**UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO**

**CENTRO DE CIÊNCIAS EXATAS, NATURAIS E DA SAÚDE**

**DEPARTAMENTO DE COMPUTAÇÃO**

**MARCOS PAULO RODRIGUES ALVES**

# Algoritmo de Kruskal para Árvore Geradora Mínima: Análise de Desempenho

**ALEGRE - ES**

**2023**

MARCOS PAULO RODRIGUES ALVES

# Algoritmo de Kruskal para Árvore Geradora Mínima: Análise de Desempenho

Trabalho de conclusão de curso apresentado ao Departamento de Computação do Centro de Ciências Exatas, Naturais e da Saúde da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

ALEGRE - ES

2023

MARCOS PAULO RODRIGUES ALVES

# Algoritmo de Kruskal para árvore geradora mínima: análise de desempenho

Trabalho de Conclusão de Curso apresentado ao Departamento de Computação do Centro de Ciências Exatas, Naturais e da Saúde da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 08 de dezembro de 2023.

**COMISSÃO EXAMINADORA**

**Prof. Dr. Edmar Hell Kampke**
**Universidade Federal do Espírito Santo**
**Orientador**

**Prof. Dr. Dayan de Castro Bissoli**
**Universidade Federal do Espírito Santo**

**Prof. Dr. Rodrigo Freitas Silva**
**Universidade Federal do Espírito Santo**

*Aos meus pais.*

## AGRADECIMENTOS

Agradeço a Deus pela bênção de uma família incrível que me apoiou incondicionalmente ao longo desta intensa jornada acadêmica. À minha família, por ter acreditado no meu potencial desde o início sem medir esforços para tornar cada desafio mais alcançável.

Ao meu orientador, minha profunda gratidão por sua compreensão, apoio e ensinamentos durante toda a minha jornada acadêmica.

Agradeço aos meus colegas de república pela parceria, apoio e obstáculos que passamos juntos.

Estendo meus agradecimentos a todos que estiveram presentes de forma direta ou indireta e contribuíram para que este objetivo fosse alcançado.

**RESUMO**

Uma árvore geradora mínima (AGM) é uma subárvore de um grafo conectado e ponderado que abrange todos os vértices do grafo, mantendo a conexão entre eles, sem ciclos e possuindo o menor peso total possível. As AGM's têm aplicações em diversos campos, como redes de comunicação, logística, design de circuitos e planejamento de rotas, onde é necessário encontrar uma estrutura de conexão ótima entre pontos específicos do grafo. Algoritmos como o de Kruskal e Prim são comumente utilizados para encontrar uma árvore geradora mínima em um grafo ponderado. Portanto, este trabalho analisará o desempenho do algoritmo de Kruskal em diferentes implementações para resolver o problema da AGM. Foram comparadas abordagens clássicas e abordagens otimizadas pela estrutura *Union-Find*, além da abordagem com a estrutura de *buckets*. Para a execução dos testes foram usadas instâncias com variados tamanhos e densidades. O intuito é analisar a complexidade temporal dos algoritmos para identificar suas vantagens e limitações.

Palavras-chave: Algoritmo de Kruskal, Árvore Geradora Mínima, *buckets*, *Union-Find*.

## LISTA DE FIGURAS

Figura 1 — Cidade de Königsberg em 1736 com destaque amarelo nas pontes.....10
Figura 2 — Representação do Grafo para o problema das pontes de Königsberg ...11
Figura 3 — Representação do grafo das fibras óticas em um campus de uma universidade .....12
Figura 4 — Árvore Geradora Mínima das fibras óticas em um campus de uma universidade .....13
Figura 5 — Grafo G .....17
Figura 6 — Grafo simples.....17
Figura 7 — Grafo G .....18
Figura 8 — Grafo Desconexo .....19
Figura 9 — Lista de três grafos .....19
Figura 10 — Floresta.....20
Figura 11 — Árvore .....21
Figura 12 — Subgrafo H é um subgrafo gerador .....21
Figura 13 — Subgrafo H induzido pelo subconjunto $V' = \{d, e, f, g\}$ .....22
Figura 14 — Árvore geradora T.....22
Figura 15 — Grafo valorado .....23
Figura 16 — Árvores Geradoras .....24
Figura 17 — Grafo G ponderado.....29
Figura 18 — Estrutura de buckets.....29

## LISTA DE TABELAS

Tabela 1 — Configuração do ambiente de testes .....31
Tabela 2 — Instâncias criadas .....32
Tabela 3 — Tempo médio em milissegundos gasto na execução .....32

**SUMÁRIO**

1 INTRODUÇÃO ..... 10
1.1 O problema e sua importância ..... 12
1.2 Objetivos ..... 14
1.2.1 Objetivo geral ..... 14
1.2.2 Objetivos específicos ..... 14
1.3 Organização do Trabalho ..... 14
2 REFERENCIAL TEÓRICO ..... 16
2.1 Grafos ..... 16
2.2 Conexidade ..... 18
2.3 Árvores ..... 20
2.4 Subgrafos ..... 21
2.5 Grafos valorados ..... 22
2.6 Árvore geradora mínima ..... 23
3 METODOLOGIA ..... 25
4 RESULTADOS OBTIDOS ..... 31
5 CONCLUSÃO ..... 34
6 REFERÊNCIAS ..... 35

# 1 INTRODUÇÃO

Em 1736, na cidade de Königsberg (atual Kalingrado, Rússia), havia duas ilhas que, na época, eram ligadas entre si por uma ponte. As duas ilhas se ligavam ainda às margens norte e sul por mais seis pontes, conforme é exibido na Figura 1. Os moradores de Königsberg se questionavam sobre a possibilidade de encontrar um percurso que partisse de uma das margens, passasse uma única vez por cada uma das sete pontes, e retornasse ao ponto de partida (NETTO; JURKIEWICZ, 2017).

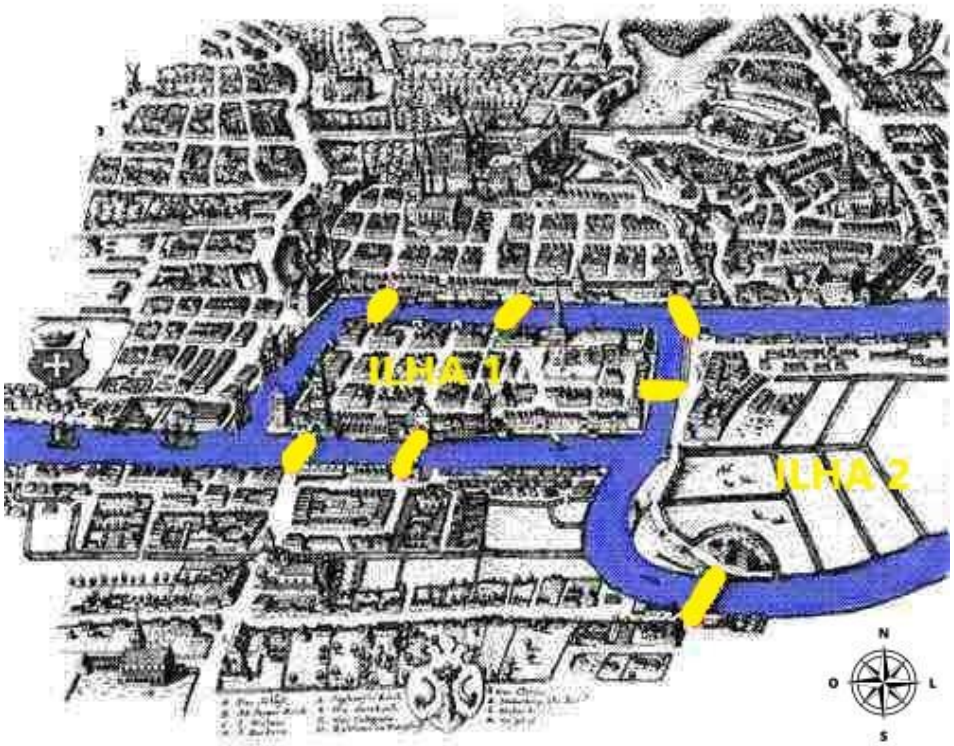


Figura 1 — Cidade de Königsberg em 1736 com destaque amarelo nas pontes

Leonhard Paul Euler (1707-1783) foi o ilustre matemático quem primeiro demonstrou a impossibilidade do desejado passeio por Königsberg. Euler observou que o número de passagens de uma das margens para as ilhas, ou entre ilhas, era sempre ímpar. Por esse motivo, para que o passeio fosse possível, cada região de Königsberg teria que ter um número par de pontes (NETTO, 2012).

Euler chegou a essa conclusão abstraindo o problema e o modelando de tal forma que as pontes fossem linhas (arestas), e as margens e ilhas fossem pontos (vértices),

conforme é apresentado na Figura 2. Essa forma de modelar com pontos e linhas posteriormente foi denominado de grafo, ou seja, Euler criou o primeiro grafo da história.

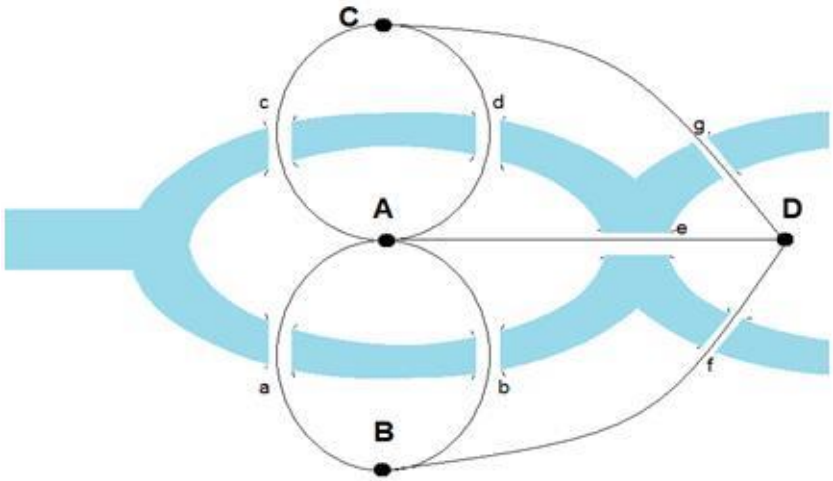


Figura 2 — Representação do Grafo para o problema das pontes de Königsberg

Esta modelagem do problema marca o nascimento da teoria dos grafos, que não se estabeleceu como teoria no século XVIII, pois não havia uma aplicação de grande importância naquela época, o que acarretou no esquecimento durante um século, até ser novamente aproveitada na física por Gustav Robert Kirchhoff, na química por Arthur Cayley, e pelo interesse de Francis Guthrie no teorema do mapa de quatro cores (NETTO; JURKIEWICZ, 2017).

Ao longo do século XX, o interesse pelo estudo dos grafos aumentou devido a problemas similares, o que serviu como estímulo para se criar um corpo teórico que modelasse de maneira geral as situações abordadas. E foi a partir da década de 1950, com a criação e o desenvolvimento dos computadores, que intensificou a utilização de modelos de grafos em busca de melhores soluções para problemas de projeto, organização e distribuição no âmbito da pesquisa operacional (NETTO; JURKIEWICZ, 2017).

Atualmente, a teoria dos grafos é um campo de interesse crescente, da qual aplicações vão desde problemas de localização e de traçado de rotas para diversos tipos de serviços, ao projeto de processadores eletrônicos, passando pelo

planejamento de horários, pelo estudo da estrutura do DNA e pelo projeto de códigos-fontes de programas (NETTO; JURKIEWICZ, 2017).

## 1.1 O problema e sua importância

Diversos problemas tem a necessidade de minimizar custos dentro de um conjunto de variáveis que compõe a solução, como exemplo, a instalação de fibras óticas em um *campus* de uma universidade. Já que a distância de fibra entre os prédios possui um custo associado, pode-se construir um grafo em que as linhas serão os cabos de fibra ótica e os pontos os prédios da universidade. Dessa maneira, pode-se obter um subconjunto dessas linhas, que conectem todos os prédios da universidade e que minimize o custo total dos cabos selecionados. Esse novo grafo formado por todos os pontos do grafo original e apenas pelas linhas desse subconjunto, é denominado de Árvore Geradora Mínima (AGM).

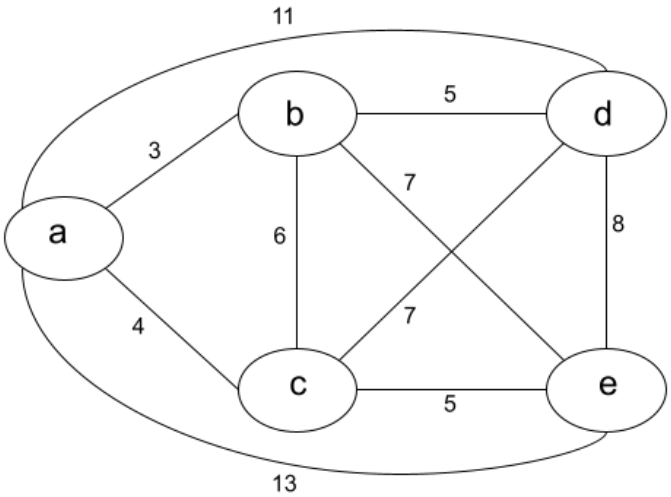


Figura 3 — Representação do grafo das fibras óticas em um *campus* de uma universidade

A Figura 3, ilustra a malha de fibras óticas em um *campus* de uma universidade, em que os vértices $a$, $b$, $c$, $d$ e $e$ são os prédios, e as linhas que ligam os vértices são o custo (distância) associado à fibra ótica.

Já na Figura 4, é ilustrado a AGM gerada a partir do grafo da Figura 3, em que se tem fibra ótica conectando todos os prédios com um custo mínimo de 17, que é a soma dos custos individuais das arestas da AGM.

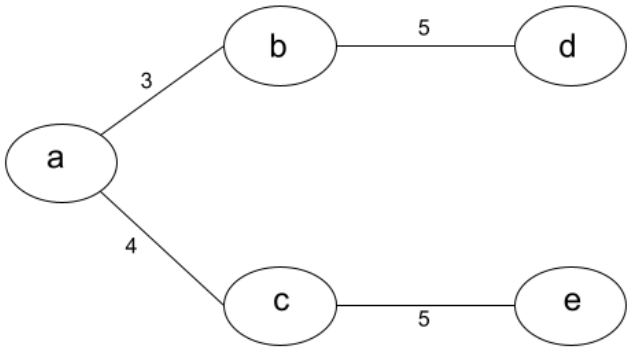


Figura 4 — Árvore Geradora Mínima das fibras óticas em um *campus* de uma universidade

Portanto, diante dessa e de outras situações, surgiu a motivação de se criar algoritmos que encontrassem a AGM de um determinado grafo.

O primeiro algoritmo para encontrar a árvore geradora mínima foi desenvolvido pelo cientista Otakar Boruvka em 1926, sendo que sua motivação foi desenvolver uma rede elétrica com o menor custo possível, mas que atendesse todos os moradores da região sul da cidade de Moravia, na República Tcheca. Posteriormente, outros cientistas como Joseph Kruskal e Robert Clay Prim resolveram criar seus próprios algoritmos, com o foco na eficiência em encontrar AGM's em qualquer situação (GRAHAM; HELL, 1985).

Todos esses algoritmos são gulosos, o que significa que o tempo de processamento é polinomial. Portanto, o problema da geração da árvore mínima pertence à classe de problemas com complexidade P, que são problemas computacionais que podem ser resolvidos por um computador real (GERSTING, 2017).

A complexidade do algoritmo implementado depende também de outros fatores como as estruturas de dados e os algoritmos de ordenação usados. Portanto, neste trabalho foi implementado o algoritmo de Kruskal em diferentes abordagens para se avaliar o desempenho e suas vantagens e desvantagens.

## 1.2 Objetivos

A seguir são descritos os objetivos geral e específicos relacionados a este trabalho.

### 1.2.1 Objetivo geral

O objetivo geral deste trabalho é avaliar o desempenho do algoritmo de Kruskal em diferentes abordagens de implementação.

### 1.2.2 Objetivos específicos

A. Estudar as diferentes formas de implementar o algoritmo de Kruskal descritas na literatura;
B. Implementar um gerador de instâncias de grafos para testar os algoritmos;
C. Implementar o algoritmo de Kruskal utilizando o algoritmo de ordenação *Merge Sort*;
D. Implementar uma nova versão do algoritmo de Kruskal com a estrutura *Union-Find*, que verifica a formação de ciclo.
E. Implementar a terceira versão do algoritmo de Kruskal, mas dessa vez sem algoritmos de ordenação e com estruturas denominadas *buckets*;
F. Executar testes de verificação nos algoritmos implementados e, se necessário, efetuar correções;
G. Avaliar o desempenho dos algoritmos nas instâncias geradas.

## 1.3 Organização do Trabalho

O restante deste trabalho está dividido da seguinte forma: o Capítulo 2 apresenta o referencial teórico necessário para o desenvolvimento deste trabalho; o Capítulo 3 detalha a metodologia a ser seguida; o Capítulo 4 apresenta os resultados obtidos; o

Capítulo 5 são as conclusões deste trabalho; e, por fim, o Capítulo 6 enumera as referências.

# 2 REFERENCIAL TEÓRICO

Este capítulo apresenta os principais conceitos da teoria dos grafos, conforme apresentado na literatura, em especial no livro de Nicoletti e Junior (2018). Estes conceitos são importantes para o entendimento do que foi proposto e para a implementação dos algoritmos deste trabalho.

## 2.1 Grafos

Segundo Nicoletti e Junior (2018), um grafo é um par ordenado que pode ser representado por $G = \big(V(G), A(G)\big)$, ou $G = (V, A)$, que consiste em dois conjuntos finitos:

- $V(G)$ ou $V$, é o conjunto de vértices do grafo, o qual é um conjunto não vazio de elementos;
- $A(G)$ ou $A$, é o conjunto de arestas do grafo, o qual é um conjunto de elementos que pode ser vazio.

Cada aresta $a \in A$ de um grafo $G$ é representado por um par não ordenado de vértice $(u, v)$, cada um chamado de vértice-extremidade de $a$. Vértices também são chamados de pontos ou nós. Os vértices que estão unidos por uma aresta, são ditos incidentes à aresta, e vice-versa. Dois vértices que estão unidos por uma aresta em comum são chamados de adjacentes ou vizinhos. Se duas ou mais arestas tem os mesmos vértices-extremidade, essas arestas são chamadas de arestas paralelas, ou arestas múltiplas. E uma aresta com vértices incidentes idênticos é chamada de laço ou *loop*. Para indicar o número de vértices de um grafo $(|V|)$, é usado o termo *ordem*, que é representado por $n$. E para indicar o número de arestas de um grafo $(|A|)$, é usado o termo *tamanho*, que é representado por $m$. Já o grau de um vértice $v$, denotado por $d(v)$, é o número de arestas de $G$ que são incidentes em $v$, contando cada *loop* duas vezes. Um vértice de grau 0, $d(v) = 0$, é, portanto, um vértice isolado.

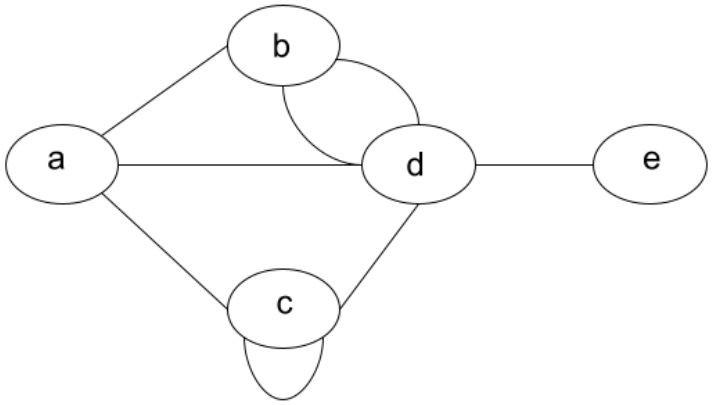


Figura 5 — Grafo G

A Figura 5 apresenta um exemplo de um grafo $G = (V, A)$ em que o conjunto de vértices é: $\mathrm{V} = \{\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}, \mathrm{e}\}$, e o conjunto de arestas é: $A = \{(a, b), (a, c), (a, d), (b, d), (c, c), (c, d), (d, b), (d, e)\}$. Assim, se analisarmos os vértices $a$ e $b$ separadamente, podemos observar que eles são incidentes a aresta $(a, b)$. Também é possível concluir que os vértices $a$ e $b$ são adjacentes, pois ambos são incidentes a mesma aresta. Por outro lado, os vértices $b$ e $c$ não são adjacentes, pois não há uma aresta incidente a eles. Além disso, podemos observar que as arestas $(b, d)$ e $(d, b)$ são arestas paralelas, pois ambas contêm os mesmos vértices-extremidades. E por fim, a aresta $(c, c)$ é um *loop*, e o grau $d(c) = 4$.

Um grafo é chamado de simples se não tem *loops* e nem arestas paralelas. Na Figura 6, podemos observar o exemplo de um grafo simples.

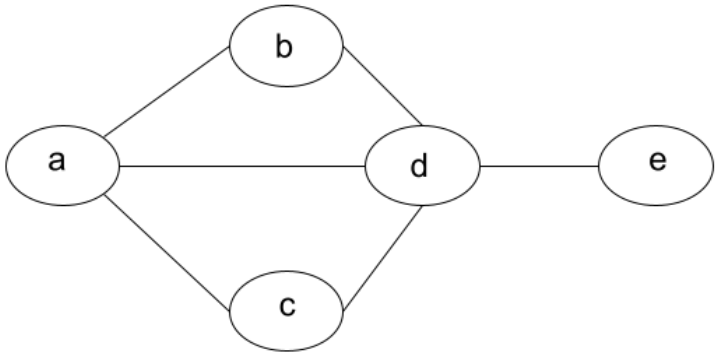


Figura 6 — Grafo simples

## 2.2 Conexidade

Um passeio em um grafo é uma sequência finita e não nula $W = v_0, (v_0, v_1), v_1, (v_1, v_2), v_2 \dots (v_{k-1}, v_k), v_k$, cujos elementos são, alternadamente, vértices e arestas incidentes. Sabe-se que o passeio $W$ é um passeio $v_0 - v_k$, ou um passeio de $v_0$ até $v_k$. Logo, o passeio é uma sequência de vértices adjacentes. Um passeio que começa e termina no mesmo vértice $v$ é chamado de passeio fechado, do contrário, é um passeio aberto.

Seja $G = (V, A)$ um grafo, e um passeio em $G$ dado pela sequência $W = v_0, (v_0, v_1), v_1, (v_1, v_2), v_2 \dots (v_{k-1}, v_k), v_k$, se as arestas $(v_0, v_1), (v_1, v_2), (v_{k-1}, v_k)$ são distintas, então $W$ é chamado de trilha. Uma trilha que começa e termina no mesmo vértice $v$ é chamada de trilha fechada ou circuito, caso contrário, é uma trilha aberta.

Considerando ainda no grafo $G$ e trilha $W$, se os vértices da trilha são distintos, então $W$ é chamado de caminho. No entanto, em um caminho, é permitido que o primeiro e o último vértice possam ser os mesmos. Um caminho que começa e termina no mesmo vértice é chamado de caminho fechado ou ciclo. Já um caminho que não é fechado é chamado de caminho aberto.

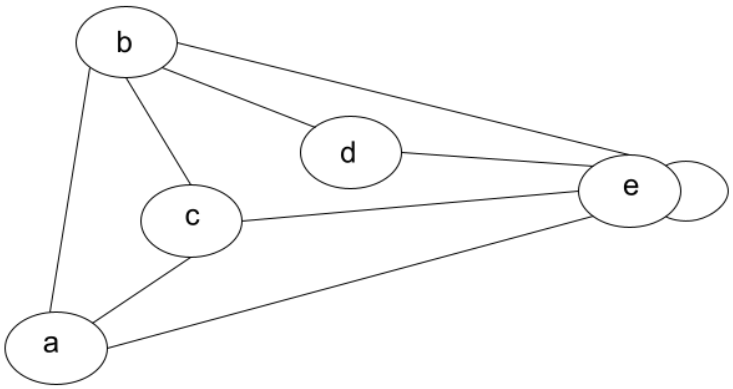


Figura 7 — Grafo G

A Figura 7 apresenta um grafo $G$ em que se pode concluir que a sequência $W_1 = a, (a, b), b, (b, e), e, (e, e), e, (e, b), b, (b, c), c$ é um passeio aberto. Já a sequência $W_2 = a, (a, b), b, (b, c), c, (c, a), a$ é um passeio fechado. Também, a sequência $W_3 =$

$a, (a, c), c, (c, b), b, (b, d), d, (d, e), e, (e, a), a$ é uma trilha fechada. E por fim, $W_4 = b, (b, d), d, (d, e), e, (e, c), c, (c, a), a$ é um caminho aberto.

Seja o grafo $G = (V, A)$ e $V = \{u, v\}$. Diz-se que o vértice $u$ está conectado ao vértice $v$ se existir um caminho de $u$ a $v$ em $G$. Logo, se quaisquer dois vértices de $G$ estão conectados, o grafo $G$ é chamado de grafo conexo. Caso contrário, $G$ é desconexo.

A Figura 7 é um exemplo de um grafo conexo, pois há pelo menos um caminho entre todos os pares de vértices do grafo. Já a Figura 8 apresentada a seguir, representa um grafo desconexo, pois não há um caminho do vértice $c$ aos vértices $d$ e $f$, por exemplo. Em um grafo desconexo, cada "pedaço" é chamado de componente conexa.

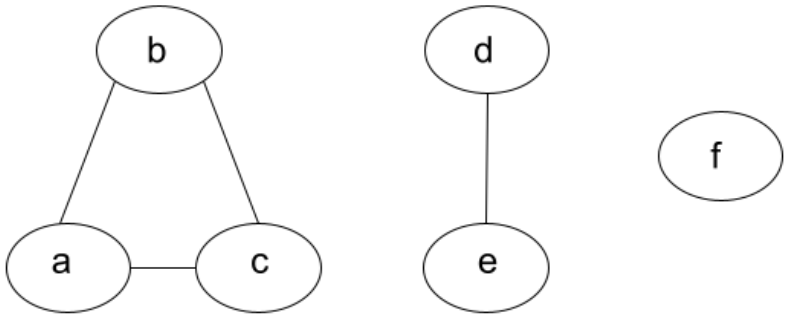


Figura 8 — Grafo Desconexo

Dado um grafo $G$, um caminho é dito caminho hamiltoniano se contém todos os vértices de $G$. Ainda, dado um grafo $G$, um ciclo é dito ciclo hamiltoniano se contém todos os vértices de $G$. Portanto, um grafo $G$ é chamado de grafo hamiltoniano se possui um ciclo hamiltoniano.

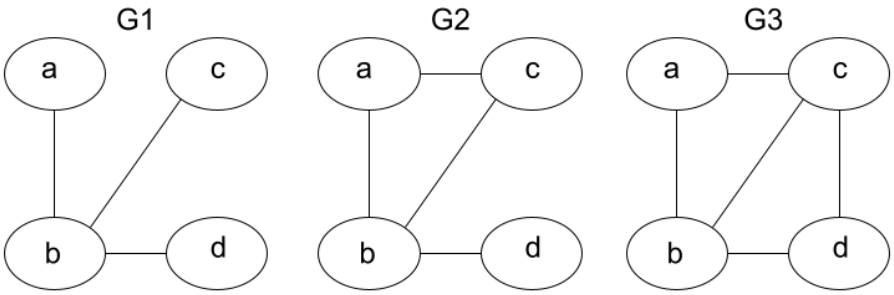


Figura 9 — Lista de três grafos

A Figura 9 apresenta três grafos. O grafo $G1$ não possui um caminho hamiltoniano, pois não há um caminho que contém todos os vértices de $G$. Já o grafo $G2$ possui um caminho hamiltoniano, como exemplo: $W = \{d, b, c, a\}$, porém, não possui ciclo hamiltoniano. Por fim, o grafo $G3$ possui um ciclo hamiltoniano, como exemplo: $W = \{a, b, d, c, a\}$.

## 2.3 Árvores

Uma floresta, é um grafo desconexo, em que cada componente conexa é uma trilha aberta. Na Figura 10, podemos observar um exemplo de uma floresta.

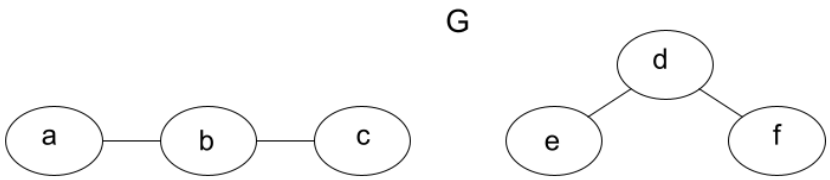


Figura 10 — Floresta

A partir disso, um grafo $G$, conexo, que possui apenas trilhas abertas, é denominado árvore. Dado uma árvore $G = (V, A)$, então $m = n - 1$. Ainda, por ser um grafo conexo, qualquer aresta removida torna $G$ desconexo. Também, por ser um grafo sem ciclo, qualquer aresta acrescentada em $G$, forma um ciclo. Além disso, a distância entre dois vértices $u$ e $v$ é o comprimento do menor caminho entre $u$ e $v$. Já a excentricidade de $u \epsilon V$ é $e(u) = max_{v \epsilon V} dist(u, v)$, ou seja, distância máxima de $u$ a outro vértice do grafo. O raio de um grafo $r(G) = min_{u \epsilon V} e(u)$, é a excentricidade mínima de seus vértices; e o diâmetro $dm(G) = max_{u \epsilon V} e(u)$, é a excentricidade máxima de seus vértices. O centro do grafo $G$ $\{u \epsilon V: e(u) = r(G)\}$, é o conjunto de todos os seus vértices, cuja a excentricidade seja igual ao raio. Por último, seja a aresta $(u, v) \epsilon A(G)$, logo, $(u, v)$ é uma ponte de $G$ se, e somente se, há um único caminho em $G$ de $u$ para $v$. Portanto, todas as arestas de uma árvore são pontes.

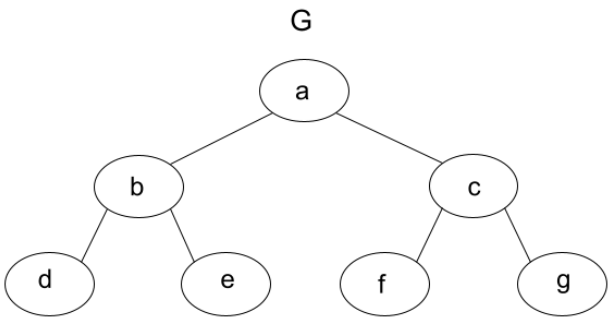


Figura 11 — Árvore

Na Figura 11, o grafo $G$ possui apenas trilhas abertas. Ou seja, não há uma trilha que começa e termina no mesmo vértice de $G$, logo, podemos concluir que o grafo é uma árvore de ordem 7, tamanho 6, raio 2, diâmetro 4 e tem $a$ como vértice central.

## 2.4 Subgrafos

Um subgrafo de um grafo $G$ é qualquer grafo $H$, representado por $H \subseteq G$, tal qual, seu conjunto de vértices e arestas estão contidos em $G$, ou seja, $V(H) \subseteq V(G)$ e $A(H) \subseteq A(G)$.

O subgrafo $H$ é um subgrafo próprio, $H \subset G$, se $H \neq G$, ou seja, $V(H) \neq V(G)$ ou $A(H) \neq A(G)$.

Levando em conta o subgrafo $H$, diz-se que $H$ é um subgrafo gerador de $G$, se possui todos os vértices de $G$, ou seja, $V(H) = V(G)$, mas não obrigatoriamente $A(H) = A(G)$. A Figura 12 mostra o exemplo de um subgrafo gerador $H$ de $G$.

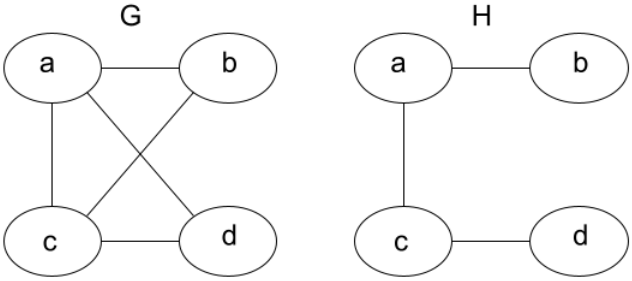


Figura 12 — Subgrafo H é um subgrafo gerador

Considerando ainda o subgrafo $H$, ele é visto como um subgrafo induzido por um subconjunto não vazio de vértices, representado como $H[V']$, se ele conter todos os vértices de $V'$ e todas as arestas adjacentes de $H$. A Figura 13 mostra o exemplo de um subgrafo induzido.

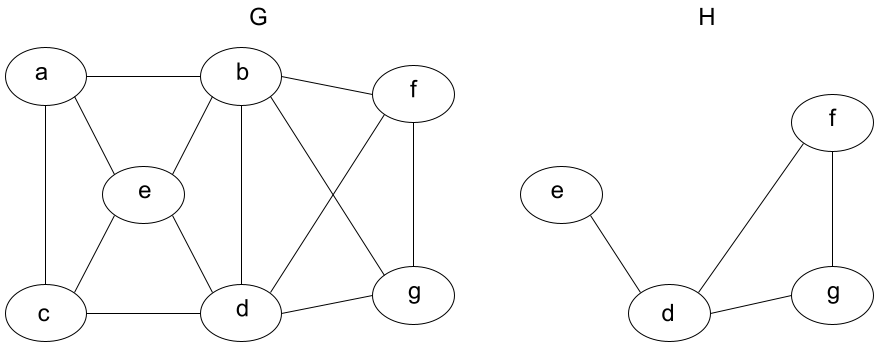


Figura 13 — Subgrafo H induzido pelo subconjunto $V' = \{d, e, f, g\}$

Dado um grafo $G$, uma árvore geradora $T$ é um subgrafo gerador de $G$, em que $T$ é uma árvore, ou seja, é um subgrafo conexo que não possui ciclo e contém todos os vértices de $G$, como pode ser observado na Figura 14.

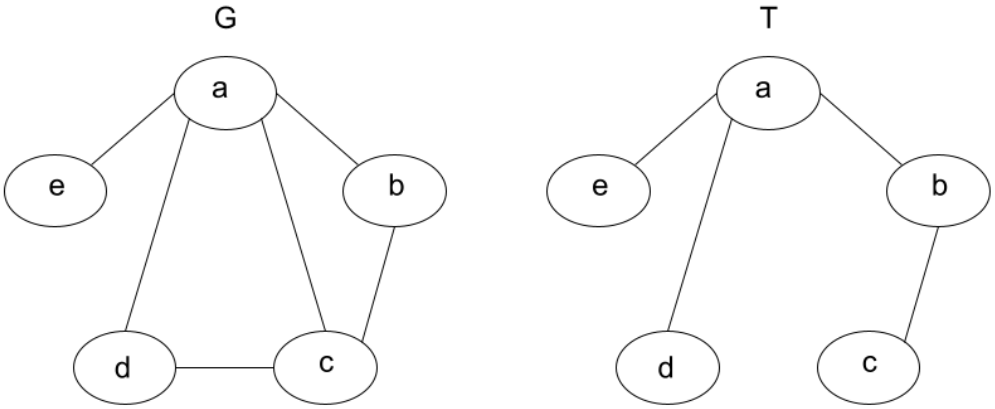


Figura 14 — Árvore geradora T

## 2.5 Grafos valorados

Um grafo $G = (V, A)$ é dito ser valorado quando existe uma ou mais funções relacionadas ao conjunto de vértices, ou ao conjunto de arestas, para um conjunto de

números. O significado das funções depende do problema, por exemplo $V = \{v | v \text{ é o prédio do campus de uma Universidade}\}$ e $A = \{\{u, v, t\} | \text{há fibra ótica ligando } u \text{ a } v, \text{sendo } t \text{ o custo associado}\}$, como pode ser visto na Figura 15.

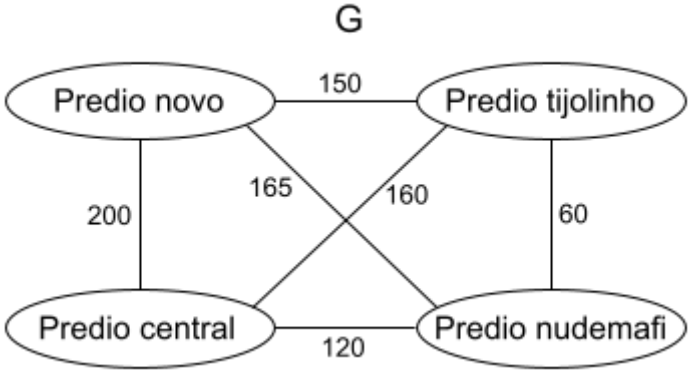


Figura 15 — Grafo valorado

## 2.6 Árvore geradora mínima

Seja $G = (V, A)$ um grafo valorado, na qual cada aresta possui um custo associado, podendo ser positivo ou negativo. O custo $w$ de um subgrafo $T$ de $G$ é a soma dos custos das arestas de $T$, $w(T) = \sum_{i=1}^{k} custo\ aresta\ i$. Portanto, uma árvore geradora mínima de $G$ é qualquer árvore geradora de $G$ que tenha o custo mínimo.

Existem várias maneiras de determinar uma AGM em um grafo ponderado. Os algoritmos de Kruskal e Prim são duas abordagens amplamente utilizadas para essa finalidade. O algoritmo de Kruskal constrói a AGM adicionando, de forma gulosa, as arestas de menor peso que não formem ciclos, até que todos os vértices estejam conectados. Por outro lado, o algoritmo de Prim começa com um vértice inicial e, em cada etapa, adiciona a aresta de menor peso que conecta um vértice na AGM parcial a um vértice fora dela, até que todos os vértices sejam alcançados.

Outra abordagem é utilizando o teorema da Matriz-Árvore, também conhecido como teorema da matriz de Kirchhoff, que enuncia que o número de árvores geradoras de um grafo pode ser calculado em tempo polinomial a partir do determinante de uma submatriz da Matriz Laplaciana do grafo, ou seja, o número é igual a qualquer cofator

da Matriz Laplaciana (JUNIOR, 2019). As árvores geradoras do grafo da Figura 15 estão ilustradas a seguir.

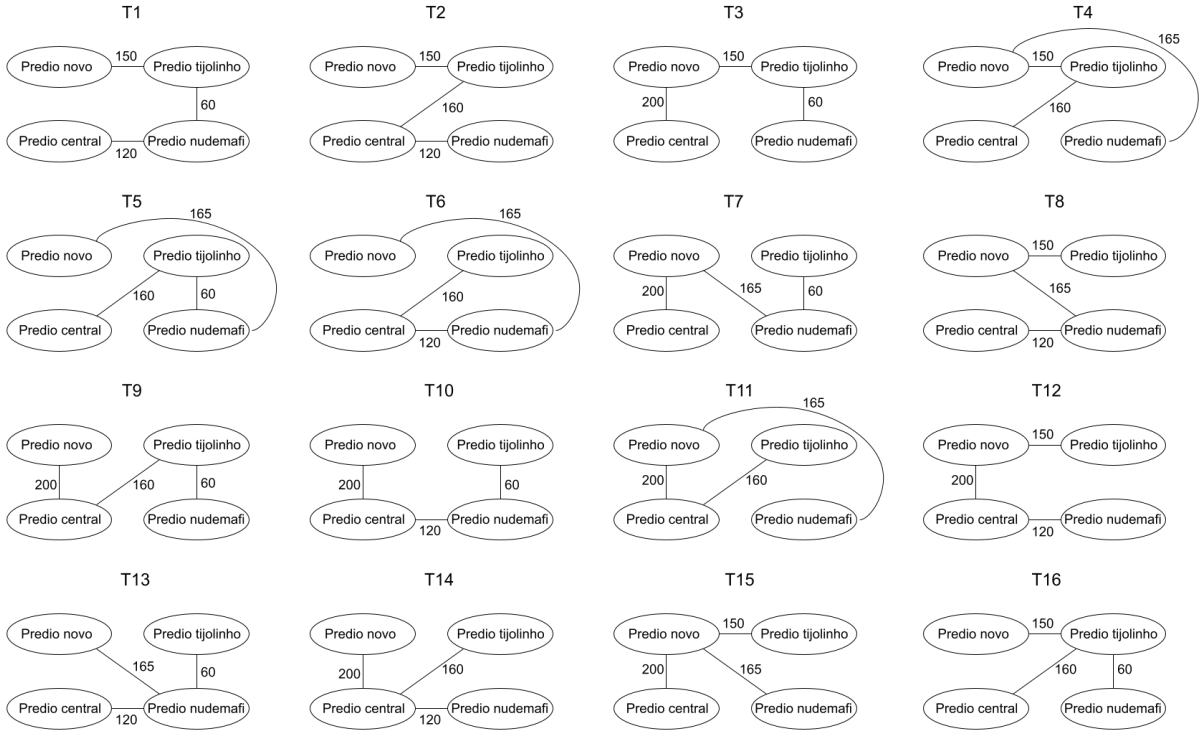


Figura 16 — Árvores Geradoras

Como pode ser observado na Figura 16, a árvore geradora $T1$ possui o menor custo, $w(T1) = 330$, portanto, ela é a árvore geradora mínima.

# 3 METODOLOGIA

O algoritmo de Kruskal é um algoritmo guloso que permite encontrar a árvore geradora mínima de um grafo conexo e ponderado. Ele faz isso escolhendo repetidamente a aresta de menor peso que conecta duas componentes conexas diferentes e adicionando-a à árvore geradora parcial. Esse processo é repetido até que todos os vértices estejam em uma única componente conexa.

Segundo Feofiloff (2020), o algoritmo de Kruskal tem uma complexidade assintótica de $O(n^3)$. Isso significa que o tempo de execução do algoritmo cresce de forma cúbica, proporcionalmente ao número de vértices.

Uma das principais vantagens do algoritmo de Kruskal é sua simplicidade e facilidade de implementação. Além disso, ele sempre produz uma solução ótima, o que significa que a árvore geradora mínima encontrada é de fato a menor possível.

A seguir é apresentado um pseudocódigo do algoritmo de Kruskal.

**Algoritmo 1:** Algoritmo de Kruskal

**Entrada:** Grafo valorado $G = (V, A)$
**Saída:** $AGM = (V, A')$

```
1  criar uma lista L com todas as arestas de G
2  ordenar L crescentemente pelo peso
3  enquanto L <> Ø faça
4      a = remove primeira aresta de L
5      se a não forma ciclo então
6          adicione a à floresta F
7      Fim
8  Fim
```

O funcionamento do algoritmo de Kruskal começa com a criação de uma lista $L$ com todas as arestas de $G$ (linha 1). Em seguida, a lista $L$ é ordenada crescentemente pelo custo das arestas (linha 2). Posteriormente, o algoritmo percorre uma série de instruções enquanto $L$ não fique vazia (linha 3). Para cada iteração, é removido a aresta de menor custo de $L$ (linha 4). É verificado então, se a inclusão dessa aresta

não cria um ciclo na árvore geradora atual (linha 5). Caso não forme ciclo, o algoritmo continua adicionando arestas à AGM até que todos os vértices estejam conectados (linha 6).

A verificação do ciclo se dá pelo conjunto de vértices que já foram conectados pelas arestas da AGM atual. Se os vértices de uma nova aresta já pertencerem ao conjunto, indica que a adição dessa aresta formará um ciclo.

A implementação acima é ineficiente, pois para cada iteração ela examina todos os vértices do grafo, para a partir disso, atualizar o conjunto de vértices. Com a intenção de deixar o algoritmo mais eficiente, podemos recorrer a uma variação do algoritmo original, utilizando a abordagem *Union-Find*, que é uma estrutura de dados para verificar se a inclusão de uma nova aresta forma ciclo.

**Algoritmo 2:** Algoritmo de Kruskal com Union-Find

```
    Entrada: Grafo valorado G = (V, A)
    Saída: AGM = (V, A')
 1  criar uma lista La com todas as arestas de G
 2  ordenar La crescentemente pelo peso
 3  criar uma lista Lc de chefes para rastrear os conjuntos conectados
 4  criar uma lista Lt de tamanhos para rastrear o tamanho de cada conjunto
 5  enquanto La <> Ø faça
 6      (u, v) = remover primeira aresta de La
 7      uc = find(u, Lc)
 8      vc = find(v, Lc)
 9      se uc <> vc então
10          adicionar (u, v) à floresta F
11          union(uc, vc, Lc, Lt)
12      fim
13  fim
14
15  Função find(v, Lc)
16      retornar Lc[v]
17  Fim
18
```

```
19 Procedimento union(uc, vc, Lc, Lt)
20     se Lt[uc] > Lt[vc] então
21         atualizar Lc para que os vértices com chefe vc passem a ter o chefe uc
22         Lt[uc] = Lt[uc] + Lt[vc]
23     Senão
24         atualizar Lc para que os vértices com chefe uc passem a ter o chefe vc
25         Lt[vc] = Lt[vc] + Lt[uc]
26     fim
27 Fim
```

O funcionamento do algoritmo de Kruskal com *Union-Find* começa com a criação de uma lista $La$ com todas as arestas de $G$ (linha 1). Em seguida, $La$ é ordenada crescentemente pelo custo das arestas (linha 2). Logo após, é criado a lista $Lc$, onde inicialmente, cada vértice de $G$ será o seu próprio chefe (linha 3); e a lista $Lt$, onde inicialmente, o tamanho de cada chefe será 1 (linha 4). Posteriormente, o algoritmo percorre uma série de instruções enquanto a lista não fique vazia (linha 5). Para cada iteração, é removido a aresta $(u, v)$ de menor custo da lista $La$ (linha 6), e é buscado o chefe de $u$ (linha 7) e o chefe de $v$ (linha 8), utilizando a função *find*. Por fim, é verificado se o conjunto que contém $u$ é diferente do conjunto que contém $v$ (linha 9). Caso verdadeiro, o algoritmo continua adicionando arestas à AGM (linha 10), e atualizando os conjuntos, realizando a união dos chefes de $u$ e $v$ até que todos os vértices estejam conectados (linha 11).

Na função *find* o algoritmo apenas retorna (linha 16) o chefe do vértice passado como parâmetro.

No procedimento *union*, é verificado se o tamanho do conjunto do chefe de $u$ é maior que o tamanho do conjunto do chefe de $v$ (linha 20); caso verdadeiro, atualiza-se a lista $Lc$ para que os vértices com o chefe $vc$ passem a ter o chefe $uc$ (linha 21), e atualiza-se a lista $Lt$ para que o tamanho do conjunto do vértice $u$ seja incrementado com o tamanho do conjunto do vértice $v$ (linha 22); caso falso, o tamanho do conjunto do vértice $u$ é menor do que o tamanho do conjunto do vértice $v$, então o contrário é realizado (linhas 24 e 25).

Sem a abordagem *Union-Find*, seria necessário percorrer a AGM atual a cada iteração para verificar se a adição de uma nova aresta formaria um ciclo. Essa verificação levaria tempo proporcional ao número de vértices na AGM, resultando em uma complexidade cúbica para o algoritmo. No entanto, com a abordagem *Union-Find*, é possível realizar essa verificação de forma muito mais eficiente. A estrutura de dados permite identificar, de maneira rápida, se dois vértices pertencem ao mesmo conjunto conectado. Dessa forma, a verificação de ciclo é feita de maneira eficiente e independente do tamanho da AGM atual. Além disso, a estrutura *Union-Find* também permite a união eficiente de conjuntos. Essa operação de união também é feita de forma rápida. Portanto, ao utilizar a abordagem *Union-Find*, o algoritmo de Kruskal consegue evitar ciclos de forma eficiente e construir a AGM de maneira mais rápida, resultando em uma melhoria significativa no desempenho do algoritmo, reduzindo sua complexidade para $O(m\ log\, n)$ (CORMEN et al., 2009).

Além da abordagem *Union-Find*, a escolha adequada do algoritmo de ordenação pode afetar o tempo de execução total do algoritmo de Kruskal.

Existem diferentes algoritmos de ordenação disponíveis, como *Bubble Sort*, *Insertion Sort*, *Selection Sort*, *Merge Sort*, *Quick Sort*, entre outros. Cada algoritmo de ordenação tem sua própria complexidade de tempo.

Os algoritmos de ordenação mais eficientes, como o *Merge Sort* ou *Quick Sort*, têm uma complexidade média de tempo de $O(n\ log\, n)$, onde $n$ é o número de elementos a serem ordenados. Esses algoritmos são geralmente preferidos para ordenar as arestas antes de aplicar o algoritmo de Kruskal, pois garantem um tempo de execução mais rápido na maioria dos casos. Por outro lado, algoritmos de ordenação menos eficientes, como *Bubble Sort* ou *Selection Sort*, têm uma complexidade de tempo pior, como $O(n^2)$, o que pode levar a um desempenho ruim quando o número de arestas é grande. Portanto, é importante escolher um algoritmo de ordenação adequado que seja eficiente para o tamanho do grafo em consideração (BACKES, 2023).

Dessa forma, uma outra possiblidade é não usar a ordenação e manter a eficiência do algoritmo com a utilização da estrutura de dados denominada *buckets*. Com essa estrutura, as arestas serão armazenadas em uma lista em que os índices da lista são referentes ao peso de cada aresta, mantendo assim, a ordenação das arestas.

Na Figura 17, é ilustrado o grafo $G$ com 5 vértices e 6 arestas. Já na figura 18, é apresentado a estrutura de *buckets* para o grafo da Figura 17, sendo que os *bucktes* são estruturas definidas como um vetor de pilhas. Logo, cada aresta é empilhada na posição do vetor com seus custos. Dessa forma, na execução do algoritmo de Kruskal, bastaria o algoritmo percorrer o vetor de pilhas removendo as arestas e verificando se cada aresta forma ciclo.

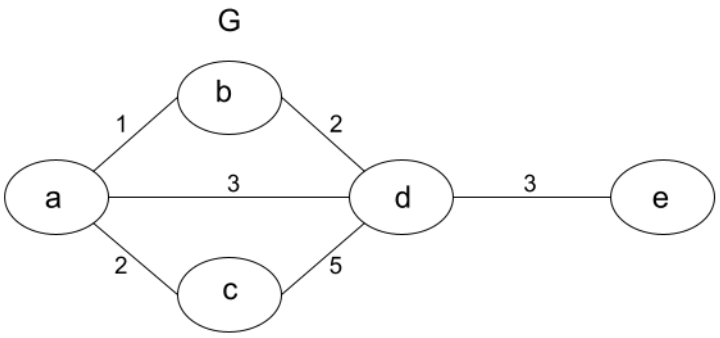


Figura 17 — Grafo G ponderado

| | | | | | |
|---|---|---|---|---|---|
| | (a,b) | (b,d)<br>(a,c) | (d,e)<br>(a,d) | | (c,d) |
| 0 | 1 | 2 | 3 | 4 | 5 |

Figura 18 — Estrutura de *buckets*

O tamanho do vetor de pilhas é determinado por $M + 1$, sendo $M$ o peso da maior aresta do grafo. Como ilustrado na Figura 18, $M = 5$, logo o número de *buckets* é 6.

Para gerar os grafos valorados que serão processados pelos algoritmos, foi desenvolvido um gerador de instâncias. Ele é responsável por gerar os grafos conexos da seguinte forma: primeira linha contém o número de vértices; a segunda linha o número de arestas; e da terceira linha em diante os vértices de cada aresta separados por vírgula, juntamente com o peso da aresta. As instâncias possuem tamanhos variados e com diferentes densidades, onde os grafos muito densos são os grafos que possuem muitas arestas, já os pouco densos, são os grafos com menos arestas.

O funcionamento do gerador de instância começa com a criação de uma estrutura de dados que representa um grafo, inicialmente com $V$ vértices, onde $V$ é o total de vértices do grafo, e zero arestas. Em seguida, o algoritmo percorre uma série de instruções, enquanto o contador não chega ao valor total de arestas do grafo. Para cada iteração, é gerado de forma aleatória valores inteiros no intervalo fechado de 0 à *RAND_MAX*, que representam os vértices $v$ e $w$ incidentes à aresta; já o peso $p$ no intervalo fechado de 1 à *RAND_P*. Se $v$ e $w$ forem diferentes, insere a aresta incidente aos vértices $v$ e $w$ no grafo, de forma que as arestas sejam distintas.

# 4 RESULTADOS OBTIDOS

Os algoritmos apresentados foram implementados na linguagem de programação Java e compilados usando o compilador javac na versão 1.8.0_321.

Os testes computacionais foram realizados em uma máquina com as configurações apresentadas na Tabela 1.

Tabela 1 — Configuração do ambiente de testes

| Item | Descrição |
|---|---|
| Processador | Intel(R) Core(TM) i7-11657G7 @ 2,80GHz |
| Memória (RAM) | 16,00GB |
| Sistema Operacional | Windows 11 |
| Tipo de Sistema | 64 bits |

Neste trabalho foram desenvolvidas as três versões do algoritmo de Kruskal. A primeira, com a implementação ineficiente para identificação de ciclos e o método de ordenação *Merge Sort*, a segunda utilizando a estrutura *Union-Find* e com o método de ordenação *Merge Sort*, e a última, utilizado a estrutura de *buckets* e *Union-Find*.

A escolha do algoritmo *Merge Sort* foi motivada por seu desempenho consistente, que se permanece estável independentemente do estado inicial da lista a ser ordenada, preservando a ordem dos elementos iguais. Além disso, sua complexidade $O(n\ log\ n)$ no pior caso supera a ineficiência quadrática do *Bubble Sort* em grandes conjuntos de dados. Adicionalmente, o *Merge Sort* adere à abordagem “dividir para conquistar”, o que simplifica a implementação do algoritmo em comparação ao *Quick Sort*.

A Tabela 2 apresenta as instâncias criadas pelo gerador de instâncias para testar as três versões do algoritmo de Kruskal.

Tabela 2 — Instâncias criadas

| Instâncias | n | Densidade (%) | m |
|---|---|---|---|
| n50_m306_25 | 50 | 25 | 306 |
| n50_m613_50 | 50 | 50 | 613 |
| n50_m919_75 | 50 | 75 | 919 |
| n100_m1238_25 | 100 | 25 | 1238 |
| n100_m2475_50 | 100 | 50 | 2475 |
| n100_m3713_75 | 100 | 75 | 3713 |
| n500_m31188_25 | 500 | 25 | 31188 |
| n500_m62375_50 | 500 | 50 | 62375 |
| n500_m93563_75 | 500 | 75 | 93563 |
| n1000_m124875_25 | 1000 | 25 | 124875 |
| n1000_m249750_50 | 1000 | 50 | 249750 |
| n1000_m374625_75 | 1000 | 75 | 374625 |

Para cada instância, foram realizadas quatro execuções de cada versão a fim de mitigar o impacto do sistema operacional nos testes. A Tabela 3 apresenta a média de tempo para obtenção dos resultados pelas três versões desenvolvidas nesse trabalho, sendo que $w_1$ representa a primeira versão, $w_2$ a segunda e $w_3$ a versão final.

Tabela 3 — Tempo médio em milissegundos gasto na execução

| Instância | $w_1$ | $w_2$ | $w_3$ |
|---|---|---|---|
| n50_m306_25 | 57,25 | 66,75 | **22,75** |
| n50_m613_50 | 56,25 | 64,25 | **17,00** |
| n50_m919_75 | 62,25 | 64,50 | **17,00** |
| n100_m1238_25 | 66,00 | 64,00 | **16,75** |
| n100_m2475_50 | 83,00 | 78,50 | **29,50** |
| n100_m3713_75 | 79,50 | 76,00 | **39,00** |
| n500_m31188_25 | 161,50 | 160,00 | **81,50** |
| n500_m62375_50 | 228,50 | 232,50 | **118,50** |
| n500_m93563_75 | 238,00 | 225,00 | **187,75** |
| n1000_m124875_25 | 265,75 | 199,75 | **143,25** |
| n1000_m249750_50 | 264,00 | 268,50 | **197,25** |
| n1000_m374625_75 | 310,00 | 316,75 | **219,25** |
| **Média** | 156,00 | 151,37 | **90,79** |

Ao analisar as médias de tempo de execução, observamos que a versão $w_3$ é a mais eficiente, com um tempo médio de 90,79 ms, seguida pela versão $w_2$ com 151,37 ms e, por fim, pela versão $w_1$ com 156 ms.

Ao compararmos individualmente as versões $w_1$ e $w_2$, percebemos que, em média, a versão $w_2$ é aproximadamente 3% mais rápido do que a versão $w_1$. No entanto, essa

diferença é relativamente pequena em comparação as melhorias significativas alcançadas pela versão $w_3$.

Quando comparamos a versão $w_3$ com a versão $w_1$, notamos uma diferença significativa. A versão $w_3$ é, em média, 41,7% mais rápida do que a versão $w_1$, indicando uma melhoria considerável na eficiência do algoritmo. Quanto à comparação entre a versão $w_3$ com a versão $w_2$, observamos uma melhoria de 40,14% em média, reforçando a notável eficácia de $w_3$.

Ao analisarmos as instâncias individualmente, destaca-se a instância “n50_m306_25”, em que $w_1$ demonstra um tempo médio de 57,25 ms, superando $w_2$ em aproximadamente 14,11%. Por outro lado, na instância “n100_m2475_50”, $w_2$ é mais eficiente, registrando um tempo médio de 78,50 ms em comparação aos 83,00 ms de $w_1$, representando uma vantagem de 5,39%. Embora $w_2$ tenha se destacado nesse cenário específico, é crucial notar que, ao considerarmos o conjunto total de instâncias analisadas, $w_1$ superou $w_2$ em cinco das doze instâncias.

Dessa forma, a estrutura de *buckets*, evidenciado pelo algoritmo $w_3$, emerge como um exemplo concreto de como a escolha estratégica de estrutura de dados pode impactar positivamente o desempenho global de algoritmos. Este resultado reforça a conclusão de que, ao considerar estratégias de otimização, a escolha de estruturas de dados eficientes desempenha um papel central na busca por soluções computacionais mais rápidas e eficazes.

# 5 CONCLUSÃO

A utilização da estrutura de b*uckets,* combinada com a estrutura *Union-Find*, mostrou-se eficiente, apresentando resultados bastante promissores quando comparada com a implementação clássica do algoritmo de Kruskal. Em particular, ao considerarmos a versão $w_1$ como referência, a implementação com a estrutura de *buckets* registrou uma notável melhoria, apresentando uma eficiência superior em cerca de 41,7%. Essa constatação ressalta não apenas a eficácia individual da estrutura de *buckets*, mas também a relevância estratégica na escolha das estruturas de dados para otimizar algoritmos.

Todo o código-fonte desenvolvido e utilizado nesse trabalho se encontra disponível no *website*: https://github.com/mpra90/kruskalsalgorithm.

# 6 REFERÊNCIAS

BACKES, A. R. Algoritmos e estruturas de dados em linguagem C. 1. ed. Rio de Janeiro: LTC, 2023.

CORMEN, T. H; LEISERSON, C. E; RIVEST, R. L; STEIN, C. Algoritmos: Teoria e Prática. 3. ed. Rio de Janeiro: LTC, 2009.

FEOFILOFF, P. Algoritmos para Grafos via Sedgewick. 2020. Disponível em: <https://www.ime.usp.br/~pf/algoritmos_para_grafos/>.

GERSTING, J. L. Fundamentos matemáticos para a ciência da computação: matemática discreta e suas aplicações. 7. ed. Rio de Janeiro: LTC, 2017.

GRAHAM, R. L; HELL, P. On the History of the Minimum Spanning Tree Problem. 1985. Disponível em: < http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15859-f09/www/handouts/cmuonly/graham-hell.pdf>.

JUNIOR, R. C. M. O Teorema da Matrix-Árvore e Aplicações. Volta Redonda: UFF, 2019.

NETTO, P. O. B. Grafos: teoria, modelos, algoritmos. 5. ed. São Paulo: Blucher, 2012.

NETTO, P. O. B; JURKIEWICZ, S. Grafos: introdução e prática. 2. ed. São Paulo: Blucher, 2017.

NICOLETTI, M. C; JUNIOR, E. R. H. Fundamentos da Teoria dos Grafos para Computação. 3. ed. Rio de Janeiro: LTC, 2018.