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. Observe o Algoritmo 1 abaixo, em que cada célula, na execução do código, será exibida como um quadrado verde.

Algoritmo 1: Classe para criar labirinto

```
1. class Maze(turtle.Turtle):
2.       def  __init__(self):
3.             super()>__init__()
4.             self.shape(“square”)
5.             self.color(“DarkGreen”)
6.             self.penup()
7.             self.speed(0)
```

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

Algoritmo 2: Classe para o quadrado final do labirinto

```
1. class Black(turtle.Turtle):
2.     def __init__(self):
3.         super().__init__()
4.         self.hideturtle()
5.         self.color("DarkOrange")
6.         self.penup()
7.         self.speed(0)
8.
9.     def stamp(self):
10.        self.goto(self.xcor(), self.ycor() - 24 / 2)
11.        self.write("🦀", align="center",
   font=("Arial", int(24 / 1.5), "normal"))
```

3. **Yellow**: O trecho no Algoritmo 3 representa o código do ponto inicial do labirinto com um quadrado amarelo.

Algoritmo 3: Classe para o ponto inicial

```
1. class Yellow(turtle.Turtle):
2.     def __init__(self):
3.         super().__init__()
4.         self.shape("square")
5.         self.color("yellow")
6.         self.penup()
7.         self.speed(0)
```

4. **White**: Marca o caminho da solução com um quadrado branco. Veja no Algoritmo 4 abaixo:

Algoritmo 4: Classe para o marcador do caminho solucionado

```
1. class White(turtle.Turtle):
2.     def __init__(self):
3.         super().__init__()
4.         self.shape("square")
5.         self.color("White")
6.         self.penup()
7.         self.speed(0)
```

Configuração do Labirinto

O labirinto é configurado a partir de uma matriz. O Algoritmo 5 ilustra um trecho de como os elementos do labirinto foram criados:

Algoritmo 5: Função para configurar o labirinto

```
1. def setup_maze(grid):
2.     global start_x, start_y, end_x, end_y
3.     for y in range(len(grid)):
4.         for x in range(len(grid[y])):
5.             character = grid[y][x]
6.             screen_x = -588 + (x * 24)
7.             screen_y = 288 - (y * 24)
8.
9.             if character == "+":
10.                 maze.goto(screen_x, screen_y)
11.                 maze.stamp()
12.                 maze.shapesize(24/20)
13.                 walls.append ((screen_x,
    screen_y))
14.            if character == " " or character ==
    "e":                path.append((screen_x,
    screen_y))
15.            if character == "e":
16.                white.goto(screen_x, screen_y)
```

```
17.                 white.shapesize(24/20)
18.                 end_x, end_y = screen_x,
   screen_y
19.                  white.stamp()
20.             if character == "s":
21.                start_x, start_y = screen_x,
   screen_y
22.                yellow.goto(screen_x, screen_y)
23.                yellow.shapesize(24/20)
```

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 uma breve explicação de cada parte do código:

Inicialização

Algoritmo 6: Busca em largura (BFS) - início

```
1. def search(x, y):
2.     frontier.append((x, y))
3.     solution[(x, y)] = (x, y)
4.
5.     while frontier:
6.          time.sleep(0.01)
7.          x, y = frontier.pop()
8.
9.          if (x,y)==(end_x, end_y):
10.              print("saída encontrada.")
11.              return
```

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

Algoritmo 7: Busca em largura (BFS)- exploração

```
12.       while frontier:
13.            time.sleep(0.01)
14.            x, y = frontier.pop()
15.
16.            if (x,y)==(end_x, end_y):
17.               print("saída encontrada.")
18.               return
```

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

Condição de Parada

Algoritmo 8: Busca em largura (BFS)- Condição de parada

```
19.   if (x,y)==(end_x, end_y):
20.       print("saída encontrada.")
21.       return
```

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

Caminho de Retorno

Algoritmo 9: Busca em largura (BFS)- Caminho de retorno

```
22.   def backRoute(x, y):
23.       white.goto(x, y)
24.       white.stamp()
25.       while (x, y) != (start_x, start_y):
26.           x, y = solution[(x, y)]
27.           white.goto(x, y)
28.           white.stamp()
```

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 começa montando o labirinto a partir de uma matriz já definida no código, mostrando o espaço onde a busca vai acontecer. Depois, usa a técnica da Busca

em Largura (BFS) para achar a solução, e garantir que o menor caminho até o ponto final seja encontrado. Após achar a solução, o algoritmo desenha o caminho de volta, mostrando visualmente a rota percorrida no labirinto. Durante a execução, o processo é mostrado em tempo real, com marcadores claros para o ponto inicial, o ponto final, as células visitadas e o caminho da solução. Essa abordagem busca tornar mais fácil entender como o algoritmo funciona e analisar seu desempenho em encontrar a solução.

7. CONCLUSÃO

O projeto apresentado foi uma tentativa de usar o algoritmo de busca em largura (BFS) em um labirinto, com o objetivo de ver como ele pode resolver problemas de busca em grafos; procurou mostrar como o BFS pode encontrar caminhos em grafos, assegurando a exploração ordenada dos nós conforme a proximidade. Além disso, a visualização gráfica feita com a biblioteca *Turtle* ofereceu uma maneira didática, permitindo observar o comportamento do algoritmo em cada fase, desde a exploração inicial até a descoberta do caminho final. Apesar das limitações deste projeto, ele ajudou a entender os conceitos básicos do BFS e destacou o valor da visualização gráfica como ferramenta para ajudar no aprendizado e na criação de algoritmos

8. REFERÊNCIAS

PYTHON SOFTWARE FOUNDATION. **Turtle**. Disponível em: https://docs.python.org/3/library/turtle.html. Acesso em: dez. 2024.

USP. **Algoritmos para grafos - Busca em largura**. Disponível em: https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/bfs.html. Acesso em: dez. 2024.

PYTHON SOFTWARE FOUNDATION. **Filas**. Disponível em: https://docs.python.org/pt-br/3/library/asyncio-queue.html. Acesso em: dez. 2024.

GEEKSFORGEEKS. **Deque in Python**. Disponível em: https://www.geeksforgeeks.org/deque-in-python/. Acesso em: dez. 2024.

YOUTUBE. **Maze generator and solver in python**. Disponível em: https://www.youtube.com/watch?v=5Kzap4DA_Gw&list=WL&index=2. Acesso em: dez. 2024.

UNIOESTE. **Modelo para formatação de resumos**. Disponível em: https://www.inf.unioeste.br/eca/arquivos/ModeloECA.doc. Acesso em: dez. 2024.