



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 experimentalmente o comportamento de algoritmos consagrados na disciplina, com o objetivo de comparar diferentes métodos apresentados durante as aulas. Nesse sentido, a representação escolhida para os grafos foi a lista de adjacências, 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 Standard Template Library (STL), como `unordered_map` e `vector`, estruturas que facilitarão 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 experimental desses algoritmos. O texto está organizado de forma que, na Seção 2, detalhamos as decisões de implementação, justificando a escolha pela lista de adjacências e explicando outras escolhas importantes na realização do trabalho. Em seguida, a Seção 3 descreve o porquê do uso de estruturas STL e como as utilizamos. Posteriormente, na Seção 4, apresentamos e analisamos os resultados dos testes de corretude e desempenho. Por fim, a Seção 5 conclui o trabalho, sintetizando as contribuições e aspectos abordados durante o trabalho.

2 Implementações

2.1 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 estruturas 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.

2.2 Estrutura de Dados: 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]).

2.2.1 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;

2.2.2 Desempenho em Grafos Esparsos X Densos

A solução implementada demonstra desempenho diferente quando comparamos grafos esparsos e densos, reflexo direto das características intrínsecas da representação por listas de adjacência. Em grafos esparsos, a implementação exibe excelente desempenho devido principalmente à localidade de referência proporcionada pelos acessos sequenciais aos vizinhos de cada vértice, à eficiência memória resultante do armazenamento proporcional apenas às conexões existentes.

Em contraste, a mesma implementação revela desvantagens significativas em grafos densos (onde $|E| \approx |V|^2$), cenário no qual o overhead de hashing é particularmente problemático, com colisões frequentes degradando o desempenho para $O(|E|)$. A falta de localidade nos acessos não sequenciais aos elementos e a progressiva perda da vantagem espacial, com consumo aproximando-se de $O(|V|^2)$, acabam por enfatizar ainda mais as limitações desta abordagem em ambientes e contextos de alta densidade.

A escolha ótima de estrutura depende do fator de densidade do grafo (Siek *et al.* [2001]). Para aplicações de processamento de imagens, onde a conectividade é tipicamente local e limitada (4 ou 8 vizinhos por pixel), a lista de adjacência ainda mantém superioridade incontestável.

2.3 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.

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.

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 $O(1)$.

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.

3 Análise de Estruturas: Fundamentação Técnica dos Containers

A implementação do sistema de grafos fundamenta-se em três estruturas de dados fundamentais da Standard Template Library (STL): `std::vector`, `std::unordered_map` e `std::unordered_set`. Esta seleção estratégica visa otimizar o equilíbrio entre desempenho computacional, eficiência de memória e flexibilidade operacional, atendendo às demandas específicas de processamento de grafos em cenários acadêmicos e aplicados. A combinação desses containers permite implementar eficientemente tanto grafos simples quanto ponderados, com operações de complexidade adequada para a maioria dos algoritmos fundamentais de teoria dos grafos.

3.1 Alocação Dinâmica e Acesso Indexado

O `std::vector` é empregado na implementação tanto para as listas de adjacência quanto para o armazenamento de rótulos dos vértices. Esta estrutura oferece alocação dinâmica de memória com acesso indexado de complexidade $O(1)$, permitindo que o grafo seja dimensionado conforme necessário durante a execução. Na representação da lista de adjacência, cada posição do vetor corresponde a um vértice e armazena seus vizinhos, enquanto o vetor de rótulos associa identificadores textuais aos vértices numericamente indexados. O uso de `std::vector` é fundamental para garantir flexibilidade na manipulação do grafo, além de otimizar operações de leitura e escrita sequenciais, essenciais para algoritmos de grafos que demandam alto desempenho e localidade de referência.

3.2 Mapeamento Eficiente para Grafos Ponderados

A estrutura `std::unordered_map` é utilizada exclusivamente na classe `WeightedGraph` para mapear vértices aos seus vizinhos e aos respectivos pesos das arestas. Cada elemento do vetor de adjacência é um `unordered_map` onde a chave representa o vértice vizinho e o valor é um vetor de pesos associados à aresta. Esta abordagem possibilita buscas,

inserções e remoções em tempo constante médio ($O(1)$), graças à implementação baseada em tabelas de dispersão (hash tables). O uso de `unordered_map` mostra-se especialmente eficiente em grafos esparsos, pois elimina a necessidade de percorrer listas extensas para localizar vizinhos ou pesos específicos, tornando as operações de acesso e atualização das informações de conectividade e ponderação altamente performáticas.

3.3 Conjuntos Eficientes de Vizinhança

Na classe `Graph`, o `std::unordered_set` é utilizado para representar o conjunto de vizinhos de cada vértice na lista de adjacência. Esta estrutura garante automaticamente a unicidade dos elementos e oferece operações de inserção, remoção e busca com complexidade média $O(1)$, beneficiando-se igualmente de tabelas de dispersão. O `unordered_set` é particularmente adequado para cenários que exigem verificação rápida da existência de arestas entre vértices e prevenção de conexões duplicadas. Sua integração sinérgica com o `std::vector` permite uma representação eficiente e escalável do grafo, facilitando significativamente a implementação de algoritmos que dependem de operações rápidas e frequentes sobre os conjuntos de vizinhos, como travessias e verificações de conectividade.

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
add_vert/all_verts	Capacidade	OK
add_edge (ND/D)	Simetria/direção	OK
add_edge ponderado	Duplicidade de peso	OK
check_edge	Existência da aresta	OK
remove_edge (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?
- Siek, J. G., Lee, L.-Q., and Lumsdaine, A. (2001). *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley Professional, 1st edition. eBook edition.