Documentação do Trabalho de Grafos 01

DCC059

9 de fevereiro de 2025

Sumário

1 Introdução

Este documento apresenta a documentação detalhada do trabalho de implementação de grafos em C++ utilizando conceitos de orientação a objetos. O projeto contempla uma estrutura de classes que permite a representação e manipulação de grafos por meio de duas abordagens:

- **Matriz de Adjacência:** Armazenamento estático dos vértices e arestas.
- **Lista Encadeada:** Armazenamento dinâmico (alocação dinâmica) tanto para os vértices quanto para as arestas.

Além disso, o sistema possui validações que impedem a inserção de laços (arestas onde a origem é igual ao destino) e arestas múltiplas (mais de uma aresta entre dois vértices).

2 Objetivos do Projeto

- Implementar uma classe abstrata `Grafo` que defina funções genéricas para:
  - Carregar o grafo a partir de um arquivo (`carrega_grafo`);
  - Obter a ordem do grafo (`get_ordem`);
  - Calcular o grau máximo (`get_grau`);
  - Determinar o número de componentes conexas (`n_conexo`);
  - Verificar se o grafo é direcionado (`eh_direcionado`);
  - Verificar se os vértices possuem peso (`vertice_ponderado`);
  - Verificar se as arestas possuem peso (`aresta_ponderada`);
  - Verificar se o grafo é completo (`eh_completo`).
- Desenvolver duas classes derivadas:
  - **GrafoMatriz**: Representa o grafo por meio de uma matriz de adjacência. Em grafos não direcionados, idealmente utiliza uma representação linear de matriz triangular, porém, para simplificação, foi empregada uma matriz completa.
  - **GrafoLista**: Representa o grafo por meio de listas encadeadas para os vértices e para as arestas, com alocação dinâmica.
- Implementar validações que impeçam:
  - **Laços:** Não permitir que uma aresta seja criada se a origem for igual ao destino.
  - **Arestas Múltiplas:** Não permitir a inserção de uma nova aresta se já houver uma aresta entre os mesmos vértices.

3 Estrutura do Projeto

A organização dos arquivos do projeto é a seguinte:

```
| include/
|    IntList.hpp
|    ListaEncadeada.hpp
|    ListaEncadeada.tpp
|    Grafo.hpp
|    GrafoMatriz.hpp
|    GrafoLista.hpp
|
| src/
|    IntList.cpp
|    Grafo.cpp
|    GrafoMatriz.cpp
|    GrafoLista.cpp
|
| entradas/
|    grafo.txt
|
| main.cpp
```

4 Descrição das Classes e Funções

4.1 Classe `Grafo` (Abstrata)

A classe `Grafo` define os atributos e métodos comuns a todas as implementações do grafo:

- **Atributos:**
  - `ordem`: Número de vértices do grafo.
  - `direcionado`: Indica se o grafo é direcionado (`true`) ou não (`false`).
  - `ponderadoVertices`: Indica se os vértices possuem peso.
  - `ponderadoArestas`: Indica se as arestas possuem peso.
- **Métodos: carrega_grafo**

  `carrega_grafo(string nomeArquivo):` Lê um arquivo de texto e carrega os dados do grafo. O arquivo tem o formato:

  ```
  ordem direcionado ponderadoVertices ponderadoArestas
  [pesos dos vértices se aplicável]
  origem destino [peso]   (para cada aresta)
  ```

  `get_ordem():` Retorna o número total de vértices.

  `get_grau():` Calcula o grau máximo dos vértices. Para grafos direcionados, soma o outdegree e o indegree.

`n_conexo()`: Utiliza uma busca em profundidade (DFS) para determinar o número de componentes conexas.

`eh_direcionado()`: Retorna se o grafo é direcionado.

`vertice_ponderado()`: Indica se os vértices têm peso.

`aresta_ponderada()`: Indica se as arestas têm peso.

`eh_completo()`: Verifica se o grafo é completo (cada vértice deve estar conectado a todos os demais).

- **Método Virtual Puro:** `get_vizinhos(int vertice)` é declarado virtual puro para que as classes derivadas forneçam sua própria implementação, retornando os vértices adjacentes.

4.2 Classe `GrafoMatriz`

A classe `GrafoMatriz` implementa a representação do grafo por meio de uma matriz de adjacência.

- **Estrutura de Armazenamento:**
  - Um array unidimensional para armazenar os pesos dos vértices.
  - Uma matriz (array de arrays) para armazenar as arestas. Em grafos não direcionados, seria ideal utilizar uma representação triangular linear; na implementação atual, utiliza-se uma matriz completa.
- **Principais Funções: inserir_vertice**

`inserir_vertice(int id, int peso)`: Inicializa e insere um vértice, armazenando seu peso.

`inserir_aresta(int origem, int destino, int peso)`: Insere uma aresta na matriz de adjacência.
Validação: Antes da inserção, verifica se:
  - A aresta não é um laço (ou seja, `origem` $\neq$ `destino`).
  - Não existe já uma aresta na posição indicada (para evitar arestas múltiplas).

`get_vizinhos(int vertice)`: Varre a linha da matriz correspondente ao vértice e retorna uma lista dos vértices adjacentes.

4.3 Classe `GrafoLista`

A classe `GrafoLista` implementa a representação do grafo usando listas encadeadas para os vértices e para as arestas.

- **Estrutura de Armazenamento:**
  - Uma lista encadeada para armazenar os vértices.
  - Cada vértice contém uma lista encadeada para armazenar suas arestas adjacentes.
- **Principais Funções: inserir_vertice**

`inserir_vertice(int id, int peso):` Cria dinamicamente um vértice e o adiciona à lista de vértices.

`inserir_aresta(int origem, int destino, int peso):` Insere uma aresta na lista de arestas do vértice de origem.
Validação: Antes de inserir, o método verifica:

- Se `origem` é igual a `destino` (impedindo laços).
- Se já existe uma aresta com o mesmo destino (impedindo arestas múltiplas).

Para grafos não direcionados, a inserção é realizada simetricamente no vértice de destino.

`get_vizinhos(int vertice):` Percorre a lista de arestas do vértice especificado e retorna os identificadores dos vértices adjacentes.

`encontrar_vertice(int id):` Função auxiliar que busca e retorna um ponteiro para o vértice com o identificador informado.

4.4 Outras Classes de Suporte

- **IntList:** Implementa uma lista dinâmica de inteiros (sem uso da STL) para armazenar, por exemplo, os vizinhos de um vértice. **add**

  `add(int value):` Adiciona um valor à lista.

  `size():` Retorna a quantidade de elementos.

  `get(int index):` Retorna o valor armazenado na posição especificada.

- **ListaEncadeada:** Implementa uma lista encadeada genérica. Utiliza uma estrutura de nó (`No<T>`) que contém o dado e o ponteiro para o próximo nó. **inserir**

  `inserir(const T& dado):` Insere um novo nó no início da lista.

  `getHead():` Retorna o ponteiro para o primeiro nó da lista, permitindo a iteração.

5 Validações Implementadas

Para garantir a integridade do grafo, foram implementadas as seguintes validações:

- **Laços:** O método `inserir_aresta` verifica se a origem é igual ao destino. Se for, exibe uma mensagem de erro e não realiza a inserção.
- **Arestas Múltiplas:** Antes de inserir uma aresta, o método percorre a lista de arestas do vértice de origem (ou verifica a matriz, no caso do `GrafoMatriz`) para confirmar que não existe uma aresta para o mesmo destino.

```
void GrafoLista::inserir_aresta(int origem, int destino, int peso) {
    // Valida  o: n o permitir la os
    if (origem == destino) {
        cerr << "Erro: La o  n  o permitido (origem e destino iguais: "
                  << origem << ")." << endl;
        return;
    }

```

```
    Vertice* v = encontrar_vertice(origem);
    if (!v) {
        cerr << "Erro: V rtice " << origem << " n o encontrado." << endl;
        return;
    }

    // Verifica se j  existe uma aresta de 'origem' para 'destino'
    No<Aresta>* no = v->arestas->getHead();
    while (no) {
        if (no->dado.destino == destino) {
            cerr << "Erro: Aresta de " << origem
                 << " para " << destino << " j  existe." << endl;
            return;
        }
        no = no->prox;
    }

    // Inser  o da aresta
    v->arestas->inserir(Aresta(destino, peso));

    // Se o grafo n o for direcionado, insere a aresta simetricamente
    if (!direcionado) {
        Vertice* v2 = encontrar_vertice(destino);
        if (!v2) {
            cerr << "Erro: V rtice " << destino << " n o encontrado." << endl;
            return;
        }
        no = v2->arestas->getHead();
        while (no) {
            if (no->dado.destino == origem) {
                cerr << "Erro: Aresta de " << destino
                     << " para " << origem << " j  existe." << endl;
                return;
            }
            no = no->prox;
        }
        v2->arestas->inserir(Aresta(origem, peso));
    }
}
```

Listing 1: Exemplo de validação no método inserir_aresta da classe GrafoLista

6 Compilação e Execução

6.1 Compilação

Para compilar o projeto, utilize um compilador C++ (por exemplo, `g++`). Um exemplo de comando é:

```
g++ -o main.out main.cpp src/*.cpp
```

6.2 Execução

O programa é executado a partir da linha de comando, utilizando os seguintes parâmetros:

- `-m` ou `-l`: Indica a estrutura de armazenamento a ser utilizada (`-m` para matriz e `-l` para lista).
- Nome do arquivo de entrada (por exemplo, `entradas/grafo.txt`).

Exemplos:

```
./main.out -d -m entradas/grafo.txt
./main.out -d -l entradas/grafo.txt
```

Durante a execução, o programa exibirá informações como:

- Grau do grafo.
- Ordem do grafo.
- Se o grafo é direcionado.
- Se os vértices são ponderados.
- Se as arestas são ponderadas.
- Se o grafo é completo.

7 Exemplo de Arquivo de Entrada

Um exemplo de arquivo `grafo.txt` que segue o formato esperado:

```
3 1 1 1
2 3 7
1 2 6
2 1 4
2 3 -5
```

- A primeira linha contém:
  - `3` – número de vértices;
  - `1` – grafo direcionado (1 = sim, 0 = não);
  - `1` – vértices ponderados (1 = sim, 0 = não);
  - `1` – arestas ponderadas (1 = sim, 0 = não).
- A segunda linha apresenta os pesos dos vértices (nesse caso, 2, 3 e 7 para os vértices 1, 2 e 3, respectivamente).
- As linhas seguintes descrevem as arestas: `origem destino [peso]`.

8 Considerações Finais

Este projeto demonstra a aplicação de conceitos de orientação a objetos em C++ para a implementação de grafos. Foram desenvolvidas duas abordagens distintas para o armazenamento (matriz e lista) e implementadas validações para garantir a integridade dos dados, impedindo a inserção de laços e arestas múltiplas.

Recomenda-se testar o programa com diferentes arquivos de entrada para verificar a robustez das implementações. Em caso de dúvidas ou necessidade de ajustes, os comentários detalhados no código-fonte podem ajudar na compreensão das funções e estruturas.