Relatório da Atividade A1 - Grafos
INE5413 - Grafos

André Pinheiro Paes (23205038)

18 de setembro de 2025

1 Introdução

Este relatório apresenta as justificativa os algoritmos implementados na Atividade A1 da disciplina de Grafos (INE5413). A implementação foi realizada em C++.

2 Estruturas de Dados

2.1 Representação do Grafo (Classe `Grafo`)

Para a implementação da classe `Grafo`, foram selecionadas as seguintes estruturas de dados principais:

2.1.1 Armazenamento de Vértices

- **Estrutura:** `std::unordered_map<int, std::string> vertices`
- **A**o mapear índices de vértices para seus rótulos, usamos O `unordered_map` que oferece acesso em tempo médio $O(1)$ para operações de busca, inserção e remoção.

2.1.2 Lista de Adjacências

- **Estrutura:** `std::unordered_map<int, std::unordered_map<int, double>> adjacencias`
- **R**epresenta as adjacências como `adjacencias[u][v] = peso`, assim essa estrutura permite:
  - Verificação de existência de aresta em $O(1)$
  - Acesso ao peso da aresta em $O(1)$
  - Listagem de vizinhos em $O(\text{grau do vértice})$
  - Inserção e remoção de arestas em $O(1)$ médio

2.1.3 Análise de Complexidade das Operações

Operação	Complexidade	Justificativa
`qtdVertices()`	$O(1)$	Variável armazenada
`qtdArestas()`	$O(1)$	Variável armazenada
`grau(v)`	$O(1)$	Tamanho do map interno
`rotulo(v)`	$O(1)$	Busca em hash table
`haAresta(u,v)`	$O(1)$	Duas buscas em hash table
`peso(u,v)`	$O(1)$	Acesso direto após verificação
`vizinhos(v)`	$O(\text{grau}(v))$	Iteração sobre adjacentes

Tabela 1: Complexidade das operações da classe Grafo

3 Algoritmos Implementados

3.1 Busca em Largura (BFS) - Questão 2

- **Fila:** `std::queue<std::pair<int, int>>` para manter pares (vértice, nível)
- **Controle de visitação:** `std::unordered_set<int>` para $O(1)$ na verificação
- **Níveis:** `std::unordered_map<int, std::vector<int>>` para agrupar vértices por nível

Ao usarmos `std::queue` garantimos a propriedade FIFO (First In, First Out) fundamental para BFS, enquanto o `unordered_set` permite verificação de vértices já visitados. A correção implementada move a inserção no conjunto `visitado` para dentro do loop principal, evitando duplicações na fila.

3.1.1 Complexidade

- **Tempo:** $O(V + E)$ onde V é o número de vértices e E o número de arestas

3.2 Ciclo Euleriano - Questão 3

A implementação utiliza duas fases:

1. **Verificação:** Confirma se existe ciclo euleriano.
2. **Construção:** Aplica o algoritmo de Hierholzer

3.2.1 Critérios para Ciclo Euleriano

Um grafo possui ciclo euleriano se e somente se:

- É conexo (verificado via DFS).
- Todos os vértices possuem grau par.

3.2.2 Algoritmo de Hierholzer

- **Estrutura:** `std::stack<int>` para construir o ciclo
- `std::unordered_map<int, std::vector<int>>` para remoção segura de arestas. A pilha permite a construção de subciclos, característica fundamental do algoritmo de Hierholzer.

3.2.3 Complexidade

- **Tempo:** $O(V + E)$ para verificação + $O(E)$ para construção = $O(V + E)$

3.3 Algoritmo de Dijkstra - Questão 4

- **Heap mínimo:** `std::priority_queue` com comparador `greater<>`
- **Distâncias:** `std::unordered_map<int, double>`
- **Predecessores:** `std::unordered_map<int, int>` para reconstrução de caminho
- **Visitados:** `std::unordered_set<int>`

O uso de `priority_queue`, nós permite extrair o vértice com menor distância em $O(\log V)$, algo fundamental para a autocorreção do algoritmo. A escolha de `double` para distâncias garante precisão adequada para pesos reais. A função `reconstruirCaminho` utiliza o vetor de predecessores para recuperar o caminho mínimo, construindo-o de trás para frente e depois invertendo.

3.3.1 Complexidade

- **Tempo:** $O((V + E) \log V)$ utilizando heap binário

3.4 Algoritmo de Floyd-Warshall - Questão 5

- **Estrutura:** `std::unordered_map<int, std::unordered_map<int, double>>`, o uso dessa estrutura pemite manter a flexibilidade de índices não sequenciais, e ainda preserva acesso $O(1)$

A matriz é inicializada seguindo os critérios:

- $\text{dist}[i][i] = 0$ (distância de um vértice para ele mesmo)
- $\text{dist}[i][j] = \text{peso}(i, j)$ se existe aresta
- $\text{dist}[i][j] = \infty$, caso contrário

3.4.1 Algoritmo Principal

O algoritmo verifica sistematicamente todos os vértices como intermediários:

```
for (int k : vertices) {
    for (int i : vertices) {
        for (int j : vertices) {
            if (dist[i][k] + dist[k][j] < dist[i][j]) {
                dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }
}
```

Listing 1: Núcleo do Floyd-Warshall

3.4.2 Complexidade

- **Tempo:** $O(V^3)$ devido aos três loops aninhados
- **Espaço:** $O(V^2)$ para armazenar a matriz de distâncias

4 Conclusão

As estruturas de dados escolidas priorizaram eficiência temporal, enquanto as operações fundamentais atendem aos requisitos de complexidade especificados. Os algoritmos implementados representam soluções eficientes para os problemas propostos.