Relatório Técnico: Biblioteca de Grafos em C++

Caio Passos Torkst Ferreira João Guilherme de Oliveira Ribeiro Kongevold

1 Introdução

Este relatório descreve a implementação de uma biblioteca de grafos em C++, que fornece suporte para diferentes representações (lista de adjacência e matriz de adjacência) e algoritmos clássicos como BFS, DFS, componentes conexas, e caminhos mínimos com Dijkstra.

2 Arquivos do Projeto

O projeto é composto pelos seguintes arquivos:

- **grafo.h**: Declarações da classe `Grafo`.
- **grafo.cpp**: Implementações dos métodos da classe.
- **README.txt**: Instruções de compilação e uso.

3 Funcionalidades Implementadas

3.1 Representações

A biblioteca permite duas representações de grafos:

- **Lista de Adjacência**: eficiente para grafos esparsos.
- **Matriz de Adjacência**: mais simples, útil para grafos densos.

3.2 Algoritmos Disponíveis

- **BFS (Busca em Largura)**: disponível entre um par de vértices, ou de um vértice para todos.
- **DFS (Busca em Profundidade)**: implementada de forma iterativa.
- **Componentes Conexas**: busca em profundidade para listar componentes.
- **Dijkstra**: para encontrar caminhos mínimos (sem pesos negativos).

3.3 Leitura e Escrita de Arquivos

O método `ler_arquivo` permite carregar grafos a partir de arquivos de texto. Os resultados são escritos automaticamente em arquivos de saída dentro da pasta `saida/`.

4 Descrição do Arquivo `grafo.h`

O arquivo `grafo.h` é o cabeçalho da biblioteca de grafos. Nele estão definidas a estrutura da classe `Grafo`, seus atributos e métodos públicos e privados. A seguir, detalhamos sua composição:

4.1 Atributos Privados

- `int V, E`: número de vértices e arestas.
- `Representacao rep`: ENUM para o tipo de representação usada.
  - **LISTA_ADJ**: lista de adjacência (mais eficiente para grafos esparsos);
  - **MATRIZ_ADJ**: matriz de adjacência (mais simples e eficiente em grafos densos).

- `vector<vector<pair<int, double>>> lista_adj`: lista de adjacência.
- `vector<vector<double>> matriz_adj`: matriz de adjacência.
- `bool tem_pesos, tem_pesos_negativos`: flags que indicam se o grafo possui pesos e se há pesos negativos.
- `string nome_base_saida`: nome base para os arquivos de saída gerados.

4.2 Métodos Privados

Métodos auxiliares usados internamente para implementação dos algoritmos:

- `bfs_interno`: executa a BFS com ou sem destino.
- `dfs_interno`: executa a DFS iterativa.
- `dfs_componentes`: realiza DFS para encontrar componentes conexas.
- `dijkstra`: implementa o algoritmo de Dijkstra com ou sem destino.

4.3 Métodos Públicos

Esses métodos representam a interface que pode ser utilizada fora da biblioteca:

- `Grafo(int V, Representacao rep)`: construtor da classe.
- `ler_arquivo(string nome_arquivo)`: lê um grafo de um arquivo de texto.
- `salvar_estatisticas()`: salva grau médio e distribuição de graus.
- `bfs(int s)`: executa busca em largura a partir de `s`.
- `dfs(int s)`: executa busca em profundidade a partir de `s`.
- `componentes_conexas()`: identifica e salva as componentes conexas.
- `caminho_minimo(int s, int t)`: retorna caminho mínimo de `s` até `t`.
- `caminhos_minimos(int s)`: retorna os caminhos mínimos de `s` até todos os vértices.

5 Decisões de Projeto

Tomamos decisões visando garantir flexibilidade, eficiência e facilidade de uso. A seguir, discutimos as principais decisões e os motivos por trás de cada uma.

5.1 Uso da Linguagem C++

A linguagem escolhida para a implementação foi o **C++**. Essa decisão se deu por uma combinação dos seguintes fatores:

- **Eficiência**: C++ oferece alto desempenho e controle preciso sobre o uso de memória e estruturas de dados, o que é essencial para algoritmos de grafos, que podem ser exigentes em termos de tempo e espaço.
- **Flexibilidade**: C++ permite misturar abordagens de alto e baixo nível. Isso foi útil, por exemplo, para leitura eficiente de arquivos (`sscanf`) combinada com strings da STL.
- **Biblioteca Padrão (STL)**: estruturas como `vector`, `pair`, `queue`, `priority_queue` e `stack` tornam a implementação mais rápida, segura e legível, sem perder desempenho.

5.2 Separação entre Interface Pública e Implementação

Tornar privados os métodos internos dos algoritmos (DFS, BFS, Dijkstra) e públicos apenas os métodos de alto nível que o usuário irá chamar.

- **Impacto:** facilita manutenção e testes, além de permitir possíveis trocas futuras na implementação sem afetar a interface.

5.3 Implementação Iterativa do DFS e BFS

- **Motivação:** evitar estouro de pilha (stack overflow) em grafos muito grandes.
- **Decisão:** usar uma pilha/fila manual (estrutura `stack/queue`) para simular a recursão da DFS e BFS.

5.4 Suporte a Pesos e Detecção de Pesos Negativos

- **Motivação:** permitir uso genérico com grafos ponderados ou não, e evitar uso incorreto de Dijkstra com pesos negativos.
- **Decisão:** detectar automaticamente, na leitura do arquivo, se os pesos estão presentes e se algum é negativo.

5.5 Geração de Arquivos de Saída

Cada método principal (BFS, DFS, componentes, caminhos mínimos) gera um arquivo `.txt` com sufixos padronizados, baseados no nome do arquivo de entrada.

- **Impacto:** separa claramente os resultados e permite uso posterior em scripts ou visualizações externas.

5.6 Suporte a Caminhos com e sem Destino

Fornecer dois métodos: `caminho_minimo(s, t)` e `caminhos_minimos(s)`.

- **Impacto:** oferece maior controle ao usuário conforme a necessidade.

5.7 Leitura de Entrada

- O método `ler_arquivo(const string& nome_arquivo)` lê o grafo de um arquivo texto. A primeira linha contém o número de vértices. As demais linhas representam arestas no formato:

```
u v peso
```

- A leitura utiliza a função `getline()` para obter cada linha do arquivo, e `sscanf()` para fazer a extração dos inteiros e do peso de forma flexível (com ou sem peso explícito).
- Durante a leitura, a função detecta automaticamente se o grafo possui pesos e se há pesos negativos, ajustando o comportamento posterior (por exemplo, Dijkstra não será executado com pesos negativos).

6 Descrição do Arquivo `grafo.cpp`

O arquivo `grafo.cpp` contém a implementação completa da classe `Grafo`. A seguir, descrevemos cada função em detalhes.

6.1 Construtor

```
Grafo::Grafo(int V, Representacao rep);
```

Inicializa um grafo com `V` vértices e o tipo de representação desejado (`LISTA_ADJ` ou `MATRIZ_ADJ`). A estrutura apropriada é alocada com base na representação.

6.2 Leitura de Arquivo

```
void Grafo::ler_arquivo(const string& nome_arquivo);
```

Lê um grafo de um arquivo texto. A primeira linha contém o número de vértices. As demais contêm arestas no formato `u v [peso]`. A função detecta automaticamente se há pesos e se algum é negativo. Também atualiza o nome base dos arquivos de saída com base no nome do arquivo de entrada.

6.3 Estatísticas

```
void Grafo::salvar_estatisticas();
```

Calcula:

- Grau de cada vértice;
- Grau médio do grafo;
- Distribuição dos graus.

Os resultados são salvos em um arquivo com sufixo `_estatisticas.txt`.

6.4 Busca em Largura (BFS)

```
void Grafo::bfs(int s);
```

Executa a BFS a partir do vértice `s` e salva, para cada vértice:

- Pai na árvore BFS;
- Nível (distância em número de arestas).

Os resultados são salvos em um arquivo com sufixo `_bfs.txt`.

6.5 Busca em Profundidade (DFS)

```
void Grafo::dfs(int s);
```

Executa DFS iterativa a partir de `s`, armazenando o pai de cada vértice. O resultado é salvo em arquivo com sufixo `_dfs.txt`.

6.6 Componentes Conexas

```
void Grafo::componentes_conexas();
```

Realiza múltiplas DFS para identificar todas as componentes conexas do grafo. As componentes são ordenadas por tamanho decrescente. O resultado é salvo em `_componentes.txt`.

6.7 Caminho Mínimo entre Dois Vértices

```
pair<vector<int>, vector<double>> Grafo::caminho_minimo(int s, int t);
```

Determina o caminho mais curto de `s` até `t`:

- Usa BFS se o grafo for não ponderado;
- Usa Dijkstra se o grafo tiver pesos positivos;
- Emite erro se houver pesos negativos.

Retorna o caminho como vetor de vértices e um vetor de distâncias. Além disso, os resultados são salvos em um arquivo com sufixo `_caminhos_s_t.txt`.

6.8 Caminhos Mínimos a partir de `s`

```
pair<vector<vector<int>>, vector<double>> Grafo::caminhos_minimos(int s);
```

Calcula os caminhos mínimos de `s` até todos os outros vértices:

- Usa BFS ou Dijkstra conforme os pesos;
- Retorna todos os caminhos reconstruídos; Os resultados são salvos em um arquivo com sufixo `_caminhos_s.txt`.

6.9 Métodos Privados

BFS Interno (sem destino):

```
void bfs_interno(int s, vector<int>& pai, vector<int>& nivel);
```

Executa BFS padrão e preenche os vetores de pais e níveis.

BFS Interno (com destino):

```
void bfs_interno(int s, int t, vector<int>& pai, vector<int>& nivel);
```

Versão modificada que interrompe a busca ao alcançar o vértice `t`.

DFS Interno:

```
void dfs_interno(int s, vector<bool>& visitado, vector<int>& pai);
```

Executa uma DFS iterativa utilizando pilha e simula o comportamento recursivo ao inserir os vizinhos em ordem reversa.

DFS para Componentes:

```
void dfs_componentes(int s, vector<bool>& visitado, vector<int>& componente);
```

Percorre todos os vértices de uma componente e adiciona ao vetor da componente.

Dijkstra Completo:

```
void dijkstra(int s, vector<int>& pai, vector<double>& dists);
```

Executa o algoritmo de Dijkstra a partir de `s` para todos os vértices. Usa uma fila de prioridade mínima para escolher o vértice com menor distância estimada.

Dijkstra com Destino:

```
void dijkstra(int s, int t, vector<int>& pai, vector<double>& dists);
```

Versão otimizada de Dijkstra que interrompe o processo quando o destino `t` é alcançado, economizando tempo.

7 Código de Testes

O arquivo `testa_lista.cpp` executa a biblioteca para três arquivos de entrada. A seguir, um exemplo do código:

```
Grafo g(0, LISTA_ADJ);
g.ler_arquivo("entrada/test.txt");
g.salvar_estatisticas();
g.bfs(0);
g.dfs(0);
g.componentes_conexas();
g.caminhos_minimos(0);
g.caminho_minimo(0, 5);
```

8 Instruções de Compilação

Conforme indicado no `README.txt`, para compilar qualquer código de teste, basta utilizar o seguinte comando:

```
g++ nome_arquivo.cpp -o nome_arquivo grafo.cpp
```

9 Considerações Finais

A biblioteca oferece uma base sólida para experimentações com algoritmos de grafos. Está preparada para leitura de grafos com pesos positivos ou não ponderados, e gera relatórios automáticos em arquivos de saída.