







Trabalho Pratico 1 de Grafos - Implementação de Grafos

Augusto Stambassi Duarte  [Pontifícia Universidade Católica de Minas Gerais (PUC Minas) | asduarte@sga.pucminas.br]
Davi Cândido de Almeida  [Pontifícia Universidade Católica de Minas Gerais (PUC Minas) | davi.almeida@sga.pucminas.br]
Gabriela de Assis dos Reis  [Pontifícia Universidade Católica de Minas Gerais (PUC Minas) | gabriela.reis@sga.pucminas.br]
Lucas Carneiro Nassau Malta  [Pontifícia Universidade Católica de Minas Gerais (PUC Minas) | lcnmalta@sga.pucminas.br]
João Pedro Torres  [Pontifícia Universidade Católica de Minas Gerais (PUC Minas) | joao.torres.1060863@sga.pucminas.br]
Vitor Leite Setragini  [Pontifícia Universidade Católica de Minas Gerais (PUC Minas) | 1526536@sga.pucminas.br]

✉ Instituto de Ciências Exatas e Informática, Pontifícia Universidade Católica de Minas Gerais (PUC Minas), Av. Dom José Gaspar, 500, Coração Eucarístico, Belo Horizonte, MG 30535-901, Brasil.

Resumo. Este artigo detalha o processo de desenvolvimento e implementação de uma biblioteca para a manipulação de grafos, uma estrutura de dados fundamental na ciência da computação. Utilizando a linguagem C++, foi desenvolvida uma estrutura flexível que suporta grafos direcionados e não-direcionados, ponderados e não ponderados. A arquitetura do projeto baseia-se em uma interface *IGrafo*, garantindo um contrato comum para duas implementações concretas: uma utilizando matriz de adjacências (*GrafoMatriz*) e outra com lista de adjacências (*GrafoLista*). Enquanto a implementação por matriz de adjacências foi focada em grafos simples, a abordagem com lista de adjacências oferece um escopo mais amplo, permitindo a representação de loops e arestas paralelas. Para validar e interagir com as estruturas implementadas, foi criado um menu de usuário que permite a manipulação dinâmica dos grafos. As funcionalidades incluem operações essenciais como a criação e remoção de grafos, vértices e arestas, bem como algoritmos clássicos para análise de conectividade e percurso, tais como a busca em largura (BFS), a busca em profundidade (DFS) e a determinação dos fechos transitivos direto e inverso de um vértice. O resultado é uma ferramenta robusta e extensível para a modelagem e análise de problemas computacionais baseados em grafos, demonstrando a aplicação prática de diferentes técnicas de representação e algoritmos fundamentais da área.

Keywords: Grafos, Matriz de Adjacências, Lista de Adjacências, Algoritmos em Grafos, C++.

1 Introdução

A Teoria dos Grafos é um ramo fundamental da matemática e da ciência da computação que se dedica ao estudo das relações entre objetos. Um grafo, em sua essência, é uma estrutura composta por um conjunto de vértices (ou nós) e um conjunto de arestas (ou arcos) que conectam pares desses vértices. Essa abstração simples é extremamente poderosa, permitindo modelar uma vasta gama de problemas do mundo real, como redes sociais (onde usuários são vértices e amizades são arestas), mapas de cidades (cruzamentos como vértices e ruas como arestas) e a própria estrutura da internet. [Sedgewick, 2002]

Para que um computador possa processar e analisar um grafo, é necessário traduzir sua estrutura abstrata para uma representação de dados concreta. A escolha de como armazenar as informações de vértices e arestas na memória é uma decisão crucial de projeto. As duas abordagens mais clássicas para esta tarefa são a Matriz de Adjacências, que utiliza uma matriz bidimensional para registrar a existência de uma aresta entre cada par de vértices, e a Lista de Adjacências, que para cada vértice mantém uma lista de todos os seus vizinhos. A escolha entre elas não é trivial e impacta diretamente a eficiência, em termos de tempo de execução e uso de memória, dos algoritmos que operam sobre o grafo.

O objetivo deste trabalho é, portanto, trazer uma solução para a implementação de uma biblioteca de software em C++ que ofereça ambas as representações. A biblioteca foi desenvolvida para ser flexível, suportando a criação e manipu-

lação de grafos direcionados e não-direcionados, que podem ou não ter pesos associados às suas arestas. Adicionalmente, foram implementados algoritmos fundamentais de percurso, como a Busca em Largura (BFS) e a Busca em Profundidade (DFS), permitindo a validação e a exploração das estruturas de dados criadas.

Este artigo está organizado da seguinte forma: a Seção 2 detalha a arquitetura do software e as decisões de implementação por trás das classes *GrafoMatriz* e *GrafoLista*. A Seção 3 apresenta os experimentos conduzidos para validar as funcionalidades e os resultados obtidos. Por fim, a Seção 4 sumariza as conclusões do trabalho e aponta direções para futuras expansões.

2 Implementação

A solução foi desenvolvida na linguagem C++, aproveitando seus recursos de orientação a objetos, templates para programação genérica e a robusta Biblioteca Padrão (STL). A arquitetura do projeto foi desenhada para ser extensível e modular, separando a interface abstrata das implementações concretas, o que facilita a manutenção e a adição de novas formas de representação de grafos no futuro.

2.1 A Interface

O pilar da arquitetura é a classe abstrata *IGrafo*, que funciona como um contrato para todas as implementações de

grafos. Conforme o código, ela foi definida como um template (`template <typename TVertice>`), garantindo que a biblioteca possa ser utilizada com diferentes tipos de dados para identificar os vértices (e.g., `int`, `std::string`).

A interface estabelece um conjunto de métodos virtuais puros que toda classe de grafo deve implementar, incluindo:

- **Operações de Tamanho:** `getQuantidadeVertices()` e `getQuantidadeArestas()`.
- **Manipulação:** Métodos para adicionar e remover vértices e arestas. Foram incluídas sobrecargas para a adição de arestas ponderadas e não ponderadas. Exemplo dos métodos: `adicionarVertice(TVertice v)`, `adicionarAresta(TVertice origem, TVertice destino, int peso)`, `removerVertice(TVertice v)`, `removerAresta(TVertice origem, TVertice destino)`.
- **Consulta:** Funções para verificar a existência de vértices e arestas, obter a lista de vizinhos de um vértice e calcular os fechos transitivos direto e inverso. Como os métodos: `getVizinhos(TVertice v)`, `fechoTransitivoDireto(TVertice v)` e `fechoTransitivoInverso(TVertice v)`.
- **Visualização:** Um método `imprimir()` para exibir o estado atual do grafo.

Essa abordagem assegura que qualquer implementação de grafo seja polimórfica e ofereça um conjunto consistente de funcionalidades.

2.2 Representação por Matriz de Adjacências

A primeira implementação concreta do contrato *IGrafo* é a classe *GrafoMatriz*, que utiliza a representação por matriz de adjacências. Esta classe especializa a interface para vértices do tipo `int`, em que cada vértice é identificado por um índice inteiro de 0 a $N-1$, sendo N o número total de vértices.

2.2.1 Estrutura de Dados

O núcleo da classe é a *matrizAdjacencias*, que representa uma matriz da forma `std::vector<std::vector<int>>`, na qual a posição `matrizAdjacencias[i][j]` armazena o peso da aresta entre o vértice i e o vértice j . Por convenção, o valor 0 indica a ausência de uma aresta. Além disso, estruturas de dados paralelas são utilizadas para armazenar metadados adicionais, como o *hash map* com os rótulos de vértices (`std::unordered_map<int, string>`) e a matriz auxiliar com os rótulos de arestas (`std::vector<std::vector<string>>`).

2.2.2 Construção e Configuração

Uma característica central desta implementação é que o número de vértices do grafo é fixo e definido em seu construtor. Além do tamanho, o construtor recebe um conjunto de *flags* booleanas (*direcionado*, *arestaPonderada*, etc.) que configuram o comportamento do grafo em tempo de execução. Essa alta configurabilidade permite que uma única classe modele diferentes tipos de grafos, mas impõe uma natureza estática à estrutura.

2.2.3 Análise das Operações

A representação por matriz oferece vantagens e desvantagens claras:

- **Eficiência em Consultas de Arestas:** A verificação de existência (*existeAresta*) e a obtenção do peso de uma aresta são operações de alta performance, com complexidade de tempo constante, $O(1)$.
- **Custo de Memória:** O espaço de armazenamento é de $O(V^2)$, onde V é o número de vértices. Isso torna a implementação inadequada para grafos grandes e esparsos (com poucas arestas).
- **Operações em Vértices:** A natureza estática da matriz torna a adição ou remoção de vértices uma operação computacionalmente cara, exigindo a reconstrução da estrutura e resultando em uma complexidade de $O(V^2)$.
- **Algoritmos:** A classe implementa algoritmos de busca de forma inteligente. O método *buscas()*, por exemplo, aplica uma Busca em Largura (BFS) para grafos não ponderados e o algoritmo de Dijkstra para grafos ponderados.

Em suma, a *GrafoMatriz* é uma implementação robusta e ideal para grafos densos e de tamanho predefinido, onde a consulta rápida por arestas é a principal prioridade.

2.3 Representação por Lista de Adjacências

A segunda implementação, *GrafoLista*, adota a abordagem de lista de adjacências com um design fortemente orientado a objetos e uma natureza dinâmica. Ela implementa a interface *IGrafo<Vertice>*, utilizando uma classe *Vertice* para encapsular as propriedades de cada nó.

2.3.1 Estrutura de Dados

A estrutura é composta por três classes principais:

1. **Vertice:** Um objeto que representa um vértice, contendo não apenas seu identificador (para diferenciar vértices com pesos e rótulos iguais), mas também, opcionalmente, um peso e um rótulo.
2. **NoVertice:** O bloco de construção da lista. Cada *NoVertice* contém um *Vertice* principal e uma lista simplesmente encadeada (`std::forward_list<Vertice>`) de todas as arestas que partem dele. Uma decisão de projeto notável é que as propriedades da aresta (peso e rótulo) são armazenadas no objeto *Vertice* de destino dentro dessa lista. Além disso, a decisão de utilizar uma lista encadeada deve-se à facilidade de inserção de novas arestas.
3. **GrafoLista:** A classe principal, que gerencia um `std::vector<NoVertice>` como a estrutura central. O vetor permite acesso em tempo constante, ou seja, em complexidade $O(1)$, a qualquer *NoVertice* a partir do identificador de seu vértice principal.

2.3.2 Natureza Dinâmica e Flexível

Diferente da matriz, o *GrafoLista* não requer um número de vértices predefinido. Vértices podem ser adicionados dinamicamente através do método *adicionarVertice*, que anexa

um novo *NoVertice* ao final do vetor principal (*listaPrincipal*). Além disso, a estrutura suporta nativamente arestas paralelas e loops, bastando adicionar o mesmo vértice múltiplas vezes na lista de adjacências.

2.3.3 Análise das Operações

As trocas de eficiência são quase o oposto da *GrafoMatriz*:

- **Eficiência de Memória e Adição:** O uso de espaço é de $O(V+E)$, ideal para grafos esparsos. Ademais, adicionar um vértice ou uma aresta são operações muito eficientes, com complexidade de tempo amortizada de $O(1)$.
- **Busca de Vizinhos:** A operação *getVizinhos* é ótima, com complexidade proporcional ao número de vizinhos do vértice, $O(\text{grau}(v))$.
- **Custo em Consultas:** Verificar a existência de uma aresta (*existeAresta*) requer uma busca linear na lista de adjacências do vértice de origem, custando $O(\text{grau}(v))$. A operação mais custosa é a busca por predecessores (*fechoTransitivoInverso*), que, na ausência de uma lista de adjacências invertida, exige uma varredura completa por todas as arestas do grafo, com complexidade $O(V+E)$.

O *GrafoLista* se destaca pela sua eficiência de memória e flexibilidade para modelar grafos dinâmicos e esparsos, onde a estrutura se altera com frequência e as operações de travessia são mais comuns que as de verificação de arestas isoladas.

2.4 Visualização das Estruturas e Funcionalidades Principais

O projeto em questão permite uma série de operações sobre as estruturas definidas, sendo estas operações controladas por um menu de controle de visão. O ponto de entrada do programa se encontra na função *main*, onde o usuário pode escolher entre as implementações via **Matriz de Adjacências** ou **Lista de Adjacências**.

A implementação *MenuMatriz*, por exemplo, fornece ao usuário a possibilidade de criar um grafo, adicionar ou remover vértices e arestas, consultar vizinhos, percorrer fechos transitivos diretos e inversos, bem como realizar buscas em profundidade (DFS) e em Largura (BFS), sendo cada uma dessas operações é organizada em um submenu específico.

De maneira similar, a classe *MenuLista* organiza operações equivalentes para a implementação via lista de adjacências, respeitando as diferenças estruturais dessa abordagem.

3 Experimentos e Análise de Resultados

Para validar as implementações e demonstrar a funcionalidade das classes *GrafoMatriz* e *GrafoLista*, foi desenvolvida uma plataforma de testes interativa. Esta seção descreve o ambiente experimental, apresenta um estudo de caso detalhado e analisa os resultados obtidos.

3.1 Ambiente Experimental

A validação das estruturas de dados foi conduzida por meio de duas interfaces de linha de comando: *MenuMatriz* e *MenuLista*. Cada uma atua como um programa para sua respectiva classe de grafo, permitindo ao usuário interagir com a biblioteca em tempo real.

A metodologia adotada foi a seguinte:

1. **Instanciação Dinâmica:** O usuário pode criar uma instância de um grafo, especificando em tempo de execução todas as suas propriedades (direcionado, ponderado, rotulado, etc.), conforme a flexibilidade oferecida pelos construtores das classes.
2. **Manipulação Interativa:** Após a criação, o menu oferece opções para executar todas as operações fundamentais de manipulação, como adicionar e remover arestas.
3. **Execução de Algoritmos:** O usuário pode invocar os algoritmos implementados (Busca em Largura, Busca em Profundidade, Fechos Transitivos) a partir de um vértice de sua escolha e observar os resultados diretamente no console.

Esse ambiente permitiu uma verificação sistemática e funcional de cada método exposto pela interface *IGrafo*, garantindo que ambas as implementações se comportassem conforme o esperado.

3.2 Estudo de Caso: Matriz de Adjacências com Grafo Direcionado e Rotulado

Para ilustrar o processo de validação, foi utilizado um grafo padrão gerado pela função *criarGrafoPadrao()* da classe *MenuMatriz*. A estrutura deste grafo de teste é definida da seguinte forma:

- **Tipo:** Grafo Direcionado.
- **Vértices:** 4 vértices, identificados pelos inteiros $\{0, 1, 2, 3\}$ e rotulados como $\{ "A", "B", "C", "D" \}$, respectivamente.
- **Arestas:** 4 arestas, todas com rótulos específicos:
 - $0 \rightarrow 1$ ("Rua 1")
 - $0 \rightarrow 2$ ("Rua 2")
 - $1 \rightarrow 2$ ("Avenida Principal")
 - $2 \rightarrow 3$ ("Ponte")

Após a instanciação, a estrutura do grafo foi primeiramente confirmada através da função de impressão.

A validação funcional prosseguiu com a execução de algoritmos-chave. Ao consultar os vizinhos do vértice 0 ("A"), o sistema corretamente retornou o conjunto $\{1, 2\}$, correspondendo às suas duas arestas de saída. O teste do fecho transitivo direto, também a partir do vértice 0, demonstrou a capacidade do algoritmo de percorrer o grafo, identificando todos os nós alcançáveis e resultando no conjunto completo $\{0, 1, 2, 3\}$.

Finalmente, a lógica de busca por predecessores foi confirmada com o cálculo do fecho transitivo inverso para o vértice 2 ("C"), que produziu o resultado esperado $\{0, 1, 2\}$, consolidando a verificação das principais funcionalidades da biblioteca.

3.3 Análise dos Resultados

Os experimentos conduzidos por meio dos menus interativos foram cruciais para a validação da biblioteca. A execução sistemática das operações de criação, manipulação e consulta em diversos cenários de teste permitiu verificar a correteza funcional de ambas as implementações, GrafoMatriz e GrafoLista.

Para cada operação testada, o resultado observado no console foi comparado com o resultado teórico esperado. Em todos os casos, a saída do programa correspondeu precisamente ao que a teoria dos grafos dita, desde a listagem de vizinhos até o cálculo de fechos transitivos. Desta forma, os experimentos cumpriram seu objetivo primário de certificar que a biblioteca implementa de maneira fiel e confiável os conceitos teóricos e os algoritmos propostos.

4 Conclusão

Este trabalho apresentou o projeto e a implementação de uma biblioteca de software em C++ para a criação e manipulação de grafos. O objetivo principal, de suportar múltiplas representações de dados, foi alcançado através de uma arquitetura flexível, fundamentada em uma interface abstrata (*IGrafo*) e duas implementações concretas: uma baseada em matriz de adjacências (*GrafoMatriz*) e outra em lista de adjacências (*GrafoLista*). A solução desenvolvida demonstrou-se robusta, suportando grafos direcionados e não-direcionados, com ou sem pesos e rótulos em seus vértices e arestas.

Os experimentos realizados, por meio de uma interface de testes interativa, validaram a correção funcional de ambas as implementações. Todas as operações de manipulação e os algoritmos de percurso, como BFS, DFS e a determinação de fechos transitivos, apresentaram resultados consistentes com a teoria. O projeto, portanto, não apenas resultou em uma ferramenta funcional, mas também serviu como uma demonstração prática das vantagens e desvantagens inerentes a cada forma de representação, consolidando o aprendizado teórico.

Como trabalhos futuros, a biblioteca pode ser expandida com a inclusão de novos algoritmos para resolver problemas clássicos, como a busca por caminhos de menor custo entre vértices ou a análise de outras propriedades de conectividade do grafo. Melhorias na interface do usuário e a implementação de funcionalidades para salvar e carregar grafos a partir de arquivos também representam evoluções naturais para o projeto, tornando-o uma ferramenta ainda mais completa para futuros estudos e aplicações.

Declarações

Authors' Contributions

Davi Cândido de Almeida e Lucas Carneiro Nassau Malta contribuíram para a concepção e o design fundamental do estudo, definindo a arquitetura do software e a interface *IGrafo*. A implementação do software foi dividida em dois subgrupos: a equipe da Matriz de Adjacências, composta por Davi Cândido de Almeida (caminhamentos e fechos transitivos), Lucas Carneiro Nassau Malta (operações estruturais) e Gabriela de Assis dos Reis (cam-

inhamentos e lógica de Dijkstra); e a equipe da Lista de Adjacências, formada por Augusto Stambassi Duarte (classes auxiliares e operações estruturais), João Pedro Torres (métodos de percurso) e Vitor Leite Setraghi (busca de vizinhos e menu de testes). O manuscrito foi escrito através de um esforço colaborativo. Todos os autores leram e aprovaram a versão final do manuscrito.

Competing interests

Os autores declaram que não há conflito de interesse.

Availability of data and materials

O código gerado e analisado durante o presente estudo estão disponíveis em: <https://github.com/DaviKandido/TP1-Grafos>.

References

Sedgewick, R. (2002). *Algorithms in C++: Volume 2, Part 5 - Graph Algorithms*. Addison-Wesley, Boston, 3rd edition.