Trabalho Prático 2 - Estrutura de Dados

Lucas Ribeiro da Silva - 2022055564
Universidade Federal de Minas Gerais
Belo Horizonte - Minas Gerais - Brasil
lucasrsilvak@ufmg.br

1 Introdução

O problema proposto foi analisar dado um grafo, seus vértices, arestas e pesos, se uma quantidade de energia seria suficiente para percorrer todo o grafo. Esse problema foi simbolizado pelo herói lendário Linque e a floresta da neblina, onde os portais indicam arestas de peso nulo e os caminhos indicam arestas de peso proporcional à distância euclidiana. Esse problema nos permitirá observar os Grafos como Estrutura de Dados, bem como avaliar suas diferentes implementações em diferentes contextos e observar alguns de seus algoritmos importantes e suas complexidades e nuances.

2 Método

2.1 Configurações de Máquina e Ambiente

O programa foi desenvolvido em C++ e compilado com G++ da GNU Compiler Collection. O ambiente de desenvolvimento utilizado foi:

- Windows 11 Home Single Language
- WSL 2 com Ubuntu 22.04.4 LTS

2.2 Estrutura de Dados

Na implementação foram utilizadas as seguintes estruturas de dados:

2.2.1 Ponto

Uma classe Ponto foi criada para encapsular as informações das coordenadas dos vértices e facilitar o cálculo de distâncias euclidianas.

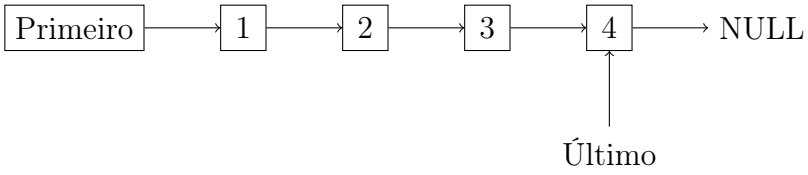
2.2.2 Tripla

Uma classe Tripla foi criada para armazenar as informações de adjacências para representar informações sobre as arestas utilizando uma 3-tupla, como indicado no código abaixo.

```
1     Tripla(int Vertice, bool Portal, float Distancia);
```

2.2.3 Lista Encadeada

A Estrutura de Dados Lista Encadeada, implementada com o paradigma simplesmente encadeado foi escolhida para ser utilizada como Lista de Adjacência, a lista foi implementada com um ponteiro para o primeiro e para o último nó para reduzir a complexidade de inserção no final para tempo constante. Foram implementados apenas os métodos necessários para o funcionamento da Lista no contexto do problema. A classe Nó também foi criada para ser o elemento da lista, cada nó possui uma Tripla e um ponteiro para o próximo nó.



2.2.4 Matriz

A Estrutura de Dados padrão matriz[][] foi utilizada para implementar eficientemente uma Matriz de Adjacência, nesse caso, cada elemento da matriz é uma Tripla contendo as informações sobre a adjacência.

2.2.5 Heap

A Estrutura de Dados Heap foi implementada para satisfazer a necessidade de uma Priority Queue nos algoritmos Dijkstra e A* implementados. A memória utilizada pela Heap foi escolhida como um array[] onde capacidade máxima foi definida como o número de vértices máximo do Grafo, podendo ser Redimensionada em caso de capacidade limítrofe.

2.2.6 Grafo

A Estrutura de Dados Grafo foi utilizada para satisfazer a necessidade de armazenar os vértices e conectá-los com arestas. Nesse trabalho, foram utilizados um Grafo com Lista de Adjacência e um Grafo com Matriz de Adjacência.

2.3 Algoritmos de Caminho

2.3.1 Dijkstra

O Algoritmo de Dijkstra **float CalculaDistancia(int numPortal)** foi implementado no trabalho para encontrar o menor caminho entre o vértice inicial (posição de Linque) e o vértice final (saída da floresta). No caso da implementação utilizada, uma informação adicional sobre o número de portais a serem utilizados é passado como parâmetro para o funcionamento do algoritmo. O algoritmo utiliza uma lista de prioridades e realiza um loop para percorrer os vértices adjacentes do vértice inicial, recursivamente, calculando a distância acumulada e guardando sempre a menor distância, se o número de portais utilizados for menor que o pré-definido.
O algoritmo retorna a distância, que será utilizada para verificar se Linque conseguiu fugir ou não da floresta com a energia armazenada.

2.3.2 A*

O Algoritmo de A* **float AcharCaminho(int numPortal)** foi implementado no trabalho para encontrar algum caminho qualquer entre o vértice inicial (posição de Linque) e o vértice final (saída da floresta). O funcionamento do algoritmo é predominantemente igual ao Dijkstra, com a adição de uma heurística que procura adivinhar o melhor caminho até o vértice final utilizando um 'chute' baseado na distância euclidiana.
O algoritmo retorna a distância, que será utilizada para verificar se Linque conseguiu fugir ou não da floresta com a energia armazenada.

3 Análise de Complexidade

3.1 Tempo

3.1.1 Lista

Lista() O construtor da Lista tem todas as operações em tempo constante e opera em O(1).
~Lista() O destrutor da Lista usa um loop para iterar todos os elementos da lista, operando em O(n).
InsereFinal() A Inserção na última posição da lista encadeada opera em O(1) devido a um ponteiro para o último elemento da lista.
GetTamanho() O método que obtém o tamanho da lista opera em O(1).
GetPrimeiro() O método que retorna o primeiro elemento da lista opera em O(1) devido a um ponteiro para o primeiro elemento da lista.

3.1.2 Heap

Heap() O construtor da Heap tem todas as operações em tempo constante e opera em O(1).
∼Heap() O destrutor da Heap simplesmente desaloca a memória e por isso opera em O(1).
HeapifyPorCima() HeapifyPorCima tem seu pior caso quando o nó é movido da folha até a raiz, nesse caso, o algoritmo opera em O(log n).
HeapifyPorBaixo() HeapifyPorBaixo tem seu pior caso quando o nó é movido da raiz até a folha, nesse caso, o algoritmo opera em O(log n).
Inserir() Inserção na Heap insere um elemento no vetor da Heap e depois chama o método HeapifyPorCima() que opera em O(log(n)), sendo essa a complexidade também do Inserir(). Entretanto, caso a capacidade esteja no limite, a Heap usará o método Redimensionar() com ordem de complexidade de O(n), sendo este também o pior caso do Inserir().
Remover() Inserção na Heap remove um elemento no vetor da Heap e depois chama o método HeapifyPorBaixo() para reorganizar o vetor, mas ele opera em O(log(n)), sendo essa a complexidade também do método Remover().
Redimensionar() O método Redimensionar() é chamado caso a capacidade da Heap seja estourada, como um loop é necessário para copiar os elementos do vetor para um novo, a ordem de complexidade é O(n).
Vazio() A Verificação se a Heap está Vazia opera em tempo constante O(1).

3.1.3 Grafo

Grafo() O construtor do Grafo tem todas suas operações em tempo constante e opera em O(1).
∼Grafo() O destrutor apenas desaloca a memória alocado pelo Grafo e por isso opera em O(1).
SetAdjacente() Inserção de uma aresta de caminho no Grafo opera em O(1).
SetPortal() Inserção de uma aresta de portal no Grafo opera em O(1).
SetPonto() Inserção de um Ponto no Grafo opera em O(1).

3.1.4 Grafo de Lista de Adjacência

CalculaDistancia() O algoritmo de Dijkstra usa um loop para preencher o vetor alocado Distancia com a constante INFINITO, o que opera em O(V). Em seguida, no loop principal, temos a operação de inserção na Heap, que opera em O(log(V)). No pior caso, quando todos os vértices e arestas são visitados, a complexidade final é:

$$O(V) + O((V + E)\log(V)) = O((V + E)\log(V))$$

AcharCaminho() O algoritmo A* usa um loop para preencher o vetor alocado Distancia com a constante INFINITO, o que opera em O(V). No loop principal, temos a operação de inserção na Heap, que opera em O(log(V)). No pior caso, quando todos os vértices e arestas são visitados, a complexidade final é:

$$O(V) + O((V + E)\log(V)) = O((V + E)\log(V))$$

3.1.5 Grafo de Matriz de Adjacência

CalculaDistancia() O algoritmo de Dijkstra usa um loop para preencher o vetor alocado Distancia com a constante INFINITO, o que opera em O(V). No loop principal, temos a operação de inserção na Heap, que opera em O(log(V)). Como precisamos percorrer toda a matriz de adjacência e as arestas visitadas, a complexidade final é:

$$O(V) + O((V^2)\log(V)) = O((V^2)\log(V))$$

AcharCaminho() O algoritmo A* usa um loop para preencher o vetor alocado Distancia com a constante INFINITO, o que opera em O(V). No loop principal, temos a operação de inserção na Heap, que opera em O(log(V)). Como precisamos percorrer toda a matriz de adjacência e as arestas visitadas, a complexidade final é:

$$O(V) + O((V^2)\log(V)) = O((V^2)\log(V))$$

3.2 Espaço

3.2.1 Lista

Lista() O construtor da Lista aloca um somente nó e opera em O(1).
∼Lista() O destrutor da Lista não aloca memória e opera em O(1).
InsereFinal() A inserção de um elemento na última posição da Lista opera em O(1).
GetTamanho() Obter o tamanho da lista não aloca memória e opera em O(1).
GetPrimeiro() Obter o primeiro elemento da lista não aloca memória e opera em O(1).

3.2.2 Heap

Heap() O construtor da Heap aloca n espaços de memória e por isso opera em O(n).
∼Heap() O destrutor da Heap não aloca memória adicional e opera em O(1).
Inserir() A Inserção na Heap não aloca memória adicional se não houver redimensionamento e opera em O(1), caso haja redimensionamento, opera em O(n).
HeapifyPorCima() O HeapifyPorCima() não aloca memória adicional e opera em O(1).
HeapifyPorBaixo() O HeapifyPorBaixo() não aloca memória adicional e opera em O(1).
Remover() A Remoção na Heap não aloca memória adicional e opera em O(1).
Redimensionar() O redimensionamento da Heap aloca um vetor adicional com o dobro do tamanho do anterior e opera em O(n).
Vazio() A Verificação se a Heap está vazia não aloca memória e opera em O(1).

3.2.3 Grafo Geral

∼Grafo() O destrutor do Grafo apenas desaloca a memória alocado e opera em O(1).
SetAdjacente() Inserção de uma aresta de caminho no Grafo não aloca memória adicional e opera em O(1).
SetPortal() Inserção de uma aresta de portal no Grafo não aloca memória adicional e opera em O(1).
SetPonto() Inserção de um Ponto no Grafo não aloca memória adicional e opera em O(1).

3.2.4 Grafo de Lista de Adjacência

Grafo() O construtor do Grafo aloca um vetor de Pontos e um vetor de Listas de Adjacências e por isso opera em O(V).

CalculaDistancia() O algoritmo de Dijkstra aloca a memória utilizada no vetor de Distancia, que necessita de armazenar a distância para cada vértice e logo opera em O(V).

AcharCaminho() O algoritmo de A* aloca a memória utilizada no vetor de Distancia, que necessita de armazenar a distância para cada vértice e logo opera em O(V).

3.2.5 Grafo de Matriz de Adjacência

Grafo() O construtor do Grafo aloca um vetor de Pontos e um vetor de Matrizes de Adjacência e por isso opera em O(V^2).

CalculaDistancia() O algoritmo de Dijkstra aloca a memória utilizada no vetor de Distancia, que necessita de armazenar a distância para cada vértice e logo opera em O(V).

AcharCaminho() O algoritmo de A* aloca a memória utilizada no vetor de Distancia, que necessita de armazenar a distância para cada vértice e logo opera em O(V).

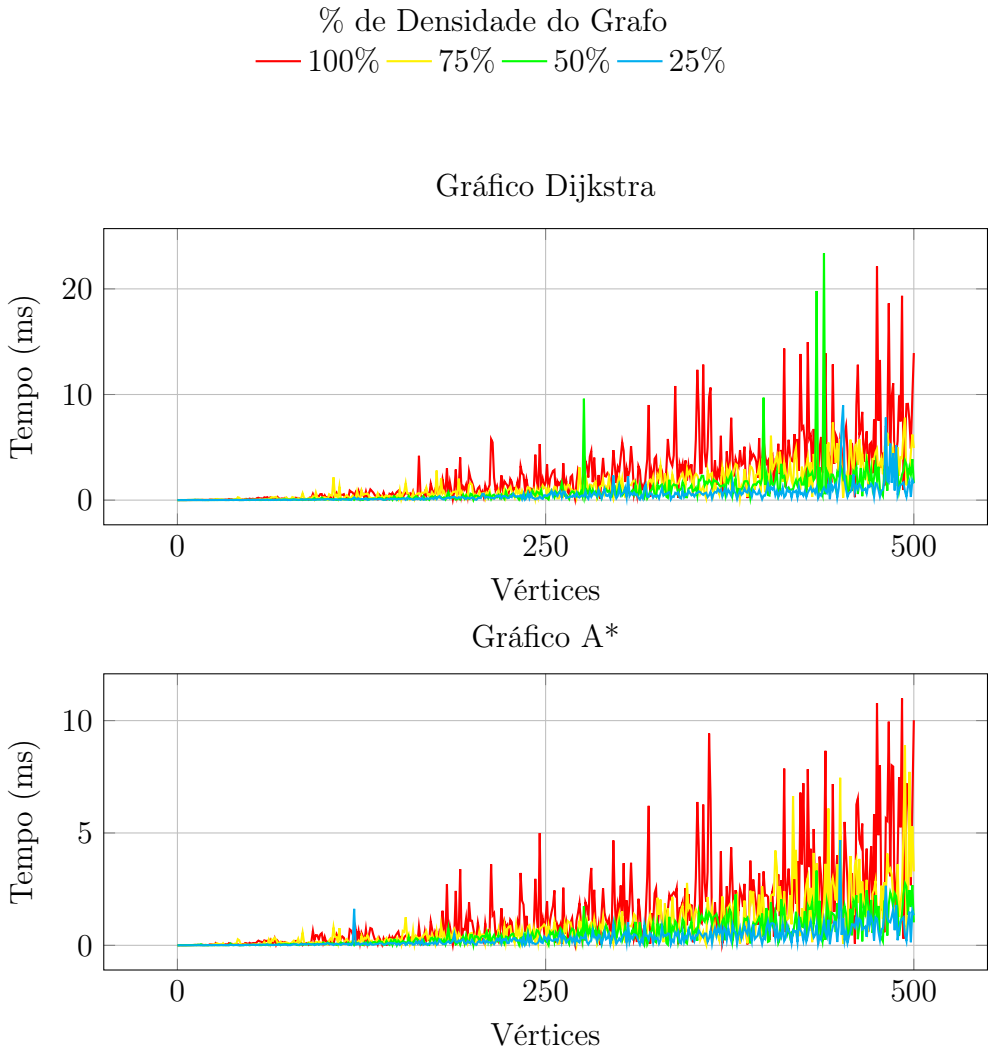
4 Análise de Robustez

Para melhorar a legibilidade, métodos foram padronizados em PascalCase e variáveis foram nomeadas em português. O código segue o paradigma de Orientação a Objetos e as estruturas de dados foram implementadas com o mínimo de funções necessárias para o funcionamento, seguindo o princípio "Keep it Simple, Stupid". O Valgrind foi utilizado para verificar vazamentos de memória. Além disso, foi utilizado um gerador de casos de testes para verificar a corretude do algoritmo em seus casos de borda e casos extremos. Nos casos em que o programa necessita de input também foi devidamente tratado com *try-catch* para evitar irregularidades, uma verificação simples na inserção de arestas e portais foi utilizada.

5 Análise Experimental

5.1 Complexidade Experimental

Para esse experimento, o gerador foi configurado para gerar grafos com número de vértices variando de 0 a 500, e os algoritmos foram testados computacionalmente com os grafos assumindo diferentes níveis de densidades de arestas no grafo de Lista de Adjacência.

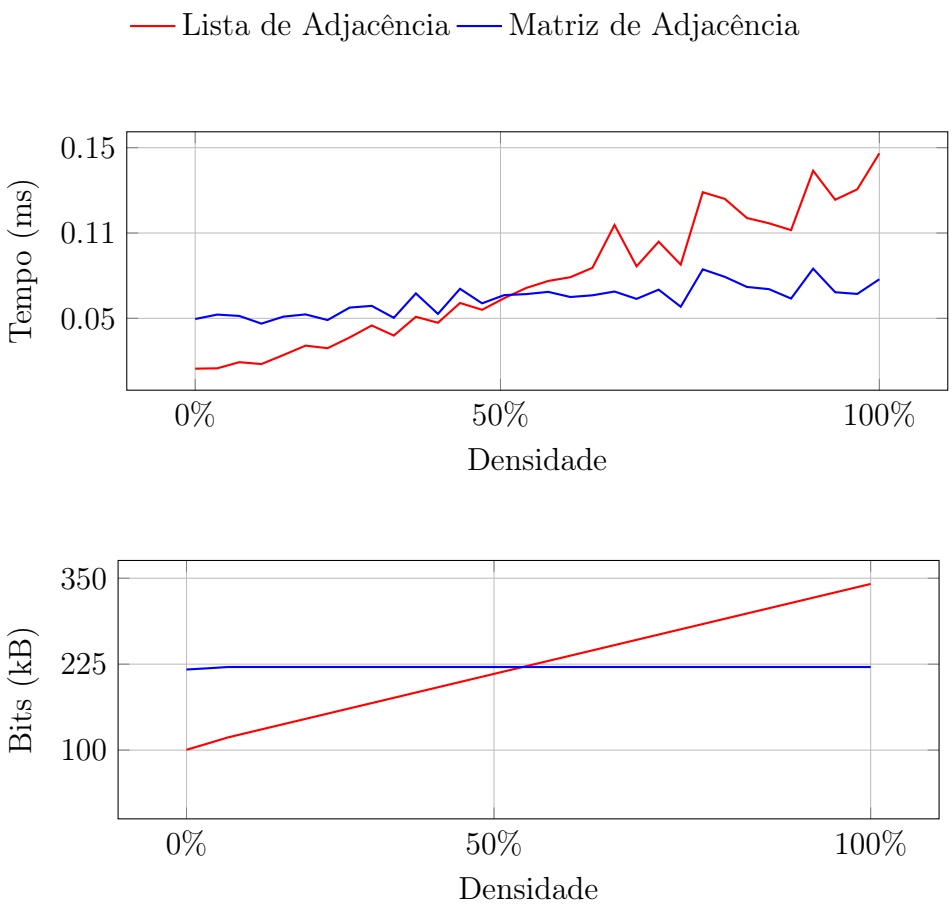


Podemos observar, pelos gráficos que a complexidade dos algoritmos é proporcional ao número de vértices e arestas, podendo ser diagnosticada a degradação rápida dos algoritmos conforme o número de arestas aumenta e comprovando o caso quadrático téorico O((V+E)logV)) degradando para O(V^2logV), enquanto os algoritmos crescem infimamente em relação ao número de vértices, comprovando o funcionamento O((V+E)logV) dos algoritmos para casos onde o número de arestas é relativamente menor.

Fica perceptível também que apesar da complexidade assintótica dos algoritmos serem a mesma, o A* é computacionalmente mais rápido que o Dijkstra, isso acontece porque o Dijkstra tem de verificar todos os caminhos de um vértice até o outro, enquanto o A* procura um caminho qualquer.

5.2 Densidade da Matriz

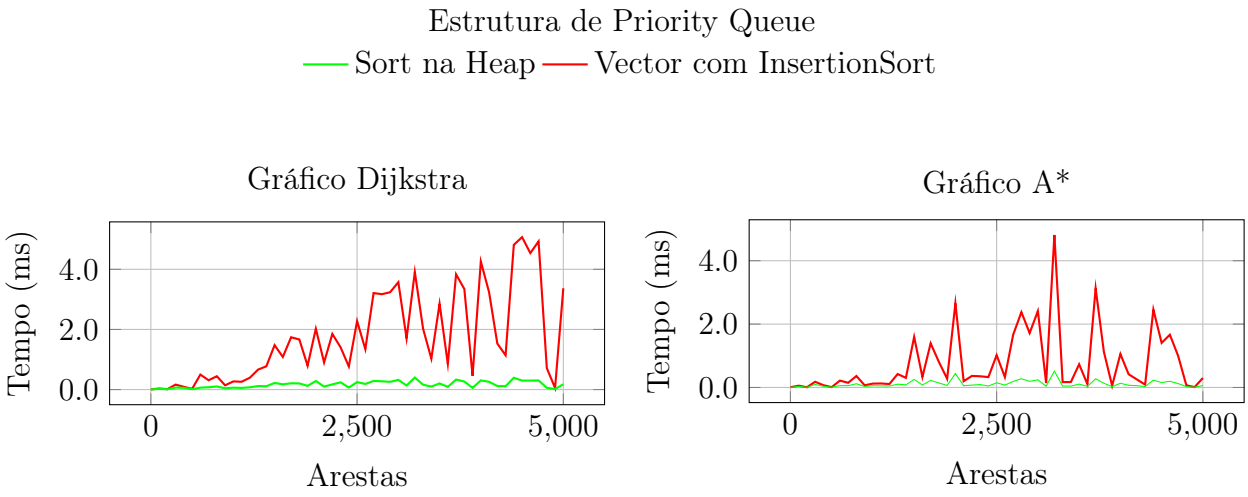
Para esse experimento, o gerador foi configurado para gerar cinco casos de teste num grafo de 100 vértices, em seguida foi calculada a média do tempo gasto nos cinco testes. O gerador operou variando de 0 até 9900 arestas e foi utilizado o algoritmo de Dijkstra para comparar as duas implementações.



Podemos avaliar pelo gráfico, que a Lista de Adjacência é mais eficiente para matrizes esparsas e embora a complexidade assintótica seja a mesma, devido a otimização de constantes para matrizes mais densas a Matriz de Adjacência torna-se mais eficiente. Para grafos com densidade média os algoritmos são praticamente equivalentes. É possível perceber também que há uma correlação positiva entre a memória usada e o tempo de execução gasto pelas duas implementações.

5.3 Comparação da Priority Queue

Para esse experimento, decidimos testar qual Estrutura de Dados se adequa melhor computacionalmente, para isso, comparamos o uso computacional da Heap com o uso de um vetor simples onde a inserção é feita com um InsertionSort(). O grafo utilizado foi o de Lista de Adjacências.

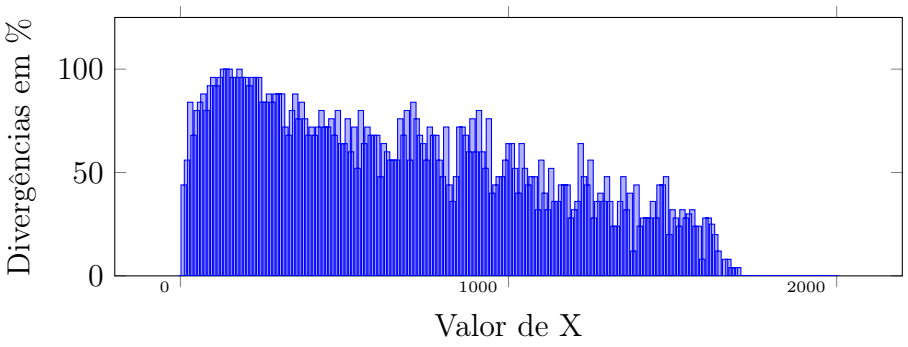


Sendo assim, fica demonstrado, e por muito, que a Heap é mais eficiente que o mais básico InsertionSort no Vetor, podemos perceber por exemplo que nos casos onde a Heap tem picos, o InsertionSort tem esses mesmos picos em tempo muito pior.

5.4 Corretude dos Métodos

Para esse experimento, o gerador foi configurado para gerar alguns casos de testes onde o número de caminhos e portais seguia a fórmula abaixo, com restrições de número de usos de portais variáveis. As configurações do problema são 50 vértices num plano de tamanho 10x10, sendo o gráfico 80% completo, com o número de Portais e Caminhos seguindo a fórmula abaixo. Onde X varia de 0 a 2000 com um passo de 10.

$$\text{Portais} = X \quad \text{e} \quad \text{Caminhos} = 2000 - X$$



Podemos avaliar pelo gráfico, que conforme o número de escolhas entre portais e caminhos aumentam, a chance de divergência de resultados entre o Dijkstra e o A* tendem a ser maiores. É possível perceber também, que se o número de portais a ser utilizados forem nulos ou se houverem somente portais, os algoritmos chegarão na mesma resposta.

6 Conclusões

O Trabalho Prático, simbolizado poeticamente pelo herói lendário Linque, possibilitou a compreensão de duas Estruturas de Dados estudadas em sala, Heap e Grafos, os diferentes métodos de implementá-los, por meio das Listas e Matrizes de adjacências e apresentou ao estudantes a importância de dois algoritmos essenciais no mundo da computação: o Dijkstra e o A*.

O projeto permitiu uma exploração divertida das Estruturas de Dados vistas em salas de aula e permitiu reforçar também que a implementação mais simples nem sempre é a mais eficiente e possibilitou a visualização de como a escolha das Estruturas de Dados influenciam diretamente na busca por soluções e otimização de um programa, dado determinado problema.

7 Bibliografia

Referências

[1] Chaimowicz, L. and Prates, R. (2020). Slides da Disciplina de Estruturas de Dados, Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, Belo Horizonte. Disponível em: https://virtual.ufmg.br/

[2] Campos Filho, F. (2007). *Algoritmos Numéricos*. 2ª edição. Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, Belo Horizonte.

[3] Cormen, T., Leiserson, C., Rivest R., Stein, C. Introduction to Algorithms, Third Edition, MIT Press, 2009. Versão Traduzida: Algoritmos – Teoria e Prática 3a. Edição, Elsevier, 2012