Centro Federal de Educação Tecnológica de Minas Gerais
Professor: Tiago Alves — Disciplina: Inteligência Artificial
Aluno: Luan Gonçalves Santos — Matrícula: 20213004695
Aluno: Pedro Henrique Pires Dias — Matrícula: 20203011622

Divinópolis, 29 de novembro de 2024.

Trabalho 01

Comparação de Algoritmos de Busca Não Informada no Problema do Labirinto

1 Introdução

O presente relatório apresenta uma análise comparativa entre dois algoritmos de busca não informada aplicados ao problema clássico do labirinto. Os algoritmos considerados são a **Busca em Largura (Breadth-First Search - BFS)** e a **Busca em Profundidade (Depth-First Search - DFS)**. O objetivo principal é avaliar o desempenho de cada método em termos de tempo de execução, consumo de memória, completude e optimalidade.

1.1 Definição do Problema

O estudo aborda o clássico problema de busca em labirinto, ilustrado na Figura 1. O objetivo principal consiste em determinar um caminho entre duas posições, navegando por um espaço representado por um conjunto de células. Para abstrair o problema, cada célula do labirinto é modelada como um vértice de um grafo, enquanto os caminhos possíveis entre as células são representados por arestas que conectam esses vértices.

O problema em questão representa uma aplicação clássica dos algoritmos de busca em grafos, onde o desafio consiste em explorar o grafo de forma eficiente para encontrar uma solução, considerando as restrições de tempo de execução e uso de memória. Nesse contexto, a aplicação dos algoritmos de **Busca em Largura (BFS)** e **Busca em Profundidade (DFS)** tem como objetivo não apenas identificar um caminho, mas também avaliar suas características, como completude e optimalidade.

Figura 1: Labirinto '*grafo.txt*' para testes.

A Figura 1 ilustra o labirinto utilizado nos testes. A célula identificada como *Start* representa o ponto de início da busca, enquanto a célula denominada *Goal* indica o objetivo a ser alcançado. O labirinto é estruturado como uma grade de células, das quais algumas são intransponíveis, funcionando como obstáculos no caminho.

2 Descrição dos Algoritmos Implementados

Nesta seção, são apresentados os dois algoritmos de busca não informada implementados para resolver o problema do labirinto: a Busca em Largura (Breadth-First Search - BFS) e a Busca em Profundidade (Depth-First Search - DFS). Ambos os métodos utilizam o grafo gerado a partir do labirinto descrito na Figura 1.

2.1 Busca em Largura (Breadth-First Search - BFS):

A Busca em Largura é um algoritmo que explora o grafo de forma sistemática, visitando todos os vértices de um mesmo nível antes de avançar para o próximo nível. A implementação utiliza uma estrutura de fila (queue) para armazenar os vértices a serem explorados. A BFS garante a completude, ou seja, encontra uma solução caso ela exista, e é optimal quando todos os custos das arestas são iguais. Seus passos principais são descritos a seguir:

1. Inicializa o vértice inicial como visitado e o adiciona à fila.
2. Enquanto a fila não estiver vazia:
   (a) Remove o vértice da frente da fila.
   (b) Se o vértice atual for o objetivo, reconstrói o caminho percorrido.
   (c) Caso contrário, adiciona todos os vértices adjacentes ainda não visitados à fila.
3. Retorna o caminho encontrado ou indica que não existe solução.

Abaixo, é possível visualizar melhor através do pseudocódigo do algoritmo BFS:

```
função BUSCA-EM-LARGURA(problema) retorna uma solução ou
falha
  nó ← um nó com ESTADO = problema.ESTADO-INICIAL, CUSTO-
DE-CAMINHO = 0
  se problema.TESTE-DE-OBJETIVO(nó.ESTADO) senão retorne
SOLUÇÃO(nó),
  borda ← uma fila FIFO com nó como elemento único
  explorado ← conjunto vazio
  repita
    se VAZIO?(borda), então retorne falha
    nó ← POP(borda) / * escolhe o nó mais raso na borda */
    adicione nó.ESTADO para explorado
    para cada ação em problema.AÇÕES(nó.ESTADO) faça
      filho ← NÓ-FILHO(problema, nó, ação),
      se (filho.ESTADO)não está em explorado ou borda então
        se problema.TESTE-DE-OBJETIVO(filho.ESTADO) então
retorne SOLUÇÃO(filho)
        borda ← INSIRA(filho, borda)
```

Figura 2: Busca em Largura em um grafo [6]

Além disso, a Figura 3 ilustra a aplicação do método BFS em um grafo simples.

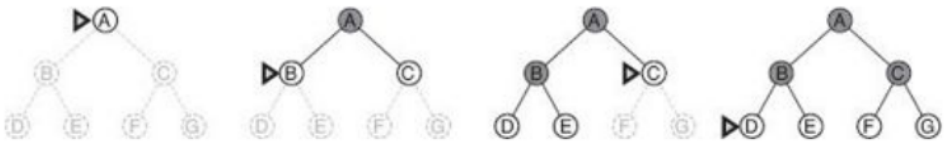

Figura 3: Busca em largura em uma árvore binária simples. Em cada fase, próximo nó a ser expandido é indicado por um marcador.

A BFS é particularmente útil em problemas como o do labirinto, pois explora todas as possibilidades de forma uniforme e muitas vezes pode encontrar o caminho mais curto em termos de número de arestas [1].

2.2 Busca em Profundidade (Depth-First Search - DFS):

A Busca em Profundidade explora o grafo ao máximo, visitando cada vértice até atingir um caminho sem saída ou o objetivo. A implementação utiliza uma estrutura de pilha (stack), seja explícita ou implícita através da recursão. Diferentemente da BFS, a DFS não garante a optimalidade do caminho encontrado. O algoritmo segue os seguintes passos principais:

1. Inicializa o vértice inicial como visitado e o adiciona à pilha.
2. Enquanto a pilha não estiver vazia:
   (a) Remove o vértice do topo da pilha.
   (b) Se o vértice atual for o objetivo, reconstrói o caminho percorrido.
   (c) Caso contrário, adiciona todos os vértices adjacentes ainda não visitados à pilha.
3. Retorna o caminho encontrado ou indica que não existe solução.

Abaixo, é possível visualizar melhor através do pseudocódigo do algoritmo DFS:

```
função BUSCA-EM-PROFUNDIDADE-LIMITADA(problema,
 limite) retorna uma solução ou falha/corte
retornar BPL-RECURSIVA (CRIAR-NÓ(problema, ESTADO-
 INICIAL), problema, limite)
função BPL-RECURSIVA(nó, problema, limite) retorna uma
 solução ou falha/corte
   se problema. TESTAR-OBJETIVO (nó.ESTADO) então,
 retorna SOLUÇÃO (nó)
   se não se limite = 0 então retorna corte
   senão
      corte_ocorreu? ← falso para cada
 ação no problema.AÇÕES(nó.ESTADO) faça
         filho ← NÓ-FILHO (problema, nó, ação)
         resultado ← BPL-RECURSIVA (criança, problema limite − 1)
         se resultado = corte  então corte_ocorreu? ← verdadeiro
         senão se resultado ≠ falha então retorna resultado
      se corte_ocorreu? então retorna corte senão retorna falha
```

Figura 4: Implementação recursiva da busca em árvore de profundidade limitada [6].

Além disso, a Figura 5 ilustra a aplicação do método DFS em um grafo simples.

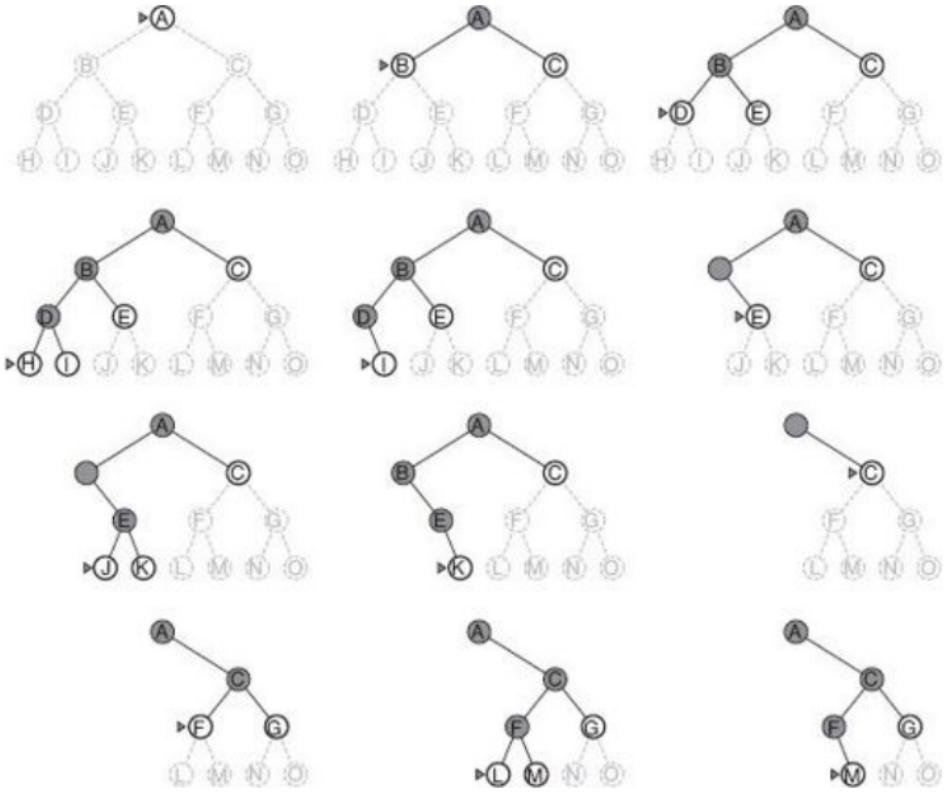

Figura 5: Busca em profundidade em uma árvore binária. A região inexplorada é mostrada em cinza-claro. Os nós explorados sem descendentes na borda são removidos da memória. Os nós na profundidade 3 não têm sucessores e M é o único nó objetivo.

A DFS é eficiente em termos de memória quando comparada à BFS, mas pode ser menos completa em casos onde o grafo contém ciclos, exigindo técnicas adicionais para evitar repetições [1].

3 Metodologia

Após a compreensão do problema e a identificação de suas particularidades, o primeiro passo para a resolução consistiu em definir como o programa desenvolvido interpreta o grafo que representa o labirinto.

3.1 Interpretação do Grafo pelo Algoritmo

Para interpretar o problema do labirinto como um grafo, o algoritmo utiliza um arquivo externo denominado `grafo.txt`, que define os vértices e arestas que compõem a estrutura do grafo. Esse arquivo serve como entrada para o construtor da classe responsável pela manipulação do grafo.

Na primeira linha do arquivo `grafo.txt`, estão listados todos os vértices do grafo. Durante a inicialização do grafo, o construtor lê essa linha e interpreta cada caractere como um vértice único. Em seguida, cada vértice identificado é adicionado à estrutura interna do grafo por meio do método `adicionarVertice`.

Essa etapa assegura que todos os vértices sejam reconhecidos antes de estabelecer suas conexões, formando a base para a construção das arestas subsequentes.

Nesse sentido, as linhas seguintes do arquivo definem as conexões entre os vértices, ou seja, as arestas do grafo. Cada linha contém dois caracteres separados por espaço, representando, respectivamente, um vértice de origem e um de destino. O construtor interpreta essas linhas como pares de vértices conectados por uma aresta. E assim essas conexões são então registradas na estrutura interna do grafo por meio do método `adicionarAresta`. Por exemplo, uma linha como `A B` indica que existe uma aresta entre os vértices A e B.

Internamente, representa-se o grafo como um mapa de adjacências, onde cada vértice é associado a uma lista de seus vértices adjacentes. Essa escolha estrutural é eficiente tanto para a construção quanto para a navegação do grafo, permitindo que os algoritmos de busca, como BFS e DFS, explorem rapidamente os vizinhos de qualquer vértice durante sua execução.

Dessa forma, torna se facil abstrair o contexto do problema do labirinto, em que cada vértice corresponde a uma posição distinta no labirinto, enquanto as arestas representam os caminhos possíveis entre essas posições. Portanto, a modelagem estabelecida transforma o problema do labirinto em um problema de busca em grafos. Ou seja, a partir dessa abordagem, o algoritmo interpreta o arquivo de entrada e constrói um grafo navegável, permitindo a aplicação dos algoritmos de busca para resolver o problema proposto.

3.2 Critérios de Avaliação

Para avaliar os algoritmos de busca implementados (BFS e DFS), foram utilizados quatro critérios principais:

- **Tempo de Execução:** Responsável por medir o intervalo de tempo necessário para que o algoritmo encontre a solução do problema. Calculou-se o tempo utilizando a biblioteca `<chrono>` [5], definindo os instantes de início e término da execução de cada um dos algoritmos. As funções `medirDesempenhoBFS` e `medirDesempenhoDFS` são as responsáveis por registrar esse resultado, retornando o tempo em milissegundos.

- **Consumo de Memória:** Responsável por medir o consumo de memória, que representa a maior quantidade de elementos simultâneos na fila para o BFS ou na pilha para o DFS. Utilizou-se o `max_tamanho_fila` para monitorar o tamanho máximo da fila no BFS. Para o DFS, utilizou-se a variável `max_tamanho_pilha`.

Ambas as variáveis são atualizadas durante a execução, afim de garantir que o valor final reflita o valor máximo de memória utilizado.

- **Completude:** Responsável por avaliar se o programa é capaz de encontrar a solução sempre que ela existir. A completude foi testada em dois cenários: um caso em que a solução existe (verificação entre os vértices U e E). Outro caso em que a solução não existe (entre U e Z). Os testes foram realizados utilizando a função `testarCompletudeBFS` e `testarCompletudeDFS`, retornando true ou false dependendo do resultado.

- **Optimalidade:** Responsável por verificar se a solução encontrada foi a melhor possível, à exemplo do caminho mais curto do grafo. Para essa verificação foi comparada o tamanho dos caminhos encontrados por ambos os algoritmos. Para o BFS, espera-se que encontre sempre o menor caminho (ótimo). Para o DFS, pode ocorrer de não ser ótimo, dependendo da estrutura do grafo.

3.3 Ferramentas Utilizadas

É importante ressaltar que, para construir o algoritmo apresentado no repositório, foi necessário utilizar algumas tecnologias, tais como C++ Orientado a Objetos, Visual Studio Code e WSL Ubuntu.

- A linguagem de programação C++ foi escolhida como a principal ferramenta para o desenvolvimento do sistema. Sendo orientada a objetos (POO), o C++ é amplamente utilizado tanto para comunicação entre máquinas quanto para o desenvolvimento de softwares acadêmicos e corporativos. A escolha baseou-se no fato de ser considerada uma das linguagens de programação mais rápidas do mundo e por oferecer robustas funcionalidades de POO. [2].

- Optou-se pelo Visual Studio Code, um editor de código-fonte desenvolvido pela Microsoft para Windows, Linux e macOS, amplamente utilizado no desenvolvimento de aplicações. [3]

- O WSL é um método para se executar o Sistema Operaciona Linux dentro do Windows de forma muito facil. Essa platoforma se torna essencial para o desenvovimento em ambiente GNU Linux, evitando tarefas como a intalação de máquinas vituais e criação de dual boot no computador pessoal. [4].

O programa ainda possui um arquivo Makefile que realiza todo o procedimento de compilação e execução. Para tanto, temos as seguintes diretrizes de execução:

Tabela 1: Comandos úteis para compilar e executar o programa de computador

Comando	**Função**
make clean	Apaga a última compilação realizada contida na pasta build
make	Executa a compilação do programa utilizando o gcc, e o resultado vai para a pasta build
make run	Executa o programa da pasta build após a realização da compilação

Nas últimas seções, apresentadam-se as conclusões e contribuições obtidas ao longo deste relatório, além de uma breve menção a ideias que podem ser exploradas em trabalhos futuros, visando aprimorar o sistema e ampliar seus resultados.

4 Resultados das Medições de Desempenho

O código desenvolvido para a realização deste trabalho está disponível publicamente no repositório GitHub. Podendo ser acessados por meio do seguinte link, clique [AQUI] (https://github.com/LuanLuL/IA_Trabalho_01).

Para avaliar o desempenho dos algoritmos BFS e DFS, foram realizados 10 testes utilizando o grafo especificado no arquivo '*grafo.txt*', disponível no repositório deste projeto. Os resultados obtidos estão apresentados nas Tabelas 2 e 3, permitindo uma análise comparativa dos algoritmos em termos de tempo de execução, consumo de memória, número de arestas exploradas, optimalidade e completude.

Tabela 2: Resultado das 10 execuções do BFS para o grafo.txt

Teste	**Tempo de Execução (ms)**	**Consumo de Memória**	**Arestas**	**Optimalidade**	**Completude**
1	0.02214	4 elementos	10	Ótimo	Sim
2	0.022279	4 elementos	10	Ótimo	Sim
3	0.022069	4 elementos	10	Ótimo	Sim
4	0.02193	4 elementos	10	Ótimo	Sim
5	0.049028	4 elementos	10	Ótimo	Sim
6	0.02186	4 elementos	10	Ótimo	Sim
7	0.022279	4 elementos	10	Ótimo	Sim
8	0.022628	4 elementos	10	Ótimo	Sim
9	0.022977	4 elementos	10	Ótimo	Sim
10	0.022908	4 elementos	10	Ótimo	Sim

Tabela 3: Resultado das 10 execuções do DFS para o grafo.txt

Teste	Tempo de Execução (ms)	Consumo de Memória	Arestas	Optimalidade	Completude
1	0.011803	4 elementos	17	Não Ótimo	Sim
2	0.012711	4 elementos	17	Não Ótimo	Sim
3	0.011943	4 elementos	17	Não Ótimo	Sim
4	0.012222	4 elementos	17	Não Ótimo	Sim
5	0.031987	4 elementos	17	Não Ótimo	Sim
6	0.012432	4 elementos	17	Não Ótimo	Sim
7	0.011454	4 elementos	17	Não Ótimo	Sim
8	0.012781	4 elementos	17	Não Ótimo	Sim
9	0.012362	4 elementos	17	Não Ótimo	Sim
10	0.012431	4 elementos	17	Não Ótimo	Sim

Os resultados das execuções do algoritmo DFS (Depth-First Search) e BFS (Breadth-First Search) para o grafo.txt revelam diferenças significativas em aspectos como tempo de execução, número de arestas exploradas e optimalidade. O DFS apresentou tempos de execução ligeiramente mais baixos na maioria dos testes, variando entre 0.011454 ms e 0.031987 ms, e explorou 17 arestas em todas as execuções. No entanto, foi classificado como "Não Ótimo"em termos de encontrar o caminho mais curto, o que é esperado devido à sua estratégia de busca profunda, que não garante optimalidade.

Em contraste, o BFS obteve tempos de execução um pouco maiores, variando de 0.02186 ms a 0.049028 ms, explorando consistentemente 10 arestas. Apesar do maior consumo de tempo, o BFS demonstrou optimalidade em todas as execuções, reforçando sua adequação para problemas que requerem caminhos mínimos em grafos não ponderados.

Ambos os algoritmos exibiram completude, confirmando que sempre encontram uma solução se esta existir. Esses resultados destacam a diferença entre a eficiência e a eficácia dos algoritmos, sendo o DFS mais rápido, mas não ideal para caminhos mínimos, enquanto o BFS é mais lento, mas garante a melhor solução.

4.1 Análise Comparativa entre Dois Grafos

Com o objetivo de obter resultados mais robustos, um novo grafo foi gerado a partir de um labirinto redesenhado, possibilitando a realização de comparações de desempenho entre diferentes cenários.

Figura 6: Labirinto '*grafo1.txt*' para testes

A média aritmética simples foi calculada a partir de 10 execuções distintas, e os resultados estão apresentados nas Tabelas 4 e 5. Essas tabelas destacam as diferenças observadas entre os algoritmos BFS e DFS ao serem aplicados em dois cenários distintos: um labirinto redesenhado (grafo1.txt) e o labirinto original (grafo.txt).

Tabela 4: Resultado das médias entre BFS e DFS (grafo1.txt - Labirinto redesenhado)

Critério	**BFS**	**DFS**
Tempo de Execução (média)	0.0302621 ms	0.0084507 ms
Consumo de Memória	10 elementos	10 elementos
Completude	Sim	Sim
Optimalidade	Sim (Ótimo)	Sim (Ótimo)

Tabela 5: Resultado das médias entre BFS e DFS (grafo.txt) - Labirinto original

Critério	**BFS**	**DFS**
Tempo de Execução (média)	0.0250098 ms	0.0142126 ms
Consumo de Memória	4 elementos	4 elementos
Completude	Sim	Sim
Optimalidade	Sim (Ótimo)	Não (Não Ótimo)

Com base nos resultados apresentados, a análise comparativa entre BFS e DFS é detalhada a seguir:

- **Qual foi o mais rápido em diferentes tamanhos de labirintos:** No labirinto redesenhado, o DFS foi mais rápido (0.0084507 ms) que o BFS (0.0302621 ms). Já no labirinto original, o DFS também foi mais rápido (0.0142126 ms) que o BFS (0.025098 ms). Portanto, em ambos os tamanhos de labirinto, o DFS foi o algoritmo mais rápido.

- **Qual consumiu menos memória:** No labirinto redesenhado, ambos os algoritmos consumiram 10 elementos de memória. Da mesma forma, no labirinto original, ambos consumiram 4 elementos de memória. Assim, não houve diferença no consumo de memória entre os algoritmos nos dois cenários.

- **Ambos são completos e ótimos:** Os algoritmos são completos nos dois casos, como indicado na coluna "Completude"(Sim). Quanto à optimalidade, no labirinto redesenhado, tanto BFS quanto DFS são ótimos. Porém, no labirinto original, apenas o BFS é ótimo, enquanto o DFS não é ótimo.

5 Conclusão e Trabalhos Futuros

Os testes realizados com os algoritmos BFS e DFS demonstraram comportamentos distintos dependendo do tipo de grafo e dos critérios de análise. O DFS se mostrou mais eficiente em termos de tempo de execução, mas não garantiu a optimalidade, frequentemente explorando mais arestas do que o BFS. Em contrapartida, o BFS garantiu sempre a solução ótima, embora com maior tempo de execução, o que o torna mais adequado para problemas onde a optimalidade do caminho é crucial.

Conclui-se entao que esse trabalho contribuiu para a compreensão das diferenças entre os dois algoritmos, ajudando a estabelecer qual deles é mais apropriado para cenários específicos, seja para um grafo simples ou para um labirinto mais complexo. Nesse sentido, em trabalhos futuros, é possível explorar otimizações do algoritmo DFS para garantir a optimalidade ou investigar outras variantes de algoritmos de busca, como A* ou Busca Gulosa, que podem oferecer resultados diferentes dependendo do contexto.

Referências

[1] CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. **Introduction to Algorithms**. 3. ed. Cambridge: MIT Press, 2009. 1312 p.

[2] CPLUSPLUS. C++ Language v3.3.4s. 2000-2024. Disponível em: https://cplusplus.com/doc/tutorial/. Acesso em: 27 nov. 2024.

[3] Microsoft. Download for free of Visual Studio Code (Version 1.84). [S.l.: s.n., s.d.]. Disponível em: https://code.visualstudio.com/. Acesso em: 27 nov. 2024.

[4] GRACIELLY, J. WSL 2 - A solução para rodar Linux dentro do Windows 10 - Root #08 [Vídeo]. YouTube, 2021. Disponível em:https://www.youtube.com/watch?v=hd6lxt5iVsg. Acesso em: 18 nov. 2023.

[5] CPLUSPLUS. <chrono> Time library. Disponível em: https://cplusplus.com/reference/chrono. Acesso em: 27 nov. 2024.

[6] RUSSELL, Stuart; NORVIG, Peter. **Inteligência Artificial**. Tradução da 3ª edição. Tradução de Ronaldo M. F. Lopes. Rio de Janeiro: Elsevier, 2013.