

**Trabalho Prático 1 (TP1)**

Algoritmos 1

**Ítalo Leal Lana Santos | Matrícula: 2024013893 | italolealanasantos@ufmg.br**

Universidade Federal de Minas Gerais (UFMG) | Belo Horizonte/MG | outubro de 2025

1. **Introdução**

O problema computacional proposto consiste em analisar a malha viária da cidade de Somatório para auxiliar a prefeita Maria Quadras em um projeto de urbanização sustentável. A cidade possui N regiões conectadas por M ruas bidirecionais, cada uma com uma distância associada. O objetivo é identificar quais ruas podem ser removidas para dar lugar a áreas verdes sem prejudicar o trânsito da cidade.

Especificamente, o problema requer:

1. Calcular a **distância mínima** entre a praça central (região 1) e o parque ecológico (região N);
2. Identificar todas as ruas que participam de pelo menos uma **rota mínima** entre esses dois pontos;
3. Descobrir as **ruas críticas**, cuja remoção aumentaria a distância mínima ou tornaria o caminho impossível.

Este é um problema clássico de teoria dos grafos aplicado ao planejamento urbano, envolvendo busca de caminhos mínimos e análise de conectividade. Este relatório apresenta a modelagem, solução e análise de complexidade desenvolvidas para resolver o problema proposto.

1. **Modelagem**

**2.1 Estrutura de Dados**

O problema foi modelado utilizando um **grafo bi-direcionado (ou não direcionado) e ponderado (com pesos estritamente positivos)**, onde:

* **Vértices**: representam as N regiões da cidade (numeradas de 1 a N)
* **Arestas**: representam as M ruas bidirecionais, cada uma com um peso (distância)

A implementação utiliza a class Graph com os seguintes atributos:

- vertex\_count: int *# Número total de regiões (N)*

- edge\_count: int *# Número total de ruas (M)*

- graph: dict[int, list] *# Lista de adjacências*

- edges: list[tuple] *# Lista de arestas com metadados*

- minimal\_edges: list[tuple] *# Cache de arestas em caminhos mínimos*

**2.2 Representação do Grafo**

**Lista de Adjacências**: O grafo é representado por um dicionário onde cada vértice aponta para uma lista de tuplas (vizinho, distância). Esta escolha foi feita por ser eficiente para o algoritmo de Dijkstra, que precisa acessar frequentemente os vizinhos de cada vértice.

graph = {

1: [(2, 5), (3, 10)],

2: [(1, 5), (3, 6) ],

3: [(1, 10), (2, 6) ] }

**Lista de Arestas**: Além da lista de adjacências, mantém-se uma lista separada contendo todas as arestas com seus metadados completos: (índice, vértice\_origem, vértice\_destino, comprimento). Isso facilita a identificação das arestas por índice nas Partes 2 e 3.

1. **Solução**

**3.1 Parte 1: Distância Mínima**

**Algoritmo:** Dijkstra **Função:** get\_minimal\_distance(start, end)

**Ideia Geral:** O algoritmo de Dijkstra encontra o caminho mais curto entre dois vértices em um grafo com pesos não negativos. Ele funciona mantendo um conjunto de vértices cuja distância mínima desde o vértice inicial já foi determinada, expandindo gradualmente este conjunto.

**Funcionamento do Algoritmo:** O algoritmo inicializa um vetor de distâncias com infinito para todos os vértices, exceto o inicial que recebe zero. Utiliza-se uma fila de prioridade (heap) para sempre processar o vértice com menor distância acumulada. A cada iteração, remove-se o vértice de menor distância do heap e relaxa-se todas as suas arestas, ou seja, verifica-se se passar por este vértice oferece um caminho mais curto para seus vizinhos. Se sim, atualiza-se a distância do vizinho e insere-se no heap. O processo continua até que o heap esteja vazio. Estados desatualizados no heap são ignorados comparando a distância extraída com a distância atual do vértice.

**Implementação:** Utilizei o módulo heapq da biblioteca padrão do Python, que implementa um heap binário eficiente. A função retorna a distância mínima do vértice 1 até o vértice N.

**3.2 Parte 2: Arestas em Caminhos Mínimos**

**Algoritmo**: Dijkstra bilateral com verificação  
**Função**: find\_minimal\_edges(start, end)

**Ideia Geral**: Para determinar se uma aresta (u, v) pertence a algum caminho mínimo entre os vértices 1 e N, verificamos se ela "conecta" corretamente as distâncias mínimas.

Executamos o algoritmo de Dijkstra duas vezes: A partir do vértice 1 (obtém **dist\_from\_start**) e a partir do vértice N (obtém **dist\_from\_end**).

Uma aresta (u, v) com comprimento L pertence a um caminho mínimo se e somente se:

**dist\_from\_start[u] + L + dist\_from\_end[v] = dist\_mínima\_total** OU

**dist\_from\_start[v] + L + dist\_from\_end[u] = dist\_mínima\_total**

**Explicação da Condição**:

* **dist\_from\_start[u]**: menor distância de 1 até u
* **L**: comprimento da aresta (u, v)
* **dist\_from\_end[v]:** menor distância de v até N
* Se a soma for igual à distância mínima total, então usar a aresta (u, v) nesta direção mantém o caminho ótimo

**Funcionamento do Algoritmo**: A solução executa o algoritmo de Dijkstra duas vezes: primeiro a partir do vértice 1, obtendo as distâncias mínimas de 1 para todos os vértices; depois a partir do vértice N, obtendo as distâncias mínimas de todos os vértices até N. Com essas informações, percorre-se todas as arestas do grafo original verificando, para cada aresta, se ela satisfaz a condição descrita acima em pelo menos uma direção. Como o grafo é direcionado, testa-se ambas as direções da aresta. As arestas que satisfazem a condição são armazenadas em uma lista e retornam ordenadas pelos índices.

**3.3 Parte 3: Arestas Críticas**

**Algoritmo**: Teste de conectividade com DFS  
**Função**: find\_critical\_edges(start, end)

**Ideia Geral**: Uma aresta é crítica se sua remoção aumenta a distância mínima ou desconecta o caminho entre 1 e N. Para otimizar, testamos apenas as arestas que pertencem a caminhos mínimos (obtidas na Parte 2), pois apenas essas podem ser críticas.

A estratégia implementada foi criar um **grafo auxiliar** contendo apenas as arestas mínimas, e para cada aresta deste grafo: Remover temporariamente a aresta, executar DFS a partir do vértice 1, verificar se o vértice N é alcançável, se não for, a aresta é crítica, e no final, restaurar a aresta novamente.

**Otimização**: Em vez de recalcular Dijkstra (que é O((V+E) log V)), uso DFS (que é O(V+E)) apenas para verificar conectividade no grafo reduzido. Como o grafo auxiliar contém apenas arestas mínimas, ele é tipicamente muito menor que o grafo original.

**Funcionamento do Algoritmo**: A solução primeiro verifica se as arestas mínimas já foram calculadas (Parte 2); caso contrário, executa a função correspondente. Em seguida, cria-se um grafo auxiliar contendo apenas as arestas que participam de caminhos mínimos. Para cada aresta deste grafo reduzido, remove-se temporariamente a aresta das listas de adjacência de ambos os vértices que ela conecta.

Realiza-se então uma busca em profundidade (DFS) a partir do vértice 1 para identificar todos os vértices alcançáveis sem aquela aresta. Se o vértice N não estiver no conjunto de vértices alcançáveis, significa que a aresta removida é crítica e seu índice é adicionado à lista de resultados. Após a verificação, a aresta é restaurada ao grafo auxiliar. O DFS utiliza uma pilha para explorar o grafo, marcando vértices como visitados e expandindo para seus vizinhos não visitados. A função retorna a lista de índices das arestas críticas em ordem crescente.

1. **Análise de Complexidade**

**4.1 Complexidade de Tempo**

|  |  |  |  |
| --- | --- | --- | --- |
| **Parte** | **Operação** | **Complexidade** | **Justificativa** |
| 1 | Dijkstra | O((V + E) log V) | Uma execução com heap binário |
| 2 | 2 \* Dijkstra | O((V + E) log V) | Dois Dijkstra + O(E) para verificar todas as arestas |
| 3 | K \* DFS | O(K × (V + E)) | K remoções, cada uma seguida de DFS |

**Dijkstra com heap binário:**Cada vértice é inserido/removido do heap no máximo uma vez:   
O(V log V), cada aresta é relaxada no máximo uma vez: O(E log V).   
Total: O((V + E) log V)

**Parte 3 - Otimização:**Número de arestas candidatas K ≤ E (geralmente K << E).   
Cada DFS no grafo reduzido: O(V + K).   
Complexidade total: O(K × (V + K)).   
No pior caso onde K = E: O(E × (V + E))

**Complexidade Total do Programa:** O(E × (V + E)).

*Obs: Embora seja O(E²) no pior caso, as otimizações tornam essa implementação eficiente para valores pequenos de K (grafos esparsos, com poucos caminhos mínimos)*

**4.2 Complexidade de Espaço**

|  |  |  |
| --- | --- | --- |
| **Estrutura** | **Complexidade** | **Descrição** |
| Lista de adjacências | O(V + E) | Armazena o grafo principal |
| Heap (Dijkstra) | O(V) | Fila de prioridade |
| Vetores auxiliares | O(V) | Distâncias e visitados |
| Lista de arestas | O(E) | Metadados completos |
| Grafo auxiliar (Parte 3) | O(V + K) | K ≤ E arestas mínimas |

**Complexidade Total de Espaço**: **O(V + E)**

O uso de memória é dominado pela representação do grafo em lista de adjacências e pela lista de arestas com seus metadados.

1. **Considerações Finais**

**5.1 Experiência de Desenvolvimento**

A **Parte 1** foi direta, utilizando o algoritmo de Dijkstra bem estabelecido. A **Parte 2** exigiu raciocínio sobre a condição de verificação usando Dijkstra bilateral, que se mostrou elegante e eficiente. A **Parte 3** foi a mais desafiadora: inicialmente recalculava Dijkstra para cada aresta, mas a otimização usando DFS em um grafo reduzido (apenas arestas mínimas) melhorou significativamente a performance.

**5.2 Decisões de Implementação**

Escolhi Python pela claridade e o módulo heapq para o Dijkstra. Implementei cache de arestas mínimas para evitar recálculos e criei um grafo auxiliar na Parte 3 para trabalhar com estruturas menores. A lista de adjacências foi escolhida por ser eficiente para acessar vizinhos frequentemente.

**5.3 Limitações**

A Parte 3 pode ser lenta para grafos muito densos com muitas arestas em caminhos mínimos. Possíveis melhorias incluiriam Dijkstra bidirecional ou algoritmos específicos para pontes em grafos.

1. **Referências**

LEVITIN, Anany. **Introduction to the Design and Analysis of Algorithms**. 3rd ed. Pearson, 2012.

KLEINBERG, Jon; TARDOS, Éva. **Algorithm Design**. Pearson, 2006.

CORMEN, Thomas H. et al. **Introduction to Algorithms**. 3rd ed. MIT Press, 2009.

Python Software Foundation. Python 3.9+ Documentation - heapq module. Disponível em: <https://docs.python.org/3/library/heapq.html>

Material didático da disciplina **Algoritmos I** - UFMG/DCC, 2025. Slides de aula sobre Grafos e Algoritmo de Dijkstra.