

UNIVERSIDADE FEDERAL DO MARANHÃO

BACHARELADO INTERDISCIPLINAR EM CIÊNCIA E TECNOLOGIA EECP0008 - INTELIGÊNCIA ARTIFICIAL

**GRAFOS**

SÃO LUÍS -MA

DEZEMBRO/2024

EMANUELLE DA SILVA LAUNE

VINICIUS ANDRE ALMEIDA PEREIRA

GRAFOS

RESOLUÇÃO DE LABIRINTO COM PYTHON

Documento apresentado como requisito parcial de avaliação da disciplina Inteligência Artificial - Turma 02, no curso Bacharelado Interdisciplinar em Ciência e Tecnologia da Universidade Federal do Maranhão.

Orientador: Prof. Dr. Thales Levi Azevedo Valente.

SÃO LUÍS -MA

DEZEMBRO/2024

INTRODUÇÃO

Teoria de grafos uma vertente da matemática e ciência da computação destinada ao estudo dos grafos, que são representações abstratas compostas de vértices/nós e arestas, e as relações que existem os objetos pertencentes a um conjunto. A utilização de grafos é validada pela modelagem de redes como dados organizacionais, redes sociais, fluxo de internet, entre outros.

A representação gráfica de grafo é usada por círculos para cada vértice, e as arestas desenhadas com arcos que unem as extremidades. Representar grafos em meio gráfico é uma forma de simplificar o assunto, ao introduzir vários algoritmos para análise e resolução de problemas em grafos.

Um caminho de grafo é a sequência de vértices que possuem a característica de ser a sequência finita ou infinita de vértices unidos por uma sequência de arestas diferentes uns dos outros.

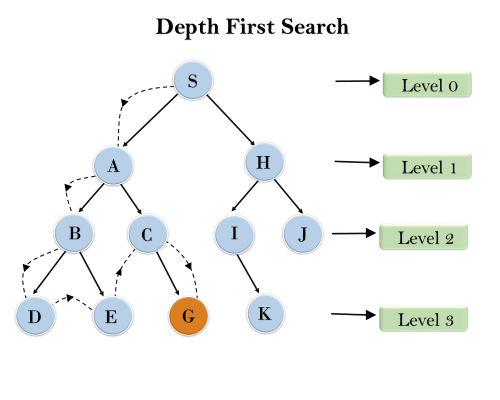
Um ciclo é o fato de o caminho coincidir o primeiro e último vértice, porém nenhum outro vértice se repete; semelhante às cadeias de carbono, são ligações simples e fechadas. É o elemento que fornece complexidade e dificulta a manipulação de grafos, caso o ciclo tenha o comprimento 1, é considerado laço/loop.

A busca em grafo é processo de percorrer arestas de um grafo, passando pelas suas arestas; para fazer esse processo, é possível utilizar algoritmos de busca.

**Depth-First Search (DFS)**

Traduzido como busca em profundidade, é o algoritmo de percorrer uma estrutura de dados toda de um grafo; o objetivo principal é ir o mais fundo possível pelo grafo. O caminho começa no nó raiz, depois vai se expandindo por todos os vizinhos deste nó até chegar no último, como não tem saída possível mais, retorna o caminho já feio e vai ao nó não explorado; esse processo se repete até uma condição definida pelo usuário ser atendida ou que todos os nós existentes sejam percorridos.

Uma desvantagem encontrada no DFS é o fato de não haver garantia de que seja poupado tempo, tendo em vista que é um sistema de visitar todos os nós de um ramo antes de percorrer o outro lado. Ela também faz uso de **pilha**, isto é estrutura de dados linear com uma ordem específica para realizar o caminho pelos nós, dessa forma, ao achar um novo nó, ele será adicionado à pilha para explorar ele e seus vizinhos.

****

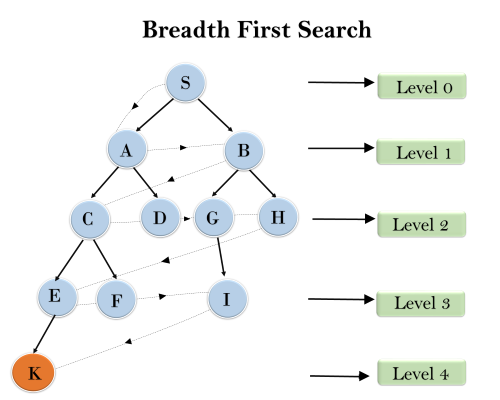
**Figura 1: Representação do percurso realizado pelo algoritmo de busca em profundidade em um grafo.**

**Breadth First Search (BFS)**

Traduzido como busca em largura, é um tipo de algoritmo de busca que utiliza um método diferente do DFS, nesse caso, o grafo é percorrido passando pelos arcos de um vértice a outro, indo de forma sistêmica pelas arestas do grafo. O algoritmo vai calcular a menor distância desde a raiz até os seus vértices acessíveis a partir dele.

Uma característica importante deste algoritmo é sua capacidade de calcular a distância menor entre o vértice inicial, no caso a raiz, e todos os outros vértices a partir dele. Essa característica faz do BFS uma escolha ideal para problemas em que é necessário determinar o caminho mais curto em grafos.

Para implementar o BFS, se usa uma **fila**, uma estrutura de dados que opera no FIFO (First In, First Out). Isso quer dizer que o primeiro elemento inserido na fila será o primeiro a ser processado. A fila organiza os vértices a serem visitados, garantindo que o algoritmo siga uma ordem de exploração nivelada. Quando um vértice é visitado, ele é removido da fila, e todos os seus vizinhos ainda não visitados são adicionados à fila, marcados como visitados para evitar repetições.



**Figura 2: Representação do percurso realizado pelo algoritmo de busca em largura em um grafo.**

**OBJETIVO**

O objetivo do trabalho foi utilizar o conhecimento sobre grafos e algoritmos de estrutura de dados para desenvolver um labirinto em Python, neste trabalho utilizamos o algoritmo A\* (A estrela) cuja principal função é realizar a busca no labirinto que encontra a rota mais curta existente entre dois pontos do grafo.

**DESENVOLVIMENTO**

Para a implementação e resolução da simulação de um labirinto, utilizamos a linguagem de programação Python, e com sua vasta gama de biblioteca empregamos a Pyamaze, pela capacidade de visualização de labirintos e trajetórias de agentes.

O algoritmo escolhido para a implementação foi o A\* (ou A Estrela), cuja principal característica é a otimização na busca do menor caminho em um labirinto. Ele utiliza uma heurística para prever o comportamento, avaliando os custos ao longo do grafo, de modo a encontrar o menor caminho entre os vértices. Essa abordagem permite que o A\* supere outros algoritmos em eficiência, desde que a heurística esteja bem definida.

**IMPLEMENTAÇÃO**

# Função heurística que calcula a estimativa de custo restante até o destino

def h\_score(celula, destino):

linhac = celula[0] # Linha da célula atual

colunac = celula[1] # Coluna da célula atual

linhad = destino[0] # Linha do destino

colunad = destino[1] # Coluna do destino

# Retorna a distância de Manhattan (distância em grade) entre a célula atual e o destino

return abs(colunac - colunad) + abs(linhac - linhad)

Nessa parte do código, pode-se ver que a função h\_score serve para calcular a estimativa de custo que sobrou para chegar no final a partir da célula atual, esse uso de função heurística é ideal para resolução de labirintos em que há apenas os movimentos vertical e horizontal.

Após essa parte, há a implementação do algoritmo em si, com o objetivo de encontrar o caminho de maneira mais otimizada (traço mais marcante do algoritmo A\*).

Variáveis de controle:

**iteracoes**: Contabiliza o número de iterações feitas pelo usuário

**analisadas**: Armazena as células já visitadas

A célula inicial é dada como a última célula da grade, na posição inferior direita, e seu custo inicial g\_score é zero, porque não tem nenhum movimento realizado até o então.O valor inicial de f\_score para essa célula é calculado somando g\_score e o valor da função heurística (h\_score), que prevê a distância ao destino.

Em seguida, a célula inicial é adicionada à fila de prioridade, organiza as células com base no menor custo estimado. Cada item inserido na fila inclui o f\_score, o valor heurístico e a própria célula.

O loop principal do algoritmo funciona enquanto houver células na fila de prioridade. Em cada iteração, a célula com o menor f\_score é tirada da fila e tida como analisada. Se essa célula for o caminho certo, a busca termina, mas caso contrário, o algoritmo explora seus vizinhos válidos (Norte, Sul, Leste e Oeste). Para cada vizinho acessível, calcula-se um novo valor de g\_score adicionando o custo do movimento (1 no caso). Um novo valor de f\_score é então calculado para o vizinho somando g\_score e o valor heurístico.

Se o novo f\_score for menor do que o valor armazenado para o vizinho, os valores de f\_score e g\_score são atualizados, o vizinho é adicionado à fila de prioridade, e a célula atual é registrada como antecedente no caminho. Isso permite reconstruir o caminho mais curto depois.

O loop garante que as células mais promissoras (menor custo estimado) sejam processadas primeiro, permitindo que o algoritmo encontre o caminho mais curto de forma eficiente.

Os valores g e f da célula vizinha são então calculados. Se o novo f for menor que o valor previamente armazenado para essa célula, os custos g e f serão atualizados. A célula vizinha é então adicionada à fila de prioridade como um item, e a célula atual é registrada no caminho como a antecessora dessa vizinha.

Esse processo se repete até que o destino seja alcançado ou não haja mais células a serem exploradas na fila, garantindo a busca pelo caminho mais curto com eficiência.

**celula\_analisada = destino**

**print("Celulas analisadas", len(caminho.keys())) # Exibe o número de células analisadas**

**# Caminha de volta do destino até a célula inicial, construindo o caminho final**

**while celula\_analisada != celula\_inicial:**

**caminho\_final[caminho[celula\_analisada]] = celula\_analisada**

**celula\_analisada = caminho[celula\_analisada]**

**print(f"Número de iterações realizadas: {iteracoes}") # Exibe o número de iterações**

**return caminho\_final, analisadas # Retorna o caminho reconstruído e as células visitadas**

Nessa parte do código, usamos a variável *celula\_analisada* definida como destino, a partir daí, o código vai começar a reconstruir o caminho de volta para a célula inicial, por meio de um loop que continuará enquanto a célula analisada não for a mesma que a célula inicial, e para cada iteração, a célula predecessora é acessada na variável *caminho*, e a célula atual na variável *caminho\_final.*

Já a função exibir\_metricas servem para a exibição das métricas de desempenho do algoritmo A\* após a sua execução e os imprime. Além disso, a função calcular\_densidade tem o propósito de calcular e mostrar a densidade do labirinto criado. Essa densidade é uma medida da proporção entre caminhos livres (sem obstáculos) em relação ao total de conexões possíveis em todas as células do labirinto.

celula\_analisada = destino

print("Celulas analisadas", len(caminho.keys())) # Exibe o número de células analisadas

# Caminha de volta do destino até a célula inicial, construindo o caminho final

while celula\_analisada != celula\_inicial:

caminho\_final[caminho[celula\_analisada]] = celula\_analisada

celula\_analisada = caminho[celula\_analisada]

print(f"Número de iterações realizadas: {iteracoes}") # Exibe o número de iterações

return caminho\_final, analisadas # Retorna o caminho reconstruído e as células visitadas

# Exibição das métricas de desempenho

def exibir\_metricas(tempo\_execucao, eficiencia, custo\_caminho, iteracoes, celulas\_analisadas, total\_celulas):

print("\n--- MÉTRICAS DE DESEMPENHO ---")

print(f"Tempo de execução: {tempo\_execucao:.4f} segundos")

print(f"Eficiência da busca: {eficiencia:.2f}%")

print(f"Custo do caminho encontrado: {custo\_caminho} passos")

print(f"Número de iterações: {iteracoes}")

print(f"Células analisadas (total): {len(celulas\_analisadas)}")

print(f"Total de células no labirinto: {total\_celulas}")

# Função para calcular a densidade do labirinto

def calcular\_densidade(labirinto):

# Conta o total de células no labirinto. Cada célula pode ter até 4 conexões possíveis (Norte, Sul, Leste, Oeste).

total\_celulas = len(labirinto.grid)

caminhos = sum([sum(cell.values()) for cell in labirinto.maze\_map.values()])

densidade = (caminhos / (4 \* total\_celulas)) \* 100 # Percentual de caminhos

print(f"Densidade do Labirinto: {densidade:.2f}%")

return densidade

# Cria um labirinto de tamanho 100x100

labirinto = maze(50, 50)

labirinto.CreateMaze(theme=COLOR.light) # Gera o labirinto com caminhos e barreiras, cor do labirinto branca

Essa parte do código representa a criação do labirinto, gerado com 100x100 células e cor clara, para facilitar a visualização. O agente vermelho representa o agente inicial, enquanto o verde é o objetivo final.

# Cálculo das métricas

tempo\_execucao = fim - inicio # Determina o tempo de execução do código

total\_celulas = len(labirinto.grid) # Número total de células

celulas\_analisadas = len(analisadas) # Células Analisadas

eficiencia = (celulas\_analisadas / total\_celulas) \* 100 # Eficiência da busca

custo\_caminho = len(caminho) # Total de passos para o fim do labirinto

# Exibe as métricas de desempenho

exibir\_metricas(

tempo\_execucao=tempo\_execucao,

eficiencia=eficiencia,

custo\_caminho=custo\_caminho,

iteracoes=len(analisadas),

celulas\_analisadas=analisadas,

total\_celulas=total\_celulas

)

# Calcula a densidade do labirinto

calcular\_densidade(labirinto)

textLabel(labirinto, "Tempo de execução", round(tempo\_execucao, 3)) # Coloca o tempo de execução na tela

# Traça o caminho no labirinto para visualização

labirinto.tracePath({agente1: caminho}, delay=15)

labirinto.tracePath({agente\_destino: []})

# Executa a visualização gráfica do labirinto com o caminho traçado

labirinto.run()

Foi optado também por exibir, ao final do código, os resultados das métricas que indicam o desempenho do algoritmo na resolução do labirinto, para avaliar a eficiência desse método.

**RESULTADO**

A aplicação e implementação do algoritmo a estrela no problema para a resolução do labirinto apresentou uma eficiência notável na busca pelo caminho mais curto (menor custo) entre o ponto inicial e o destino. Utilizando a biblioteca Pyamaze, foi possível criar um labirinto de 50x50 células, com visualização clara dos agentes e dos caminhos percorridos. As principais métricas coletadas durante a execução incluem:

1. Tempo de execução:
   * O algoritmo apresentou tempos de execução extremamente rápidos, inferiores a 0,1 segundo, mesmo para labirintos grandes (25000 células).
2. Eficiência:
   * O algoritmo a estrela analisou aproximadamente 68% das células disponíveis no labirinto para encontrar a solução, devido ao tamanho do labirinto e a quantidade de paredes (obstáculos), indicando que em labirintos menores o algoritmo é mais eficiente no processo.
3. Custo do caminho:
   * O custo do caminho mais curto encontrado foi de 390 passos, o que reflete a eficiência do algoritmo na minimização de custos.
4. Densidade do labirinto:
   * A densidade calculada indicou uma distribuição equilibrada de caminhos livres e barreiras, o que reforça a validade do labirinto para teste de algoritmos de busca.
5. Visualização gráfica:
   * O traçado do caminho, exibido no labirinto com agentes de cores diferentes, possibilitou uma compreensão clara do funcionamento do algoritmo. Identificando o ponto de início e o destino

Os resultados confirmam que o A\* é plenamente eficaz para a resolução de labirintos em que os movimentos são restritos a direções horizontais e verticais, oferecendo uma otimização no tempo de busca e precisão no cálculo do menor caminho.

**--- MÉTRICAS DE DESEMPENHO —**

Células analisadas 2499

Número de iterações realizadas: 2500

Tempo de execução: 1.0643 segundos

Eficiência da busca: 100.00%

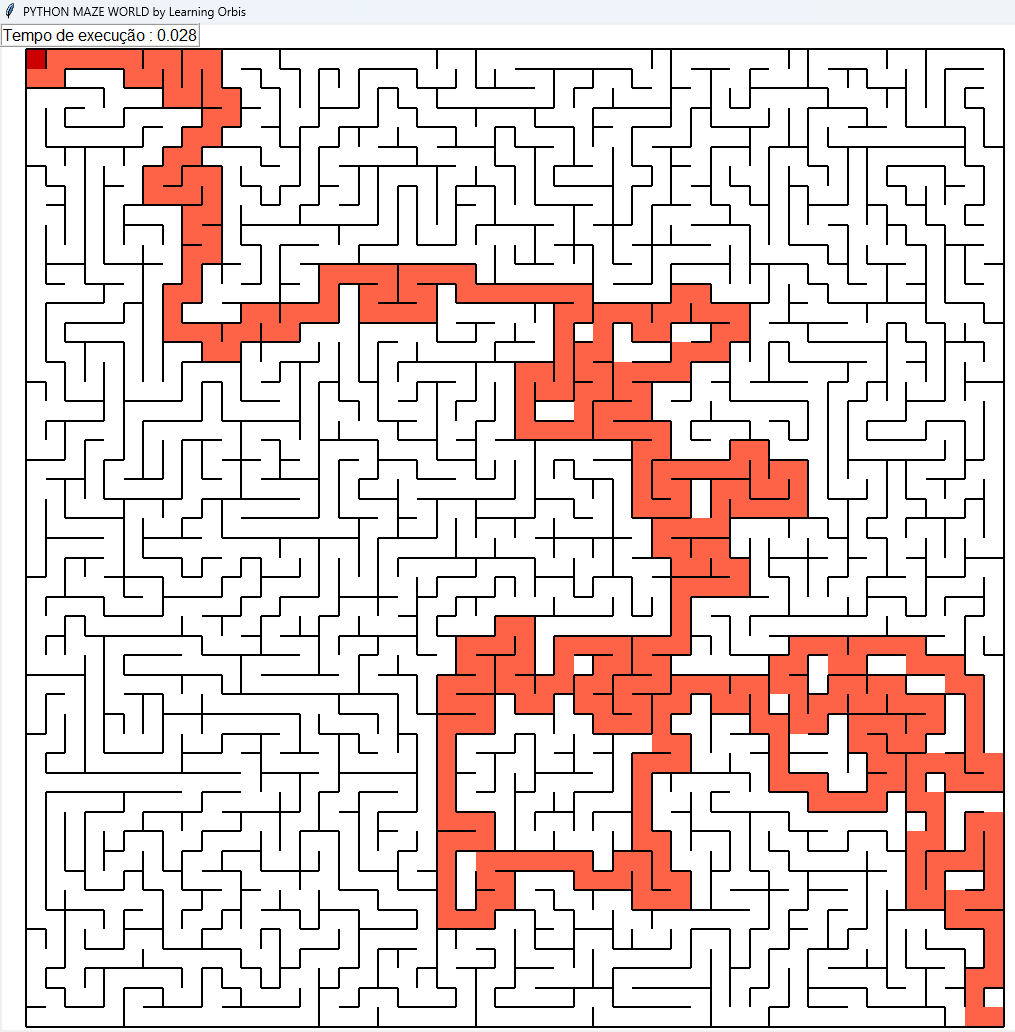
Custo do caminho encontrado: 732 passos

Número de iterações: 2500

Células analisadas (total): 2500

Total de células no labirinto: 2500

Densidade do Labirinto: 49.98%



**Figura 3: Labirinto finalizado após o agente encontrar a saída**

**CONCLUSÃO**

O trabalho conseguiu atingir o objetivo proposto de implementar o algoritmo a estrela para resolver um labirinto modelado como um grafo. A utilização da biblioteca Pyamaze permitiu a construção do ambiente e a visualização dos resultados, enquanto o a estrela demonstrou ser uma solução eficiente para a resolução do problema de busca do menor caminho (menor custo).

O uso de uma heurística baseada na distância de Manhattan mostrou-se adequado para o contexto do labirinto, fazendo com que o algoritmo priorizasse os caminhos mais promissores (menores custos) e minimizasse o número de células analisadas (maiores custos). Apesar disso, observou-se que a eficiência pode ser impactada pela densidade do labirinto, o que sugere que mudanças na geração de barreiras podem impactar os resultados.

Este estudo reforça a aplicabilidade de algoritmos de grafos na resolução de problemas reais e destaca o A\* como uma ferramenta robusta em contextos onde a otimização de caminhos é essencial.

**REFERÊNCIAS**

BIN, Miao. *Graph Traversal in Python: A* algorithm\*. Disponível em:<https://medium.com/nerd-for-tech/graph-traversal-in-python-a-algorithm-27c30d67e0d0>. Acesso em: 17 dez. 2024.

HERMUCHE, Anwar. *Métodos de Busca em Grafos — BFS & DFS*. Disponível em:<https://medium.com/@anwarhermuche/m%C3%A9todos-de-busca-em-grafos-bfs-dfs-cf17761a0dd9>. Acesso em: 18 dez. 2024.

MAFRA JUNIOR, Cristiano. *Algoritmo A* (A Estrela)\*. Disponível em:<https://www.tabnews.com.br/CristianoMafraJunior/algoritmo-a-a-estrela>. Acesso em: 16 dez. 2024.

*Programa Python para Breadth First Search ou BFS para um gráfico*. Disponível em:<https://www.geeksforgeeks.org/python-program-for-breadth-first-search-or-bfs-for-a-graph/>. Acesso em: 18 dez. 2024.