

Trabalho de Teoria dos Grafos

Integrantes do grupo:

Vítor Fernandes Reis - 202065096AB

Carlos Gustavo Ferreira Rezende – 202065503B

Gabriel De Oliveira Vieira - 202265029AC

Maria Luiza Dornelas Corrêa - 201665194C

1. Introdução

Este trabalho tem como objetivo a implementação de estruturas de dados para representação e manipulação de grafos em C++. O projeto abrange a representação de grafos tanto por listas de adjacência quanto por matrizes de adjacência, permitindo operações como inserção e remoção de nós e arestas, verificação de propriedades do grafo, cálculo de menores distâncias e resolve o problema da árvore de Steiner.

2. Estrutura do Projeto

```
TrabalhoGrafosGrupoX.zip
|
| docs/
| | documentacao.pdf
| | apresentacao.pdf
| | contribuidores.pdf (o que cada membro fez, junto com o link do github)
| | descricao.pdf (este documento)
|
| include/
| | *.h
| | *.hpp
| | *.tpp
|
| src/
| | *.c
| | *.cpp
|
| entradas/
| | *.txt
|
| main.cpp
|
| README (explicação simples e considerações sobre a execução do programa)
```

- **docs/**: Contém documentos de apoio, como a documentação técnica, apresentação e contribuições dos membros do grupo.
- **include/**: Arquivos de cabeçalho (.h, .hpp) e templates (.tpp).
- **src/**: Implementação do código-fonte em C++.
- **entradas/**: Arquivos de entrada utilizados para testar o programa. Estão presentes 10 instâncias de grafos de pelo menos 5 mil vértices para os testes da função que resolve a árvore de Steiner.

- **main.cpp**: Arquivo principal de execução do programa.
- **README**: Guia rápido de uso do programa.

3. Descrição das Classes e Funções

3.1. Grafo

Classe base abstrata que define as operações fundamentais para grafos. As funções marcadas como virtuais devem ser implementadas pelas classes derivadas (GrafoMatriz e GrafoLista).

- **virtual void novo_no(int id, float peso)**: Adiciona um novo nó.
- **virtual void deleta_no(int id)**: Remove um nó.
- **virtual void nova_aresta(int origem, int destino, float peso)**: Adiciona uma nova aresta.
- **virtual void deleta_aresta(int origem, int destino)**: Remove uma aresta.
- **virtual float getPesoAresta(int origem, int destino)**: Retorna o peso de uma aresta.
- **virtual bool existeNo(int id)**: Verifica se um nó existe.
- **virtual bool existeAresta(int origem, int destino)**: Verifica se uma aresta existe.
- **virtual int getGrau(int id)**: Retorna o grau de um nó.
- **virtual int* getVizinhos(int id)**: Retorna os vizinhos de um nó.
- **virtual int getNumArestas()**: Retorna o número total de arestas do grafo.
- **void carrega_grafo(const std::string& arquivo)**: Carrega um grafo a partir de um arquivo.
- **int get_ordem()**: Retorna a ordem do grafo (número de nós).
- **bool eh_direcionado()**: Verifica se o grafo é direcionado.
- **bool vertice_ponderado()**: Verifica se os vértices são ponderados.
- **bool aresta_ponderada()**: Verifica se as arestas são ponderadas.
- **int get_grau()**: Retorna o grau máximo do grafo.
- **bool eh_completo()**: Verifica se o grafo é completo.
- **int n_conexo()**: Retorna o número de componentes conexas do grafo.
- **void DFS(int no, bool* visitado)**: Realiza uma busca em profundidade.

- **void calculaMenorDistancia():** Calcula as menores distâncias usando o algoritmo de Floyd-Warshall.
- **void deleta_primeira_aresta(int id):** Deleta a primeira aresta de um nó.
- **void imprimeGrafo():** Exibe informações do grafo.
- **virtual void imprimeLista():** Imprime a lista de adjacência do grafo.
- **void steinerTree(int *terminais, int tamanho, bool randomizado, bool reativo, float alpha):** Resolve o problema da Árvore de Steiner, pelos métodos guloso, guloso randomizado e guloso randomizado reativo, todos implementados na mesma função, passados pelos parâmetros bool recebidos pela função.

3.2. GrafoLista

Implementação do grafo usando listas de adjacência. Esta classe implementa todos os métodos virtuais da classe base Grafo.

- **void novo_no(int id, float peso) override:** Adiciona um novo nó.
- **void deleta_no(int id) override:** Remove um nó.
- **void nova_aresta(int origem, int destino, float peso) override:** Adiciona uma nova aresta.
- **void deleta_aresta(int origem, int destino) override:** Remove uma aresta.
- **bool existeNo(int id) override:** Verifica se o nó existe.
- **bool existeAresta(int origem, int destino) override:** Verifica se a aresta existe.
- **int getGrau(int id) override:** Retorna o grau de um vértice.
- **int* getVizinhos(int id) override:** Retorna os vizinhos de um nó.
- **int getNumArestas() override:** Retorna o número total de arestas.
- **float getPesoAresta(int origem, int destino) override:** Retorna o peso da aresta.
- **void imprimeLista() override:** Imprime a lista de adjacência do grafo.

3.3. GrafoMatriz

Implementação do grafo usando matrizes de adjacência. Também implementa todos os métodos virtuais da classe base Grafo.

- **void novo_no(int id, float peso) override:** Adiciona um novo nó.
- **void deleta_no(int id) override:** Remove um nó.

- **void nova_aresta(int origem, int destino, float peso) override:** Adiciona uma nova aresta.
- **void deleta_aresta(int origem, int destino) override:** Remove uma aresta.
- **bool existeAresta(int origem, int destino) override:** Verifica se uma aresta existe.
- **float getPesoAresta(int origem, int destino) override:** Retorna o peso da aresta.
- **int getGrau(int id) override:** Retorna o grau de um nó.
- **int* getVizinhos(int id) override:** Retorna os vizinhos de um nó.
- **int getNumArestas() override:** Retorna o número total de arestas.
- **void imprimeMatriz() override:** Imprime a matriz de adjacência do grafo.

3.4. ListaEncadeada

Suporte para a representação de listas de nós e arestas:

- **No* getNo(int id):** Retorna o nó correspondente.
- **Aresta* getAresta(int origem, int destino):** Retorna a aresta entre dois nós.
- **void setCabeca(No* novoCabeca):** Define o nó cabeça da lista.
- **No* getCabeca():** Retorna o nó cabeça da lista.
- **void imprimeLista():** Imprime a lista de adjacência do grafo.

4. Compilação e Execução

4.1. Compilação

Para compilar o programa, utilize o seguinte comando:

```
g++ -o main.out main.cpp src/*.cpp -I./include/ -g -Wall -Werror
```

4.2. Execução

O programa pode ser executado com o seguinte comando:

- **time|valgrind main.out -p -m|-l ./entradas/grafos.txt**
- **-p:** Ativa a análise de performance do código.
- **-m ou -l:** Define se o grafo será representado por matriz (-m) ou lista (-l).
- **./entradas/grafos.txt:** Arquivo de entrada contendo a descrição do grafo.

5. Exemplos de Entrada/Saída

Exemplo de Arquivo de Entrada (entradas/exemplo.txt):

```
30 0 1 1
10 20 30 40 50
1 2 5
1 3 3
2 3 4
2 5 9.6
2 4 2.588
3 5 7.32
3 4 9.1
3 6 4
4 5 1.5
4 6 2
5 7 3.2
6 8 4.5
7 8 2.1
7 9 5.7
8 10 3.8
9 10 4.2
9 11 6.1
10 12 2.9
11 12 3.3
11 13 7.8
12 14 4.4
13 14 5.6
13 15 8.2
14 16 3.7
15 16 6.9
15 17 2.3
16 18 4.8
17 18 5.1
17 19 7.4
18 20 3.6
19 20 6.2
19 21 4.7
20 22 2.8
21 22 5.3
21 23 8.1
22 24 3.9
23 24 6.5
23 25 2.4
24 26 4.1
25 26 7.1
25 27 3.5
26 28 5.8
27 28 6.7
27 29 2.6
28 30 4.3
29 30 7.9
```

Saída Esperada:

```
INFORMAÇÕES DO GRAFO:
Ordem: 30
Direcionado: Não
Guloso Normal
Árvore de Steiner normal encontrada com os nós: 8 10 12 14 16 18 20 22 24
E com as arestas: (10, 12) (8, 10) (14, 16) (12, 14) (22, 24) (20, 22) (18, 20) (16, 18)

Guloso Randomizado
Árvore de Steiner randomizada encontrada com os nós: 8 10 12 14 16 18 20 22 24
E com as arestas: (10, 12) (8, 10) (14, 16) (12, 14) (22, 24) (20, 22) (18, 20) (16, 18)

Guloso Randomizado Reativo
Melhor custo encontrado: 29.9
Melhor solução com os nós: 8 10 12 14 16 18 20 22 24
E com as arestas: (10, 12) (8, 10) (14, 16) (12, 14) (22, 24) (20, 22) (18, 20) (16, 18)
```

6. Conclusão

O trabalho permitiu a compreensão prática de conceitos de Teoria dos Grafos, como a representação de grafos por listas e matrizes de adjacência, além de algoritmos de busca, e, com as expansões feitas durante o trabalho 2, pudemos adicionar o cálculo de distâncias, e fazer a adição e remoção de nós e arestas. Com as expansões feitas durante o trabalho 3, pudemos resolver problemas da Árvore de Steiner, encontrando uma árvore que conecta um subconjunto específico de vértices com o menor custo total, incluindo vértices e arestas adicionais para criar um caminho mais curto.