Journal of the Brazilian Computer Society, 2025.

Relatório do trabalho 1 da disciplina de Teoria de Grafos e Computabilidade

Iasmin Oliveira[**PUC Minas** | ***iasminfeo@gmail.com***]
Laura Persilva[**PUC Minas** | ***lapersilva@gmail.com***]
Arthur Signorini[**PUC Minas** | ***arthursigmiranda@gmail.com***]
Otávio Augusto[**PUC Minas** | ***otavioaugustoafm@gmail.com***]
Cauã Alves[**PUC Minas** | ***cauacostalves@gmail.com***]
Andriel Mark[**PUC Minas** | ***dieuhmark@icloud.com***]

Institute of Exact Sciences and Information, Pontifícia Universidade Católica, R. Dom José Gaspar, 500, Belo Horizonte, MG, Brazil.

Abstract. This report presents the implementation and experimental analysis of four graph representations: undirected/unweighted, directed/unweighted, undirected/weighted, and directed/weighted. Two data structures were considered—adjacency matrix and adjacency list—and three classical algorithms were evaluated: Depth-First Search (DFS), Breadth-First Search (BFS), and Dijkstra's shortest path. Experiments on sparse graphs with 5,000 vertices and 10,000 edges show that adjacency lists consistently outperform adjacency matrices, reducing running times by more than an order of magnitude for traversals and by over two hundred times for Dijkstra when combined with a priority queue. The results empirically validate the theoretical complexities and reinforce the central role of data structure choice in algorithmic performance.

Keywords: Grafos, Não-Direcionado, Direcionado, Ponderado, Não-Ponderado

1 Introdução

Ao nos aproximar da metade do curso de Ciência da Computação, é visto como necessário, senão imprescindível, o aprendizado de Teoria dos Grafos. Nosso objetivo nesse trabalho foi produzir os códigos de 4 grafos diferentes (Não-Direcionado Não-Ponderado, Direcionado Não-Ponderado, Não-Direcionado Ponderado e Direcionado Ponderado) e observar seu comportamento.

Abaixo, descrevemos as responsabilidades de cada um dos membros do grupo.

- Andriel Mark: Experimentos, resultados obtidos e documentação.
- Arthur Signorini: Produção dos códigos.
- Cauã Costa: Produção dos códigos.
- Iasmin Oliveira: Experimentos, resultados obtidos e documentação.
- Laura Persilva: Experimentos, resultados obtidos e documentação.
- Otávio Augusto: Produção dos códigos.

2 Análise da Implementação

Inicialmente, testamos duas abordagens de grafos, com criação a partir de uma matriz e a partir de uma lista de adjacências. Na seção de experimentos, conseguimos expressar uma inspeção de qual seria a melhor opção entre ambos, mas, nesta análise de implementação, assumiremos apenas a de lista de adjacências.

2.1 Grafo Não-Direcionado Não-Ponderado

2.1.1 Matriz

A implementação do grafo não direcionado e não ponderado por matriz de adjacência utilizou uma estrutura simples e direta, baseada em uma matriz booleana (bool**) que representa a existência ou ausência de arestas entre pares de vértices. Como se trata de um grafo não direcionado, a inserção de uma aresta entre dois vértices é refletida simetricamente na matriz, com a marcação de ambas as posições [i][j] e [j][i] como true. A validação de entrada evita loops e restrições de vértices inválidos, assegurando a integridade de um grafo simples. Essa forma de representação favorece a clareza visual e o acesso rápido às conexões, sendo adequada para grafos densos ou de tamanho reduzido.

2.1.2 Lista

Na implementação do grafo não direcionado e não ponderado por lista de adjacência, optou-se por representar o grafo como um vetor de listas, onde cada lista contém os vizinhos de um vértice. Como as arestas são bidirecionais, a adição é feita nos dois sentidos: ao inserir (u, v), também se adiciona (v, u). Essa estrutura é muito mais eficiente em termos de memória, especialmente em grafos esparsos, e oferece uma forma dinâmica e flexível de percorrer os vizinhos de um vértice. A simplicidade dos dados armazenados torna a estrutura leve, e sua construção é rápida e intuitiva.

2.2 Grafo Direcionado Não-Ponderado

2.2.1 Matriz

Para o grafo direcionado e não ponderado por matriz, a implementação mantém a estrutura booleana para representar a existência de arestas, porém elimina a necessidade de simetria ao adicionar conexões: a marcação ocorre apenas na posição [u][v], respeitando a direcionalidade da aresta. Essa mudança torna a matriz assimétrica, condizente com a definição do grafo. A simplicidade da matriz ainda oferece rápida verificação de conectividade entre dois vértices, mas sua limitação de espaço continua presente em casos onde o número de arestas é pequeno em relação ao número de vértices.

2.2.2 Lista

No grafo direcionado e não ponderado por lista, a lista de adjacência é construída com vetores simples de inteiros, sem necessidade de duplicidade de inserção, já que as conexões são unilaterais. Cada vértice armazena apenas os vértices para os quais possui saída direta, o que simplifica a estrutura e reflete com fidelidade a natureza direcionada do grafo. Essa implementação é especialmente eficiente em casos onde há poucas conexões e a direção importa, como em representações de fluxos de dados ou dependências.

2.3 Grafo Não-Direcionado Ponderado

2.3.1 Matriz

No grafo não direcionado e ponderado por matriz, a estrutura da matriz foi adaptada para armazenar mais informações por aresta, utilizando uma struct que inclui campos para indicar a existência da aresta, seu peso e um marcador de visitação. A simetria na inserção das arestas é mantida, já que a natureza não direcionada do grafo exige que o peso e a presença da aresta sejam refletidos em ambas as direções. Essa abordagem garante uma representação fiel dos pesos, com clareza na leitura e visualização das conexões entre os vértices, além de preparar o grafo para futuras aplicações de algoritmos como Prim ou Kruskal, que dependem dos pesos. No entanto, assim como no caso não ponderado, o uso de matriz pode desperdiçar memória em grafos esparsos.

2.3.2 Lista

O grafo não direcionado e ponderado por lista de adjacência acrescenta à estrutura anterior a capacidade de armazenar o peso das arestas. Utilizando listas de pares (v, peso) em cada posição da lista de adjacência, cada vértice armazena os vizinhos e os respectivos custos de conexão. A bidirecionalidade continua sendo respeitada ao adicionar o mesmo par inverso na lista do vértice oposto. Essa estrutura se mostra bastante eficiente, combinando economia de memória com a expressividade necessária para representar os pesos das ligações. É uma escolha adequada para a maioria dos algoritmos clássicos de grafos ponderados.

2.4 Grafo Direcionado Ponderado

2.4.1 Matriz

Já o grafo direcionado e ponderado por matriz evolui a estrutura booleana para uma matriz de structs com campos para peso e existência de aresta, semelhante ao não direcionado ponderado, mas sem simetria na adição. A estrutura permite mapear com precisão o sentido e o custo de cada ligação entre os vértices, permitindo o uso em aplicações mais avançadas que dependem de grafos direcionados com pesos, como redes de fluxo ou algoritmos de caminhos mínimos. A matriz oferece uma representação clara e direta, mas pode ser ineficiente para grafos com poucos arcos em relação ao total de vértices.

2.4.2 Lista

Por fim, o grafo direcionado e ponderado por lista de adjacência adota uma estrutura baseada em vetores de listas de pares (v, peso), onde cada entrada representa uma aresta saindo de um vértice para outro, com um determinado custo. Essa representação é bastante eficiente em termos de espaço, adequada para grafos esparsos, e permite acesso direto às conexões reais do grafo sem desperdício de memória. Além disso, a organização por pares facilita diretamente a aplicação de algoritmos como Dijkstra e Bellman-Ford. A estrutura se mostra robusta, clara e eficiente para representar grafos direcionados com pesos, sendo a mais indicada para aplicações complexas que envolvam otimizações ou busca de caminhos mínimos.

3 Experimentos

Os experimentos realizados demonstram, de forma concreta e quantificável, o impacto que a escolha da estrutura de dados exerce sobre o desempenho de algoritmos em grafos. Foram testados três algoritmos clássicos: Busca em Profundidade (DFS recursiva), Busca em Largura (BFS) e Dijkstra utilizando duas representações distintas de grafos: matriz de adjacência e lista de adjacência. Os testes foram aplicados sobre dois grafos com 5.000 vértices e 10.000 arestas, sendo um ponderado e outro não ponderado.

3.1 Representações: Matriz vs. Lista de Adjacência

A matriz de adjacência ocupa espaço fixo de $O(V^2)$, independentemente da quantidade de arestas presentes. Em grafos esparsos, isso resulta em alto desperdício de memória e tempo, pois a maioria das posições na matriz representa conexões inexistentes. Em contrapartida, a lista de adjacência armazena apenas as conexões reais, com uso de memória proporcional a $O(V + E)$, sendo muito mais eficiente nesse cenário.

A operação fundamental em algoritmos de travessia (obter os vizinhos de um vértice) é um exemplo claro dessa diferença:

- Matriz: exige percorrer toda a linha do vértice, com custo $O(V)$.

- Lista: acessa diretamente os vizinhos reais, com custo O(grau(u)).

3.2 Análise dos resultados

A DFS recursiva e a BFS exibiram tempos muito próximos dentro de cada representação, com pequenas variações decorrentes da implementação, como esperado. Ambos os algoritmos percorrem todos os vértices e arestas acessíveis, com complexidade $O(V^2)$ para matriz e $O(V + E)$ para a lista.

A diferença entre os dois está na estratégia de busca (profundidade vs. largura), mas não no custo total. O fator decisivo nos tempos observados está na representação do grafo. Com lista de adjacência, os tempos caíram de dezenas de milissegundos para frações de milissegundo, confirmando a eficiência dessa estrutura em grafos esparsos.

Já o algoritmo de Dijkstra realiza operações mais custosas: cálculo e atualização de distâncias mínimas, e seleção repetida do próximo vértice de menor distância. Na implementação com matriz, a busca linear pelo mínimo (em O(V)) em cada passo compromete seriamente a performance, resultando em tempo de execução de 615 ms.

Sua versão com lista de adjacência, associada a uma fila de prioridade (heap), atinge complexidade O(E log V), ideal para grafos esparsos. O resultado prático — execução em apenas 2 a 3 ms — confirma que a escolha de estrutura e algoritmos auxiliares (como heap) tem impacto direto e profundo na eficiência do algoritmo.

Os tempos de execução dos algoritmos nas duas representações podem ser resumidos nas tabelas abaixo.

Matriz

DFS REC		
	Não direcionado	Direcionado
Não ponderado	44 ms	43 ms
Ponderado	49 ms	51 ms
BFS		
	Não direcionado	Direcionado
Não ponderado	46 ms	45 ms
Ponderado	48 ms	52 ms
Dijkstra		
Ponderado (Não direcionado / Direcionado)		615 ms

Lista

DFS REC		
	Não direcionado	Direcionado
Não ponderado	943600 ns	580200 ns
Ponderado	1 ms	677700 ns
BFS		
	Não direcionado	Direcionado
Não ponderado	906400 ns	654500 ns
Ponderado	914800 ns	674200 ns
Dijkstra		
	Não direcionado	Direcionado
Ponderado	3 ms	2 ms

4 Conclusão

Os experimentos realizados neste trabalho demonstram de forma clara e concreta o impacto prático que a escolha da estrutura de dados exerce sobre o desempenho de algoritmos em grafos. Ao comparar algoritmos clássicos e suas diferentes formas de implementação foi possível validar conceitos teóricos de complexidade e observar diferenças significativas de desempenho.

Assim, os dados mostraram, de forma consistente, que para grafos esparsos (com muitas arestas ausentes em relação ao número de vértices), a lista de adjacência supera amplamente a matriz de adjacência em termos de eficiência. Esse ganho é especialmente perceptível nos algoritmos de travessia, como DFS e BFS, nos quais o uso da lista evita iterações desnecessárias sobre vértices desconectados. Os tempos de execução caíram de dezenas de milissegundos (com matriz) para frações de milissegundo (com lista), evidenciando uma melhoria de mais de uma ordem de grandeza.

No caso do algoritmo de Dijkstra, a diferença foi ainda mais acentuada. A versão com matriz, que exige varreduras completas a cada iteração, teve desempenho mais de 200 vezes inferior ao da versão com lista de adjacência e fila de prioridade. Esse resultado reforça a ideia de que as otimizações algorítmicas só atingem seu potencial máximo quando combinadas com estruturas de dados adequadas.

Além dos ganhos práticos, os experimentos também têm valor didático. Eles tornam tangível a diferença entre as complexidades assintóticas dos algoritmos, ilustrando com dados reais conceitos que muitas vezes permanecem abstratos. Mais do que isso, reforçam uma lição essencial da Ciência da Computação: o projeto de algoritmos deve estar sempre alinhado à escolha correta da estrutura de dados.

Do ponto de vista prático, os resultados sugerem que, ao lidar com grafos grandes e esparsos — como redes sociais, mapas viários, grafos de dependência ou hiperlinks da web — a lista de adjacência deve ser a estrutura preferencial, especialmente quando combinada com estruturas auxiliares como filas ou heaps, dependendo do algoritmo em questão.

Seria interessante, em algum outro momento, avaliar:

- Repetir os testes com grafos densos, para observar se os resultados se invertem;
- Avaliar outros algoritmos de caminhos mínimos, como Bellman-Ford ou A*;
- Estudar o impacto de paralelismo ou otimizações específicas de hardware (como o uso de cache com matrizes);
- Medir também o uso de memória, além do tempo de execução.

Declarations

Authors' Contributions

Andriel Mark, Iasmin Oliveira, and Laura Persilva were responsible for conducting the experiments, analyzing the results, and documenting the project. Arthur Signorini, Cauã Costa, and Otávio Augusto developed the source code for the graph algorithms used in the study.

All authors contributed to the project design, reviewed, and approved the final version of the manuscript.

Competing interests

The authors declare that they have no competing interests.

Acknowledgements

The authors would like to thank Prof. Silvio Jamil for his valuable teaching of the algorithms and scientific thinking, and the Laboratory of Computational Science for providing the test environment.

Data and Materials Availability

The source code for this work can be accessed at: Github Link. This is an implementation developed for academic purposes.