

**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 proposto consiste em analisar a infraestrutura viária da cidade de Somatório, modelada como um grafo não direcionado e ponderado, onde os vértices representam regiões da cidade e as arestas representam ruas com seus respectivos comprimentos. O objetivo é responder a três perguntas:

**Parte 1:** Encontrar a distância mínima entre a praça central (vértice 1) e o parque ecológico (vértice N).

**Parte 2:** Identificar todas as ruas que fazem parte de pelo menos um caminho mínimo entre o vértice 1 e o vértice N.

**Parte 3:** Identificar as ruas críticas, ou seja, aquelas que, se removidas, causam um aumento na distância mínima entre 1 e N.

1. **Modelagem**

O problema foi modelado usando um grafo não direcionado e ponderado. Cada rua é uma aresta com um comprimento (peso) e conecta duas regiões (vértices). A solução foi implementada em Python, utilizando as seguintes estruturas de dados e algoritmos:

* **Grafo**: Representado por um dicionário de listas de adjacências, onde cada vértice aponta para uma lista de tuplas (vértice vizinho, peso da aresta). Além disso, mantém uma lista de arestas com seus índices, vértices de origem e destino e comprimento.
* **Algoritmo de Dijkstra**: Utilizado para calcular a distância mínima de um vértice inicial para todos os outros. Esse algoritmo é eficiente para grafos com pesos não negativos.
* **Identificação de arestas em caminhos mínimos**: Para cada aresta, verifica-se se ela pertence a algum caminho mínimo entre 1 e N. Isso é feito comparando a distância mínima de 1 até N com a soma das distâncias mínimas de 1 até um dos vértices da aresta, o comprimento da aresta e a distância mínima do outro vértice até N.
* **Identificação de arestas críticas**: Para cada aresta que pertence a um caminho mínimo, remove-se temporariamente a aresta e recalcula-se a distância mínima entre 1 e N. Se a distância aumentar, a aresta é considerada crítica.

1. **Solução**

**3.1 Parte 1: Distância Mínima**

A distância mínima entre o vértice 1 e o vértice N é calculada usando o algoritmo de Dijkstra a partir do vértice 1. O algoritmo mantém um heap (fila de prioridade) para selecionar o próximo vértice a ser visitado, garantindo eficiência. A distância para o vértice N é então retornada.

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

Para identificar as arestas que participam de pelo menos um caminho mínimo, utilizamos duas execuções do algoritmo de Dijkstra: uma a partir do vértice 1 e outra a partir do vértice N. Com isso, obtemos as distâncias mínimas de 1 para todos os vértices e de N para todos os vértices. Em seguida, para cada aresta (u, v) com comprimento L, verificamos se:

* A distância de 1 até u, somada com L e com a distância de v até N, é igual à distância mínima de 1 até N, ou
* A distância de 1 até v, somada com L e com a distância de u até N, é igual à distância mínima de 1 até N.

Se uma dessas condições for verdadeira, a aresta pertence a pelo menos um caminho mínimo.

**3.3 Parte 3: Arestas Críticas**

Para cada aresta identificada na Parte 2, removemos temporariamente a aresta do grafo e recalculamos a distância mínima entre 1 e N. Se a nova distância for maior que a original, a aresta é considerada crítica. A remoção é feita retirando a aresta da lista de adjacências de ambos os vértices e, após o cálculo, a aresta é readicionada.

1. **Análise de Complexidade**

**4.1 Complexidade de Tempo**

* **Algoritmo de Dijkstra**: Utilizando um heap binário, a complexidade é O((V + E) log V), onde V é o número de vértices e E o número de arestas.
* **Parte 1**: Uma execução de Dijkstra: O((V + E) log V).
* **Parte 2**: Duas execuções de Dijkstra (uma de 1 e uma de N) e depois percorre todas as arestas E. Portanto, O(2\*(V+E) log V + E) = O((V+E) log V).
* **Parte 3**: Para cada aresta na lista de arestas mínimas (que pode ser O(E)), remove a aresta, executa Dijkstra (O((V+E) log V)) e readiciona a aresta. No pior caso, a lista de arestas mínimas tem O(E) arestas, resultando em O(E \* (V+E) log V). No entanto, note que a lista de arestas mínimas é um subconjunto de E, mas no pior caso pode ser O(E). Isso pode ser muito custoso para grafos grandes.

**4.2 Complexidade de Espaço**

* O grafo é armazenado em listas de adjacências, ocupando O(V + E).
* As estruturas auxiliares do Dijkstra (distância, heap) usam O(V).
* A lista de arestas mínimas e críticas usa O(E).

Portanto, a complexidade de espaço total é O(V + E).

1. **Considerações Finais**

A implementação do trabalho foi desafiadora, especialmente a Parte 3, devido à necessidade de remover e readicionar arestas e recalcularem-se as distâncias mínimas para cada aresta candidata. Isso torna a solução da Parte 3 computacionalmente pesada para instâncias grandes.

As Partes 1 e 2 foram mais diretas, pois envolvem algoritmos clássicos e bem conhecidos. A modelagem do grafo foi eficiente, utilizando estruturas de dados apropriadas.

Uma possível melhoria para a Parte 3 seria buscar um algoritmo mais eficiente para identificar arestas críticas, talvez usando fluxo em redes ou algoritmos específicos para pontes em grafos ponderados, mas isso ia além do escopo deste trabalho.

1. **Referências**

Cormen, T. H., et al. "Introduction to Algorithms" - Capítulo sobre algoritmos em grafos

Documentação oficial Python 3.9+ - Módulos heapq e estruturas de dados

Material da disciplina Algoritmos I - Notas de aula sobre Dijkstra e grafos

**Documentação - Trabalho Prático 01**

**1. Introdução**

Este trabalho visa resolver um problema de otimização de rotas em uma malha viária urbana, onde precisamos analisar as conexões entre diferentes regiões de uma cidade chamada Somatório. O problema consiste em três partes principais: determinar a distância mínima entre dois pontos estratégicos da cidade (praça central e parque ecológico), identificar todas as ruas que participam de rotas mínimas entre esses pontos, e encontrar as ruas críticas que não podem ser removidas sem prejudicar o tráfego da cidade.

A prefeitura deseja liberar terrenos para áreas verdes removendo ruas pouco utilizadas, mas precisa garantir que as rotas principais permaneçam eficientes. Este é um problema clássico de teoria dos grafos aplicado ao planejamento urbano.

**2. Modelagem**

**2.1 Estrutura de Dados**

O problema foi modelado usando um **grafo não direcionado e ponderado**:

* **Vértices**: Representam as regiões da cidade (numeradas de 1 a N)
* **Arestas**: Representam as ruas, cada uma com um comprimento (peso)
* **Grafo**: Implementado como lista de adjacências para eficiência em consultas de vizinhança

**2.2 Classes e Estruturas**

A classe Graph foi implementada com os seguintes componentes:

* vertex\_count: Número total de regiões (vértices)
* edge\_count: Contador de ruas (arestas) adicionadas
* graph: Dicionário que mapeia cada vértice para sua lista de vizinhos e distâncias
* edges: Lista contendo todas as arestas com seus metadados completos

**3. Solução**

**3.1 Parte 1: Distância Mínima**

**Algoritmo**: Dijkstra

**Ideia Geral**:

* Encontrar o caminho mais curto entre a região 1 (praça) e região N (parque)
* Utiliza uma fila de prioridade (heap) para processar vértices na ordem de menor distância
* Mantém um vetor de distâncias mínimas desde o vértice inicial
* Complexidade: O((V + E) log V)

**Pseudocódigo**:

text

Copy

Download

função dijkstra(vertice\_inicial):

inicializar heap com (0, vertice\_inicial)

distancias[vertice\_inicial] = 0

enquanto heap não vazio:

remover vértice u com menor distância

para cada vizinho v de u:

nova\_distancia = distancia[u] + peso(u,v)

se nova\_distancia < distancia[v]:

atualizar distancia[v]

adicionar (distancia[v], v) no heap

**3.2 Parte 2: Ruas em Rotas Mínimas**

**Algoritmo**: Verificação baseada em Dijkstra bilateral

**Ideia Geral**:

* Executar Dijkstra a partir do vértice 1 para obter distâncias mínimas de partida
* Executar Dijkstra a partir do vértice N para obter distâncias mínimas de chegada
* Para cada aresta (u, v), verificar se ela pertence a algum caminho mínimo usando a condição:  
  distancia\_de\_1[u] + peso + distancia\_ate\_N[v] == distancia\_minima\_total OU  
  distancia\_de\_1[v] + peso + distancia\_ate\_N[u] == distancia\_minima\_total

**Complexidade**: O((V + E) log V) para os dois Dijkstra + O(E) para verificação

**3.3 Parte 3: Ruas Críticas**

**Algoritmo**: Remoção e teste incremental

**Ideia Geral**:

* Para cada aresta identificada na Parte 2 (que participa de rotas mínimas):
  + Remover temporariamente a aresta do grafo
  + Recalcular a distância mínima entre 1 e N
  + Se a nova distância for maior que a original, a aresta é crítica
  + Restaurar a aresta no grafo

**Complexidade**: O(K × (V + E) log V), onde K é o número de arestas candidatas

**4. Análise de Complexidade**

**4.1 Complexidade de Tempo**

| Parte | Complexidade | Justificativa |
| --- | --- | --- |
| Parte 1 | O((V + E) log V) | Algoritmo de Dijkstra com heap |
| Parte 2 | O((V + E) log V) | Dois Dijkstra + verificação linear O(E) |
| Parte 3 | O(K × (V + E) log V) | K execuções do Dijkstra (K ≤ E) |

Onde:

* V = número de vértices (regiões)
* E = número de arestas (ruas)
* K = número de arestas candidatas da Parte 2

**4.2 Complexidade de Espaço**

| Estrutura | Complexidade | Descrição |
| --- | --- | --- |
| Grafo (lista adj.) | O(V + E) | Armazenamento das conexões |
| Heap | O(V) | Fila de prioridade do Dijkstra |
| Vetores auxiliares | O(V) | Distâncias e pais |
| Lista de arestas | O(E) | Metadados das ruas |

**Complexidade total de espaço**: O(V + E)

**5. Considerações Finais**

**5.1 Desafios Encontrados**

**Parte 2**: A identificação de arestas em caminhos mínimos exigiu cuidado especial. Inicialmente considerei verificar todos os caminhos mínimos possíveis, mas isso seria computacionalmente inviável. A solução usando Dijkstra bilateral mostrou-se eficiente e correta.

**Parte 3**: A implementação de remoção temporária de arestas precisou garantir que o grafo fosse restaurado corretamente após cada teste. Também foi necessário otimizar para testar apenas as arestas candidatas da Parte 2, reduzindo significativamente o número de execuções do Dijkstra.

**5.2 Decisões de Implementação**

1. **Heap para Dijkstra**: Escolhi usar heapq do Python por ser eficiente e fazer parte da biblioteca padrão permitida.
2. **Representação do Grafo**: Lista de adjacências foi escolhida por ser mais eficiente para o algoritmo de Dijkstra, que precisa acessar frequentemente os vizinhos de cada vértice.
3. **Armazenamento de Arestas**: Mantive uma lista separada com metadados completos para facilitar a identificação por índice nas Partes 2 e 3.

**5.3 Limitações e Melhorias**

* A Parte 3 pode se tornar lenta para grafos muito densos com muitas arestas em caminhos mínimos
* Para grafos extremamente grandes, poderiam ser exploradas técnicas de otimização como:
  + Bidirecional Dijkstra para a Parte 1
  + Armazenamento de caminhos mínimos alternativos

**5.4 Aprendizados**

Este trabalho reforçou a importância de:

* Escolher estruturas de dados adequadas ao problema
* Compreender profundamente os algoritmos antes da implementação
* Testar com casos extremos (grafos esparsos, densos, com muitas alternativas)

**6. Referências**

1. Cormen, T. H., et al. "Introduction to Algorithms" - Capítulo sobre algoritmos em grafos
2. Documentação oficial Python 3.9+ - Módulos heapq e estruturas de dados
3. Material da disciplina Algoritmos I - Notas de aula sobre Dijkstra e grafos