



UNIOESTE

Universidade Estadual
do Oeste do Paraná

Projeto 1 - Algoritmos de Busca

Professora:

Huei Diana Lee

Alunos:

Victor Hugo Almeida Alicino,
Victor Emanuel Almeida

December 5, 2022

Conteúdo

1	Introdução	2
2	Executando o programa	2
2.1	Requisitos	2
2.2	Compilando	3
2.3	Executando	3
2.4	Saída	4
3	Implementação	4
3.1	Estruturas de dados	5
3.2	Algoritmos de busca	5
3.2.1	Algoritmo A* (melhor solução)	6
3.2.2	Busca em profundidade (extra)	6
3.2.3	Busca em largura (pior solução)	6
4	Testando o sistema	6
4.1	Descrição do teste	6
4.2	Arquivo de entrada	7
4.3	Resultados dos testes	8
5	Conclusão	9

Lista de Figuras

1	Exemplo de arquivo de saída do programa	4
2	Grafo utilizado para os testes	7
3	Tempos de execução A*	8
4	Tempos de execução BFS	8
5	Tempos de execução DFS	8
6	Média dos tempos de execução dos algoritmos	9

1 Introdução

O Projeto tem como objetivo a implementação de um sistema para auxiliar turistas em Veneza a chegarem aos diversos museus da cidade, localizados em diferentes ilhas.

Implementado inteiramente na linguagem de programação C++ na revisão 2020 (C++20) sem utilização de bibliotecas externas além das padrões da linguagem, podendo assim ser compilado em qualquer sistema operacional que possua um compilador baseado no padrão ISO/IEC 14882:2020 ^[1].

O C++20 foi escolhido pois além de possuir abstrações alto nível com o uso de classes, ainda é extremamente eficiente ao ser compilado diretamente para linguagem de máquina. Além do disso a dupla responsável pelo projeto possui familiaridade com a linguagem.

O sistema é capaz de calcular a menor rota entre dois pontos, utilizando os algoritmos A*, Busca em Largura (*Breadth-first Search*) e Busca em Profundidade (*Depth-first Search*).

Visando a simplicidade e levar os algoritmos a seus limites de eficiência, o sistema não implementa uma interface gráfica, todos os parâmetros devem ser passados como argumentos de linha de comando, ou coletados em tempo de execução. (Para ver os argumentos para o programa veja a seção Executando).

Uma vez que todos os parâmetros para execução do programa podem ser passados antes que ele inicie, pode-se fazer uso de scripts para automatizar a execução do programa gerando resultados para diferentes cenários. Dessa maneira foi implementado um script para testar o desempenho, explicado na seção Testando o sistema.

2 Executando o programa

2.1 Requisitos

- cmake;
- make;
- g++;
- git (opcional).

2.2 Compilando

Antes de realizar a compilação é necessário definir o nível de mensagens que serão exibidas durante a execução do programa.

Para isso basta definir a expressão `DEFAULT_LOG_LEVEL`, no arquivo `logger.hpp`, para um dos valores definidos no enum `LogLevel`:

Listing 1: Enum `LogLevel`

```
1  enum LogLevel { DEBUG, INFO, WARNING, ERROR, FATAL };
```

Caso o valor seja definido como:

- **DEBUG**: Todas as mensagens serão exibidas, tais como objeto construído, objeto destruído, tempos de execução de funções intermediárias, etc.
- **INFO**: Exibe mensagens “Modo iterativo” dos algoritmos, como a ilha que está sendo visitada, entre outras.
- **WARNING**: Exibe mensagens de aviso como quando um arquivo não é encontrado.
- **ERROR**: Exibe mensagens de erro na qual ainda é possível continuar a execução do programa.
- **FATAL**: Exibe apenas mensagens de erro que impossibilitam a execução do programa, nível que menos exibe mensagens.

Uma vez definido qual o nível de mensagens que serão exibidas, basta compilar o programa com os comandos:

- Entrar na pasta build: `cd build`
- Configurar o projeto: `cmake .`
- Compilar o projeto: `make`

2.3 Executando

Para executar o programa basta executar o arquivo

`Graph_Search_Algorithms_1.0.0` gerado na pasta build.

Caso nada seja passado como argumento, o programa irá coletar os argumentos em tempo de execução, sendo eles:

1. O caminho para o arquivo de entrada;

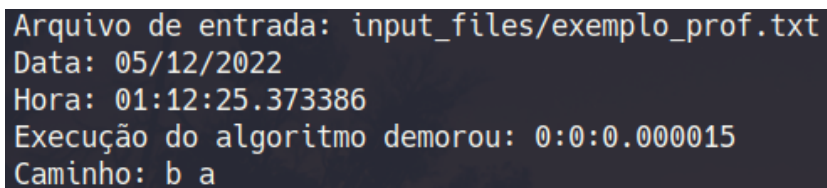
2. O caminho para o arquivo de saída contendo o resumo da execução;
3. O algoritmo a ser utilizado, podendo ser:
 - $A \rightarrow A^*$;
 - $B \rightarrow BFS, Breadth First Search$ (busca em largura);
 - $D \rightarrow DFS, Depth First Search$ (busca em profundidade).

Exemplos de execução:

- **Sem argumentos:** `./Graph_Search_Algorithms_1.0.0`
- **Com argumentos:** `./Graph_Search_Algorithms_1.0.0 input_files/exemplo_prof.txt output.txt A`

2.4 Saída

Caso a execução seja bem sucedida, o programa irá gerar um arquivo de saída contendo o resumo da execução, que deverá ser semelhante ao exemplo abaixo:

A imagem mostra um terminal com o seguinte texto: Arquivo de entrada: input_files/exemplo_prof.txt, Data: 05/12/2022, Hora: 01:12:25.373386, Execução do algoritmo demorou: 0:0:0.000015, Caminho: b a.

```
Arquivo de entrada: input_files/exemplo_prof.txt
Data: 05/12/2022
Hora: 01:12:25.373386
Execução do algoritmo demorou: 0:0:0.000015
Caminho: b a
```

Figura 1: Exemplo de arquivo de saída do programa

3 Implementação

Essa seção visa explicar a implementação do programa, especificando seus algoritmos e estruturas de dados.

Para mais detalhes da implementação, tais como todas as classes e seus métodos, pode-se consultar a documentação em <https://graph-search-algorithms.pages.dev/>

3.1 Estruturas de dados

Segue abaixo a descrição das estruturas de dados utilizadas no programa.

Listing 2: Estrutura de dados para representar um nó do grafo.

```

1  enum NodeState { HAS_NONE = 0, HAS_WEIGHT, HAS_HEURISTIC,
    HAS_BOTH };
2
3  class AdjacencyNode {
4  private:
5      std::string _id;
6      int16_t _weight;
7      int16_t _heuristic;
8      NodeState _state;
9  };

```

Listing 3: Estrutura de dados para representar o grafo.

```

1  class Graph {
2  private:
3      std::unordered_map<std::string, std::vector<AdjacencyNode>>
        _nodes;
4      std::string _start_node;
5      std::string _end_node;
6      Algorithms _algorithm;
7  };

```

Para representação do grafo foi utilizado uma lista de adjacência, porém ao invés de utilizar um vetor, como normalmente é utilizado, optou-se por usar um *unordered_map*, tabela hash, para armazenar os nós do grafo. Visto que no arquivo de entrada possui nomes de ilhas e não números para identificar os nós, dessa forma para utilizar um vetor seria necessário um mapeamento entre os nomes e os números, o que poderia dificultar a implementação.

A estrutura grafo também armazena o identificador do nó inicial e final, bem como o algoritmo a ser utilizado. Sendo assim o grafo é o objeto que contém todos os dados necessários para a execução do programa.

3.2 Algoritmos de busca

A seguir é apresentado os algoritmos de busca implementados no programa.

3.2.1 Algoritmo A* (melhor solução)

Algoritmo de busca ótimo que utiliza uma heurística para encontrar a melhor solução.

Escolhe o próximo nó a ser visitado com base no custo total do caminho^[2], $g(n)$, até o nó somado com o custo estimado do nó até o nó final, $h(n)$, sendo esse custo estimado a heurística do nó. Sendo representado matematicamente por^[3]:

$$f(n) = g(n) + h(n) \quad (1)$$

A complexidade de tempo deste algoritmo é $O(b^d)$, sendo b a ramificação do grafo e d a profundidade da solução, porém caso a possua uma boa heurística a resposta será encontrada em menos iterações. Por esse motivo foi escolhido como melhor solução.

3.2.2 Busca em profundidade (extra)

Algoritmo de busca cega não ótimo e completo, devido ao fato de existir um número finito de ilhas (espaço de busca ser finito).

Sempre escolhe o nó mais profundo na árvore de busca para visitar.

A complexidade de tempo deste algoritmo é $O(b^m)$, sendo b a ramificação do grafo e m a profundidade da solução, e a complexidade de espaço é $O(bm)$.

3.2.3 Busca em largura (pior solução)

Algoritmo de busca cega completo e ótimo, com implementação simples.

Escolhe o próximo nó a ser visitado com base na profundidade do nó, sendo o nó mais próximo do nó inicial escolhido primeiro.

A complexidade de tempo deste algoritmo é $O(b^d)$, sendo b e d explicados na seção Algoritmo A* (melhor solução) a cima.

Pelo fato deste algoritmo gastar mais memória que o algoritmo de busca em profundidade, $O(b^d)$, foi escolhido como pior solução.

4 Testando o sistema

Com o sistema implementado, foi realizado testes utilizando o script `build/run_tests.sh`

4.1 Descrição do teste

- Para cada um dos algoritmos de busca implementados, foi executado X vezes passando os parâmetros `arquivo saidai algoritmosj`, sendo

$i \in \{1, \dots, 100\}$ e $j \in \{A^*, \text{DFS}, \text{BFS}\}$.

- Após cada execução espera-se um tempo de 1s para que o sistema possa ser executado novamente.
- Os testes foram realizados em um computador com as seguintes características:
 - **Processador:** i3-1115G4 4.100GHz;
 - **Memória RAM:** 8GB;
 - **SSD:** 256GB;
 - **Sistema operacional:** Arch Linux ⁱ.
- A execução dos testes foi realizada logo após a inicialização do computador, sem nenhum outro processo de usuário em execução.
- Os arquivos de saída gerados foram salvos em `build/tests_output`.

4.2 Arquivo de entrada

Para a realização dos testes foi utilizado o arquivo de entrada disponibilizado pela professora na especificação do trabalho, esse arquivo está em `build/input_files/exemplo_prof.txt`.

Que gera um grafo semelhante ao da figura 2.

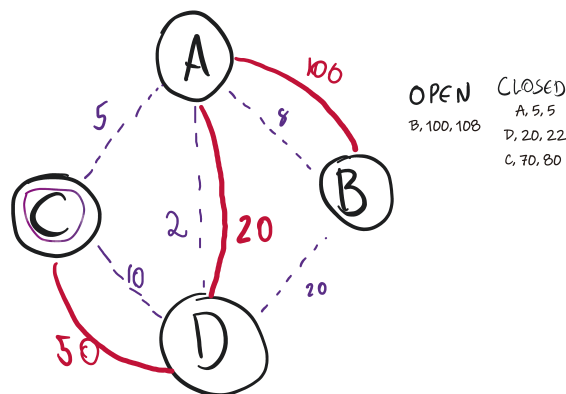


Figura 2: Grafo utilizado para os testes

ⁱLink para download do sistema operacional <<https://archlinux.org/download/>>

4.3 Resultados dos testes

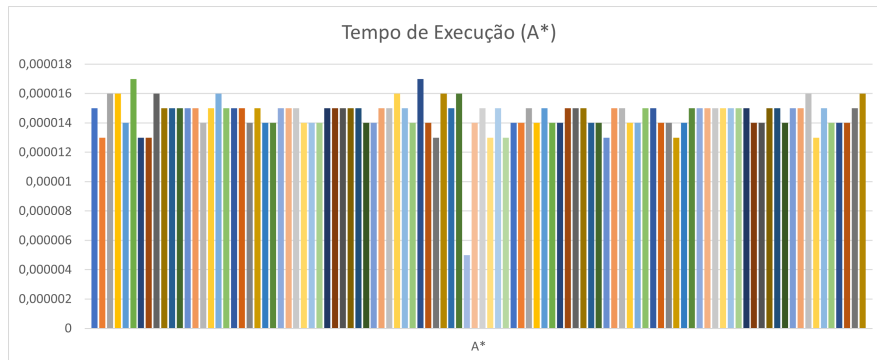


Figura 3: Tempos de execução A*

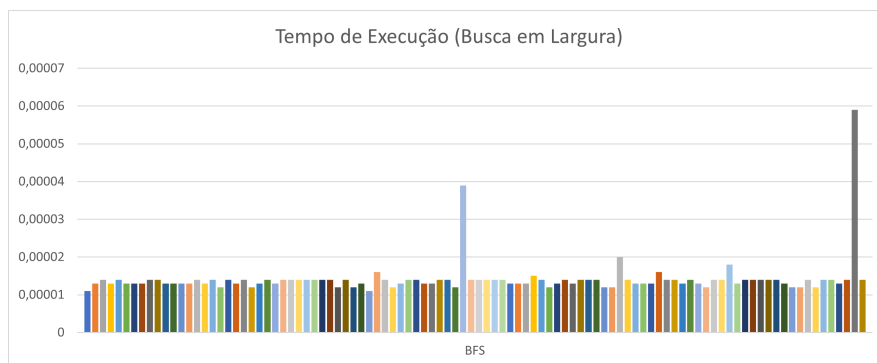


Figura 4: Tempos de execução BFS

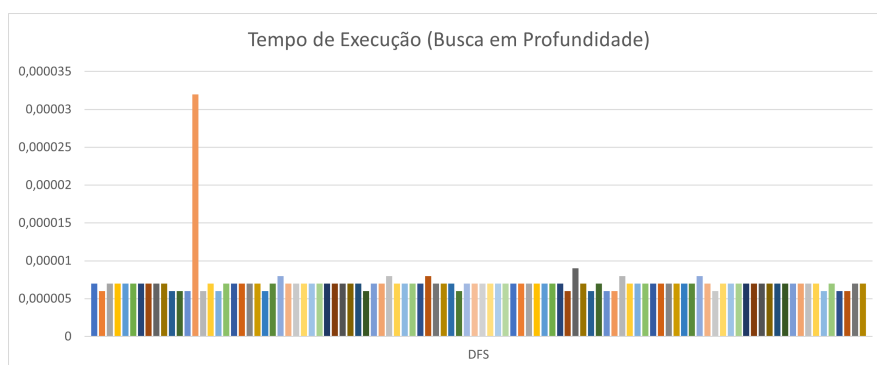


Figura 5: Tempos de execução DFS

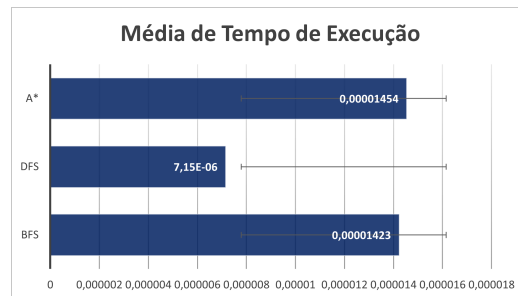


Figura 6: Média dos tempos de execução dos algoritmos

5 Conclusão

Nesse trabalho foi implementado um sistema para encontrar a melhor rota entre duas ilhas, utilizando os algoritmos de busca A^* , BFS e DFS .

Observando os resultados o algoritmo BFS foi o melhor em termos de tempo de execução, devido ao fato do grafo possuir um caminho muito pequeno entre as ilhas, o que faz com que o algoritmo simplesmente busque em apenas duas ilhas.

Caso o grafo fosse maior, o algoritmo A^* seria o melhor, visto que suas iterações são mais custosas computacionalmente, porém tende a realizar um número menor de iterações.

Referências

- 1 ISO/IEC 14882:2020 Programming languages — C++. 6. ed. [S.l.], 2020. Citado na página 2.
- 2 POUND, M. **A^* Search Algorithm - Computerphile**. 2017. Disponível em: <<https://www.youtube.com/watch?v=ySN5Wnu88nE>>. Acesso em: 05 de dezembro de 2022. Citado na página 6.
- 3 SINGHAL, A. **A^* Algorithm AI Gate Vidyalyay**. Gate Vidyalyay. Disponível em: <<https://www.gatevidyalay.com/a-algorithm-a-algorithm-example-in-ai/>>. Acesso em: 05 de dezembro de 2022. Citado na página 6.