




Trabalho 01 - Implementação de Grafos

Alice Salim Khouri Antunes   [Pontifícia Universidade Católica de Minas Gerais | alice.salim.khouri@gmail.com]

Arthur Carvalho Rodrigues  [Pontifícia Universidade Católica de Minas Gerais | arthurcarvalhorodrigues2409@gmail.com]


Bruna Furtado da Fonseca  [Pontifícia Universidade Católica de Minas Gerais | brunafurfon@gmail.com]

Daniel Victor Rocha Costa  [Pontifícia Universidade Católica de Minas Gerais | danivitorcosta@gmail.com]

Felipe Barros Ratton de Almeida  [Pontifícia Universidade Católica de Minas Gerais | felipebarrosratton.almeida@gmail.com]

Gustavo Henrique Rodrigues de Castro  [Pontifícia Universidade Católica de Minas Gerais | guguh1612@gmail.com]

Mariana Almeida Mendonça  [Pontifícia Universidade Católica de Minas Gerais | marianaalmeidaafa@gmail.com]

 Instituto de Ciências Exatas e Informática, Pontifícia Universidade Católica de Minas Gerais, R. Dom José Gaspar, 500 - Coração Eucarístico, Belo Horizonte - MG, 30535-901, Brazil.

Resumo. Este trabalho apresenta o desenvolvimento de um sistema para manipulação de grafos, implementado como parte da disciplina de Teoria de Grafos e Computabilidade. O projeto inclui a criação de grafos não direcionados e não ponderados, além de grafos direcionados e ponderados, permitindo operações como inserção e remoção de vértices e arestas, consulta de conexões e pesos, e impressão da estrutura. A implementação foi organizada em módulos, com arquivos de cabeçalho e código-fonte separados, garantindo clareza e reutilização do código. O sistema foi desenvolvido com o objetivo de consolidar conceitos teóricos por meio de uma abordagem prática.

Keywords: Teoria dos Grafos, Grafos Direcionados, Grafos Não-Direcionados, Grafos Ponderados, Implementação, Estrutura de Dados

© Published under the Creative Commons Attribution 4.0 International Public License (CC BY 4.0)

1 Introdução

O estudo de grafos é uma área essencial da ciência da computação, com aplicações em diversas áreas, como redes de comunicação, análise de sistemas complexos, redes sociais e sistemas de transporte. Algoritmos baseados em grafos são utilizados em diversas áreas para resolver uma ampla gama de problemas. Um grafo pode ser definido como $G = (V, E)$, onde V representa o conjunto de vértices e E o conjunto de arestas. Este trabalho foi desenvolvido como parte da disciplina de Teoria de Grafos e Computabilidade, com o objetivo de implementar estruturas básicas de grafos e explorar suas operações e propriedades.

O projeto abrange a implementação de quatro tipos principais de grafos (apresentados respectivamente na **Figura 1**), considerando as combinações entre direção e peso: o grafo não-direcionado e não-ponderado apresenta conexões simples entre os vértices, enquanto o grafo direcionado e não-ponderado inclui arestas com direção. Já o grafo não-direcionado e ponderado associa pesos às conexões, e o grafo direcionado e ponderado combina pesos e direção nas arestas. Essas estruturas permitem a aplicação de conceitos como listas de adjacência, iteração sobre vértices e arestas, manipulação de conexões e atualização de pesos. A implementação foi realizada em C++, utilizando conceitos de orientação a objetos para garantir flexibilidade e reutilização. O programa permite operações como inserção e remoção de vértices e arestas, consulta de conexões e pesos, e impressão da estrutura do grafo.

Este documento apresenta a descrição detalhada do projeto, incluindo sua organização, funcionalidades implemen-

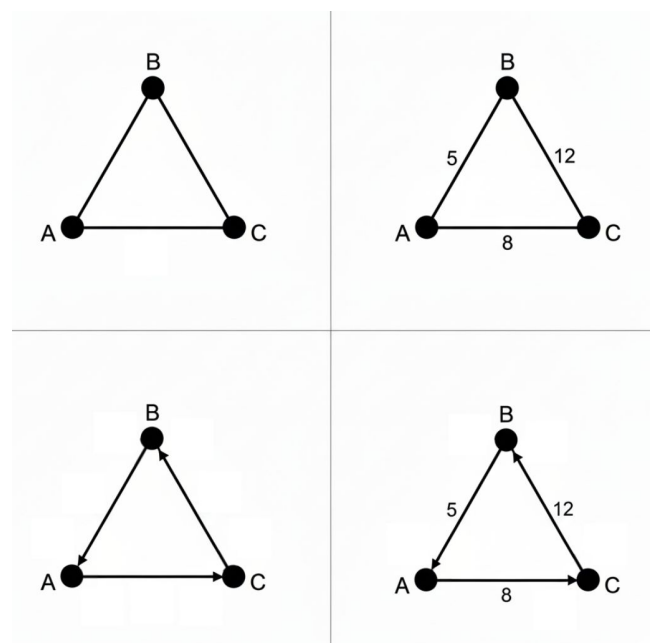


Figure 1. Representação dos quatro tipos de grafos.

tadas e os resultados obtidos, além de destacar a aplicação prática dos conceitos de grafos.

2 Metodologia

Para o desenvolvimento do projeto realizamos a definição da estrutura até a implementação e os testes. O trabalho foi estruturado de forma modular, utilizando conceitos de orien-

tação a objetos e boas práticas de programação.

2.1 Ferramentas Utilizadas

- **Linguagem de Programação:** C++.
- **Compilador:** g++.
- **Controle de Versão:** GitHub, para colaboração e gerenciamento de tarefas.
- **Ambiente de Desenvolvimento:** Visual Studio Code, com extensões para C++.

2.2 Estruturação do Código

O projeto foi organizado em módulos para garantir clareza e reutilização. A estrutura de pastas foi definida da seguinte forma:

- **include/:** Contém os arquivos de cabeçalho (.h) com as definições das classes e estruturas.
- **src/:** Contém os arquivos de implementação (.cpp) e o arquivo principal (main.cpp).
- **main:** Arquivo executável gerado após a compilação.

Os principais arquivos e suas responsabilidades são:

- **GraphBase.h:** Define a classe base para os grafos, com características comuns a todos os tipos.
- **Edge.h e WeightedEdge.h:** Representam as arestas simples e ponderadas, respectivamente.
- **Graph.h e Graph.cpp:** Implementam grafos não ponderados.
- **WeightedGraph.h e WeightedGraph.cpp:** Implementam grafos ponderados.

2.3 Implementação

A estrutura de dados escolhida para representar os grafos foi definida com base na busca por eficiência e simplicidade nas operações mais comuns, como inserção, remoção e consulta de vértices e arestas. Optou-se pelo uso de listas de adjacência, uma abordagem amplamente reconhecida por seu baixo consumo de memória e excelente desempenho em grafos esparsos. Cada grafo foi modelado como uma coleção de vértices, onde cada vértice armazena uma lista de suas arestas adjacentes. Essa representação foi projetada de forma modular, com classes específicas para cada componente do grafo. A implementação foi desenvolvida em C++ e fez uso de conceitos de orientação a objetos. Cada tipo de grafo foi estruturado como uma classe distinta, contendo métodos próprios para suas operações. A seguir, são apresentadas as principais etapas dessa implementação.

Os **grafos não direcionados e não ponderados** foram implementados na classe `Graph`, que utiliza uma lista de adjacência para armazenar as conexões entre os vértices. Essa lista é representada por um vetor de vetores (`std::vector<std::vector<Edge>>`), onde cada posição do vetor principal corresponde a um vértice, e o vetor interno armazena as arestas conectadas a esse vértice. Cada aresta é representada pela classe `Edge`, que contém informações sobre o vértice de destino. Como o grafo é não direcionado,

ao adicionar uma aresta entre dois vértices, a conexão é registrada em ambas as direções, garantindo a simetria. Essa estrutura é eficiente para operações como inserção e consulta de arestas, especialmente em grafos esparsos.

Os **grafos direcionados e não ponderados** também utilizam a classe `Graph`, com a diferença de que as conexões são registradas apenas na direção especificada. A lista de adjacência armazena as conexões direcionais, ou seja, uma aresta entre dois vértices é registrada apenas no vetor correspondente ao vértice de origem. Essa abordagem permite modelar relações assimétricas, como dependências ou fluxos direcionais. A estrutura de dados é a mesma utilizada para grafos não direcionados, mas o comportamento das operações de inserção e consulta é ajustado para refletir a direção das conexões.

Os **grafos não direcionados e ponderados** foram implementados na classe `WeightedGraph`, que estende a funcionalidade da classe base para incluir pesos nas arestas. A lista de adjacência é representada por um vetor de vetores (`std::vector<std::vector<WeightedEdge>>`), onde cada posição do vetor principal corresponde a um vértice, e o vetor interno armazena as arestas conectadas a esse vértice. Cada aresta é representada pela classe `WeightedEdge`, que armazena o vértice de destino e o peso da conexão. Como o grafo é não direcionado, as conexões são registradas em ambas as direções, garantindo que o peso seja consistente em ambas as representações. Essa estrutura é útil para modelar cenários onde as conexões possuem custos associados, como redes de transporte ou mapas de rotas.

Os **grafos direcionados e ponderados** também utilizam a classe `WeightedGraph`, com a diferença de que as conexões são registradas apenas na direção especificada. A lista de adjacência armazena conexões direcionais com seus respectivos pesos, permitindo modelar cenários onde as conexões possuem um sentido e um custo associado, como redes de fluxo ou sistemas de transporte com restrições de capacidade. Cada aresta é representada pela classe `WeightedEdge`, que armazena o vértice de destino e o peso da conexão. Essa estrutura permite operações como consulta de custos e atualização de pesos, além de suportar algoritmos específicos para grafos direcionados, como o cálculo de caminhos mínimos.

2.4 Testes

Os testes foram realizados no arquivo `main.cpp`, que serve como ponto de entrada do programa. Foram criados exemplos práticos para verificar o funcionamento correto de cada tipo de grafo e suas operações utilizando as classes `Graph` e `WeightedGraph`. Os testes incluíram:

- Criação de grafos com diferentes combinações de direção e peso.
- Inserção e remoção de vértices e arestas.
- Consulta de conexões e pesos.
- Impressão da estrutura do grafo.

3 Resultados

Os resultados obtidos na implementação foram organizados para avaliar sistematicamente o cumprimento dos objetivos

propostos, com foco na funcionalidade, correção e robustez das estruturas implementadas. A seguir, são apresentados os resultados detalhados para cada variante de grafo implementada, demonstrando o correto funcionamento das operações básicas e das propriedades estruturais características de cada tipo.

Grafo Não-Direcionado e Não-Ponderado

Neste teste, foi criada uma instância da classe `Graph` com 5 vértices e sem direção nas arestas. As operações realizadas incluem a inserção de arestas entre os vértices 0-1 e 0-2, além de consultas sobre o número de vértices, arestas e a existência de conexões específicas. A saída confirma que as arestas são bidirecionais, ou seja, uma conexão entre 0 e 1 implica uma conexão entre 1 e 0. Essa estrutura é útil para modelar relações simétricas.

Grafo Direcionado e Não-Ponderado

Neste caso, foi criada uma instância da classe `Graph` com 5 vértices e direção nas arestas. As operações incluem a inserção de arestas direcionais (como 0->1 e 1->3), consultas sobre conexões direcionais e a manipulação da estrutura do grafo, como a inserção de um novo vértice e a remoção de arestas e vértices. A saída demonstra que as conexões são unidirecionais, ou seja, uma aresta de 0 para 1 não implica uma conexão de 1 para 0. Além disso, o método `print()` exibe a lista de adjacência atualizada após cada operação, confirmando a consistência da estrutura.

Grafo Direcionado e Ponderado

Para este teste, foi utilizada a classe `WeightedGraph` com 5 vértices e direção nas arestas. As operações realizadas incluem a inserção de arestas direcionais com pesos (como 0->1 com peso 2.5), a atualização de pesos (1->2 atualizado para 10.0) e a remoção de arestas. As consultas verificam a existência de conexões direcionais e os pesos associados. A saída confirma que os pesos são corretamente armazenados e atualizados, e que as conexões são unidirecionais. Esse tipo de grafo é ideal para modelar cenários onde as conexões possuem um custo associado e direção, como redes de transporte.

Grafo Não-Direcionado e Ponderado

Neste teste, foi criada uma instância da classe `WeightedGraph` com 5 vértices e sem direção nas arestas. As operações incluem a inserção de arestas ponderadas bidirecionais (como 0-1 com peso 2.5), a atualização de pesos (1-2 atualizado para 10.0) e a remoção de arestas. As consultas verificam a existência de conexões e os pesos associados. A saída demonstra que as conexões são bidirecionais e que os pesos são consistentes em ambas as direções. Esse tipo de grafo é útil para modelar relações simétricas com custos, como mapas de rotas.

3.1 Contribuições do Trabalho

Este projeto contribuiu para o aprendizado prático dos integrantes, consolidando conceitos teóricos de grafos e algoritmos. Além disso, o sistema desenvolvido pode servir como base para estudos futuros, permitindo a expansão com novas funcionalidades e algoritmos.

4 Conclusão

Este trabalho atingiu com sucesso seu objetivo principal de implementar e validar as quatro variantes fundamentais de estruturas de grafos (não-direcionado não-ponderado, direcionado não-ponderado, não-direcionado ponderado e direcionado ponderado) utilizando a linguagem C++. A implementação baseada em listas de adjacência mostrou-se eficiente e adequada para as operações básicas de manipulação de grafos, incluindo inserção, remoção e consulta de vértices e arestas.

Os resultados demonstraram que todas as estruturas implementadas apresentaram o comportamento esperado de acordo com a teoria de grafos, mantendo as propriedades fundamentais de cada tipo: simetria nas conexões para grafos não-direcionais, assimetria controlada para grafos direcionais, e consistência no armazenamento e recuperação de pesos para as variantes ponderadas. A abordagem orientada a objetos permitiu uma arquitetura modular e extensível, facilitando a manutenção do código e o reaproveitamento de componentes. A organização do projeto em diretórios distintos para cabeçalhos e implementações, aliada ao uso de controle de versão, contribuiu para a robustez do desenvolvimento.

Por fim, este projeto não apenas cumpriu seus objetivos educacionais, mas também resultou em uma base sólida para aplicações práticas em problemas reais que requerem modelagem por grafos, servindo como ponto de partida para desenvolvimentos mais complexos na área.

Declarações

Contribuição dos Autores

Gustavo contribuiu com a concepção e desenvolvimento da estrutura genérica de grafos. Mariana implementou o grafo não-direcionado e não-ponderado. Felipe desenvolveu o grafo direcionado e não-ponderado. Alice implementou o grafo não-direcionado e ponderado. Daniel desenvolveu o grafo direcionado e ponderado. Arthur realizou a validação e testes de todas as estruturas implementadas. Bruna documentou o projeto seguindo os padrões JBCS. Todos os autores revisaram e aprovaram o manuscrito final.

Conflito de Interesse

Os autores declaram que não possuem conflitos de interesse.

Disponibilidade de Dados e Materiais

Os conjuntos de dados e softwares gerados e/ou analisados durante o presente estudo estão disponíveis no repositório GitHub: https://github.com/marialmeida1/study-grafos_tp01.