# Relatório de Implementação e Estudo de Casos – Parte 2
# Trabalho de Grafos

Aluno: Pedro Tuttman Cysne Soares
DRE: 124023584

Universidade Federal do Rio de Janeiro
Engenharia de Computação e Informação
2025

## 1 Introdução

Na primeira parte deste trabalho, a implementação foi conduzida de forma mais direta, utilizando uma única classe de grafos capaz de representar tanto a estrutura por vetor de adjacência quanto por matriz, e os algoritmos de busca (BFS, DFS) e de menor caminho (Dijkstra) eram definidos em funções separadas, mas com lógica semelhante. Essa abordagem exigia duplicação parcial de código, além de dificultar a expansão e a reutilização.

A segunda parte propôs uma reformulação orientada a objetos, baseada em **classes abstratas e herança**. Essa modificação visou garantir uma única implementação dos algoritmos principais, permitindo que apenas as partes específicas das representações ou estruturas de dados fossem sobrescritas. Isso promoveu um código mais limpo, modular e extensível.

## 2 Estrutura e Esqueleto das Classes

A seguir, é apresentada uma descrição mais detalhada das classes e da lógica de herança adotada.

### 2.1 Classe Base: `Graph`

A classe `Graph` é uma classe abstrata que define a interface comum para todas as representações de grafos. Ela fornece a estrutura e os métodos que serão herdados pelas classes derivadas. Os métodos principais são:

- `addEdge(u, v, w)`: adiciona uma aresta de peso $w$ entre os vértices $u$ e $v$;
- `forEachAdj(u, f)`: percorre todos os vértices adjacentes de $u$, aplicando a função `f(v, w)`.

As classes derivadas são:

- `GraphVector`: implementa o grafo com vetor de adjacência (memória $O(n+m)$);
- `GraphMatrix`: implementa o grafo com matriz de adjacência (memória $O(n^2)$).

### 2.2 Classe Base: `Dijkstra`

A classe `Dijkstra` é também abstrata e define a estrutura geral do algoritmo, possuindo:

- Vetores `dist`, `pai` e `fechado`;
- Método principal `run(s)`, responsável pelo cálculo das distâncias a partir da origem $s$;
- Métodos abstratos `init(s)`, `extrair()` e `atualiza(v)`, que variam conforme a estrutura usada (vetor ou heap).

As classes concretas são:

- `DijkstraVector`: utiliza varredura linear para encontrar o vértice com menor distância (complexidade $O(n^2)$);
- `DijkstraHeap`: utiliza uma fila de prioridade (heap) para otimizar a extração do mínimo (complexidade $O((n+m)\log n)$).

## 3 Arquitetura do Código e Integração

### 3.1 Visão geral das classes e dependências

A Figura 1 ilustra, de forma compacta, como as classes se relacionam. As caixas marcadas como *abstratas* estão na hierarquia superior; as demais são especializações concretas.

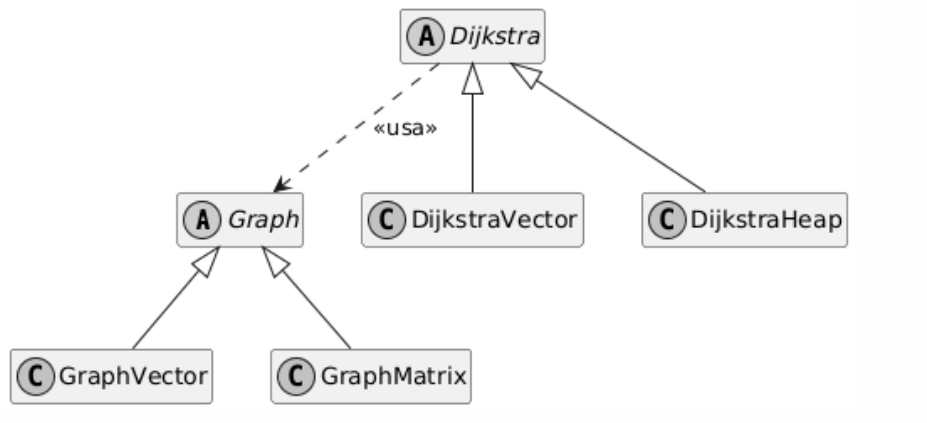


Figura 1: Relações de herança (setas contínuas) e dependência/uso (seta tracejada) entre as classes.

**Legenda do diagrama:**

- **Dijkstra (abstrata)**: define a estrutura do algoritmo (`run`, `init`, `extrair`, `atualiza`).

- **Graph (abstrata)**: define a interface do grafo (`forEachAdj`, `addEdge`).
- **DijkstraVector (concreta)**: extração do mínimo por varredura linear ($O(n^2)$).
- **DijkstraHeap (concreta)**: extração do mínimo via heap ($O((n+m)\log n)$).
- **GraphVector (concreta)**: representação por **vetor de adjacência**, memória $O(n+m)$.
- **GraphMatrix (concreta)**: representação por **matriz de adjacência**, memória $O(n^2)$.
- **Seta tracejada «usa»**: o algoritmo **Dijkstra** opera *sobre* uma instância de **Graph** passada em tempo de execução.

## 3.2 Fluxo de execução

1. **Entrada e flags** (`main.cpp`).
2. **Construção do grafo**: `GraphVector` (vetor de adjacência) ou `GraphMatrix`.
3. **Escolha do algoritmo**: `DijkstraVector` (varredura) ou `DijkstraHeap` (fila de prioridade).
4. **Execução**: `run(src)` calcula `dist`/`pai`; reconstrução do caminho `src`→`dst`.
5. **Benchmark opcional**: repete $K$ vezes e imprime tempo médio em ms.

## 3.3 Principais funções por classe

`GraphVector`

- `addEdge(u,v,w)`: insere $(v, w)$ em `adj[u]` (e $(u, w)$ se não-direcionado).
- `forEachAdj(u,f)`: itera pares $(v, w)$ de `adj[u]` e aplica `f(v,w)`.

`GraphMatrix`

- `addEdge(u,v,w)`: ajusta `mat[u][v]` (e `mat[v][u]` se não-direcionado).
- `forEachAdj(u,f)`: varre a linha $u$ ($O(n)$), chamando `f(v,mat[u][v])` quando houver aresta.

`Dijkstra` **(abstrata)**

- `run(s)`: laço único; extrai $u$, chama `G->forEachAdj(u, relax)` e delega `init`/`extrair`/`atualiza`
- `init(s)`, `extrair()`, `atualiza(v)`: pontos de variação implementados nas derivadas.

`DijkstraVector` `extrair()` por varredura linear.

`DijkstraHeap` `init(s)` com `push(0,s)`, `extrair()` consistente, `atualiza(v)` com `push(dist[v],v)`.

### 3.4 Responsabilidades por arquivo

`graph.cpp/graph.hpp` Definições e implementações das classes (abstratas e concretas); rotinas auxiliares (reconstrução de caminho, impressão de distância/caminho).

`main.cpp` Parsing das flags, construção do grafo, execução do menor caminho (imprime distância e caminho) e bloco de benchmark (média em ms).

### 3.5 Dificuldades e decisões

- **Memória em matriz** ($O(n^2)$): instâncias grandes causaram `std::bad_alloc`; decisão prática por vetor de adjacência para escalar.
- **Evitar duplicação de código**: centralizar algoritmos em classes abstratas com pontos de variação mínimos.
- **Medição de tempo**: média sobre $K$ execuções e "uso" do resultado para evitar eliminação por otimização.
- **Caminho mínimo**: reconstrução por vetor `pai` padronizada em todas as combinações (vetor/matriz × heap/vetor).

## 4 Flags de Execução

O programa inclui diversas *flags* para controle de entrada e experimentação:

- `--in=arquivo.txt`: define o arquivo de entrada contendo o grafo;
- `--rep=vector|matrix`: seleciona a representação do grafo (vetor ou matriz);
- `--src=X` e `--dst=Y`: determinam os vértices de origem e destino;
- `--bench-dijkstra-vec=K`: executa o Dijkstra com vetor $K$ vezes e calcula o tempo médio;
- `--bench-dijkstra-heap=K`: executa o Dijkstra com heap $K$ vezes e imprime o tempo médio.

## 5 Complexidade Computacional

As complexidades teóricas são as seguintes:

- **BFS/DFS (vetor)**: $O(n + m)$;
- **BFS/DFS (matriz)**: $O(n^2)$;
- **Dijkstra (vetor)**: $O(n^2)$;
- **Dijkstra (heap)**: $O((n + m) \log n)$.

O uso de matriz implica custo alto de memória ($O(n^2)$), explicando as falhas por `std::bad_alloc` nas instâncias maiores.

# 6 Resultados e Análise Experimental

Os experimentos foram realizados com base nos dados da planilha `Estudo_de_Casos_Tabelas_2.x` que apresenta tempos médios de execução (em milissegundos) para diferentes instâncias e configurações.

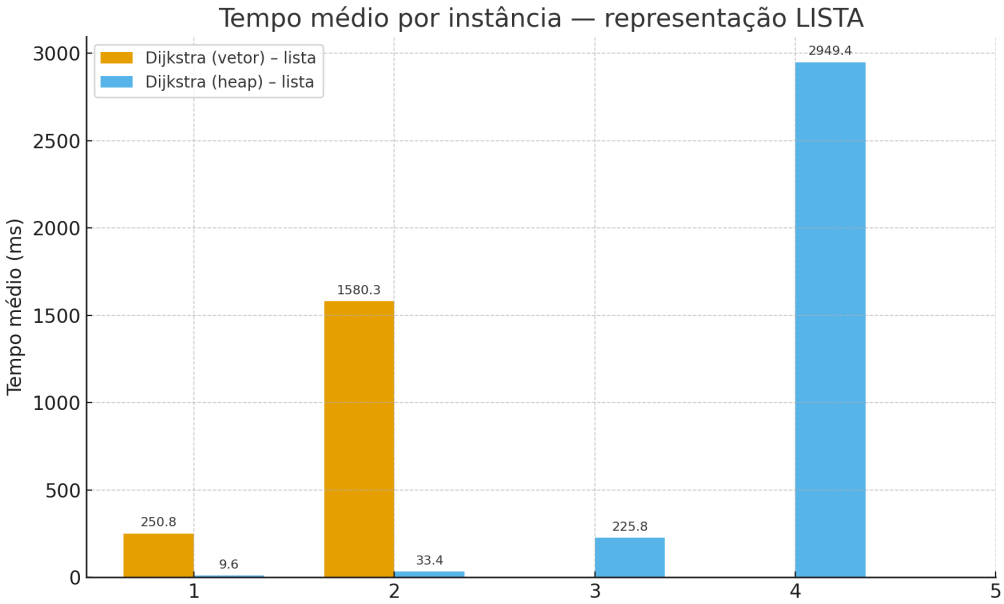


Figura 2: Tempo médio (ms) — Dijkstra por vetor vs heap (representação por vetor de adjacência).

## 6.1 Análise dos Resultados

Os resultados confirmam que o **Dijkstra com heap** supera amplamente o **Dijkstra com vetor**, especialmente em grafos grandes e esparsos. A aceleração observada cresce proporcionalmente ao número de vértices e arestas. Isso se deve à diferença estrutural das abordagens: enquanto o vetor varre todos os vértices a cada iteração ($O(n)$), o heap permite extração eficiente ($O(\log n)$), resultando em menores tempos totais.

O **erro** `std::bad_alloc` ocorre quando o sistema não consegue atender uma requisição de memória dinâmica. Na representação **matricial**, o consumo é $O(n^2)$. Em grandes instâncias, o espaço exigido ultrapassa a memória física disponível, causando falhas de alocação. Por exemplo, um grafo de 25 mil vértices demandaria cerca de 5 GB apenas para armazenar a matriz de pesos. Já a representação por vetor de adjacência, que cresce linearmente com o número de arestas ($O(n+m)$), é muito mais leve e escalável.

## 6.2 Erros recorrentes e contornos adotados

**(#1)** `std::bad_alloc` **nas instâncias grandes.** Falha de alocação devido ao custo $O(n^2)$ da matriz. Contorno: preferir **vetor de adjacência** em instâncias grandes e documentar a limitação da matriz.

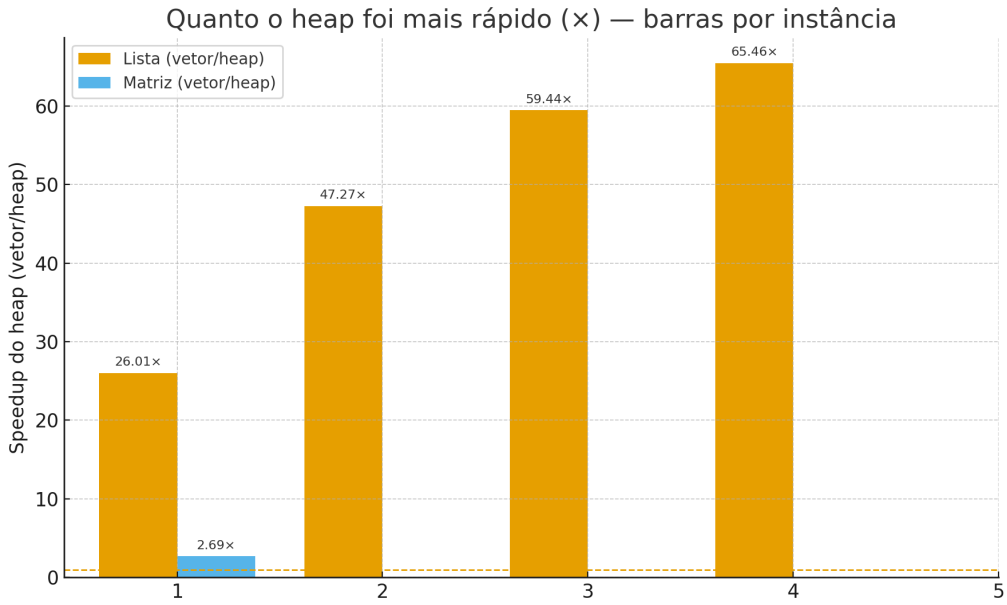


Figura 3: **Speedup do heap (*vetor/heap*)**. Cada barra mostra quantas vezes o *heap* foi mais rápido que o *vetor*. A linha tracejada em 1 indica empate; valores acima de 1 representam vantagem do *heap*.

**(#2) Tempo "rápido demais" por otimização do compilador.** Se o resultado do Dijkstra não é usado no benchmark, o compilador pode eliminar trabalho (*dead-code elimination*), gerando tempos artificiais. Contornos adotados:

- **Checksum**: somar valores de `dist` (ou `dist[dst]`) a uma variável e imprimir esse valor;
- **(Opcional)** `volatile`: em cenários extremos, acumular em uma variável `volatile`;
- **Cronometragem correta**: medir apenas a função do algoritmo; I/O fora; warm-up e **média** de $K$ repetições.

**Trecho resumido do benchmark (pseudocódigo).**

```
// Pseudocódigo do benchmark:
double soma_ms = 0.0, checksum = 0.0;
for (int r = 0; r < K; ++r) {
    int s = fontes[r];                // origem (fixa ou pseudo-aleatória)
    auto t0 = agora();
    auto dist = dijkstra.run(s);      // mede só o algoritmo
    auto t1 = agora();
    soma_ms += ms(t1 - t0);

    checksum += dist[dst_fixo];       // usa o resultado para evitar otimização
}
double media_ms = soma_ms / K;
imprime("tempo_medio_ms=", media_ms, " checksum=", checksum);
```

# 7 Conclusão

A segunda parte do projeto consolidou os conceitos de orientação a objetos e modularização. A utilização de **classes abstratas** e **herança** eliminou redundâncias e favoreceu a clareza estrutural. Além disso, a análise experimental evidenciou o ganho real do Dijkstra com heap, validando a teoria.

A representação por vetor de adjacência mostrou-se a mais eficiente e escalável, enquanto a matriz, embora simples, não é viável em grandes volumes de dados devido ao consumo de memória. O erro `std::bad_alloc` confirmou essa limitação física.

Todos os códigos e scripts desenvolvidos estão disponíveis publicamente em meu repositório do GitHub:
`https://github.com/pedrotuttman/Grafos-Parte2`

Assim, este trabalho não apenas demonstra a aplicação prática dos conceitos estudados em sala, mas também reforça a importância de boas práticas de engenharia de software na implementação de algoritmos eficientes e sustentáveis.