Universidade Federal de Minas Gerais
Ciência da Computação

DCC205: Estruturas de Dados

Autor: Luisa Lopes Carvalhaes
Matrícula: 2023028064

Trabalho Prático II

"Escapando da floresta da neblina"

July 24, 2024

1 Introdução

O objetivo deste trabalho é utilizar estruturas de dados e algoritmos para resolver um problema que envolve o percorrimento de um grafo. O problema que buscamos resolver, de forma resumida, é o seguinte:

"Linque, um herói lendário, está perdido na Floresta da Neblina, que está ficando cada vez mais densa e mágica. Ele precisa escapar rapidamente antes que a neblina o aprisione para sempre. A floresta tem n clareiras e m trilhas unidirecionais, algumas das quais têm portais que não consomem energia. Linque está exausto e possui s unidades de energia, podendo usar os portais no máximo k vezes. Começando na clareira 0, ele precisa chegar à clareira $n-1$ sem exceder o limite de energia e portais. O objetivo é determinar se Linque consegue escapar da floresta dadas essas restrições."

Para resolver esse problema, utilizaremos algoritmos de percorrimento em grafos, em especial, os algoritmos de Dijkstra e A* (A estrela). No entanto, para que seja possível a implementação desses algoritmos, é essencial o uso de algumas estruturas de dados para armazenar e manipular os dados do nosso problema, como o grafo que modelará a estrutura da floresta e o uso da fila de prioridade para a implementação dos algoritmos, entre outros. Portanto, como o foco da disciplina é em estruturas de dados, daremos ênfase em como essas estruturas são trabalhadas, como sua implementação impacta o resultado do problema, e em quais contextos cada tipo de estrutura é mais adequada analisando seus impactos no desempenho e na eficiência da solução.

2 Método e Análise de Complexidade

Nesta seção, descreveremos detalhadamente a implementação da solução para o problema apresentado anteriormente, com ênfase nas estruturas de dados utilizadas e na análise de complexidade. Decidimos unir os tópicos de *"Método"* e *"Análise de Complexidade"* para proporcionar uma visão integrada da implementação de cada estrutura de dados e algoritmo, com a análise teorica de complexidade das operações sobre essas estruturas.

Para modelar o problema, foi utilizada uma estrutra `Graph`, representando cada clareira como um vértice do grafo e cada trilha como uma aresta. Cada clareira é modelada como uma estrutura do tipo `Point`, que possui coordenadas `x` e `y`, e cada trilha (aresta) tem o peso correspondente à distância euclidiana entre os vértices. Os portais são modelados como arestas de peso zero, conectando as clareiras.

Para armazenar essa estrutura de grafo, consideramos duas possíveis estruturas de dados: a matriz de adjacência, `Adjacency Matrix`, e a lista de adjacência, `Adjacency List`. Para a implementacao da segunda, utilizamos um tipo de dado como base, a Lista encadeada (`Linked List`). Mais adiante, explicaremos em detalhes cada uma dessas estruturas e faremos uma comparação entre elas.

Com a principal estrutura do problema definida, resta desenvolver os algoritmos de resolução. Os dois algoritmos propostos são o `Dijkstra` e o `A* (A estrela)`. Ambos os algoritmos têm uma implementação similar, diferenciando-se apenas pela utilizacao de uma heurística. Para sua implementação, utilizamos a fila de prioridade, `PriorityQueue`, baseada em um `Heap`. Esta estrutura de dados também será analisada posteriormente com detalhes.

Além disso, utilizamos tipos de dados auxiliares básicos, como `Node`, `Edge`, `Pair` entre outros, para suportar a implementação das estruturas principais e dos algoritmos.

2.1 Estruturas de Dados

Descrevemos a seguir a implementção e a complexidade das estruturas de dados utilizadas:

2.1.1 Grafo

O grafo é representado utilizando a classe `Graph`, que pode ser implementada tanto com uma matriz de adjacência quanto com uma lista de adjacência. Para isso, a classe `Graph` é implementada como a classe pai de `AdjList` e `AdjMatrix`, assim possui métodos virtuais como seu destrutor e este método (`virtual LinkedList<Edge> neighbors(int v)`) que é especificado pelas classes filhas. De forma geral, a classe

`Graph` armazena um vetor de pontos `Points` como vértices, (`n`) número de vértices, (`m`) número de arestas e (`p`) número de portais, e possui funções como `getPoint` que retorna o ponto que representa aquele vértice. O resto é especificado nas classes filhas, definindo se o grafo armazena as arestas como lista ou matriz de adjacência. A seguir, são detalhadas as duas principais representações para grafos, suas operações e suas complexidades.

- **Lista de Adjacência:** Uma lista de adjacência é uma estrutura de dados onde cada vértice é associado a uma lista encadeada contendo todos os seus vizinhos. Esta estrutura é mais eficiente em termos de espaço ao armazenar grafos. Na implementação da lista encadeada, criamos uma classe que herda de `Graph`. Nela, utilizamos a estrutura `LinkedList` que será detalhada mais à frente. Esta lista utiliza templates `Node`, que são os nós da lista, e neles são armazenados dados do tipo `Edge`, ambos também serão detalhados posteriormente. Em resumo, a lista de adjacência é uma lista de listas encadeadas de `Edge` instanciada da seguinte forma: `LinkedList<Edge>* list;`. A classe em si é bem simples, tendo um construtor e destrutor próprio, e especificando a função `LinkedList<Edge> neighbors(int v) const override;` que retorna a lista de vizinhos do vértice v. Além disso, também possui a função `void addEdge(int u, int v, double weight)` que utiliza a lista encadeada para adicionar vértices. Outras funções não estão detalhadas pois não foram necessárias para a resolucão do problema proposto.

  - **Verificação de Existência de Aresta:** A complexidade é $O(d)$, onde d é o grau do vértice. Para verificar se existe uma aresta entre dois vértices u e v, é necessário percorrer a lista de adjacência do vértice u, procurando o vértice v. O tempo necessário para isso é proporcional ao número de vizinhos de u.
  - **Inserção de Aresta:** A complexidade é $O(1)$. Inserir uma aresta em uma lista de adjacência envolve adicionar um novo nó na lista de vizinhos do vértice u. Esta operação pode ser realizada no início da lista de forma constante.
  - **Listagem de Vizinhos:** A complexidade é $O(d)$, onde d é o grau do vértice. Listar todos os vizinhos de um vértice u envolve percorrer toda a lista de adjacência de u, o que é linear em relação ao número de vizinhos.

  O espaço utilizado para armazenar a lista de adjacência é $O(n + m)$, onde n é o número de vértices e m é o número de arestas. Este espaço é proporcional ao número de vértices e arestas reais no grafo. Esta estrutura é mais eficiente em termos de espaço para grafos esparsos, onde m é menor que n^2.

- **Matriz de Adjacência:** Uma matriz de adjacência é uma matriz $n \times n$, onde n é o número de vértices no grafo. Cada elemento `matrix[i][j]` da matriz representa o peso da aresta entre os vértices i e j. Esta estrutura é ideal para grafos densos, onde a maioria dos pares de vértices possui arestas, já que ocupa mais espaço. Assim como a lista de adjacências, essa classe foi implementada herdando de `Graph`, porém, diferente dela, não é baseada em outra estrutura de dados como a lista encadeada. Ela é apenas uma matriz `double ** matrix;` e possui os mesmos métodos da lista de adjacência, para que dessa forma seja possível utilizar ambas da mesma forma no código principal.

  - **Verificação de Existência de Aresta:** A complexidade é $O(1)$. Isso ocorre porque o acesso a um elemento da matriz é feito por índice, o que é uma operação constante. Para verificar se há uma aresta entre dois vértices i e j, basta acessar o elemento `matrix[i][j]` diretamente.
  - **Inserção de Aresta:** A complexidade é $O(1)$. Inserir uma aresta na matriz de adjacência envolve simplesmente atualizar o valor na posição `matrix[i][j]` com o peso da aresta. Este acesso e atualização são operações de tempo constante.
  - **Listagem de Vizinhos:** A complexidade é $O(n)$. Para listar todos os vizinhos de um vértice v, é necessário percorrer toda a linha v da matriz de adjacência, verificando quais colunas contêm um valor diferente de zero (indicando a presença de uma aresta). Isso exige uma varredura completa da linha, resultando em uma complexidade linear em relação ao número de vértices.

  O espaço utilizado para armazenar a matriz de adjacência é $O(n^2)$, onde n é o número de vértices. Este espaço é necessário para armazenar todos os pares possíveis de vértices, independentemente de haver ou não uma aresta entre eles. Em grafos esparsos, onde o número de arestas é muito menor que n^2, essa estrutura pode ser ineficiente em termos de espaço.

2.1.2 Lista Encadeada

A classe `LinkedList` representa uma lista encadeada genérica, onde cada elemento é armazenado em um nó que contém um ponteiro para o próximo nó na sequência. A lista encadeada utilizada foi criada como um template para ser versátil com diferentes tipos de dados. A estrutura é composta por nós, `Node`. Cada nó contém um dado genérico `T` e um ponteiro para o próximo nó. A lista possui dois ponteiros: `head` e `tail`. O `head` aponta para o primeiro nó da lista, enquanto o `tail` aponta para o último nó. Assim como outras estruturas, a lista possui um construtor, destruidor e também sobrecargas do operador de atribuição e construtor de cópias, caso seja necessário fazer cópias por atribuição. A seguir, detalhamos alguns dos principais métodos e analisamos a complexidade associada a cada operação.

- **Inserção:** Insere um novo nó no final da lista. Se a lista estiver vazia, o novo nó se torna tanto o `head` quanto o `tail`. Caso contrário, o nó atual `tail` é atualizado para apontar para o novo nó, e o `tail` é atualizado para o novo nó. A complexidade de tempo é $\mathcal{O}(1)$, pois a operação de inserção é realizada em tempo constante, atualizando apenas o ponteiro do `tail` e criando o novo nó. A complexidade de espaço também é $\mathcal{O}(1)$, já que é necessário um espaço adicional constante para o novo nó.
- **Limpeza (`clear`):** Remove todos os nós da lista, liberando a memória alocada para cada nó. O ponteiro `head` é atualizado para `nullptr` após a remoção de todos os nós. A complexidade de tempo é $\mathcal{O}(n)$, onde n é o número de elementos na lista, pois cada nó é visitado e removido, resultando em um tempo linear em relação ao número de elementos. A complexidade de espaço é $\mathcal{O}(1)$, uma vez que a operação de limpeza utiliza um espaço adicional constante para variáveis temporárias durante a remoção dos nós.
- **Métodos de Iteração (`begin` e `end`):** O método `begin` retorna o ponteiro para o primeiro nó da lista (`head`), enquanto o método `end` retorna um ponteiro nulo (`nullptr`), indicando o final da lista. A complexidade de tempo é $\mathcal{O}(1)$ para ambos os métodos, pois eles apenas retornam ponteiros sem realizar iterações ou cálculos adicionais. A complexidade de espaço é $\mathcal{O}(1)$, já que a operação de obtenção dos ponteiros utiliza um espaço adicional constante.

De maneira geral, a lista encadeada ocupa um espaço de $\mathcal{O}(n)$, onde n é o número de elementos na lista. No entanto, seu uso no programa está diretamente atrelado à estrutura da lista de adjacência, definindo a forma como a lista é utilizada. Álem disso, muitos métodos comuns a listas encadeadas, como inserção e remoção no meio da lista, foram excluídos, pois não são necessários para a solução do problema específico.

2.1.3 Fila de Prioridade

A fila de prioridade é frequentemente implementada utilizando um min-heap, uma estrutura de dados crucial em algoritmos como Dijkstra e A*, para selecionar o próximo vértice a ser explorado com base na menor distância ou na heurística. Um min-heap é uma árvore binária completa onde cada elemento tem uma prioridade associada e o elemento com a menor prioridade está sempre no topo da estrutura. Em outras palavras, cada nó na árvore é menor ou igual aos seus filhos. Essa propriedade garante que as operações de acesso e remoção do elemento de menor prioridade sejam eficientes, tipicamente realizadas em tempo logarítmico.

A inserção de novos elementos também é eficiente, mantendo uma complexidade logarítmica, pois se ajusta a estrutura após cada inserção ou remoção para manter sua propriedade de heap.

No contexto da implementação, a fila de prioridade armazena um tipo de dado `T` em um array `heap`, acompanhado por um tamanho atual `size`, um tamanho máximo definido `MAX_SIZE`, e operações básicas para a manipulação dos dados do heap. As operações principais incluem `heapify_up` e `heapify_down`, que são usadas para ajustar a estrutura do heap durante as operações de inserção e remoção, assegurando que a propriedade do heap seja preservada. Além disso, métodos auxiliares como `get_parent`, `get_left_child`, e `get_right_child` são utilizados para acessar os nós pai e filhos em uma dada posição do heap. A complexidade de espaço é `O(1)`, já que não requer memória adicional.

Além disso, a implementação dos operadores de comparação é crucial, pois define o critério para determinar qual elemento da fila é maior ou menor que o outro. No caso do algoritmo de Dijkstra, os operadores de comparação devem basear-se nas distâncias armazenadas: a menor distância é utilizada como critério de ordenação e, consequentemente, como a prioridade no código.

Por outro lado, na implementação do algoritmo A*, o critério de ordenação é a soma da distância já percorrida e da estimativa da distância restante até o objetivo (heurística). Isso foi necessário para a implementação do comparador do A*, como `struct CompareAstar`, garantindo que a fila de prioridade seja ordenada adequadamente de acordo com a necessidade do algoritmo.

A seguir, exploraremos em detalhes as complexidades de tempo e espaço associadas a essa estrutura.

- **Heapify_up:** Usado para restaurar a propriedade de heap após a inserção de um novo elemento. Quando um elemento é inserido, ele é colocado na última posição do heap e, em seguida, pode precisar ser movido para cima na árvore até que a propriedade de heap seja restabelecida. A complexidade de tempo é `O(log n)`, onde `n` é o número de elementos no heap. Isso ocorre porque, no pior caso, o método pode precisar subir desde a folha até a raiz, e a altura do heap é logarítmica em relação ao número de elementos.

- **Heapify_donw:** Responsável por restaurar a propriedade de heap após a remoção do elemento no topo do heap. Após a remoção, o último elemento é movido para a posição do topo e pode precisar ser movido para baixo na árvore até que a propriedade de heap seja restaurada. A complexidade de tempo é `O(log n)`. Isso porque o método pode precisar percorrer a altura do heap para garantir que a estrutura de heap seja mantida. A altura do heap é logarítmica. A complexidade de espaço é `O(1)`, pois o método não requer memória adicional.

- **Métodos Insere e Remove:** O método `insert` adiciona um novo elemento ao heap. O elemento é inicialmente colocado na última posição do heap e, em seguida, o `heapify_up` é chamado para restaurar a propriedade de heap. Esse método segue as complexidade do `Heapify_up`, `O(log n)`. Já o `remove` remove o elemento de menor prioridade (o topo do heap) e substitui-o pelo último elemento do heap. Em seguida, o `heapify_down` é chamado para restaurar a propriedade de heap, logo tem compelxidade `O(log n)`, na mesma lógica da inserção.

- **Top:** Retorna o valor do elemento de menor prioridade na fila, que é o elemento localizado no topo do min-heap. Esse valor é sempre o menor elemento. A complexidade de tempo para o método `top` é `O(1)`, pois é uma operação direta, o mesmo vale para complexidade de espaço `O(1)`.

2.1.4 Tipos Auxiliares/Simples

Utilizamos diversos tipos de dados auxiliares básicos, que são parte integrante das estruturas de dados mencionadas:

- **Node**: Tipo de dado utilizado na lista encadeada, implementado como um template. Armazena um tipo genérico `T` e um ponteiro para o próximo nó.

- **Edge**: Tipo de dado usado nos nós da lista encadeada para modelar o grafo com listas de adjacência. Armazena um inteiro `V` que representa o vértice de chegada e o peso da aresta.

- **Pair**: Tipo utilizado na fila de prioridades, que armazena um par genérico de dados `T1` e `T2` como `first` e `second`, além de sobrecargas para os operadores mais comuns.

- **Point**: Implementado para o grafo, armazena as coordenadas dos vértices como `double x` e `y`. Inclui funções para retornar `x` e `y`, e calcular a distância euclidiana entre o ponto e outro ponto.

2.2 Algoritmos de Resolução

Descrevemos a seguir os algoritmos de resolução utilizados:

2.2.1 Algoritmo de Dijkstra

O algoritmo de Dijkstra é um método clássico utilizado para encontrar o caminho mais curto de um vértice a todos os outros em um grafo ponderado com arestas não negativas.

De forma clássica, o funcionamento do algoritmo de Dijkstra envolve o uso de uma fila de prioridade para explorar os vértices com a menor distância acumulada desde o ponto de partida. A cada iteração, o

vértice com a menor distância é removido da fila e seus vizinhos são atualizados. Esse processo é repetido até que todos os vértices sejam processados ou o vértice destino seja alcançado. Assim, cada aresta do grafo é processada apenas uma vez, quando a extremidade do seu vértice é processada. Isso ocorre pois temos m arestas, o loop que percorre os vizinhos vai no máximo iterar m vezes, ou seja, m inserções na fila de prioridades, mas como se utiliza um min-heap, é garantido que o vértice será processado pela primeira vez com o custo mínimo.

Tendo isso em vista, podemos ver a complexidade do algoritmo. A complexidade de tempo do algoritmo de Dijkstra, quando implementado com uma fila de prioridade usando um min-heap, possui uma complexidade de $O((V+E)\log V)$, onde V é o número de vértices e E é o número de arestas no grafo. Isso ocorre já que cada vértice é processado uma vez, se precisar processar cada aresta, no pior caso, nos leva a essa complexidade. A inserção e remoção na fila de prioridade têm complexidade $O(\log V)$, e cada aresta é processada uma vez, por conta da natureza de armazenamento do min-heap que já foi comentado, resultando numa complexidade final de $O((V+E)\log V)$.

Para o problema em questão, devemos levar em consideração que, devido à presença de portais, rotas distintas podem visitar o mesmo vértice. Logo, devemos considerar os diferentes caminhos com o uso de uma quantidade j de portais, para cobrir todos os casos. Para isso, foi utilizada uma matriz de $n \times (k+1)$, assim `dist`$[i][j]$ representa a menor distância até o vértice i usando j portais. Assim, enquanto houverem caminhos no grafo de trilhas e posteriormente no grafo de portais, atualizaremos a fila de prioridade de acordo com a lógica inicial do algoritmo. Se alguma das rotas ultrapassar o número máximo de portais a ser utilizado, ela é descartada. Por fim, quando o vértice final for processado, checamos se a energia utilizada é menor que a energia s do personagem. Depois, pegando a matriz de distâncias, é simples saber o menor caminho, é só checar todas as menores distâncias com as j possibilidades de uso de portal.

Adaptando a complexidade teórica para o nosso problema específico, temos o seguinte: cada vértice é processado no máximo $O(k)$ vezes, logo $O((V+E)\cdot k\cdot\log(n\cdot k))$. A complexidade de espaço é $O(V\cdot(k+1))$ devido à matriz de distâncias `dist`, que armazena a distância mínima para cada combinação de vértices e contagem de portais.

2.2.2 Algoritmo A*

O algoritmo A* é uma extensão do algoritmo de Dijkstra que utiliza uma heurística para guiar a busca na direção do objetivo. A heurística pode variar, mas a que estamos utilizando é a distância euclidiana. Portanto, a implementação e resolução do problema são bem similares ao que já foi explicado no algoritmo de Dijkstra.

O algoritmo A* combina a distância acumulada do ponto de partida com uma estimativa da distância restante até o objetivo, fornecida pela heurística. A cada iteração, o algoritmo explora os vértices, priorizando aqueles que têm a menor soma da distância real percorrida adicionada à estimativa heurística. Na prática, o vértice com a menor soma da distância real e heurística é removido da fila de prioridade, e seus vizinhos são atualizados. Se um caminho mais curto para um vizinho for encontrado, a heurística desse vizinho é atualizada e o vizinho é inserido novamente na fila.

A complexidade de tempo é análoga à do algoritmo de Dijkstra, sendo $O((V+E)\log V)$ quando se utiliza uma fila de prioridade baseada em min-heap. No entanto, o desempenho real pode ser significativamente melhor do que o de Dijkstra se a heurística usada for eficiente naquele contexto, pois A* pode explorar menos vértices ao usar a heurística para guiar a busca. Adaptando ao contexto do nosso problema, teríamos uma complexidade de $O((V+E)\cdot k\cdot\log(n\cdot k))$, já que utilizamos a mesma fila de prioridade e cada vértice é processado $O(k)$ vezes no máximo. Quanto às inserções no heap, como no Dijkstra, no máximo $O(n\cdot k)$, resultando na complexidade analisada.

A complexidade de espaço também é $O(V\cdot(k+1))$, semelhante ao Dijkstra, devido às matrizes 'dist' que armazenam a distância acumulada para cada vértice.

Discussão sobre o impacto da Heurística: Tanto o algoritmo de Dijkstra quanto o A* são ferramentas poderosas para encontrar o caminho mais curto em grafos, com Dijkstra sendo mais generalizado e A* aproveitando heurísticas para otimizar a busca. No entanto, a presença de arestas nulas em um grafo direcionado, como o problema específico que estamos enfrentando, pode impactar significativamente o desempenho e a precisão do algoritmo A*.

No contexto de grafos direcionados com arestas nulas, a heurística do A* pode se tornar problemática. A heurística, que é uma estimativa do custo restante para o destino, assume que a soma das distâncias reais e heurísticas fornece uma estimativa razoável do custo total do caminho. No entanto, a presença de arestas com peso zero (ou nulo) pode quebrar a desigualdade triangular, uma propriedade matemática que a heurística geralmente utiliza para garantir que a estimativa seja válida. Isso pode levar a situações onde o caminho mais curto não é imediatamente evidente com base na heurística. Em que a heurística está contabilizando uma distância que na prática não existiria, já que a presença de um portal a tornaria 0. Por exemplo, se uma aresta nula permite um atalho significativo entre dois vértices, a heurística pode não capturar a importância desse atalho de forma adequada, resultando em caminhos incorretos ou ineficientes.

Portanto, ao lidar com grafos que possuem arestas nulas, o algoritmo A* pode enfrentar desafios adicionais em comparação com o algoritmo de Dijkstra. A heurística do A* pode não fornecer uma estimativa precisa do custo total do caminho, o que pode levar a soluções subótimas ou incorretas. Em tais casos, Dijkstra, que não depende de heurísticas e explora todos os caminhos possíveis para encontrar a solução ótima, pode ser mais confiável. A escolha do algoritmo deve considerar o contexto inserido.

3 Estratégias de Robustez

Durante a implementação das estruturas de dados e dos algoritmos, foram adotadas várias práticas para garantir a robustez do código. A modularização foi uma das principais estratégias adotadas, resultando em estruturas de dados com baixo acoplamento e alta coesão. Isso facilitou a manutenção e a compreensão do código, permitindo que as partes individuais fossem desenvolvidas e testadas de forma isolada. Estruturas que precisavam ser genéricas para manipular tipos variados de dados foram implementadas como templates, como nos casos de `Node`, `PriorityQueue` e outras estruturas descritas no tópico "Método". Essa abordagem não só facilita a reutilização do código, como também assegura que ele se mantenha organizado e compreensível.

Os arquivos de código foram devidamente organizados em arquivos de cabeçalho (.h) e arquivos de implementação (.cpp), o que ajuda a manter uma estrutura limpa e facilita o entendimento e a manutenção do código. Cada classe e função foi projetada com cuidado para garantir que todos os construtores e destrutores fossem implementados corretamente. Esse cuidado é essencial para evitar problemas como vazamentos de memória. A alocação e desalocação de memória foram gerenciadas de forma cuidadosa, e o Valgrind foi utilizado para verificar a correta liberação de memória.

Além disso, exceções foram utilizadas de maneira extensiva para garantir a robustez do código. Verificações foram implementadas para assegurar que a alocação de memória fosse bem-sucedida e que a memória fosse corretamente liberada. Quando novos vértices, nós ou itens eram inseridos nas estruturas de dados, exceções foram usadas para validar se a inserção foi realizada corretamente. Da mesma forma, ao acessar ou manipular dados, o código inclui verificações rigorosas para garantir que todos os acessos estejam dentro dos limites válidos e que não haja referências inválidas.

Outra questão é a não existência de interação direta com o usuário. Isso reduz significativamente a possibilidade de erros causados por entradas inesperadas ou mal formatadas. Em vez disso, o código é executado por meio de um script que gera as entradas conforme necessário, eliminando a necessidade de validações extensivas de entrada dentro do próprio código.

4 Análise Experimental

Nesta seção, serão apresentados os resultados dos experimentos realizados, abordando o desempenho computacional e as análises correspondentes. Buscamos explorar três principais questões:

- Análise da complexidade experimental para verificar os resultados teóricos apresentados na documentação.
- Comparação das implementações de matriz e lista de adjacência. Como elas impactam a execução? Existem tipos de instâncias que favorecem a utilização de alguma das implementações?
- Análise da qualidade das soluções obtidas. Os algoritmos sempre encontram as mesmas soluções? Se você variar ou restringir alguns dos parâmetros de entrada, essa resposta muda? Por quê?

Para isso, foi desenvolvido um script em Python que gera diferentes tipos de grafos. Foram definidas cargas de trabalho variando tamanhos de grafos, grafos que geram representações esparsas versus densas, e variações entre muitos, poucos ou nenhum portal (arestas de peso 0). Também foram considerados casos extremos, com valores extremamente pequenos ou grandes dessas cargas de trabalho. No geral, os tamanhos dos grafos gerados variaram de 0 a 10^5. É importante ressaltar que os grafos criados são sempre conexos, com um caminho de 0 a n e as distâncias das coordenadas variam de 0 a 10^5.

Análise da Complexidade Experimental

Verificação dos Resultados Teóricos

Nesta análise, buscamos verificar se os resultados obtidos experimentalmente estão alinhados com a análise teórica apresentada anteriormente. Focaremos na avaliação de ambos os algoritmos utilizando grafos de diferentes tamanhos e compararemos o tempo de execução em função do tamanho do grafo. Para esta análise, utilizaremos listas de adjacência como estrutura de dados que armazena o grafo. A análise detalhada dessa estrutura será abordada na seção seguinte. Além disso, não nos concentraremos no tipo de resposta fornecido pelos algoritmos, o qual será explorado na próxima seção.

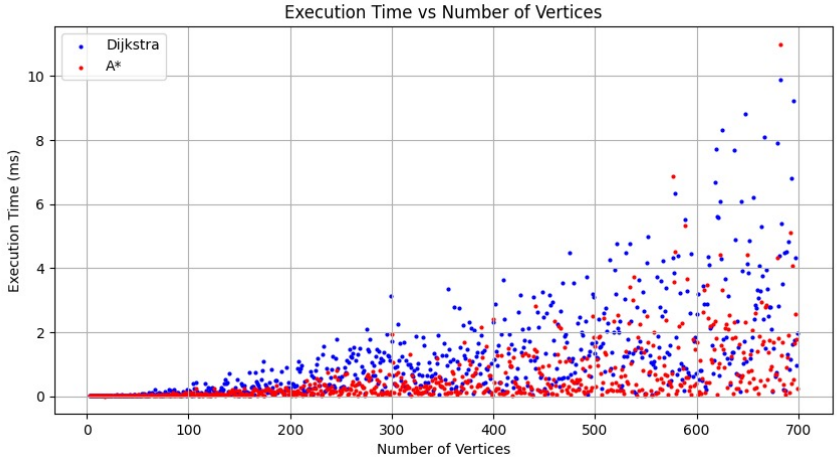

Figure 1: A star x Dijkstra - Tempo de execução x Quantidade de Vértices

Os gráficos mostram que a análise computacional está de acordo com a análise teórica. Vemos uma curva tanto do A* quanto do Dijkstra que aumenta à medida que o número de vértices cresce. Assim, observamos que o tempo cresce conforme o tamanho da entrada aumenta, ficando limitado pela complexidade assintótica analisada de $O(V \cdot (k+1))$.

No entanto, percebemos que em grafos com densidade média, que é o caso desse experimento, definidos como foi explicitado anteriormente, o algoritmo A* é significativamente mais rápido devido à heurística que guia a busca por caminhos. Entretanto, aqui não estamos avaliando a validade dos caminhos encontrados, apenas o tempo que cada algoritmo leva para concluir sua execução. Esse tópico será abordado mais adiante.

Matriz de Adjacência X Lista de Adjacência

Impacto de Cada Tipo de Estrutura no Desempenho do Programa

Neste tópico, buscamos analisar como a utilização de listas de adjacência ou matrizes de adjacência impacta o desempenho do programa. Conforme discutido nas seções de método e análise teórica, a matriz de adjacência ocupa significativamente mais espaço (ordem de $O(V^2)$) em comparação com a lista de adjacência (ordem de $O(V+E)$). Portanto, o objetivo é comparar em quais cenários cada estrutura oferece vantagens distintas. Realizaremos testes com grafos que apresentam representações mais esparsas e grafos mais densos, para avaliar como cada estrutura lida com esses cenários distintos e qual estrutura se revela mais eficiente em termos de desempenho e uso de espaço.

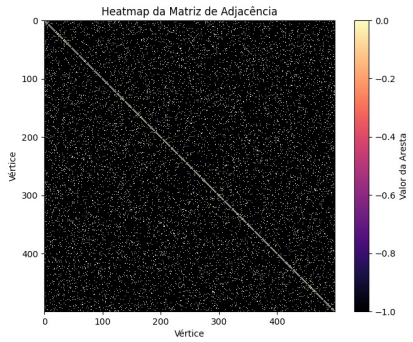

Figure 2: Heatmap - Grafo Densidade 5%

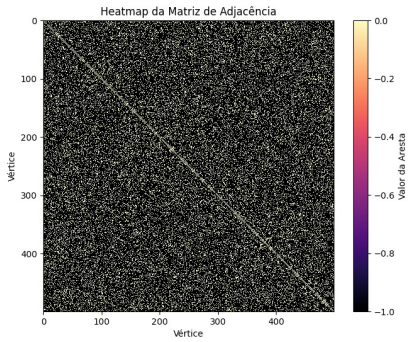

Figure 3: Heatmap - Grafo Densidade 20%

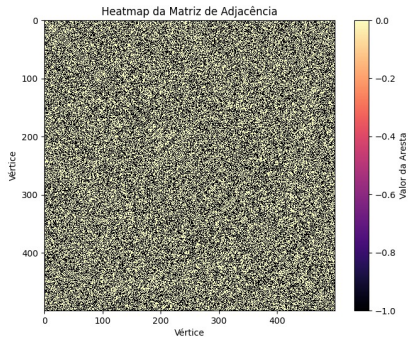

Figure 4: Heatmap - Grafo Densidade 50%

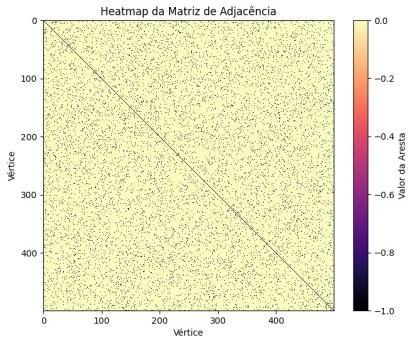

Figure 5: Heatmap - Grafo Densidade 95%

Os gráficos a seguir mostram como o espaço de uma matriz que armazena um grafo de tamanho 500 é preenchido dependendo da densidade de arestas desse grafo. É visível o desperdício de memória ao armazenar valores nulos quando a matriz é esparsa, e à medida que a matriz fica mais densa, começa a valer a pena usar esse tipo de representação. Isso ocorre principalmente porque o acesso a qualquer item da matriz é $O(1)$ e encontrar os vizinhos é $O(n)$, enquanto na lista de adjacência encontrar os vizinhos é $O(d)$, onde d é o grau do vértice. À medida que a matriz fica mais densa, o grau do vértice se aproxima de n, logo essa operação tem complexidade similar em ambas as representações. Abaixo vemos uma distribuição da quantidade de bytes de memória utilizados e desperdiçados em cada configuração de densidade da matriz. Além disso, temos também o gráfico de tempo de execução por número de vértices, comparando a eficiência de ambas as representações no algoritmo em si. Observamos que, de maneira geral, o uso adicional de memória torna o uso da matriz de adjacência mais lenta para matrizes de densidade média.

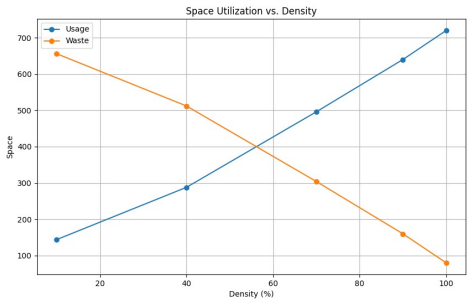

Figure 6: Uso de memória - Uso x Desperdício

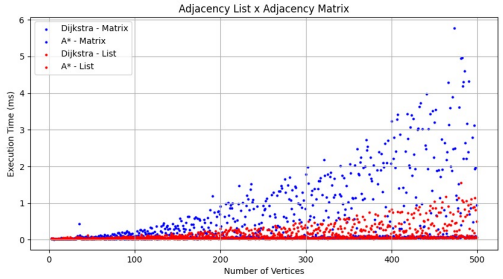

Figure 7: Impactos - Matriz de Adjacência x Lista de Adjacência

Qualidade das Soluções Obtidas

Os Algoritmos Sempre Encontram as Mesmas Soluções?
Como explorado na seção "Método", é sabido que a heurística do algoritmo A* influencia os caminhos que o algoritmo segue. Consequentemente, o A* pode encontrar soluções diferentes em comparação com o algoritmo de Dijkstra, e nem sempre garantir um caminho. Neste tópico, vamos analisar como a qualidade das soluções pode variar. Investigaremos se, ao alterar ou restringir parâmetros como o número de portais (arestas com peso zero), as respostas dos algoritmos mudam e discutiremos as razões por trás dessas variações.

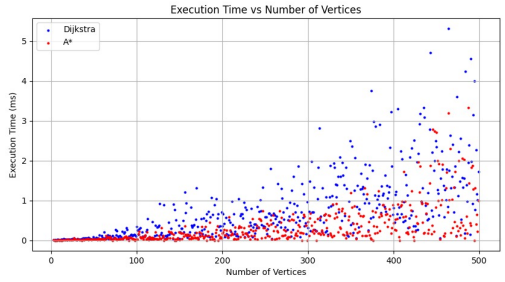

Figure 8: Tempo de Execução x Número de Vértices - Grafo com Portais

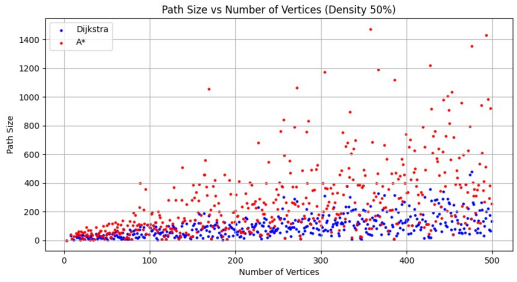

Figure 9: Tamanho do Caminho Encontrado x Número de Vértices - Grafo com Portais

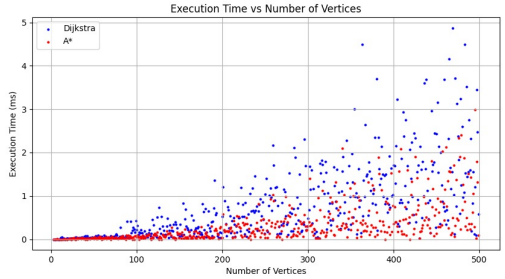

Figure 10: Tempo de Execução x Número de Vértices - Grafo sem Portais

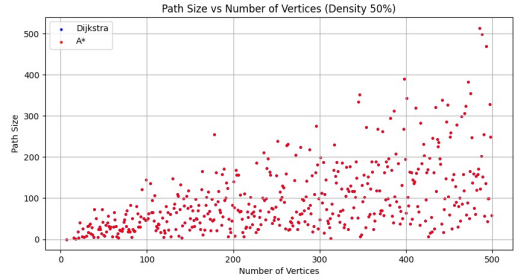

Figure 11: Tamanho do Caminho Encontrado x Número de Vértices - Grafo sem Portais

Nos gráficos acima, apresentamos duas configurações de parâmetros para o nosso problema: uma com o uso de portais e outra sem utilizar portais. Com isso, podemos verificar o que já suspeitávamos na parte teórica sobre o uso da heurística.

Os gráficos superiores mostram a relação entre o tempo de execução (independente da qualidade da resposta) e o tamanho do caminho encontrado em relação ao número de vértices quando há portais no grafo. Observamos que o algoritmo A* muitas vezes encontra caminhos significativamente mais longos do que o Dijkstra. Isso corrobora o que foi discutido na seção "Impacto da Heurística", onde destacamos que os portais, por quebrarem a desigualdade triangular, fazem com que o A* não encontre caminhos ótimos ou, até mesmo, não encontre caminhos quando há um caminho possível. Por outro lado, o Dijkstra sempre encontra o caminho ótimo, se ele existir.

Nos dois gráficos abaixo, foi realizada uma restrição em um dos parâmetros: o número de portais foi definido como 0. Nesse cenário, vemos que o Dijkstra continua demorando mais do que o A*, mas ambos retornam sempre a mesma resposta e o mesmo caminho. Portanto, a eficiência do A* se mostra útil neste contexto, pois não só é eficiente, como também é confiável em termos de resposta.

Finalmente, o gráfico abaixo mostra a distribuição das diferentes respostas encontradas pelos algoritmos, evidenciando como o A* encontra menos caminhos do que o Dijkstra.

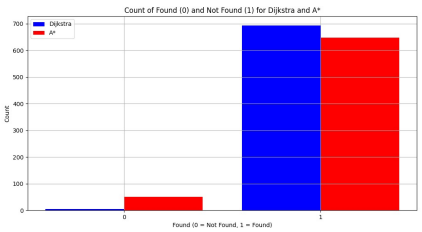

Figure 12: Qualidade da Resposta - Sucesso em Achar o Caminho

5 Conclusões

Com este trabalho, foi possível explorar as nuances envolvidas na resolução de problemas de menor caminho em grafos, levando em consideração a implementação e as estruturas de dados utilizadas. Através das análises realizadas, foram obtidos importantes insights que podem ser aplicados a uma variedade de problemas envolvendo estruturas de dados.

Primeiramente, ficou evidente a necessidade de compreender o contexto específico do problema e as estruturas de dados adequadas para modelá-lo. Cada estrutura de dados possui características que podem impactar diretamente a eficiência da solução. Por exemplo, o conhecimento sobre matrizes e listas de adjacência ajudou a determinar a melhor forma de armazenar o grafo para o problema em questão, onde, de maneira geral, a lista de adjacência mostrou-se mais adequada.

Além disso, é crucial entender a natureza dos algoritmos utilizados. Por exemplo, o algoritmo A*, embora eficiente devido à sua heurística, pode não encontrar caminhos ótimos em grafos com arestas de custo zero, o que pode prejudicar seu desempenho comparado ao algoritmo de Dijkstra. Essa análise destaca a importância de selecionar o algoritmo mais apropriado com base nas características específicas do grafo e do problema.

Portanto, as conclusões deste trabalho reforçam a importância de uma escolha cuidadosa das estruturas de dados e algoritmos, levando em consideração as particularidades do problema para alcançar soluções eficazes e eficientes. Esses aprendizados são fundamentais para a aplicação em problemas futuros e para a melhoria contínua na área de computação.

References

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, *Introduction to Algorithms*, 3ª edição, MIT Press, 2009.

[2] N. Ziviani, *Projeto de Algoritmos com Implementações em Pascal e C*, 3ª edição, Cengage Learning, 2011.