Trabalho Prático 2 - Estruturas de Dados

Wesley Marques Daniel Chaves - 2020053246

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil
wesleymdc@ufmg.br

1. Introdução

O objetivo deste trabalho é solucionar o problema enfrentado pelo heroi Linque, perdido na floresta da neblina. Este estudo foca na implementação e análise dos algoritmos Dijkstra e A* (A-estrela) para determinar se é possível que Linque escape da floresta, considerando as restrições de energia e o uso limitado de portais mágicos. Serão apresentados os algoritmos, suas complexidades, e a análise experimental dos resultados obtidos.

A seção 2 trata de como o programa está organizado, quais métodos foram implementados, além de informações concernentes a configurações da máquina utilizada para a execução dos testes. Já na seção 3, são apresentadas análises de complexidade de tempo e espaço para cada algoritmo, além de hipóteses julgadas importantes. Por fim, a seção 4 busca apresentar e discutir efetivamente os resultados dos experimentos, trazendo informações pertinentes na utilização dos algoritmos escolhidos.

2. Método

2.1. Configuração

Para o desenvolvimento e teste do programa, foram utilizadas as seguintes configurações:

- Sistema Operacional: Windows 11 Home Single Language
- Linguagem de Programação: C++
- Compilador: G++ (GNU Compiler Collection)
- Hardware: Processador Intel Core i5-1135G7, com 16gb de RAM

2.2. Estruturas de Dados

A implementação teve como base a estrutura de dados grafo, na qual as clareiras foram mapeadas para vértices e trilhas para arestas. Foi utilizada duas abordagens para representar o grafo: a lista de adjacência e a matriz de adjacência. Além disso, foi empregada uma fila de prioridades, que é utilizada para realizar operações eficientes de extração do vértice com a menor distância acumulada, sendo de grande importância para os algoritmos utilizados.

2.2.1. Grafo com Matriz de Adjacência

A implementação utiliza uma matriz bidimensional para armazenar as informações sobre os vértices e suas conexões. Cada célula (i, j) armazena uma estrutura, que contém o índice do vértice j vizinho de i, suas coordenadas e a distância para i. Para sabermos se existe uma relação de vizinhança entre i e j, foi utilizada uma flag, que é verdadeiro se existe uma aresta partindo de i para j, e falso caso contrário. Embora a matriz permita acesso direto, sua construção possui complexidade de tempo e espaço de $O(V^2)$.

2.2.2. Grafo com Lista de Adjacência

A lista de adjacência usa um array de estruturas, onde cada posição do array representada por uma estrutura contém um apontador para uma lista de vértices adjacentes, referenciando as arestas do grafo. Cada estrutura contém o índice do vértice, suas coordenadas, a distância para o vértice 'cabeça' e um ponteiro 'next', que mantém a lista encadeada. Embora a lista encadeada não permita acesso direto, sua construção possui complexidade de tempo e espaço de $O(V)$.

2.2.3. Fila de Prioridade

A fila de prioridade foi implementada utilizando a estrutura de dados Min-Heap, que é uma árvore binária na qual para todo nó C, se P é o pai de C, então a chave de P é menor ou igual a chave de C. Em nossa solução, utilizamos um array de estruturas, na qual cada estrutura armazena o índice de um vértice, a menor distância acumulada para se chegar ao vértice por um determinado caminho, a quantidade de portais utilizada e a distância heurística até o destino (no caso do algoritmo A*).

A escolha de adicionar os portais na estrutura utilizada na fila de prioridades foi feita devido a necessidade de isolar a quantidade de portais utilizada para cada caminho, pois caso contrário, a alteração da utilização de portais em um caminho impactaria no uso dos portais por um caminho alternativo, obtendo resultados incorretos. Além disso, armazenar a distância até o destino é essencial para que a fila de prioridades priorize os vértices com base na soma da distância do vértice inicial até ele e a distância estimada até o destino, permitindo que a fila de prioridades organize os vértices de maneira apropriada para o algoritmo A*.

2.3. Classes

2.3.1. Graph

Foi criada uma classe abstrata $Graph$ onde são definidas as operações utilizadas por ambas as abordagens listadas anteriormente. Em particular, $addVertex$, que adiciona um vértice no grafo, $addEgde$, que adiciona uma aresta entre dois vértices e $getWeight$, utilizada para pegar a distância entre dois vértices vizinhos.

Além disso, na implementação de cada subclasse $GraphMatrix$ e $GraphList$, definimos as funções $getVertex$ e $getNeighbours$, que retornam, respectivamente, a estrutura representada por um vértice e todos os vizinhos de um determinado vértice.

2.3.2. MinHeap

Para o TAD fila de prioridades, foi criada a classe $MinHeap$, que implementa a estrutura de dados Min-Heap e suas funções. Entre as operações principais, o método $insert$ adiciona novos elementos ao heap, ajustando a ordem do array para manter a propriedade do Min-Heap. O método $remove$ retira e retorna a raiz da árvore, sendo o elemento com a menor distância computada pertencente à estrutura.

2.3.3. Dijkstra

Implementa o algoritmo de Dijkstra para encontrar o caminho mais curto entre o vértice inicial e o final. A principal função é run, que executa o algoritmo e retorna a distância caso seja possível chegar no objetivo, e retorna um número arbitrariamente alto caso contrário. Para adaptá-lo à lógica dos portais, antes de adicionar um vizinho de um vértice na fila de prioridades, é verificado se a distância entre eles é zero (portal), e caso verdadeiro, é verificado também se a quantidade de portais utilizada para se chegar ao vértice atual é menor que a quantidade permitida. Caso sim, adicionamos o vizinho na fila, caso contrário, apenas passamos para o próximo vizinho.

2.3.4. Star

TAD que implementa o algoritmo A*, que é muito parecida com o algoritmo de Dijkstra, possuindo como função principal run**.** Em tal método, diferentemente do Dijkstra, antes de adicionarmos um vizinho de um vértice na fila de prioridades, é calculado a distância desse vizinho até o vértice final (heurística), sendo também armazenada junto com o vértice.

3. Análise de complexidade

3.1. Graph

addVertex: Envolve apenas a atribuição de valores a um array, sem utilizar espaço adicional significativo de memória, dessa forma, possui complexidade de tempo e espaço tanto para matriz de adjacência quanto para lista de adjacência de $O(1)$.

addEdge: Pode exigir a travessia da lista de adjacência para encontrar o final, tendo complexidade de tempo no pior caso de $O(V)$. Porém, para a matriz de adjacência, temos atribuição direta a célula, sendo $O(1)$. Com relação ao espaço, ambas as implementações alocam um novo nó que é armazenado no grafo, tendo complexidade de espaço de $O(1)$.

getWeight: Exige a travessia da lista de adjacência caso o vizinho seja o último da lista, tendo complexidade de tempo de O(V) no pior caso. Novamente, para a matriz de adjacência teremos O(1). Como utiliza um espaço adicional constante, a complexidade de espaço para ambas as implementações é O(1).

getVertex: Em ambas as implementações, o método retorna o vértice diretamente do array, com complexidade de tempo O(1). O espaço adicional utilizado é constante em ambas as implementações, resultando em complexidade de espaço O(1).

getNeighbours: Para a lista de adjacência, o método retorna a lista de vizinhos de um vértice diretamente, enquanto para a matriz de adjacência, retorna um ponteiro para a linha correspondente ao vértice. Em ambos os casos, teremos a complexidade de tempo $O(1)$. O espaço adicional utilizado é constante em ambas as implementações, resultando em complexidade de espaço O(1).

3.2. MinHeap

insert: A complexidade de tempo é dominada pela reorganização, que pode realizar a troca do elemento com seus ancestrais até a raiz, sendo $O(log(n))$. Como a operação é realizada in-place, teremos a complexidade de espaço $O(1)$.

remove: Novamente, a complexidade de tempo é dominada pela reorganização. Após colocarmos o último elemento na raiz, pode ser realizada até $log(n)$ trocas para colocá-lo na posição correta, tendo portanto complexidade de $O(log(n))$. A complexidade de espaço é $O(1)$, sendo in-place.

3.3. Dijkstra e A* (A-estrela)

Para a matriz de adjacência, a inserção e remoção no heap têm complexidade de $O(log(V))$ e acessar todos os vizinhos para cada vértice requer $O(V^2)$, resultando em uma complexidade de tempo total de $O(V^2 log(V))$. Já para a lista de adjacência, a complexidade de tempo é $O((V + E)log(V))$, pois acessar todos os vizinhos exige $O(E)$.

A matriz de adjacência utiliza $O(V^2)$ de espaço, enquanto outras estruturas, como o heap, utilizam $O(V)$, resultando em uma complexidade de espaço total de $O(V^2)$. A complexidade de espaço é $O(V + E)$, devido ao uso da lista de adjacência e do heap.

4. Estratégias de Robustez

Para aumentar a confiabilidade do código, algumas práticas foram empregadas. Foram introduzidas exceções para lidar com entradas incorretas e evitar operações inválidas, como adicionar elementos a uma fila de prioridades que já está cheia ou tentar remover de uma fila vazia. A validação de alocações dinâmicas de memória foi implementada para garantir que qualquer falha na alocação fosse devidamente tratada. Além disso, o código passou por uma análise detalhada com o Valgrind para detectar e resolver problemas de gerenciamento de memória, garantindo a ausência de vazamentos e acessos não autorizados.

5. Análise Experimental

Nos experimentos, foi executado tanto o Dijkstra quanto o A* em diversos grafos, na qual a quantidade de vértices foi incrementada a cada execução. Como tanto as coordenadas dos vértices quanto as arestas e portais se alternavam de acordo com o grafo, foi possível testar ambos os algoritmos em diferentes configurações, permitindo uma análise precisa dos pontos fortes e fracos de cada um. Além disso, foi feita uma comparação entre a execução dos algoritmos em grafos implementados como lista de adjacência e matriz de adjacência. Dessa forma, não foi analisado apenas os procedimentos em si, mas sim a melhor estrutura subjacente do grafo em cada caso.

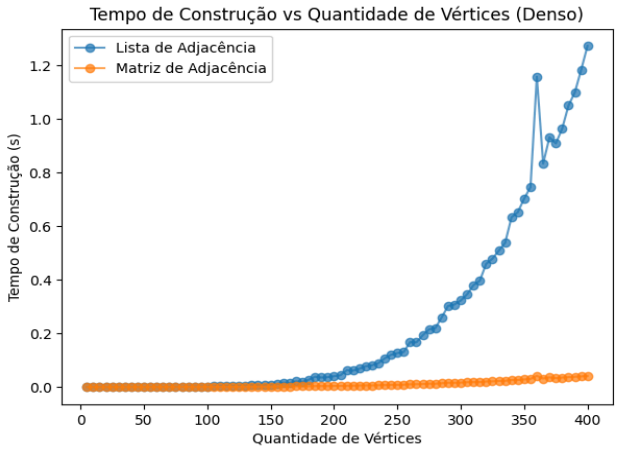
Como será possível observar, a análise foi feita separadamente em grafos densos e esparsos, onde um grafo denso possui densidade de 0.7, e um esparso densidade de 0.3. Dessa forma, para se realizar o cálculo da quantidade de arestas de um grafo dado uma densidade D e uma determinada quantidade de vértices V, foi utilizada a seguinte fórmula:

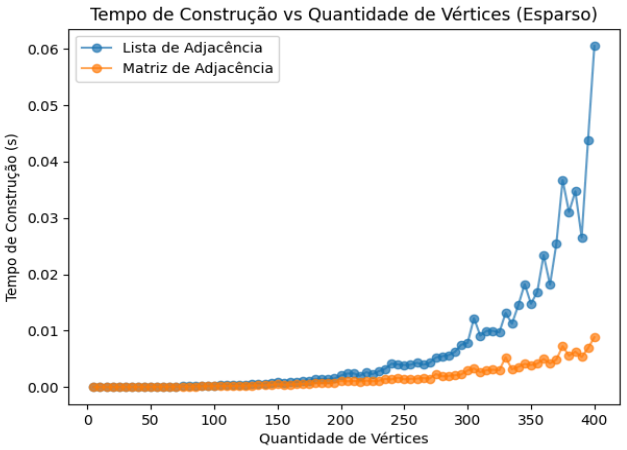
$$D = \frac{|E|}{|V| \times (|V| - 1)}$$

Onde o numerador é a quantidade de arestas (E) e o denominador a quantidade máxima de arestas em um grafo direcionado com V vértices.

5.1. Construção

Inicialmente, foi comparada o tempo de inicialização de cada grafo, na qual são realizadas a criação e inserção de todas as arestas necessárias.

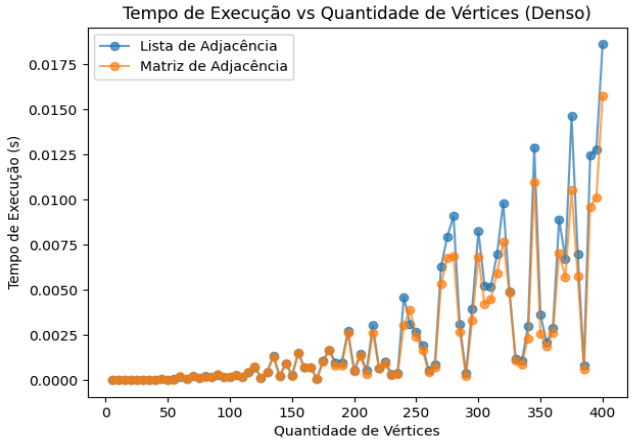


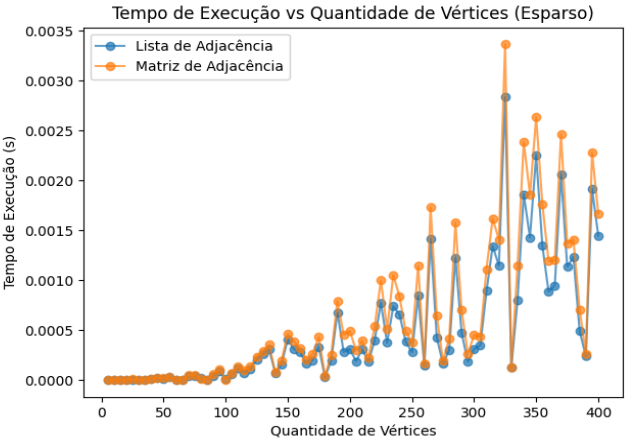


Em ambos os casos, o tempo de construção em grafos que utilizam a lista de adjacência como estrutura de dados demoraram muito mais em comparação com os que utilizam matriz de adjacência, demonstrando que o acesso direto possui efeito substancial. No entanto, para grafos com poucos vértices, tal diferença não é muito visível, o que pode tornar o uso da lista de adjacência vantajoso, dependendo do objetivo. Além do mais, a diferença entre os tempos não é tão acentuada em grafos esparsos como nos densos.

5.2. Dijkstra

É apresentado os tempos de execução do algoritmo em ambas as implementações do TAD grafo, considerado grafos densos e esparsos.





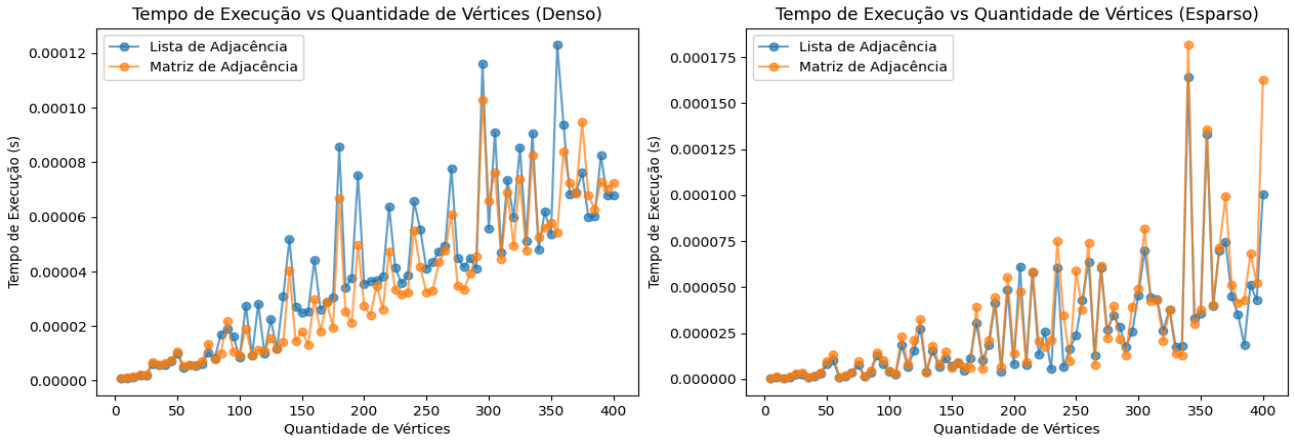
Em ambas as estruturas de dados, o algoritmo possui um aumento substancial de tempo conforme aumenta a quantidade de vértices, tendo valores maiores em grafos densos em comparação aos grafos esparsos. Além disso, é possível visualizar uma dinâmica adicional: em grafos densos, a matriz de adjacência performa melhor, já em grafos esparsos, vemos uma vantagem da lista de adjacência.

Em grafos densos, o acesso constante permitido pela matriz de adjacência compensa a complexidade alta do algoritmo. Adicionalmente, o tempo para percorrer as longas listas de

adjacências se tornam mais um fator que explica a diferença observada. Para os grafos esparsos, as listas de adjacências são pequenas em comparação com as linhas da matriz de adjacência que representam os vértices, na qual a maioria das células não representam vizinhos, dessa forma, a implementação com listas requer menos tempo e espaço que a matriz de adjacência.

5.3. A* (A-estrela)

Similarmente, é apresentado os tempos de execução do algoritmo em ambas as implementações do TAD grafo, considerado grafos densos e esparsos.



De forma equivalente ao algoritmo Dijkstra, a matriz de adjacência é mais eficiente em grafos densos, já a lista de adjacência é melhor em grafos esparsos. No entanto, não é possível observar uma diferença substancial de tempo entre grafos densos e espaços, o que pode indicar que a heurística utilizada pode não ser muito informativa, prejudicando o andamento do algoritmo em grafos esparsos.

Outro aspecto importante de se considerar é a diferença brutal de tempo entre o Dijkstra e o A*. Nas mesmas entradas, o A* encontrou sua solução em tempos muitos menores, evidenciando um outro aspecto ainda não percebido.

5.4. Qualidade

Com relação aos resultados, foi observado uma diferença de resultados entre Dijkstra e o A* em 15% dos casos, ou seja, supondo que o Dijkstra sempre retorna a resposta correta, o A* retorna a resposta correta apenas em 85% dos casos. Considerando a questão levantada anteriormente com relação aos tempos de execução do A* em comparação ao Dijkstra, deve-se analisar o tradeoff verificado, ou seja, o tempo de execução versus precisão.

Em busca de explicar o motivo da discrepância, foi realizado diversos testes, que mostrou que a discrepância entre A* e Dijkstra ocorre na maior parte dos casos, quando existem portais, dando uma evidência de que a utilização do A* com a heurística sendo a distância euclidiana em problemas nas quais existem arestas nulas, pode não ser adequada, caso nenhum outro fator seja determinante na escolha.

6. Conclusão

Este trabalho prático focou na implementação e análise dos algoritmos Dijkstra e A* para resolver o problema do heroi Linque na floresta da neblina. Através da aplicação desses algoritmos em diferentes representações de grafos, isto é, lista de adjacência e matriz de adjacência, e a utilização de uma fila de prioridades, foi possível explorar os desempenhos e complexidades associadas a cada abordagem.

Os resultados experimentais evidenciaram que a escolha da estrutura de dados influencia significativamente o desempenho dos algoritmos. Em grafos densos, a matriz de adjacência mostrou-se superior devido ao seu acesso direto eficiente, enquanto a lista de adjacência apresentou melhor desempenho em grafos esparsos. A análise também destacou que o algoritmo A* oferece soluções de forma mais rápida, embora com uma perda razoável de precisão em comparação ao Dijkstra, refletindo um trade-off entre a corretude e o tempo.

Em resumo, este trabalho reforça a importância de adaptar a escolha do algoritmo e da estrutura de dados às características do problema. A experimentação prática confirmou que não existe uma solução única para todos os cenários. A eficiência dos algoritmos depende do contexto, e a análise cuidadosa das necessidades e restrições do problema é crucial para alcançar os melhores resultados.

7. Bibliografia

Chaimowicz, L. and Prates, R. (2020). Slides virtuais da disciplina de estruturas de dados. Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

MEIRA, Wagner; FIGUEIREDO, Éder. Slides da disciplina. Estrutura de dados. Universidade Federal de Minas Gerais, 2024.

Wikipédia. Dijkstra's Algorithm. Wikipédia, a enciclopédia livre. Disponível em: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm.

Wikipédia. A* Search Algorithm. Wikipédia, a enciclopédia livre. Disponível em: https://en.wikipedia.org/wiki/A*_search_algorithm.