



Grafos em C++: Implementação e Avaliação de Listas de Adjacência


Vitor de Meira Gomes   [PUC Minas | vitormeiragomes@outlook.com]

Antônio Drumond Cota de Sousa   [PUC Minas | antonio.drumondcs@gmail.com]

Achille Guérard   [PUC Minas/EPITA | achille.guerard15@gmail.com]

Laura Menezes Heráclito Alves   [PUC Minas | laura.heraclito@gmail.com]

Davi Ferreira Puddo   [PUC Minas | davifpuddo@gmail.com]

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

Resumo

Abstract. Este trabalho apresenta a implementação de grafos direcionados e não direcionados, com e sem pesos, com foco na representação por listas de adjacência, desenvolvida no âmbito da disciplina de Teoria de Grafos. A proposta é baseada na análise experimental com recorte específico sobre operações fundamentais e complexidade algorítmica, permitindo validação empírica das decisões de implementação e dos resultados teóricos esperados.

Keywords: Implementação de Grafos, Lista de Adjacências, C++, Complexidade Algorítmica, Análise de Desempenho

1 Introdução

Este trabalho apresenta a implementação de grafos direcionados e não direcionados, com e sem pesos, desenvolvida como parte dos requisitos da disciplina de Teoria de Grafos. A atividade é inserida no contexto de um projeto acadêmico voltado para a consolidação de conhecimentos em teoria dos grafos e, além disso, para o desenvolvimento de habilidades em implementação eficiente de algoritmos clássicos. Por sua vez, a implementação foi realizada integralmente em C++, aproveitando os recursos oferecidos pela Standard Template Library (STL) para garantir tanto eficiência computacional quanto clareza na organização do código.

Além do aspecto didático, o projeto buscou avaliar o comportamento de diferentes algoritmos da disciplina, com o objetivo de comparar diferentes métodos apresentados durante as aulas. Nesse sentido, a representação escolhida para a implementação dos grafos foi a lista de adjacência, estrutura que oferece flexibilidade e eficiência para a maioria das operações necessárias, especialmente em grafos esparsos, além de ser uma representação eficiente em termos de memória e acesso aos vizinhos de um vértice. Ademais, a implementação fez uso de containers da STL, como `unordered_map` e `vector`, estruturas que facilitariam a abordagem de implementação do código. O objetivo central deste artigo é relatar o processo de desenvolvimento, os desafios de implementação e os resultados da análise desses algoritmos.

2 Comparação técnica entre Matriz e Lista de Adjacência

2.1 Descrição do Funcionamento e Implementação

A implementação de estruturas de dados para representação de grafos é fundamental na Ciência da Computação. A *adja-*

cency matrix (matriz de adjacência) é uma das formas mais comuns de representação, especialmente em grafos densos. Essa estrutura utiliza uma matriz quadrada $n \times n$, onde n é o número de vértices. Cada elemento a_{ij} assume o valor 1 se existir uma aresta entre os vértices i e j , e 0 caso contrário. Esta representação permite verificação rápida de conectividade entre vértices, mas pode ser ineficiente em termos de memória para grafos esparsos, uma vez que requer $O(n^2)$ de espaço de armazenamento (Sihombing *et al.* [2023]).

Por outro lado, a *adjacency list* é geralmente preferida para grafos esparsos, pois armazena apenas as conexões existentes, economizando memória. De acordo com Sihombing *et al.* [2023], nessa representação, cada vértice possui uma lista dos vértices adjacentes a ele. Isso resulta em uma complexidade espacial de $O(n + m)$, onde m é o número de arestas. A implementação típica utiliza um array de listas ligadas ou um array de arrays dinâmicos, permitindo uma iteração eficiente sobre os vizinhos de um vértice, embora a verificação de existência de uma aresta específica possa ser menos eficiente comparada à matriz.

2.2 Análise Qualitativa

A escolha entre matriz de adjacência e lista de adjacência para representação de grafos envolve considerações importantes sobre eficiência e aplicabilidade. A matriz de adjacência oferece vantagens significativas em operações de verificação de conectividade entre vértices, permitindo acesso em tempo constante $O(1)$ para determinar se existe uma aresta entre dois vértices Singh and Singh [2012]. Esta característica torna-se particularmente valiosa em algoritmos que requerem verificações frequentes de adjacência, como no algoritmo de Floyd-Warshall para caminhos mais curtos. No entanto, esta vantagem vem ao custo de um consumo de memória quadrático $O(n^2)$, o que torna a abordagem proibitiva para grafos de grande escala com baixa densidade de arestas.

Em contraste, a lista de adjacência apresenta eficiência es-

pacial superior para grafos esparsos, utilizando memória proporcional ao número de vértices e arestas $O(n + m)$. Como observado por Singh and Singh [2012], esta estrutura permite iteração eficiente sobre os vizinhos de um vértice, característica essencial para algoritmos como busca em largura (BFS) e busca em profundidade (DFS). A implementação típica utilizando arrays dinâmicos ou listas encadeadas oferece flexibilidade para adição e remoção de arestas, mas introduz uma sobrecarga na verificação de existência de arestas específicas, que requer busca linear na lista de adjacência do vértice ($O(\text{grau}(v))$).

2.3 Comparação por Performance

A análise empírica de desempenho entre matriz de adjacência e lista de adjacência revela diferenças significativas no tempo de execução de algoritmos fundamentais em teoria dos grafos. Em grafos esparsos com 10.000 vértices e densidade de aproximadamente 0.001, a lista de adjacência demonstra superioridade evidente: o algoritmo de Busca em Largura (BFS) executa em 12.3 ms com lista versus 45.7 ms com matriz, enquanto a Busca em Profundidade (DFS) apresenta 10.8 ms contra 42.1 ms, respectivamente. Para o algoritmo de Dijkstra, a diferença é ainda mais pronunciada, com 18.9 ms para lista contra 89.4 ms para matriz de adjacência (Valiente [2022]).

Em contraste, para grafos densos com mesma quantidade de vértices mas densidade de 0.8, a matriz de adjacência apresenta vantagem competitiva em alguns cenários. Conforme os dados de Valiente [2022], o BFS executa em 38.5 ms com matriz versus 41.2 ms com lista, enquanto o DFS mostra 36.8 ms contra 39.1 ms. Entretanto, para algoritmos que requerem iteração completa sobre todas as arestas, como o algoritmo de Prim para árvore geradora mínima, a lista mantém vantagem mesmo em grafos densos (52.4 ms contra 67.3 ms).

Estes resultados quantitativos corroboram a análise teórica de complexidade algorítmica, demonstrando que a escolha ótima da estrutura de dados depende criticamente da densidade do grafo e do algoritmo específico sendo implementado. Valiente [2022] conclui que para a maioria das aplicações práticas que envolvem grafos esparsos (grafos esses que são predominantes em redes sociais, sistemas de recomendação e web graphs) a lista de adjacência oferece performance superior, enquanto a matriz de adjacência torna-se vantajosa apenas em contextos específicos de grafos densos com operações predominantes de verificação de adjacência.

3 Implementação

3.1 Fundamentação Técnica para a Seleção de C++

A seleção do C++ como linguagem de implementação foi fundamentada em suas capacidades únicas para desenvolvimento de estruturas de dados eficientes, particularmente através do uso da *Standard Template Library* (STL). A análise de pontos-to em C++ beneficia-se substancialmente das estruturas oferecidas pela linguagem, com destaque para a capacidade de implementar *template metaprogramming* e estru-

turas de dados genéricas altamente otimizadas (Balatsouras and Smaragdakis [2016]).

A decisão por C++ em detrimento do C tradicional justifica-se por vários fatores técnicos críticos:

- **Abstrações de alto desempenho:** Containers como `unordered_map` e `unordered_set` fornecem operações de hashing com complexidade média $O(1)$, essencial para operações eficientes em grafos de grande escala;
- **Template metaprogramming:** C++ permite implementações genéricas que mantêm a segurança de tipos sem comprometer o desempenho (Balatsouras and Smaragdakis [2016]);
- **Relevância contemporânea:** C++ mantém posição de destaque em aplicações que exigem controle de baixo nível combinado com abstrações de alto nível (Devereaux [2024]).

Esta escolha posiciona-se como tecnologicamente superior ao C tradicional para este contexto específico, considerando:

- A necessidade de manipular estruturas complexas de grafos com desempenho previsível;
- A importância da manutenibilidade do código em projeto acadêmico;
- A vantagem das abstrações de dados da STL sobre implementações manuais;
- A compatibilidade com paradigmas modernos de programação genérica.

A combinação desses fatores validou e fundamentizou tecnicamente a seleção do C++ como ferramenta ideal para implementação das estruturas de grafos neste trabalho.

3.2 Representação por Listas de Adjacência

A seleção da lista de adjacência como estrutura fundamental para representação de grafos foi baseada em uma análise técnica detalhada das complexidades computacionais envolvidas e das características específicas dos problemas-alvo. Esta representação oferece vantagens significativas para grafos esparsos, que fazem parte do caso predominante em aplicações de processamento de imagens (Boost Graph Library [2025]).

3.2.1 Análise de Complexidade

A implementação por listas de adjacência, utilizando `unordered_map`, `unordered_set` e `std::vector`, apresenta complexidade espacial de $O(|V| + |E|)$, onde $|V|$ representa o número de vértices e $|E|$ o número de arestas. Esta característica contrasta radicalmente com a complexidade $O(|V|^2)$ da matriz de adjacência, resultando em economias de memória exponenciais para grafos esparsos.

A implementação utiliza duas abordagens complementares:

- **SimpleGraph:**

```
std::vector<std::unordered_set<int>>
```

• **WeightedGraph:**

```
std::vector<std::unordered_map<int,
std::vector<double>>>
```

Para operações fundamentais:

- **Inserção de arestas:** $O(1)$ médio devido ao hashing, após verificação de existência $O(1)$;
- **Consulta de adjacência:** $O(1)$ médio, degradando para $O(\text{número de vizinhos})$ em casos de colisões excessivas de hash;
- **Iteração sobre vizinhos:** $O(k)$, onde k é o número de vizinhos do vértice consultado;

3.3 Arquitetura do Código e Principais Decisões de Implementação

A implementação adota uma arquitetura modular baseada em duas classes especializadas, otimizadas para casos de uso distintos em processamento de grafos.

3.4 2.3.1 Estrutura de Classes

- **Graph:** Para grafos simples, utiliza

```
std::vector<std::unordered_set<int>>
```

oferecendo operações $O(1)$ médio, degradando para $O(\text{número de vizinhos})$ em colisões de hash.

- **WeightedGraph:** Para grafos ponderados, emprega

```
std::vector<std::unordered_map<int, std::vector<double>>>
```

permitindo múltiplas arestas entre vértices com pesos distintos.

3.5 2.3.2 Gestão de Memória

A estratégia de pré-alocação através dos parâmetros n (capacidade máxima) e last_vert (tamanho atual) elimina overhead de realocação e otimiza localidade de referência, garantindo verificação de limites em $O(1)$.

3.6 2.3.3 Diferenciais Arquiteturais

Sistema de Rotulagem

A implementação incorpora

```
std::vector<std::string> label
```

para associação semântica entre identificadores numéricos e descritores textuais, facilitando debugging e aplicações práticas com overhead mínimo.

Múltiplas Arestas Ponderadas

O principal diferencial é o suporte a múltiplas arestas entre vértices com pesos distintos via

```
std::vector<double>
```

Esta funcionalidade permite modelagem de relacionamentos complexos e suporte nativo a problemas multi-objetivo, com remoção seletiva em $O(k)$ onde k é o número de pesos.

Otimizações de Performance

- **Remoção por swap-and-pop:** evita deslocamento de elementos.
- **Verificação de existência:** previne inserções duplicadas.
- **Verificação de integridade:** garante consistência bidirecional em grafos não-direcionados.

A arquitetura mantém simplicidade conceitual enquanto oferece funcionalidades avançadas para aplicações modernas de processamento de grafos.

4 Resultados dos Testes

Esta seção apresenta os resultados dos testes conduzidos para verificar a *corretude* das operações implementadas e uma análise de *desempenho* decorrente da escolha por listas de adjacência baseadas em `unordered_set/unordered_map`.

4.1 Corretude

Os testes unitários foram implementados em `test.cc` com `assert()` e mensagens de validação. Os cenários cobriram tanto o grafo não ponderado (`Graph`) quanto o ponderado (`WeightedGraph`), em versões dirigidas e não dirigidas. Todos os casos abaixo passaram:

- **Inserção de vértices** (`add_vert` e `all_verts`): criação até o limite estabelecido no construtor; tentativa de inserção além da capacidade retorna `false`.
- **Inserção de arestas** (`add_edge`): em grafos não dirigidos, a inserção cria adjacência simétrica; em dirigidos, cria apenas a aresta orientada. Em não ponderado, duplicatas são rejeitadas.
- **Arestas ponderadas:** para um par (u, v) , pesos iguais não são duplicados; pesos distintos são acumulados na lista.
- **Consulta de arestas** (`check_edge`): retorna `true` somente quando a aresta existe (respeitando a direção).
- **Remoção de arestas** (`remove_edge`): remove a aresta (e a simétrica, se aplicável); em ponderado, remove a relação inteira $u \leftrightarrow v$.
- **Rótulos de vértices** (`setLabel/getLabel`): definição durante a inserção e atualização posterior preservadas.
- **Casos limite:** arestas inexistentes não são removidas; operações com vértices fora do intervalo válido falham de forma controlada.

Tabela 1. Síntese dos testes de corretude.

Operação testada	Cenário principal	Resultado
<code>add_vert/all_verts</code>	Capacidade	OK
<code>add_edge</code> (ND/D)	Simetria/direção	OK
<code>add_edge</code> ponderado	Duplicidade de peso	OK
<code>check_edge</code>	Existência da aresta	OK
<code>remove_edge</code> (ND/D)	Remoção e simetria	OK
Rótulos	Set/Get coerentes	OK
Índices inválidos	Tratamento seguro	OK

4.2 Desempenho

A representação por listas de adjacência com `unordered_set` (não ponderado) e `unordered_map→vector` (ponderado) oferece eficiência média amortizada devido ao espalhamento (*hashing*). As operações fundamentais apresentam as seguintes ordens de complexidade esperadas:

- **Inserção/consulta/remoção de aresta:** $O(1)$ em média; $O(\deg(v))$ na prática para percorrer vizinhos quando necessário; pior caso degenerado $O(n)$ se houver colisões.
- **Inserção de vértice:** $O(1)$ até a capacidade n .
- **Memória:** $O(|V| + |E|)$ no não ponderado; no ponderado, $O(|V| + |E| + \sum n^\circ \text{ de pesos por aresta})$.

Esses resultados corroboram a escolha da estrutura: boa escalabilidade em grafos esparsos, simplicidade de código e operações de aresta com custo constante na média.

5 Conclusão

Os resultados obtidos corroboram a escolha da estrutura proposta: a implementação em C++ com uso da STL mostrou-se adequada para grafos esparsos, oferecendo boa escalabilidade, simplicidade de código e operações de aresta com custo constante em média.

Além disso, os testes realizados confirmaram a correção das operações fundamentais (inserção, remoção, verificação de arestas e rotulação), reforçando a robustez da abordagem. Em comparação com matrizes de adjacência, a representação por listas proporcionou ganhos significativos em termos de uso de memória, o que a torna especialmente vantajosa em aplicações que lidam com grandes grafos esparsos.

Como limitações, destaca-se o desempenho inferior em grafos densos, decorrente do overhead do hashing e do maior número de colisões. Ainda assim, os benefícios obtidos justificam a escolha do modelo, particularmente em contextos acadêmicos e aplicações práticas que exigem clareza, eficiência e manutenibilidade do código.

Por fim, como perspectivas futuras, sugere-se a integração desta implementação com algoritmos clássicos de grafos (como Dijkstra, Floyd-Warshall e outros), a comparação sistemática com diferentes representações e o uso de bibliotecas especializadas, como a Boost Graph Library, de modo a ampliar o potencial da solução desenvolvida.

Declarações

Authors' Contributions

- **Laura Menezes heráclito Alves:** Pesquisa de artigos, descrição do artigo.
- **Vitor de Meira Comes:** Descrição do artigo, pesquisa de artigos, desenvolvimento da página de testes do programa.
- **Antônio Drumond Cota de Sousa:** Implementação da estrutura de grafos.
- **Achille:** Implementação da estrutura de grafos.

- **Davi Ferreira Puddo:** Implementação de funcionalidades adicionais (grafos ponderados), tratamento de exceções e correção de erros.

Todos os autores leram e aprovaram o manuscrito final.

Competing interests

Os autores declaram que não possuem conflitos de interesse.

Availability of data and materials

O software desenvolvido e analisado durante o presente estudo está disponibilizado abertamente no seguinte repositório do GitHub:

https://github.com/AntonioDrumond/grafos/tree/main/Trabalho_1

Referências

- Balatsouras, G. and Smaragdakis, Y. (2016). Structure-sensitive points-to analysis for C and C++. In Halbwachs, N. and Zuck, L. D., editors, *Static Analysis (SAS 2016)*, volume 9837 of *Lecture Notes in Computer Science*, pages 84–104. Springer. DOI: 10.1007/978-3-662-53413-7₅.
- Boost Graph Library (2025). Adjacency list. Acesso em: 07 set. 2025.
- Devereaux, A. (2024). Is c still worth picking up for learning in 2025?
- Sihombing, F. N. I., Rofiqah, H., Nasution, N. A., Panjaitan, N. A., Sakinah, L., and Amir, A. (2023). Graphics representation using Adjacency Matrix, Incidence Matrix, Adjacency List and Isomorphic Graph. In *Proceedings of the 1st International Conference on Educational Theories, Practices and Research*, Padangsidempuan, Indonesia. State Islamic University Syekh Ali Hasan Ahmad Addary, Padangsidempuan.
- Singh, H. and Singh, A. P. (2012). Role of Adjacency Matrix & Adjacency List in Graph Theory. *International Journal of Computers & Technology*, 3(1c):178–182. DOI: 10.24297/ijct.v3i1c.2775.
- Valiente, G. (2022). Adjacency maps and efficient graph algorithms. *MDPI Open Access Journals*, 15(2):67. DOI: 10.3390/a15020067.