as conexões têm o mesmo peso (Busca em Largura).

## 4. METODOLOGIA

A abordagem do projeto foi organizada em três etapas principais para facilitar o desenvolvimento e a compreensão do trabalho. Na primeira etapa, foi feita a configuração do ambiente. Isso incluiu a criação de uma matriz para representar o labirinto e a definição dos elementos gráficos que seriam utilizados para visualizá-lo.

Na segunda etapa, foi implementado o algoritmo de busca BFS (Busca em Largura). Ele foi desenvolvido para percorrer o labirinto e encontrar o melhor caminho até a solução. Por fim, na terceira etapa, foi realizada a visualização da solução. Utilizamos a biblioteca Turtle para desenhar o caminho encontrado pelo algoritmo, mostrando tanto o percurso até o objetivo quanto o retorno à origem.

## 5. ESTRUTURA E FUNÇÕES DO CÓDIGO

### Configuração do Ambiente

O projeto foi implementado em Python, utilizando a biblioteca ***turtle*** para a criação e representação do labirinto. A estrutura do labirinto é definida por uma matriz de caracteres. Cada caractere representa uma parte do labirinto:

- **+:** Parede.
- **espaço**: Caminho.
- **s:** Ponto inicial.
- **e**: Ponto final.

**Estruturas e Classes**

O código utiliza quatro classes principais para representar os elementos gráficos do labirinto:

1. **Maze**: Representa as paredes do labirinto. Cada célula é exibida como um quadrado verde.

Imagem 1: Classe para criar labirinto

```
# Classe para criar labirinto
class Maze(turtle.Turtle):
    def __init__(self):
        super().__init__()
        self.shape("square")
        self.color("DarkGreen")
        self.penup()
        self.speed(0)
```

Fonte: Acervo pessoal

2. **Black**: representa o ponto final do labirinto. O método ***stamp*** exibe o ícone "🦀" para representar a IA e marcar visualmente o ponto final no labirinto.

Imagem 2: Classe para quadrado final do labirinto

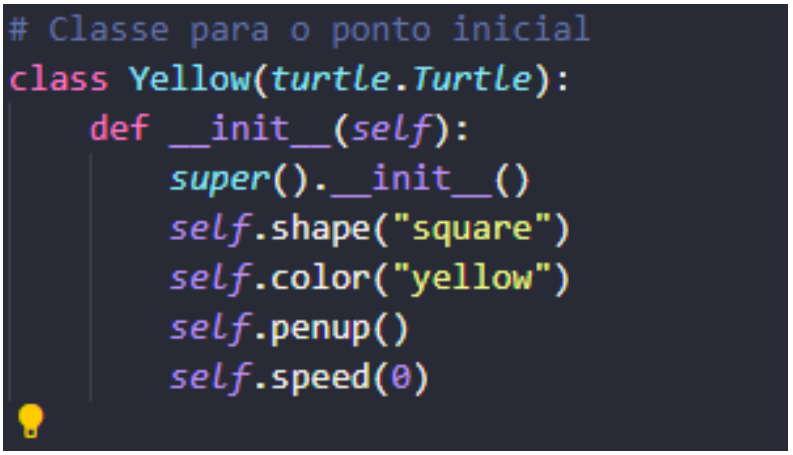
```
# Classe para o quadrado final do labirinto
class Black(turtle.Turtle):
    def __init__(self):
        super().__init__()
        self.hideturtle()
        self.color("DarkOrange")
        self.penup()
        self.speed(0)

    def stamp(self):
        self.goto(self.xcor(), self.ycor() - 24 / 2)
        self.write("🦀", align="center", font=("Arial", int(24 / 1.5), "normal"))
```

Fonte: Acervo Pessoal

3. **Yellow**: Representa o ponto inicial do labirinto com um quadrado amarelo.

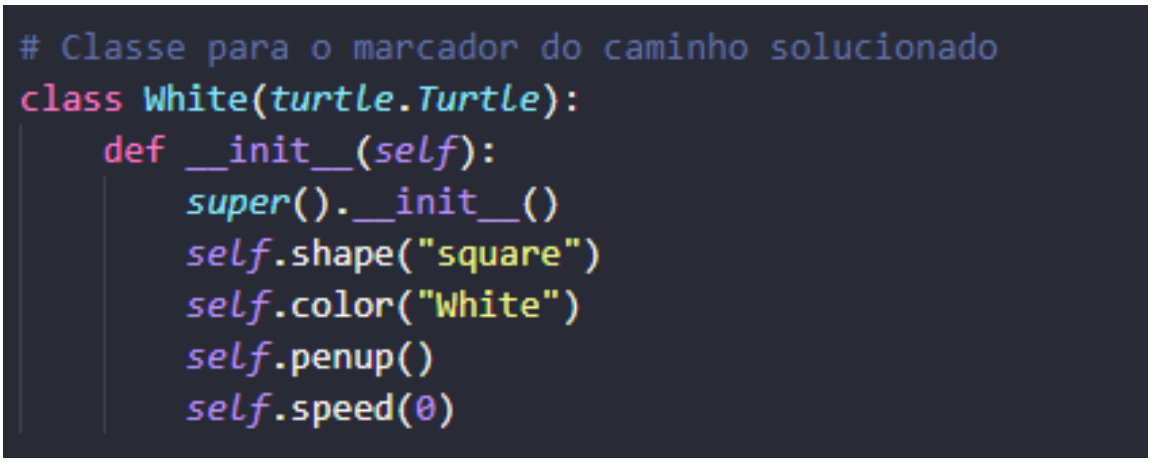
Imagem 3: Classe para o ponto inicial




Fonte: Acervo Pessoal

4. **White**: Marca o caminho da solução com um quadrado branco.

Imagem 4: Classe para o marcador do caminho solucionado




Fonte: Acervo Pessoal

### Configuração do Labirinto

O labirinto é configurado a partir de uma matriz. A imagem abaixo ilustra um trecho de como os elementos do labirinto foram criados:

Imagem 5: Função do labirinto

```
# Função para configurar o labirinto
def setup_maze(grid):
    global start_x, start_y, end_x, end_y  # Variáveis para início e fim do labirinto
    for y in range(len(grid)):
        for x in range(len(grid[y])):
            character = grid[y][x]
            screen_x = -588 + (x * 24)
            screen_y = 288 - (y * 24)

            if character == "+":  # Se o caractere é "+", desenha uma parede
                maze.goto(screen_x, screen_y)
                maze.stamp()
                maze.shapesize(24/20)
                walls.append((screen_x, screen_y))

            if character == " " or character == "e":  # Caminho ou saída
                path.append((screen_x, screen_y))

            if character == "e":  # Marca a posição de saída
                white.goto(screen_x, screen_y)
                white.shapesize(24/20)
                end_x, end_y = screen_x, screen_y
                white.stamp()

            if character == "s":  # Marca a posição de início
                start_x, start_y = screen_x, screen_y
                yellow.goto(screen_x, screen_y)
                yellow.shapesize(24/20)
```

Fonte: Acervo Pessoal

Este trecho realiza a leitura da matriz ***grid***, posiciona cada elemento na tela e armazena informações sobre paredes e caminhos.

## Algoritmo BFS

A busca em largura é implementada na função ***search.*** Segue a explicação detalhada de cada parte do código:

### Inicialização

Imagem 6: Algoritmo de Busca

```
# Algoritmo de busca em largura (BFS)
def search(x, y):
    frontier.append((x, y))  # Adiciona o ponto inicial à fronteira
    solution[(x, y)] = (x, y)  # Marca a célula inicial na solução
```

Fonte: Acervo Pessoal

A fronteira é uma fila que guarda as células a serem exploradas. O ponto inicial é adicionado à fronteira, e ele é registrado como o primeiro nó na solução.

**Exploração**

Imagem 7: Algoritmo de Busca

```
while frontier:  # Continua enquanto houver células na fronteira
    time.sleep(0.01)  # Controla o tempo de execução
    x, y = frontier.pop()  # Remove a célula da fronteira

    #Verifica se encontrou a saída
    if (x,y)==(end_x, end_y):
        print("saída encontrada.")
        return

    # Marca a célula atual como visitada
    visited.add((x, y))

    # Cria uma lista de vizinhos e embaralha para escolha aleatória
    neighbors = [
        (x - 24, y),
        (x + 24, y),
        (x, y - 24),
        (x, y + 24)
    ]
    random.shuffle(neighbors)  # Embaralha os vizinhos

    # Percorre os vizinhos da célula atual
    for neighbor in neighbors:
        if neighbor in path and neighbor not in visited and neighbor not in frontier:
            solution[neighbor] = (x, y)  # Registra o caminho
            frontier.append(neighbor)  # Adiciona o vizinho na fronteira
```

Fonte: Acervo Pessoal

Neste bloco, cada célula é explorada, e seus vizinhos são adicionados à fronteira, desde que sejam válidos e ainda não visitados.

**Condição de Parada**

Imagem 8:Algoritmo de Busca

```
#Verifica se encontrou a saída
if (x,y)==(end_x, end_y):
    print("saída encontrada.")
    return
```

Fonte: Acervo Pessoal

A busca termina quando o ponto final é alcançado. Isso é verificado comparando as coordenadas atuais com as do ponto final.

**Caminho de Retorno**

Imagem 9: Algoritmo de Busca

```
# Traça a rota de volta para o início
def backRoute(x, y):
    white.goto(x, y)
    white.stamp()
    while (x, y) != (start_x, start_y):
        x, y = solution[(x, y)]
        white.goto(x, y)
        white.stamp()
```

Fonte: Acervo Pessoal

Após encontrar a saída, o caminho é reconstruído a partir da relação pai-filho registrada na solução. Este processo destaca o caminho final no labirinto.

## 6. RESULTADOS E ANÁLISES

O algoritmo executa os seguintes passos:

1. Configura o labirinto baseado na matriz definida no código.
2. Encontra a solução através da Busca em Largura.
3. Traça o caminho de volta, destacando a rota no labirinto.

A execução é visualizada em tempo real, com marcadores para o ponto inicial, o ponto final, as células visitadas e o caminho da solução.

## 7. CONCLUSÃO

O projeto foi uma tentativa de implementar o algoritmo de busca BFS em um cenário de labirinto. Ele buscou mostrar como o BFS pode encontrar diferentes caminhos em grafos. A visualização gráfica, criada com a biblioteca Turtle, ajudou a compreender o comportamento do algoritmo em cada uma de suas etapas.

## 8. REFERÊNCIAS


- Documentação da biblioteca Turtle: https://docs.python.org/3/library/turtle.html.
- Algoritmos para grafos - Busca em largura: https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/bfs.html.
- Documentação da biblioteca Filas:https://docs.python.org/pt-br/3/library/asyncio-queue.html.
- Deque in Python: https://www.geeksforgeeks.org/deque-in-python/.
- Maze generator and solver in python: https://www.youtube.com/watch?v=5Kzap4DA_Gw&list=WL&index=2.