Pontifícia Universidade Católica Do Rio De Janeiro

Árvore geradora mínima em tempo esperado linear

Francisco Geiman Thiesen

Projeto Final de Graduação

Centro Técnico Científico - CTC

Departamento de Informática

Curso de Graduação em Ciência da Computação

Rio de Janeiro, Setembro de 2020

Francisco Geiman Thiesen

Árvore geradora mínima em tempo esperado linear

Relatório de Projeto Final, apresentado ao programa **Ciência da Computação** da PUC-Rio como requisito parcial para a obtenção do titulo de Bacharel em Ciência da Computação.

Orientador: Prof. Marcus Poggi

Rio de Janeiro

Setembro de 2020.

Mathematical problems, or puzzles, are important to real mathematics (like solving real-life problems), just as fables, stories and anecdotes are important to the young in understanding real life

– Terence Tao

Agradecimentos

À minha mãe, que através de seu esforço incansável pôde me proporcionar a educação que tive.

À minha família e amigos por todo apoio ao longo dos anos.

Ao meu orientador Professor Marcus Poggi, por sua participação fundamental nesse trabalho, suas valiosas dicas e sua parceria.

Aos meus professores e professoras que me inspiraram e inspiram até hoje.

A todos os funcionários da PUC-Rio, que com seu trabalho garantem o bom funcionamento da universidade.

À comunidade de programação competitiva, por todas as competições organizadas, encontros promovidos e amizades proporcionadas. Os desafios de programação dos quais participo desde os 15 anos de idade foram cruciais em minha escolha profissional e me fascinam até hoje.

Resumo

Geiman Thiesen, Francisco. Poggi, Marcus. Árvore geradora mínima em tempo esperado linear. Rio de Janeiro, 2020. 59 p. Relatório de Projeto Final – Departamento de Informática. Pontifícia Universidade Católica do Rio de Janeiro.

Neste trabalho foi implementado e testado o algoritmo de Karger-Klein-Tarjan (KKT) para encontrar uma árvore geradora mínima. Além disso, seu desempenho foi avaliado com relação aos algoritmos clássicos em diferentes grafos aleatórios.

Palavras-Chave

Árvore, Árvore Geradora Mínima, Karger-Klein-Tarjan, Kruskal, Prim, Borůvka, C++, Grafos, Teoria de Grafos, Estruturas Discretas

Sumário

1 Introdução

1.1 Definição do problema

Dado um grafo $G = (V, E)$ com custos nas arestas, não orientado e conexo, uma árvore geradora do grafo é um subgrafo $G' \in G$ que satisfaz as seguintes 3 propriedades.

1. Conecta todos os vértice de G
2. Possui exatamente $|V| - 1$ arestas
3. Não contém ciclos

Uma árvore geradora mínima (AGM) é uma árvore geradora de G que possui o menor custo total dentre todas as possíveis árvores geradoras de G, onde o custo de uma árvore é dado pela soma dos custos de suas arestas. É importante atentar para o fato de que podem existir múltiplas árvores geradoras mínimas para um determinado grafo.

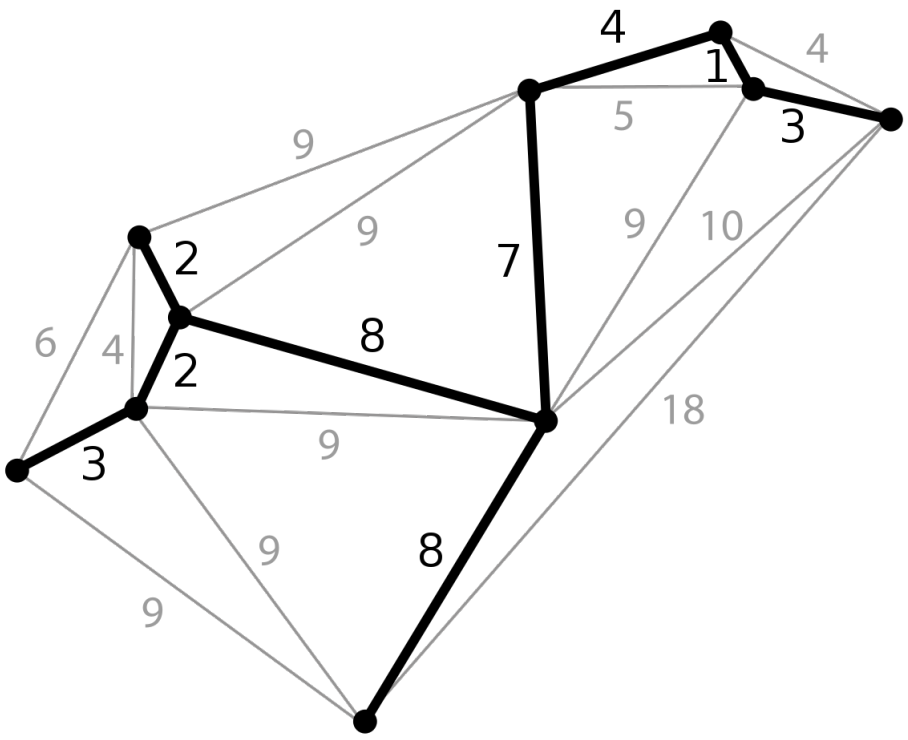


Figura 1: Exemplo de árvore geradora mínima de um grafo (em negrito). (Wikipedia, the free encyclopedia 2001h)

Outro termo relevante ao longo deste trabalho é o de floresta geradora mínima (FGM). Quando trabalhamos com um grafo que não é conexo, isto é, que não possui ao menos um caminho entre quaisquer dois vértices, este grafo não irá possuir uma árvore geradora. Entretanto, esse grafo pode ser separado em componentes conexas (que em último caso serão vértices isolados) e cada uma dessas componentes conexas admite uma árvore. A união das árvores geradoras mínimas de cada componente conexa do grafo define a floresta geradora mínima.

1.2 Aplicações

1.2.1 Construção de redes

AGMs tem aplicação direta em problemas de design e construção de redes, incluindo redes de computadores, redes de telecomunicação, redes de transporte, redes de fornecimento de água e grids elétricos (Borůvka 1926b)(Li, Hou e Sha 2005)(Kadiva e Shiri 2011)(*Understanding Multiple Spanning Tree Protocol (802.1s)* 2020)(Santhi e Padmaja 2016)(Antoš 2016). O primeiro algoritmo para encontrar uma AGM foi proposto pelo cientista tcheco Otakar Borůvka em 1926, justamente com o propósito de achar uma forma eficiente de implantar uma rede elétrica em Moravia (Borůvka 1926a).

1.2.2 Problemas com rotas

AGMs também são amplamente utilizadas como sub-rotinas em algoritmos de redes/grafos/rotas. O famoso algoritmo de Christofides que dá uma 1.5-aproximação para o problema do caixeiro viajante usa AGM (Christofides 1976). Outros exemplo são o algoritmo de corte mínimo multi-terminal e também o que aproxima o matching perfeito de menor custo em um grafo com pesos (Dahihaus et al. 1994)(Supowit, Plaisted e Reingold 1980).

1.2.3 Outras aplicações

O uso de AGMs não é restrito as duas áreas supracitadas, para fins de completude outras aplicações conhecidas estão listadas abaixo.

- Taxonomia. (Sneath 1957)
- Clustering: Clustering de pontos no plano (Asano, Bhattacharya e Keil 1988), single-linkage clustering (Gower e Roos 1969), clustering de teoria dos grafos

(Päivinen 2005) e clustering de expressão genética. (Y. Xu, Olamn e D. Xu 2002).

- Construção de árvores para roteamento de pacotes em redes de computadores. (Dalal e Metcalfe 1978)
- Registro e Segmentação de imagens. (Ma et al. 2000)(Felzenszwalb e Huttenlocher 2000)
- Extração de características curvilíneas de imagens em visão computacional. (Suk e Song 1984)
- Reconhecimento de expressões matemáticas manuscritas. (Tapia e Rojas 2004)
- Design de circuitos. (Ohlsson 2004)
- Regionalização de áreas socio-geográficas (agrupamento de áreas em regiões contínuas e homogêneas). (Assunção et al. 2006)
- Comparação de dados ecotoxicológicos. (Devillers e Dore 1989)
- Observabilidade topológica em sistemas elétricos. (Mori e Tsuzuki 1991)
- Medida de homogeneidade de materiais bidimensionais. (Filliben, Kafadar e Shier 1983)
- Processo de controle minimax. (Kalaba 1963)

2 Situação atual

Atualmente são conhecidas diversas formas para resolver o problema árvores geradoras mínimas. Nessa seção vamos falar um pouco das diferentes formas e buscar entender o panorama atual.

2.1 Algoritmos populares

Para entender quais são os algoritmos mais populares para resolução do problema de árvores geradoras mínimas, recorremos a página do Google Trends (Google 2020b).

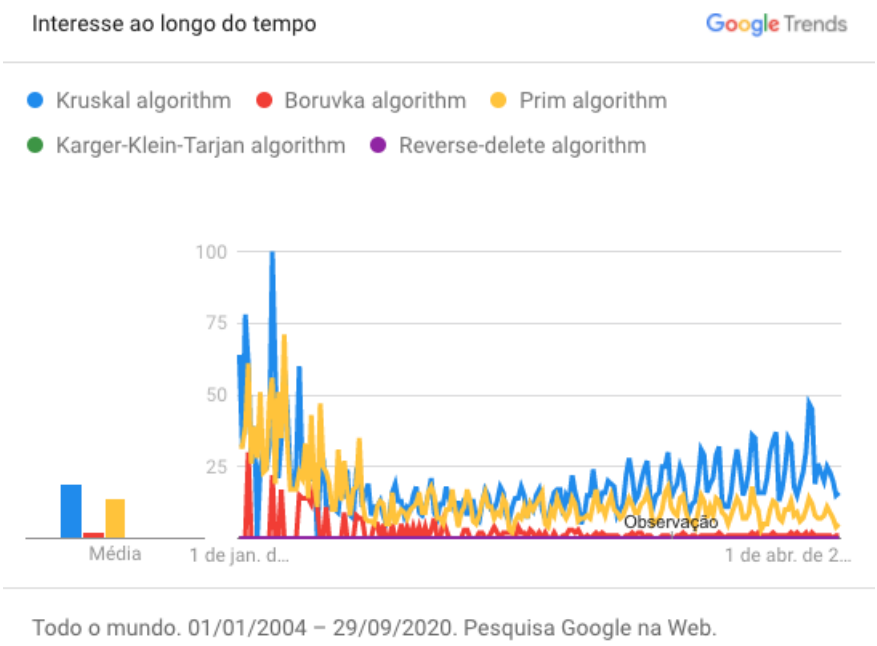


Figura 2: Interesse ao longo do tempo nos diferentes algoritmos de árvores geradoras mínimas. (2004-Hoje)

2.1.1 Top 3 de algoritmos mais populares (de acordo com o Google Trends)

1. Algoritmo de Kruskal. (Kruskal 1956)
2. Algoritmo de Prim. (Prim 1957)
3. Algoritmo de Borůvka. (Borůvka 1926a)

Os algoritmos Reverse-Delete (Kruskal 1956) e Karger-Klein-Tarjan (D. R. Karger, Klein e Robert E. Tarjan 1995) tem uma procura baixa quando comparados aos outros e isso fica nítido no gráfico que foi exibido acima.

Para não nos limitarmos ao que é ou não pesquisado no Google, também foram analisadas algumas bibliotecas populares de grafos que tem licenças de código aberto. São elas, respectivamente: NetworkX (Python) (Hagberg, Swart e Schult 2005), JGraphT (Java) (*JGraphT* 2000), Boost Graph Library (C++) (*Boost Graph Library* 2001) e Cytoscape (Javascript) (*Cytoscape* 2002).

Algoritmo	NetworkX	JGraphT	Boost Graph Library (BGL)	Cytoscape
Kruskal	Sim	Sim	Sim	Sim
Prim	Sim	Sim	Sim	Não
Borůvka	Sim	Sim	Não	Não
Karger-Klein-Tarjan	Não	Não	Não	Não
Reverse-Delete	Não	Não	Não	Não

Tabela 1: Tabela de algoritmos implementados por cada biblioteca.

2.2 Breve discussão dos algoritmos

Nessa seção, vamos falar um pouco mais dos algoritmos conhecidos para computar AGMs.

2.2.1 Algoritmos clássicos

Aqui, a discussão será a respeito dos algoritmos mais populares discutidos acima. A qualquer momento do trabalho, estaremos utilizando n para dizer a quantidade de vértices de um grafo e m para se referir ao número de arestas.

1. **Algoritmo de Kruskal** - Descoberto por Joseph Kruskal, também permite encontrar a floresta geradora mínima. A complexidade do algoritmo é de $O(m \cdot \log(n))$. (Kruskal 1956)

2. **Algoritmo de Prim** - Descoberto por Vojtěch Jarník (Jarník 1930) e redescoberto por (Prim 1957) e (Dijkstra 1959). Sua complexidade varia de acordo com a implementação, mas a mais eficiente que se conhece é $O(m + n \cdot \log(n))$ (Fredman e R. E. Tarjan 1987) utilizando heap de fibonacci.

3. **Algoritmo de Borůvka** - Descoberto por Otakar Borůvka, foi o primeiro algoritmo especificado para esse problema. Possui a mesma complexidade do Algoritmo de Kruskal, $O(m \cdot \log(n))$. (Borůvka 1926a)

2.2.2 Estado da arte

Durante os últimos 50 anos tentou-se reduzir a complexidade de algoritmos que encontram AGMs. Graças a esses esforços temos algoritmos que são mais eficientes em termos de complexidade computacional.

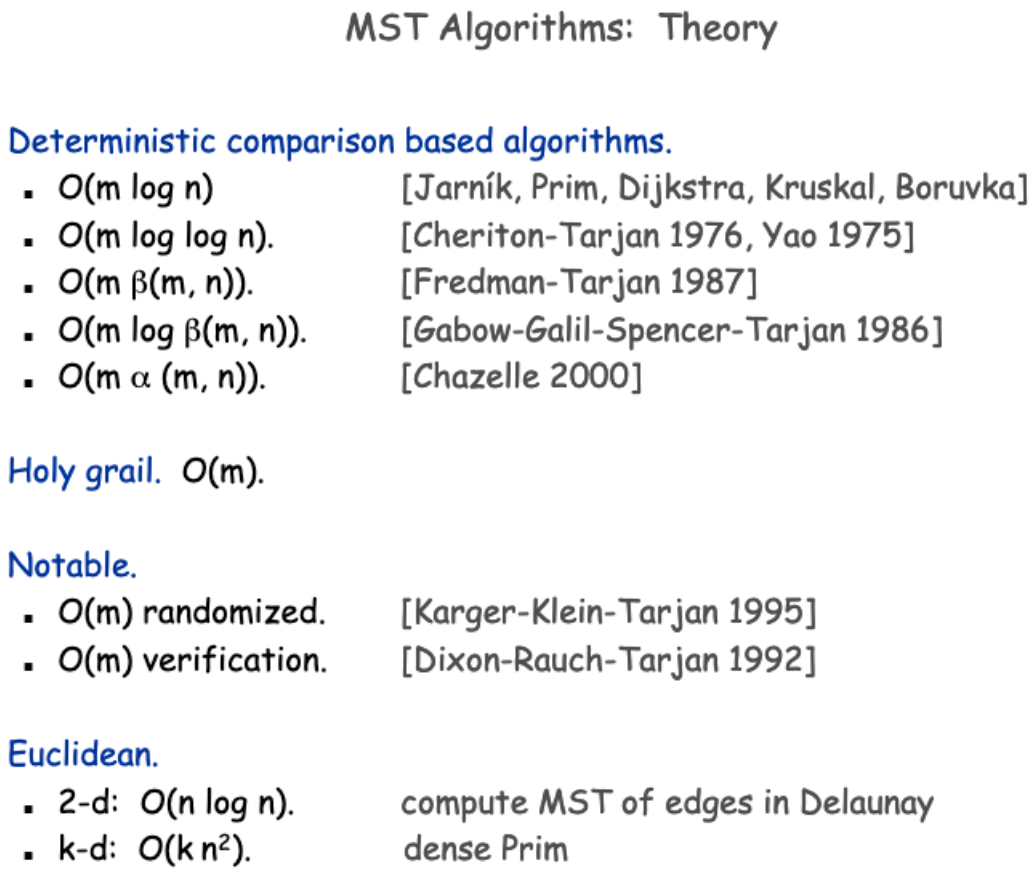


Figura 3: Tabela listando os avanços no problema de AGM ao longo dos anos. (Wayne 2003)

Dos algoritmos citados acima, dois ocupam certa posição de destaque.

1. **Karger, Klein & Tarjan (KKT) 1995** - Algoritmo randomizado que com alta probabilidade funciona com complexidade $O(n+m)$. (D. R. Karger, Klein e Robert E. Tarjan 1995)

2. **Chazelle 2000** - Nesse trabalho, Bernard Chazelle propôs o algoritmo não-randomizado com a melhor complexidade da atualidade, baseado na implementação de uma estrutura de dados chamada soft-heap, que nada mais é do que uma fila de prioridade aproximada. Esse algoritmo tem complexidade $O(m * \alpha(m, n))$ (Chazelle 2000), onde α representa a função inversa de Ackermann, uma função que tem um crescimento extremamente lento. Inclusive, na imensa maioria dos grafos esse termo $\alpha(m, n)$ é ≤ 5 (Wikipedia, the free encyclopedia 2001i).

3 Motivação

A seção anterior mostra que existe uma certa distância entre os resultados teóricos mais recentes e os que são mais populares e implementados em bibliotecas.

Para investigar a predominância do uso de algoritmos clássicos em detrimento de algoritmos mais recentes, nada mais natural do que buscar uma implementação do algoritmo de KKT, que com alta probabilidade funciona em tempo linear no tamanho do grafo, que é a melhor complexidade teórica já encontrada para o problema de AGM (D. R. Karger, Klein e Robert E. Tarjan 1995) e comparar seu desempenho com o dos algoritmos clássicos. No entanto, não há atualmente uma implementação publicamente disponível do KKT. Isto é surpreendente, pois o algoritmo foi publicado inicialmente em 1995.

Essa ausência de uma implementação foi encarada como oportunidade de estudar em detalhe o algoritmo, a fim de poder implementar uma versão correta do mesmo e compará-lo com os algoritmos mais populares.

4 Objetivos

Este projeto foi elaborado em torno de dois questionamentos, que são, respectivamente:

1. Como funciona o algoritmo KKT?
2. Tendo em vista que o KKT tem uma complexidade melhor na teoria, como ele se sai na prática quando comparado com os algoritmos mais populares (Kruskal, Prim, Borůvka)?

Além de tentar responder aos questionamentos, outra motivação foi a de poder disponibilizar esse código com uma licença de código aberto para que terceiros possam usar essa implementação como base para estudos, testes e experimentos.

5 Teoria

Essa seção aborda o estudo teórico realizado ao longo do trabalho. Vamos cobrir os algoritmos de Kruskal, Prim, Borůvka e KKT. A todo o momento estaremos falando de grafos não-direcionados com pesos nas arestas.

5.1 Kruskal

O algoritmo de Kruskal permite encontrar a AGM quando o grafo é conexo e a FGM quando o grafo não é conexo (Kruskal 1956). Funciona de forma gulosa, pois cada passo adiciona a próxima aresta de menor custo possível que não forma um ciclo na FGM selecionada até o momento.

5.1.1 O algoritmo

O algoritmo de Kruskal funciona da seguinte forma:

- Cria uma floresta F (conjunto de árvores), onde cada vértice no grafo é uma árvore separada
- Cria um conjunto S que contém todas as arestas do grafo.
- Enquanto o conjunto S não estiver vazio e F ainda não for gerador do grafo.
  - Remove a aresta de menor custo de S
  - Se a aresta removida conecta duas árvores diferentes de F, então adicionamos essa aresta a F, de modo a combinar as duas árvores antes desconexas em uma única árvore.

Ao final do algoritmo, a floresta F forma a floresta geradora mínima do grafo. Se ele for conexo, a floresta tem apenas uma componente e é também uma árvore geradora mínima.

5.1.2 Pseudocódigo

Nesse pseudocódigo, faremos uso da estrutura de união-busca (R. Tarjan 1975). Essa estrutura servirá para particionar os vértices em grupos, de modo a poder testar em tempo quase constante se uma aresta está ou não ligando vértices de grupos distintos.

Algorithm 1 Kruskal

1: **procedure** KRUSKAL($G = (V, E, w)$)
2: $F \leftarrow \{\}$
3: **for** $v \in G.V$ **do**
4: CRIA-GRUPO(v)
5: Ordenamos aresta de $G.E$ de modo que $w(e_1) < w(e_2) < ... < w(e_m)$
6: **for** $(u, v) \in G.E$ (percorrido de forma crescente nos pesos) **do**
7: **if** $GRUPO(u) \neq GRUPO(v)$ **then**
8: $F \leftarrow F \cup \{(u, v)\}$
9: UNE-GRUPOS($GRUPO(u), GRUPO(v)$)
10: **return** F

5.1.3 Exemplo

A seguir, será exibida uma aplicação do algoritmo de Kruskal em um grafo de exemplo com o intuito de tornar mais tangível uma execução do algoritmo.

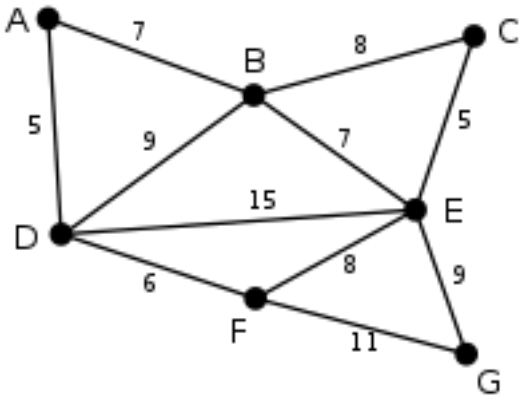


Figura 4: Grafo ao qual o algoritmo de Kruskal será aplicado. Números nas arestas são os pesos e nenhuma aresta foi selecionada por enquanto. (Wikipedia, the free encyclopedia 2001j)

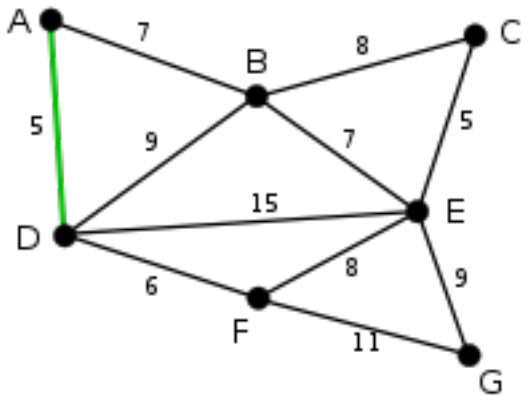


Figura 5: Arestas AD e CE são as mais leves do grafo. Empates são quebrados arbitrariamente e AD é escolhido (Wikipedia, the free encyclopedia 2001j)

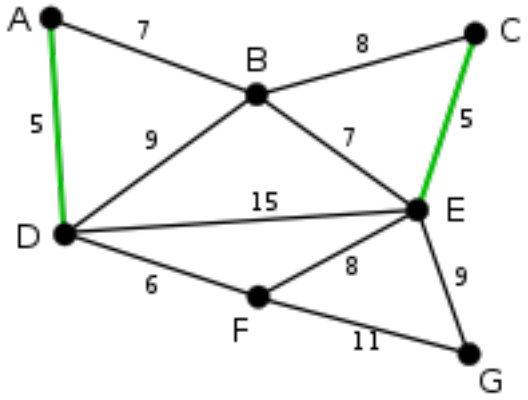


Figura 6: Aresta CE é a mais leve das que sobraram, ela é selecionada já que não forma laço com arestas já selecionadas. (Wikipedia, the free encyclopedia 2001j)

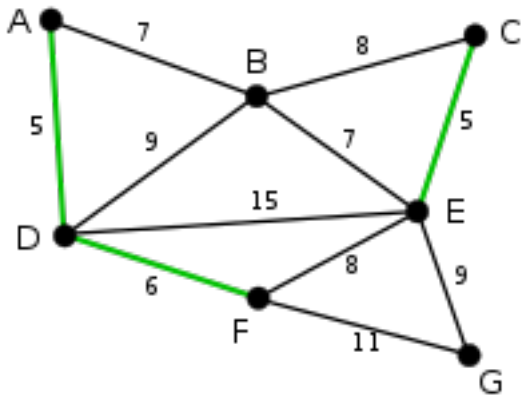


Figura 7: A próxima aresta é a DF com peso 6. Ela não forma uma laço com as arestas já selecionadas e por isso é selecionada. (Wikipedia, the free encyclopedia 2001j)

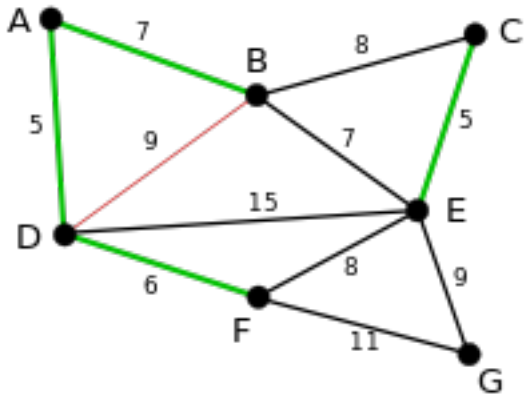


Figura 8: Agora temos duas arestas com peso 7, AB é escolhida ao acaso. A aresta BD de peso 9 passa a ser inválida e por isso é marcada de vermelho, porque ela forma um ciclo e por isso não será selecionada pelo algoritmo. (Wikipedia, the free encyclopedia 2001j)

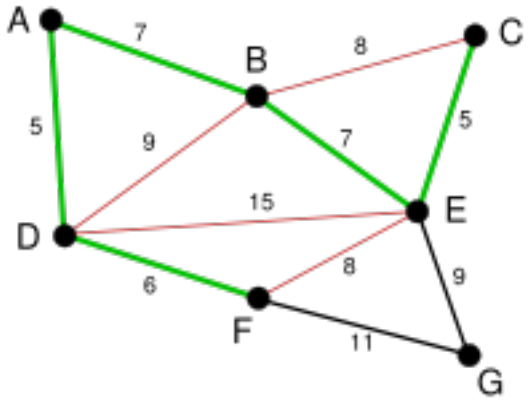


Figura 9: Selecionamos agora BE que também tem peso 7 e atende a todos os critérios do algoritmo. Outras arestas ficam invalidadas após a inclusão de BE. (Wikipedia, the free encyclopedia 2001j)

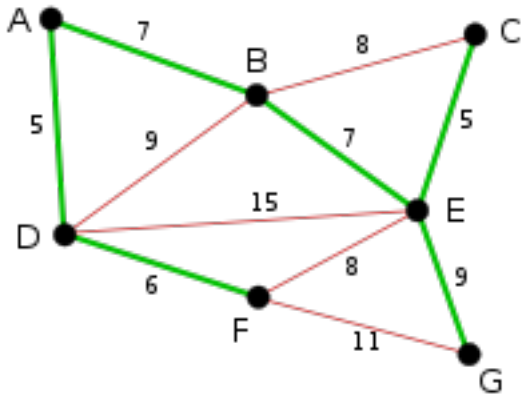


Figura 10: Por último, a aresta EG com peso 9 é selecionada. A aresta FG é invalidada e marcada de vermelho. Agora, como todas as arestas restantes formariam um laço, significa que chegamos ao final do algoritmo e a árvore geradora foi encontrada (em verde). (Wikipedia, the free encyclopedia 2001j)

5.1.4 Corretude

Vamos argumentar sobre a corretude do Kruskal para grafos conexos. A prova será feita em duas partes.

Prova de que o algoritmo produz uma árvore geradora

Seja G um grafo conexo, com pesos e seja Y o subgrafo de G produzido pelo algoritmo. Y não pode conter ciclo, pois nunca adicionamos a Y uma aresta que forma

ciclos. Y não pode ser desconexo, pois se fosse teria ao menos duas componentes distintas A e B e como G é conexo, existe ao menos uma aresta $\in G$ que liga as componentes A e B e essa aresta teria sido adicionada pelo algoritmo ao processá-la, o que contradiz a noção Y não é conexo. Logo, Y é uma árvore geradora de G. □

Prova de que a árvore geradora produzida é mínima

Vamos provar a proposição **P** por indução nas arestas.

P: Se F é o conjunto de arestas escolhido em qualquer momento do algoritmo, então existe uma árvore geradora mínima que contém F e não contém nenhuma das arestas recusadas pelo algoritmo.

- Certamente **P** é verdade no início, quando o F está vazio. Qualquer AGM é suficiente, e existe ao menos AGM uma porque o grafo G é conexo.
- Assuma que **P** é verdade para algum conjunto não-final F e seja T uma AGM que contém F.
  - Se a próxima aresta considerada pelo algoritmo e também $\in T$, então **P** também é verdade para $F + e$.
  - Caso contrário, se $e \notin T$ é porque $T + e$ tem um ciclo C. Esse ciclo contém arestas que não pertencem a F, pois e não forma um ciclo quando é adicionado a F (caso contrário, e seria descartado pelo algoritmo), mas forma ciclo em T. Seja f uma aresta que está $\in C$, mas $\notin F+e$. Note que f também pertence a T, e por **P** ainda não foi processada pelo algoritmo. f tem um peso tão grande quanto o peso de e, se não teria sido processada pelo algoritmo antes de e, e portanto, $T - f + e$ é uma árvore geradora mínima contendo $F + e$, confirmando a proposição **P**.
- Logo, pelo princípio de indução, **P** funciona também para quando F se torna uma árvore geradora, e isso só é possível quando F é a própria árvore geradora mínima. □

5.1.5 Complexidade

Para um grafo com m arestas e n vértices, o algoritmo de Kruskal funciona em tempo $O(m \log(m))$, e equivalentemente, $O(m \log(n))$. Os termos acima são equivalentes em termos de complexidade assintótica, pelo seguinte motivo:

- m é no máximo n^2 e $\log(n^2) = 2*\log(n)$. Logo, $\log(m)$ é no máximo $2*\log(n) \in O(\log(n)$

Vamos agora efetuar a análise detalhada da complexidade, levando em conta a equivalência entre $O(n \log(n))$ e $O(m \log(n))$.

1. Ordenamos as arestas do grafo em ordem crescente de pesos. Se utilizarmos o mergesort para fazer tal ordenação, esse passo tem complexidade de $O(m \log(m))$ que é equivalente a $O(m \log(n))$.

2. Agora, para cada aresta (u, v) do grafo G, teremos que possivelmente executar os seguintes passos:

   - Checar o grupo do vértice u
   - Checar o grupo do vértice v
   - Unir os grupos de u e v, caso sejam distintos.

   Todos esses procedimentos executados no passo 2, podem ser feitos em tempo $\log(n)$ se utilizarmos uma estrutura de união-busca com a heurística de união por rank. Logo, como G tem no exatamente m arestas e para cada uma delas o trabalho do algoritmo é $O(\log(n))$, então o passo 2 também tem complexidade $O(m \log(n))$.

Tanto o passo 1, quanto o passo 2 tem complexidade $O(m \log(n))$. Isso nos permite concluir que o algoritmo de Kruskal também tem essa complexidade.

5.2 Prim

O algoritmo de Prim (também conhecido como algoritmo de Jarník) é um algoritmo guloso que encontra uma árvore geradora mínima de um grafo conexo. Ele opera construindo essa árvore um vértice de cada vez, começando de um vértice arbitrário e adicionando a cada passo a conexão mais barata possível da árvore para um novo vértice.

5.2.1 O algoritmo

O algoritmo de Prim funciona da seguinte forma:

1. Inicializa a árvore com um único vértice qualquer, escolhido arbitrariamente no grafo.

2. Aumenta a árvore selecionada em uma aresta, selecionando a aresta de menor custo que conecta algum vértice da árvore atualmente selecionada com algum vértice que ainda não faz parte da árvore.

3. Repetimos o passo 2 (até que todos os vértices do grafo estejam na árvore).

5.2.2 Pseudocódigo

Aqui teremos uma versão mais elaborada de como implementar o algoritmo.

Algorithm 2 Prim

1: **procedure** PRIM($G = (V, E, w)$)
2: $F \leftarrow \{\}$
3: **for** $v \in G.V$ **do**
4: CUSTO-DESCOBERTA[v]$\leftarrow \infty$
5: ARESTA-INCIDENTE[v]$\leftarrow -1$
6: $Q \leftarrow G.V$
7: **while** Q não está vazio **do**
8: Procura e remove de Q o vértice v' que tem o menor valor possível de CUSTO-DESCOBERTA[v']
9: $F \leftarrow F + v'$, e se ARESTA-INCIDENTE[v'] $\neq -1$ também adicionamos a aresta incidente a F.
10: **for** cada aresta $(v', w) \in G.E$ **do**
11: **if** custo da aresta $<$ CUSTO-DESCOBERTA[w] e $w \in Q$ **then**
12: CUSTO-DESCOBERTA[w] = custo da aresta
13: ARESTA-INCIDENTE[w] = aresta (v', w)
14: **return** F

Vale notar que a forma de armazenar o grafo e de implementar o conjunto Q altera a complexidade do algoritmo.

5.2.3 Exemplo

Para deixar mais claro o funcionamento do algoritmo de Prim, será exibido um exemplo de sua execução em um grafo.

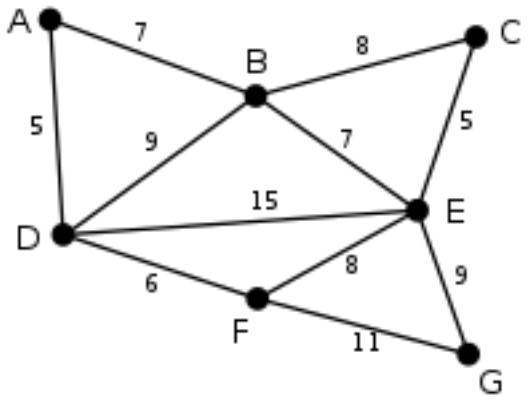


Figura 11: Grafo original. Números das arestas representam seus pesos

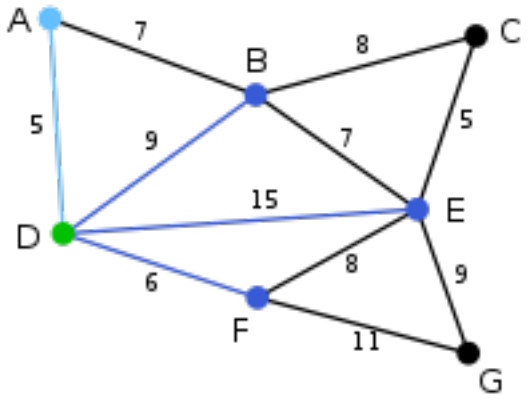


Figura 12: Vértice D foi selecionado como ponto de partida do algoritmo. Vértices A, B, E e F estão conectados a D através de uma aresta e o valor CUSTO-DESCOBERTA e ARESTA-INCIDENTE desses vértices será atualizado. A aresta AD é a mais barata dentre as possibilidades e será selecionada para formar a árvore.

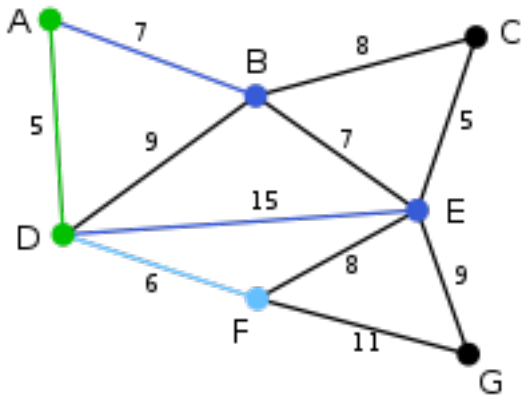


Figura 13: Agora, temos que selecionar qual vértice está mais próximo de D ou A. Nessa instância o vértice mais próximo é F com distância 6, logo a aresta DF será selecionada.

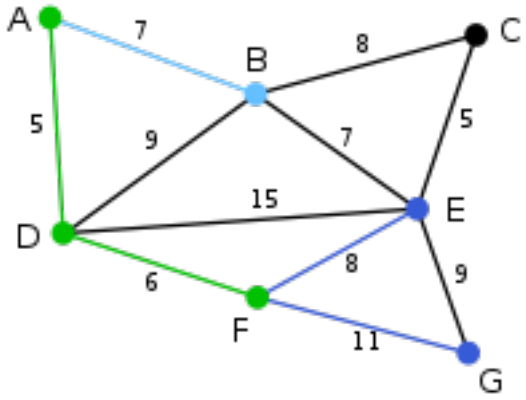


Figura 14: Nesse momento, queremos encontrar o vértice mais próximo de A, D ou F. Esse critério faz com que a aresta AB seja selecionada

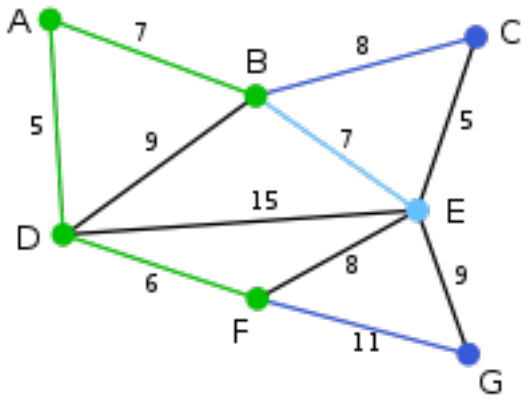


Figura 15: Pelo critério análogo ao anterior, selecionamos a aresta BE.

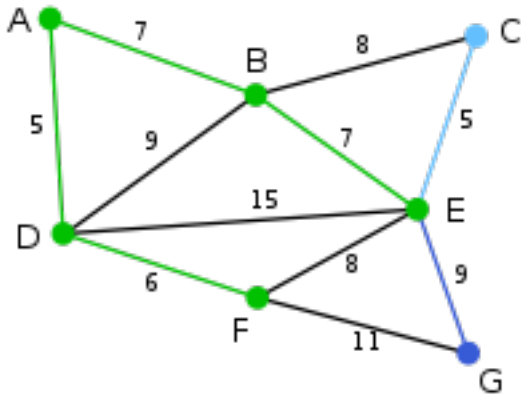


Figura 16: Agora, restam somente os vértices C e G. A aresta EC é escolhida e adicionada a nossa árvore.

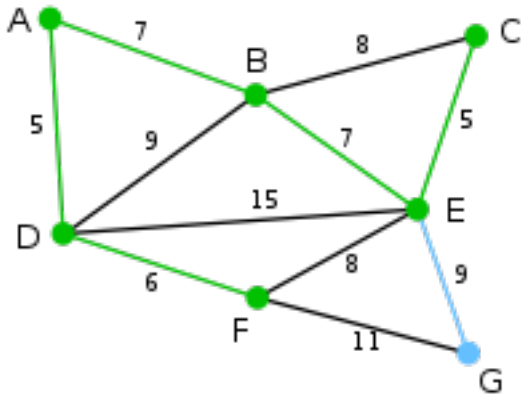


Figura 17: Por fim, a aresta EG é selecionada pelo algoritmo, por ter o menor custo entre todas as possibilidades

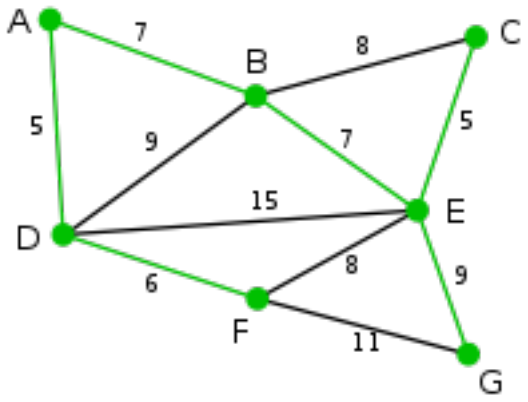


Figura 18: Aqui temos a árvore geradora encontrada pelo algoritmo de Prim em verde.

5.2.4 Corretude

Seja P um grafo conexo, com pesos. A cada iteração, o algoritmo adiciona um vértice novo ao subgrafo selecionado até não poder adicionar mais nenhum vértice. Como P é conexo, sempre vamos ter um caminho para qualquer vértice. A saída T do algoritmo de Prim é uma árvore, porque cada aresta adicionada a T é conectada a um vértice previamente selecionado, e nunca redescobrimos um vértice que já foi adicionado a T previamente.

Seja T_1 uma árvore geradora mínima de P.

Se $T = T_1$, então T é mínima e a prova está concluída.

Caso contrário, seja e a primeira aresta adicionada durante a construção de T que não está em T_1, e seja V o conjunto de vértices conectados antes da adição de $e = (u, v)$. Então, uma extremidade de e está em V e a outra não. Como T_1 é uma árvore geradora de P, existe um caminho em T_1 que u a v. Nesse caminho existe uma aresta f que une o conjunto V ao conjunto que não está em V. Agora, na iteração em que e foi adicionada pelo algoritmo de Prim, a aresta f também poderia ter sido adicionada e ela teria certamente sido adicionada no lugar de e, se $peso(f) < peso(e)$. Como e foi adicionada, e não f, podemos concluir que:

$peso(f) \geq peso(e)$.

Seja T_2 o grafo obtido ao remover a aresta f e adicionando a aresta e a T_1. É fácil ver que T_2 será conexa, tem a mesma quantidade de vértices que T_1 e a soma dos pesos das arestas não vai ser maior do que a soma dos pesos de T_2, portanto temos que T_2 também é uma árvore geradora mínima do grafo P e contém a aresta e e todas as arestas selecionadas pelo algoritmo de Prim antes de e. Esse processo pode ser aplicado repetidas vezes, e garante que a árvore gerada pelo algoritmo de Prim também é mínima. □

5.2.5 Complexidade

Como dito na subseção 5.2.1, a complexidade do algoritmo varia de acordo com a estrutura de dados selecionada para armazenar o grafo e também para representar o conjunto Q de vértices a serem explorados.

Para o escopo desse trabalho a análise será limitada a versão que implementa Q através de uma heap binária e usa lista de adjacência para representar o grafo de entrada.

Podemos resumir o trabalho do algoritmo em dois passos:

1. Inicializa o conjunto Q, CUSTO-DESCOBERTA e ARESTA-INCIDENTE
2. Repita n vezes o seguinte passo, onde n é a quantidade de vértices do grafo
   (a) Selecione o vértice v' de Q que tem o menor CUSTO-DESCOBERTA associado e remova-o de Q.
   (b) Para cada vizinho de v', possivelmente atualize o valor de CUSTO-DESCOBERTA.

O passo 1 pode ser facilmente implementado em proporcional ao número de vértices.

Para entender o custo do passo 2, é necessário observar que cada aresta é considerada para atualizar o valor do CUSTO-DESCOBERTA no máximo duas vezes, quando alguma de suas extremidades está sendo processada e removida de Q. Isso garante que atualizamos o valor de CUSTO-DESCOBERTA no máximo m vezes e a seleção do vértice com menor CUSTO-DESCOBERTA de Q ocorre exatamente n vezes. Uma heap binária permite selecionar o vértice de menor CUSTO-DESCOBERTA, atualizar o CUSTO-DESCOBERTA do vértice e remover o vértice de Q em $\log(n)$, uma vez que essa heap nunca terá mais que n elementos. Essas observações nos

permitem concluir que a complexidade de Prim implementado com heap binária é de $O(m \log(n))$.
É possível atingir uma complexidade de $O(m + n \log(n))$ com heap de fibonacci, mas isso não será abordado nesse trabalho.

5.3 Borůvka

5.3.1 O algoritmo

O algoritmo de Borůvka foi o primeiro algoritmo a resolver o problema de encontrar uma árvore geradora mínima, publicado pela primeira vez em 1926 por Otavar Borůvka (Borůvka 1926a) (Borůvka 1926b). O algoritmo começa por encontrar a aresta mais barata incidente em cada vértice do grafo e adicionando essa aresta a floresta, depois o algoritmo faz o mesmo, mas considerando a aresta mais barata que sai de cada árvore da floresta selecionada até o momento. Esse processo é interessante, pois reduz a quantidade de árvores do grafo resultante a um fator de no mínimo 2 e após um número logarítmico de iterações ele produz uma árvore geradora mínima se o grafo de entrada for conexo, caso contrário, ele retornará a floresta geradora mínima.

5.3.2 Pseudocódigo

Algorithm 3 Borůvka

1: **procedure** BORŮVKA($G = (V, E, w)$)
2: $F \leftarrow G.V$ ▷ F é uma floresta, que inicialmente recebe cada vértice de $G.V$ como uma árvore separada
3: **while** F tem mais de um componente conexo **do**
4: Encontre as componentes conexas de F e marque cada vértice de $G.V$ com o índice de sua componente.
5: Inicializa a aresta mais barata de cada componente conexa para $NULL$
6: total-arestas-selecionadas $\leftarrow 0$
7: **for** aresta $(u, v) \in G.E$ **do**
8: **if** u e v pertencem a duas componentes conexas distintas **then**
9: **if** custo de (u, v) é menor do que o custo da aresta mais barata para o componente de u **then**
10: Atribua (u, v) como a aresta mais barata incidente na componente de u
11: **if** custo de (u, v) é menor do que o custo da aresta mais barata para o componente de v **then**
12: Atribua (u, v) como a aresta mais barata incidente na componente de v
13: **for** componente cuja aresta mais barata incidente não é $NULL$ **do**
14: Adicione a aresta mais barata incidente a essa componente a F
15: total-arestas-selecionadas++
16: **if** total-arestas-selecionadas == 0 **then**
17: **break** ▷ Se o grafo não for conexo, isso garante que retornaremos a floresta geradora mínima
18: **return** F

5.3.3 Exemplo

Abaixo será exibida uma execução do algoritmo em um grafo, para melhor visualização de seu funcionamento.

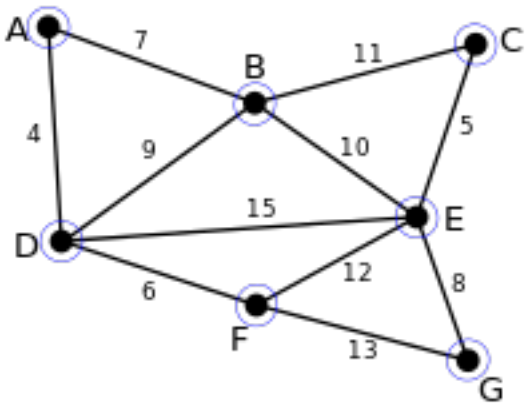


Figura 19: Grafo inicial no qual será aplicado o algoritmo. Inicialmente, cada vértice representa uma componente individual. (Wikipedia, the free encyclopedia 2001a)

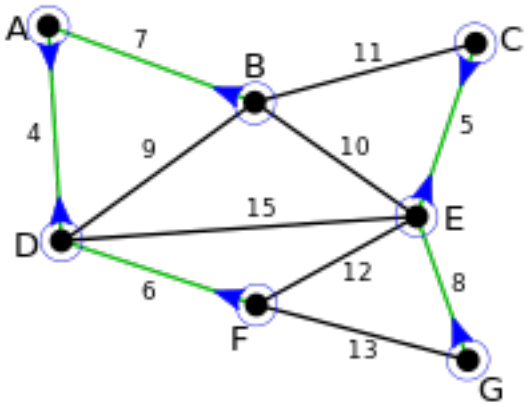


Figura 20: Na primeira iteração do loop principal, o peso mínimo incidente em cada componente é computado. Essas arestas são adicionadas a resposta e ficamos com duas componentes, uma formada por $\{A, B, D, F\}$ e outra formada por $\{C, E, G\}$. (Wikipedia, the free encyclopedia 2001a)

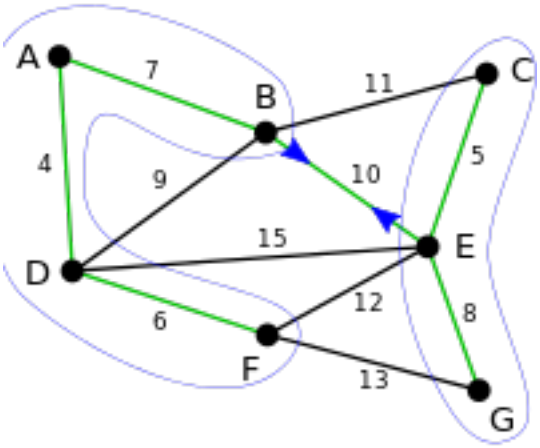


Figura 21: Agora, selecionamos a aresta mais barata incidente em cada uma das duas componentes conexas. Isso faz com que a aresta BE seja adicionada a resposta e isso faz com que tenhamos uma única componente conexa, que é a árvore geradora mínima retornada pelo algoritmo. (Wikipedia, the free encyclopedia 2001a)

5.3.4 Corretude

Para argumentar sobre a corretude, vamos introduzir duas propriedades fundamentais. Elas serão utilizadas também na subseção sobre o algoritmo de Karger-Klein-Tarjan.

5.3.4.1 Propriedade de Ciclo

Para todo ciclo C em um grafo G, a aresta e de maior peso $\in C$ não pode fazer parte de nenhuma floresta geradora mínima. Assumindo que os pesos das arestas são distintos, caso haja empate entre duas arestas, basta introduzir um critério de desempate para que possamos considerar os pesos distinctis. (Wikipedia, the free encyclopedia 2001f)

Prova da propriedade do Ciclo (Por Contradição)
Assuma que a aresta $e = (u, v)$, que é a maior aresta de um determinado ciclo C pertence a uma floresta geradora mínima F_1. Então, deletar a aresta e irá quebrar F_1 em ao menos duas sub-florestas, de tal modo que u e v estarão em sub-florestas distintas. Note que todos os vértices de C pertencem a mesma componente conexa G, e portanto, estão necessariamente na mesma componente conexa em F_1, caso contrário, F_1 não seria floresta geradora e já chegaríamos em uma contradição.

Sabemos também que a aresta e faz parte de um ciclo C, e que ao remover e do ciclo C, ainda temos um caminho conectando os vértices u e v em $C \setminus \{e\}$. Esse caminho contido em $C \setminus \{e\}$ liga u a v, pela definição de ciclo e isso garante que vai existir uma aresta f nesse caminho que liga as duas sub-florestas, sendo que o custo da aresta f é estritamente menor que o da aresta e, porque estamos falando de um grafo onde os pesos das arestas são únicos e assumimos que a aresta e era a mais pesada $\in C$. Obtemos, portanto, uma segunda floresta F_2 que conecta as mesmas componentes que F_1 com um custo inferior e isso diz que F_1 não é uma floresta geradora mínima, ou seja, chegamos em uma contradição. □

5.3.4.2 Propriedade de Corte

Para qualquer subconjunto próprio C de um grafo G, a aresta mais barata e que possui exatamente uma extremidade em C faz parte da floresta geradora mínima. (Wikipedia, the free encyclopedia 2001e)

Prova da propriedade do Corte. (Por Contradição)
Assuma que existe uma floresta geradora mínima F que não contém e. Adicionar e a F irá formar um ciclo e esse ciclo contém pelo menos duas arestas que possuem exatamente uma extremidade $\in C$, sejam essas arestas e e e'. Deletar a aresta e' vai gerar uma outra floresta geradora $F \setminus \{e'\} \cup \{e\}$ e essa nova floresta tem custo total $< F$, o que contradiz a afirmação de que F é uma floresta geradora mínima. □

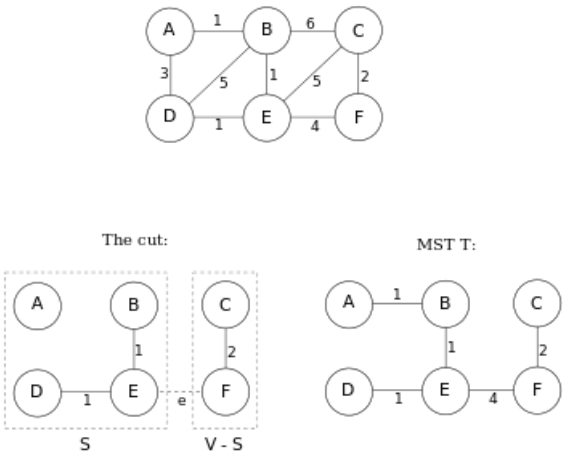


Figura 22: Essa figura ilustra a propriedade do corte de AGMs. T é a única AGM desse grafo. Se $S = \{A, B, D, E\}$, então $V - S = \{C, F\}$ e existem 3 possibilidades de arestas cruzando o corte, são elas BC, EC, EF no grafo original. Então, e, é a aresta de menor custo que atravessa esse corte, portanto $S \cup \{e\}$ é parte da AGM T.

5.3.4.3 Argumento sobre a corretude

Tendo visto a propriedade do corte, a corretude do Borůvka advém de dois fatos, são eles:

1. Sempre adicionamos arestas que pertencem a floresta geradora mínima. Isto é resultado direto da propriedade do corte.

2. A cada passo, dividimos a quantidade de componentes conexos do grafo por um fator de no mínimo 2. □

5.3.5 Complexidade

Como a cada passo do algoritmo reduz-se a quantidade de componentes conexas por um fator de no mínimo 2, por isso, podemos dizer que o loop principal do algoritmo é executado no máximo $\log(V)$ vezes. A cada iteração do loop principal as m arestas do algoritmo são percorridas, e encontrar as componentes conexas de F para a marcação inicial pode ser feito em tempo proporcional a n. Com isso, podemos dizer que a complexidade do algoritmo de Borůvka é $O(m \log(n))$.

5.4 Karger-Klein-Tarjan

O algoritmo de Karger-Klein-Tarjan (KKT) resolve não só o problema de encontrar uma árvore geradora mínima, mas também pode ser usado para encontrar

uma floresta geradora mínima (D. R. Karger, Klein e Robert E. Tarjan 1995). A essência do algoritmo está em combinar a propriedade do ciclo e a propriedade do corte apresentadas na subseção de corretude do algoritmo de Borůvka, com duas sub-rotinas que são, respectivamente, passo Borůvka e amostragem aleatória.

Durante toda essa subseção, vamos assumir que todas as arestas do grafo tem pesos distintos. Se os pesos das arestas não foram distintos na prática, é possível numerar arbitrariamente as arestas e usar essa numeração para quebrar possíveis empates. Essa garantia de lidar com arestas de pesos distintos garante que a AGM é única. A todo momento estaremos nos referindo a grafos não-direcionados, onde cada aresta tem um peso associado.

5.4.1 Propriedades fundamentais

5.4.2 Passo Borůvka

Esse passo é fundamental para o KKT, e tem como papel principal reduzir a quantidade de vértices do grafo por um fator de no mínimo 2.

Para cada vértice, selecione a aresta de menor custo incidente ao vértice. Efetue a contração de todas as arestas selecionadas, criando um único super-vértice para cada componente conexa e deletando todos os vértices isolados do grafo resultante, loops (arestas com ambas extremidades em um mesmo super-vértice), e todas menos a aresta de menor custo entre dois super-vértices.

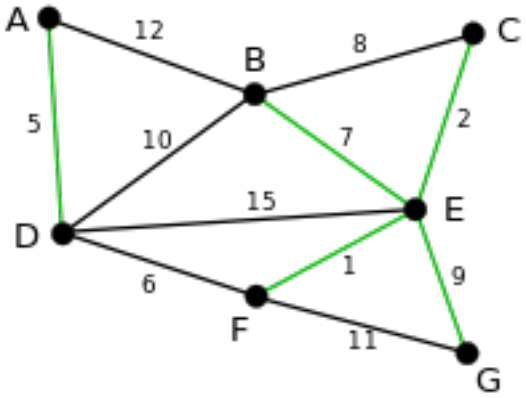


Figura 23: Aresta mais barata incidente em cada vértice destacada em verde. (Wikipedia, the free encyclopedia 2001b)

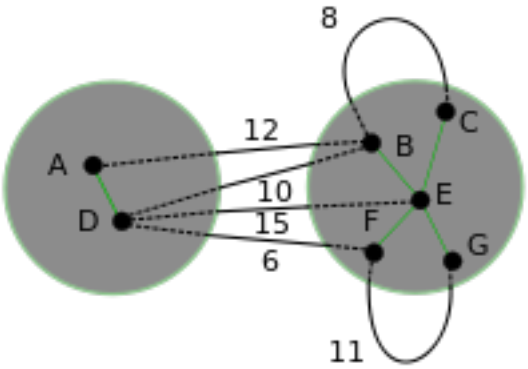


Figura 24: Todas as arestas verdes do passo anterior são contraídas e cada componente conexo verde do passo anterior é substituído por um único super-vértice. Todas as arestas do grafo continuam presentes. (Wikipedia, the free encyclopedia 2001b)

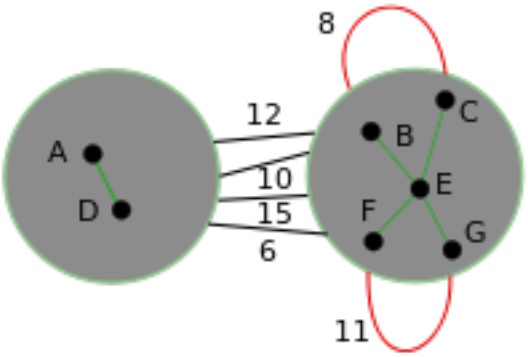


Figura 25: Arestas que passam a ser loops em algum dos super-vértices são removidas. (Wikipedia, the free encyclopedia 2001b)

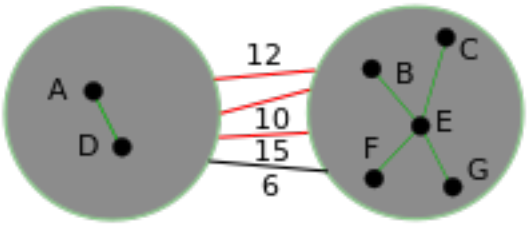


Figura 26: Arestas redundantes (que não são mínimas) entre dois super-vértices são removidas. Na imagem elas estão destacadas em vermelho. (Wikipedia, the free encyclopedia 2001b)

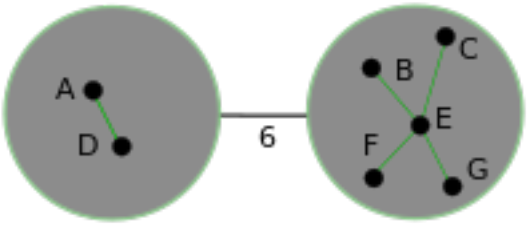


Figura 27: Grafo resultante após aplicar o passo Borůvka uma vez, com dois super-vértices e uma única aresta entre eles. (Wikipedia, the free encyclopedia 2001b)

5.4.3 O Lema da amostragem aleatória

Outro passo essencial do KKT para chegar na complexidade esperada linear é o passo de amostragem aleatória para descartar arestas que não podem ser parte da floresta geradora mínima (Wikipedia, the free encyclopedia 2001g). A eficácia desse passo será exposta abaixo, mas antes é necessário introduzir algumas notações e noções essenciais. (Wikipedia, the free encyclopedia 2001k)

Noções inciais

Seja G um grafo não-direcionado com pesos únicos associados a cada aresta.

1. Vamos usar $w(x, y)$ para se referir ao peso da aresta $\{x, y\}$.

2. Se F é uma floresta de G, usaremos $F(x, y)$ para representar o caminho (se existir) conectando x e y em F. Além disso, vamos usar $w_F(x, y)$ para se referir ao maior peso de uma aresta pertencente a $F(x, y)$. Vamos convencionar que $w_F(x, y) = \infty$ quando não houver caminho entre x e y em F.

3. Uma aresta $\{x, y\}$ é F-Heavy se $w(x, y) > w_F(x, y)$, caso contrário, chamaremos ela de F-Light. Note que todas as arestas de F são F-Light. Note que, para qualquer floresta F, nenhuma aresta F-Heavy pode pertencer a árvore geradora mínima de G. Isto é consequência direta da propriedade do ciclo.

Com isso, estamos prontos para enunciar o Lema.

Lema da Amostragem Aleatória: Seja $G = (V, E, w)$ um grafo não-direcionado com pesos e tome $0 < p < 1$. Tome $G' = (V, E', w)$ um sub-grafo aleatório de G obtido escolhendo incluir ou não cada aresta de G, de forma independente e com probabilidade p. Seja F uma floresta geradora mínima de G'. Então, o número esperado de arestas de G que são F-Light é no máximo $\frac{n}{p}$, onde $n = |V|$.

Observação 1: No lema acima, o limite de arestas F-Light depende de apenas da quantidade de vértices n e não da quantidade de arestas do grafo m.

Prova do Lema:

Abaixo será descrito um algoritmo que gera um subconjunto aleatório de arestas $E' \subseteq E$, em que cada aresta é selecionada de forma independente com probabilidade p. Vamos usar o algoritmo de Kruskal para construir a floresta geradora mínima F do subgrafo $G' = (V, E')$. No processo também vamos construir um conjunto $L \subseteq E$ de arestas de $G = (V, E)$ que são F-Light. Note que o algoritmo *ProvaLemaAmostragem* será utilizado apenas na prova do lema, então não estamos preocupados com implementá-lo de forma eficiente.

Algorithm 4 ProvaLemaAmostragem

1: **procedure** KRUSKAL MODIFICADO($G = (V, E, w)$) ▷ Construção de H e F
2: $E', F, L \leftarrow \{\}$
3: Ordenamos as arestas de modo que $w(e_1) < w(e_2) < ... < w(e_m)$
4: **for** $i \leftarrow 1$ to m **do**
5: Com probabilidade p: $E' \leftarrow E' \cup \{e_i\}$
6: **if** e_i conecta duas componentes conexas distintas de F **then**
7: $L \leftarrow L \cup \{e_i\}$
8: **if** $e_i \in E'$ **then**
9: $F \leftarrow F \cup \{e_i\}$
10: **return** E', F, L

A floresta F produzida por esse procedimento é a floresta gerada pelo algoritmo de Kruskal aplicado as arestas de E', logo F é exatamente a floresta geradora mínima de E'. (Kruskal 1956)

Observação 2: Toda aresta e_i que conecta duas componentes conexas distintas u e v de F em um determinado momento do algoritmo acima é F-Light, porque nenhuma aresta de F com custo inferior conectou essas duas componentes e isso faz com que $w_F(u, v) = \infty$. E essa aresta e_i permanece F-Light até o fim, pois após a inserção da aresta e_i só vamos considerar arestas com peso $> w(e_i)$.

Observação 3: Toda aresta e_i que foi considerada F-Heavy no momento em que foi processada, vai continuar sendo processada até o final do algoritmo. Isso advém do fato de que não removemos arestas de F, e o caminho que tornou $w_f(u, v) < w(e_i)$ continuam presentes.

Como *ProvaLemaAmostragem* é aleatorizado, $|L|$ e $|F|$ são variáveis aleatórias. Sempre que uma aresta e é adicionada a L, também é adicionada a F com probabilidade p. Logo $\mathbb{E}[|F|] = p\mathbb{E}[|L|]$. Entretanto, como F é uma floresta geradora de $G' = (V, E')$, temos que $|F| \leq n - 1$ e claramente $\mathbb{E}[|F|] \leq n - 1$. Logo, $\mathbb{E}[|L|] \leq \frac{n-1}{p} \leq \frac{n}{p}$, como enunciado no *Lema*. □

5.4.4 Pseudocódigo do algoritmo

Tendo visto as propriedades fundamentais, o passo Borůvka (Wikipedia, the free encyclopedia 2001b) e o procedimento de amostragem aleatória (Wikipedia, the free encyclopedia 2001k) estamos prontos para combinar essas ideias e explicitar o algoritmo KKT. (D. R. Karger, Klein e Robert E. Tarjan 1995)

Algorithm 5 Karger-Klein-Tarjan

1: **function** KKT($G = (V, E, w)$)
2: **if** $E = \{\}$ **then**
3: **return** $\{\}$
4: $(G', F') \leftarrow PassoBoruvka(G)$ ▷ Aplicando o passo Borůvka - primeira vez
5: **if** G'.vazio() **then**
6: **return** F' ▷ Passo Borůvka encontrou todas as arestas restantes
7: $(G'', F'') \leftarrow PassoBoruvka(G')$ ▷ Aplicando o passo Borůvka - segunda vez
8: **if** G''.vazio() **then**
9: **return** $F' \cup F''$ ▷ Passos Borůvka encontraram todas as arestas restantes
10: $G_1 \leftarrow RandSubgraph(G'', \frac{1}{2})$
11: $F_1 \leftarrow KKT(G_1)$
12: $E_2 \leftarrow LightEdges(G'', F_1)$ ▷ Arestas F-Light de G'', com respeito a F_1
13: $G_2 \leftarrow (V[G''], E_2, w)$
14: $F_2 \leftarrow KKT(G_2)$
15: **return** $F' \cup F'' \cup F_2$ ▷ F' são as arestas contraídas no primeiro passo Borůvka, F'' são as arestas contraídas no segundo passo Borůvka e F_2 são as arestas retornadas pela aplicação do algoritmo em um grafo reduzido G_2.

5.4.5 Corretude do Algoritmo

Teorema: Assuma que G não tem loops, porque caso tivesse eles poderiam ser removidos facilmente em uma etapa de pre-procesasmento e não fazem parte de nenhuma floresta geradora mínima, isto é consequência direta da *propriedade do ciclo*. Nesse caso, O algoritmo KKT encontra a floresta geradora mínima de qualquer grafo $G = (V, E, w)$.

Prova: Utilizaremos as duas propriedades fundamentais discutidas anteriormente, a *propriedade do corte* e a *propriedade do ciclo*. Utilizaremos uma indução no nú-

mero de vértices.

Caso base: Grafo com exatamente 1 vértice. Como não temos loops, o conjunto de arestas E será vazio e o algoritmo retorna o conjunto vazio de arestas.

Passo indutivo: Suponha que o algoritmo funciona corretamente para grafos com quantidade de vértices $n = \{1, ..., x-1\}$. Vamos analisar o funcionamento do algoritmo. Os dois passos inicias de Borůvka selecionam sempre arestas que pertencem a floresta geradora mínima, segundo a *propriedade do corte* e reduzem o x por um fator de no mínimo 4. Podemos assumir que a chamada recursiva do algoritmo para um grafo com uma quantidade de vértices $\leq \frac{x}{4}$ funciona, pela hipótese de indução e nos retorna uma floresta geradora mínima F_1, que foi obtida de G_1. É necessário notar também que as arestas de G'' que são removidas por serem F-Heavy com relação a F_1 não podem pertencer a nenhuma floresta geradora mínima, pois se uma aresta é F-Heavy com relação a um subgrafo é porque ela é a maior aresta de um ciclo do grafo original e a propriedade do ciclo nos diz que essa aresta não pode fazer parte de nenhuma floresta geradora mínima. Por fim, basta recorrer novamente a hipótese indutiva, porque ao final do processo, se necessário, será feita uma chamada recursiva do algoritmo para um grafo com no máximo $\frac{x}{4}$ vértices. □

Recapitulando a prova do passo

1. Começamos com um grafo G de x vértices.

2. Primeiro passo do Borůvka reduz seu tamanho por um fator de pelo menos 2. Se ele ficou vazio, retornamos a floresta geradora mínima correta, pois só adicionamos a resposta arestas válidas pela *propriedade do corte*. Vamos chamar o grafo reduzido de G'.

3. Segundo passo do Borůvka reduz o tamanho do grafo G' por um fator de pelo menos 2, incluindo na resposta somente arestas que fazem parte da resposta, pela *propriedade do corte*. Repare que já temos uma redução por um fator de pelo menos 4 do grafo original. Vamos chamar o grafo ainda mais reduzido de G''.

4. Chamamos a rotina de amostragem aleatória para obter um subgrafo de G'', seja ele G_1

5. Chamamos o algoritmo de forma recursiva para obter a floresta geradora mínima de G_1. Aqui usamos a hipótese de indução e assumimos que o algoritmo funciona nesse caso, pois G_1 tem no máximo $\frac{x}{4}$ vértices. Vamos chamar essa floresta geradora mínima de F_1.

6. Removemos as arestas de G'' que são F1-Heavy. Note que remover uma aresta F-Heavy com relação a qualquer floresta é válido, uma vez que ela tem o maior peso de um ciclo e por isso não pode pertencer a uma floresta geradora mínima (*propriedade do ciclo*). Seja G_2 o grafo resultante após a remoção das arestas F1-Heavy.

7. Chamamos o algoritmo recursivamente para G_2, sua corretude é garantida pela hipótese indutiva. A quantidade de vértices em G_2 é a mesma de G'', isto é, $\leq \frac{x}{4}$.

8. Por fim, retornamos as arestas contraídas nos dois passos de Borůvka juntamente com as arestas da chamada recursiva de G_2.

5.4.6 Complexidade

Aqui, vamos estabelecer uma equação para analisar a complexidade do algoritmo KKT. Seja $T(m, n)$ a complexidade esperada do algoritmo KKT em um grafo com m arestas e n vértices. O passo Borůvka percorre todas as arestas para encontrar a aresta de menor custo incidente em cada vértice e depois percorre o grafo para agrupar os vértices de cada componente conexa em super-vértices, esse processo tem complexidade $O(n+m)$. Como dois passos do algoritmo de Borůvka reduz a quantidade de vértices por um fator de pelo menos 4 e a quantidade de arestas que sobram é no máximo $\frac{m}{2}$ (Dado que ao menos $\frac{m}{2}$ são colapsadas durante o passo Borůvka em um grafo sem vértices isolados. Pois a quantidade de vértices diminui pela metade ao colapsarmos as arestas incidentes de menor custo e para cada super-vértice vamos manter apenas a aresta mais barata incidente nele), a primeira chamada recursiva tem complexidade esperada $T(\frac{n}{4}, \frac{m}{2})$.

Um resultado importante obtido por Dixon-Rauch-Tarjan, nos diz que a verificação de arestas F-Light de um grafo pode ser implementado em tempo $O(n + m)$ (Dixon, Rauch e Robert E. Tarjan 1992). Além disso, a chamada recursiva final toma tempo $T(\frac{n}{4}, |L|)$ onde L é a quantidade de arestas que não são F-Light com

relação a floresta produzida pela primeira chamada recursiva. Isso nos dá a seguinte recorrência para T:

$$T(n, m) = O(n + m) + T(\tfrac{n}{4}, \tfrac{m}{2}) + \mathbb{E}(T(\tfrac{n}{4}, |L|))$$

O Lema da Amostragem Aleatória enunciado anteriormente, nos diz que se escolhermos $p = 0.5$ na etapa de amostragem, teremos que o valor esperado de $|L| = \frac{\frac{n}{4}}{0.5} = \frac{n}{2}$. Isso faz com que a equação mude para:

$$T(n, m) = O(n + m) + T(\tfrac{n}{4}, \tfrac{m}{2}) + T(\tfrac{n}{4}, \tfrac{n}{2})$$

Munidos da equação acima, podemos aplicar indução e linearidade do valor esperado para mostrar a que a complexidade esperada do algoritmo é da ordem de $O(n + m)$.

5.4.6.1 Prova da complexidade por indução

Lembrando, temos a equação anterior da complexidade esperada do algoritmo de KKT em um grafo aleatório com n vértices e m arestas:

$$T(n, m) = O(n + m) + T(\tfrac{n}{4}, \tfrac{m}{2}) + T(\tfrac{n}{4}, \tfrac{n}{2})$$

Caso base: Grafo com $n \leq 3$ e uma quantidade de arestas m qualquer, note que estamos falando de grafos sem self-loops e arestas paralelas, uma vez que isso pode ser removido em tempo linear antes de aplicar o algoritmo. Nesse caso, a floresta ou árvore geradora mínima será encontrada apenas através das duas chamadas iniciais do passo Borůvka. As chamadas recursivas não serão feitas, porque $\frac{n}{4}$ nesse caso específico implica que os G'' e G_2 estarão vazios.

Passo indutivo: Suponha que o algoritmo tem complexidade esperada $O(n + m)$ para grafos aleatórios com $n \leq x$. Queremos mostrar que isso continua válido para grafos com $x + 1$ vértices e uma quantidade qualquer de arestas m.

$$T(x + 1, m) = O((x + 1) + m) + T(\tfrac{x+1}{4}, \tfrac{m}{2}) + T(\tfrac{x+1}{4}, \tfrac{x+1}{2})$$

Após a aplicação inicial dos 2 passos Borůvka, existem dois possíveis cenários:

- **Caso 1:** Os dois passos Borůvka podem por si só já encontram a AGM/FGM. Isso acontece garantidamente para grafos com $x + 1 \leq 3$, mas pode acontecer para grafos gerais, uma vez que a cota de $\frac{v}{4}$ vértices obtidos após duas aplicações

do passo em um grafo de v vértices é uma cota superior no número de vértices do grafo resultante. Nesse caso não é feito trabalho recursivo algum e, trivialmente temos que $T(x+1, m) = O((x+1) + m)$.

- **Caso 2:** Os dois passos Borůvka não encontram todas as arestas da AGM/FGM. Nesse caso vamos ter chamadas recursivas, $T(\frac{x+1}{4}, \frac{m}{2})$ e $T(\frac{x+1}{4}, \frac{x+1}{2})$. O caso acima trata todos os grafos com $x+1 \leq 3$, e temos com folga que $\frac{x+1}{4} < x+1$. Dessa forma, podemos invocar a hipótese indutiva, assumindo que $T(\frac{x+1}{4}, \frac{m}{2}) = O(\frac{x+1}{4} + \frac{m}{2})$ e $T(\frac{x+1}{4}, \frac{x+1}{2}) = O(\frac{x+1}{4} + \frac{x+1}{2})$.
  Com isso, podemos dizer que:
  $T(x+1, m) = O((x+1) + m) + O(\frac{x+1}{4} + \frac{m}{2}) + O(\frac{x+1}{4} + \frac{x+1}{2})$
  $T(x+1, m) \leq O((x+1) + m) + O((x+1) + m) + O((x+1) + m)$
  $T(x+1, m) \leq O((x+1) + m)$
  □

6 Prática

Essa seção aborda o trabalho prático desenvolvido ao longo do projeto. Por prático, estamos considerando a implementação dos algoritmos estudados acima, testes exaustivos para validação da implementação e uso de ferramentas para facilitar e aprimorar a execução do projeto.

6.1 Implementação

6.1.1 Kruskal

Pode-se dizer que o algoritmo de Kruskal tem a implementação mais simples entre todos os estudados nesse trabalho. Seu funcionamento depende basicamente de uma sub-rotina de ordenação e de uma estrutura de União-Busca para selecionar apenas as arestas que juntam componentes conexas distintas. O código é relativamente pequeno e pode ser visto abaixo:

Cód. 1: Implementação do algoritmo de Kruskal

```
// Cada aresta tem (origem, destino, custo, id). O campo id n o
// tem relev ncia para essa implementa  o.
vector<tuple<int, int, int, int>> kruskal(
    const vector<tuple<int, int, int, int>>& graph_edges,
    int n)
{
        UnionFind graph(n);
        vector<tuple<int, int, int, int>> edges;
        vector<tuple<int, int, int, int>> spanning_tree;

        edges = graph_edges;

        sort(edges.begin(), edges.end(),
                 [&](const tuple<int, int, int, int>& a,
                         const tuple<int, int, int, int>& b) {
                         return get<2>(a) < get<2>(b);
                 });

        for (const auto& edge: edges) {
                int from, to, cost, id;
                tie(from, to, cost, id) = edge;
                if (graph.unite(from, to))
                    spanning_tree.emplace_back(from, to, cost, id);
        }

        return spanning_tree;
}
```

Na implementação da estrutura de União-Busca foi utilizada a heurística de união por ranking e compressão de caminhos para diminuir a altura dos conjuntos e acelerar as operações. Mais detalhes sobre a estrutura de União-Busca e as técnicas mencionadas podem ser encontrados em (Wikipedia, the free encyclopedia 2001l).

6.1.2 Prim

O algoritmo de Prim (Prim 1957) admite algumas implementações distintas, com diferentes complexidades. São elas:

1. Implementação para grafos densos, utilizando matrizes de adjacência. Complexidade => $O(n^2)$

2. Implementação para grafos esparsos, utilizando lista de adjacência e heap binária. Complexidade => $O(m \log(n))$

3. Implementação para grafos esparsos, utilizando lista de adjacência e heap de fibonacci. Complexidade => $O(m + n \log(n))$

Como a comparação dos algoritmos foi feita em cima de grafos esparsos, a primeira implementação não era adequada, pois ia ter uma performance desnecessariamente

lenta. A terceira opção com heap de fibonacci é um pouco mais eficiente em teoria, mas sua implementação é bem mais complexa do que a que usa uma simples heap binária e por isso a opção 2 foi a escolhida. A implementação final tem aproximadamente 35 linhas e será exibida abaixo, um comentário a ser feito é que essa implementação foi fortemente baseada em (http://e-maxx.ru/ 2014).

Cód. 2: Implementação do algoritmo de Prim

```
1  struct Edge {
2          int w = INF, to = -1, id;
3          bool operator<(Edge const& other) const {
4                  return make_pair(w, to) < make_pair(other.w, other.to);
5          }
6          Edge() {
7                  w = INF;
8                  to = -1;
9          }
10         Edge(int _w, int _to, int _id) : w(_w), to(_to), id(_id) {}
11 };
12
13 vector<tuple<int, int, int, int>> _prim(const vector<vector<Edge>>& adj, int n)
14 {
15         vector<tuple<int, int, int, int>> spanning_tree;
16
17         vector<Edge> min_e(n);
18         min_e[0].w = 0;
19         set<Edge> q;
20         q.insert({0, 0, -1});
21
22         vector<bool> selected(n, false);
23         for (int i = 0; i < n; ++i) {
24                 int v = q.begin()->to;
25                 selected[v] = true;
26                 q.erase(q.begin());
27
28                 if (min_e[v].to != -1)
29             spanning_tree.emplace_back(min_e[v].to, v, min_e[v].w, min_e[v].id);
30
31                 for (Edge e: adj[v]) {
32                         if (!selected[e.to] && e.w < min_e[e.to].w) {
33                                 q.erase({min_e[e.to].w, e.to, e.id});
34                                 min_e[e.to] = {e.w, v, e.id};
35                                 q.insert({e.w, e.to, e.id});
36                         }
37                 }
38         }
39         return spanning_tree;
40 }
```

Outro aspecto a ser observado é que a implementação acima não usa uma heap binária, mas sim o *set* de C++ que funciona como uma árvore rubro-negra (*C++ implementation of set* 2020). Isso não muda a complexidade, pois o *set* também permite inserções e remoções com complexidade logarítmica no número de elementos (*C++ implementation of set* 2020).

6.1.3 Borůvka

O algoritmo de Borůvka acabou ficando com uma implementação um pouco mais extensa, mas ainda é plausível de ser exibido em uma única página. (Borůvka 1926a)

Cód. 3: Implementação do algoritmo de Borůvka

```
vector<tuple<int, int, int, int>>
        boruvka(const vector<tuple<int, int, int, int>>& edges, int n) {
        vector<int> component(n, -1), cheapest(n, -1);
        vector<tuple<int, int, int, int>> ans;
        vector< vector<int> > selected_graph(n);

        int m = static_cast<int>(edges.size()), graph_cc = n;
        while (graph_cc > 1) {
                fill(cheapest.begin(), cheapest.end(), -1);
                fill(component.begin(), component.end(), -1);
                int cur_cc = 0;

                for(int i = 0; i < n; ++i) {
                        if(component[i] == -1) {
                mark-cc(i, cur_cc, component, selected_graph);
                                cur_cc++;
                        }
                }

                for (int i = 0; i < m; ++i) {
                        int from, to, cost;
                        tie(from, to, cost, ignore) = edges[i];
                        to = component[to]; from = component[from];
                        if (from == to) continue;
                        if (cheapest[from] == -1 || cost < get<2>(edges[cheapest[from]]))
                                cheapest[from] = i;
                        if (cheapest[to] == -1 || cost < get<2>(edges[cheapest[to]]))
                                cheapest[to] = i;
                }

                unordered_set<int> inserted_edges_id;

                for (int i = 0; i < n; ++i) {
                        if (cheapest[i] == -1) continue;
                        int from, to, cost, id;
                        tie(from, to, cost, id) = edges[cheapest[i]];
                        if(inserted_edges_id.find(id) != inserted_edges_id.end()) continue;
                        else {
                                selected_graph[from].push_back(to);
                                selected_graph[to].push_back(from);
                                inserted_edges_id.insert(id);
                                ans.emplace_back(from, to, cost, id);
                        }
                }
                graph_cc = cur_cc;
        }
        return ans;
}
```

Vale notar que existe uma chamada para uma função auxiliar *mark-cc* dentro do código da função. O papel dessa função é explorar todos os vizinhos de um determinado vértice e sinalizar que eles pertencem a mesma componente conexa em

um determinado passo do algoritmo. A *mark-cc* pode ser implementada tanto como uma busca em profundidade (Wikipedia, the free encyclopedia 2001d), como busca em largura (Wikipedia, the free encyclopedia 2001c) sem afetar na complexidade, apesar de que nesse projeto a escolha foi por utilizar busca em profundidade. A complexidade final dessa implementação é também de $O(m \log(n)$.

6.1.4 Karger-Klein-Tarjan

A discussão da implementação do algoritmo de KKT será feita sem expôr o código completo, pois este tem um tamanho considerável. A implementação deste algoritmo foi de longe a parte mais desafiadora de todo o projeto, uma vez que durante sua elaboração não havia nenhuma outra implementação publicamente disponível para utilizar como referência. Essa discussão será quebrada em partes distintas, porém igualmente importantes.

6.1.4.1 Passo Borůvka

A implementação do passo Borůvka (Wikipedia, the free encyclopedia 2001b) se assemelha a implementação do algoritmo de Borůvka, mas ela precisa realizar alguns procedimentos extras que não são necessários na implementação do algoritmo de Borůvka, como:

- Remover vértices do grafo que ficam isolados.
- Realizar a compressão dos vértices em super-vértices para retornar o grafo reduzido. Isso é um detalhe essencial para que o KKT tenha a tão desejada complexidade linear.
- Remover todas as arestas entre dois super-vértices, exceto a de menor custo.

E tudo isso tem que ser suportado sempre mantendo complexidade linear no tamanho do grafo recebido na entrada. A parte mais delicada foi manter a aresta mais barata entre cada super-vértice resultante, que foi resolvido mantendo uma tabela hash que com pares de inteiros como chave.

6.1.4.2 Amostragem Aleatória

A amostragem em si é uma sub-rotina essencial para o algoritmo, onde um subgrafo H é induzido de G ao decidirmos incluir ou não cada aresta de G com pro-

babilidade $\frac{1}{2}$. Seu código é relativamente simples e será exibido abaixo. (Wikipedia, the free encyclopedia 2001k)

Cód. 4: Implementação da amostragem aleatória

```
problem random_sampling(problem& P, unsigned int seed = 0)
{
        srand(seed);
        vector<tuple<int, int, int, int>> H;
        for (const auto& E: P.graph_edges)
        {
                int b = (rand() & 1);
                if (b) H.push_back(E);
        }

        problem P_H = problem(P.num_vertices, H);
        return remove_isolated_vertices(P_H);
}
```

Aqui chamamos também uma sub-rotina que remove os vértices isolados de H, pois eles são irrelevantes para o cálculo da AGM/FGM. Note que um vértice é chamado de isolado, se e somente se, nenhum outro vértice puder alcançá-lo usando arestas do grafo.

6.1.4.3 Filtragem de arestas F-Light de um grafo G, com relação à uma árvore F (Wikipedia, the free encyclopedia 2001g)

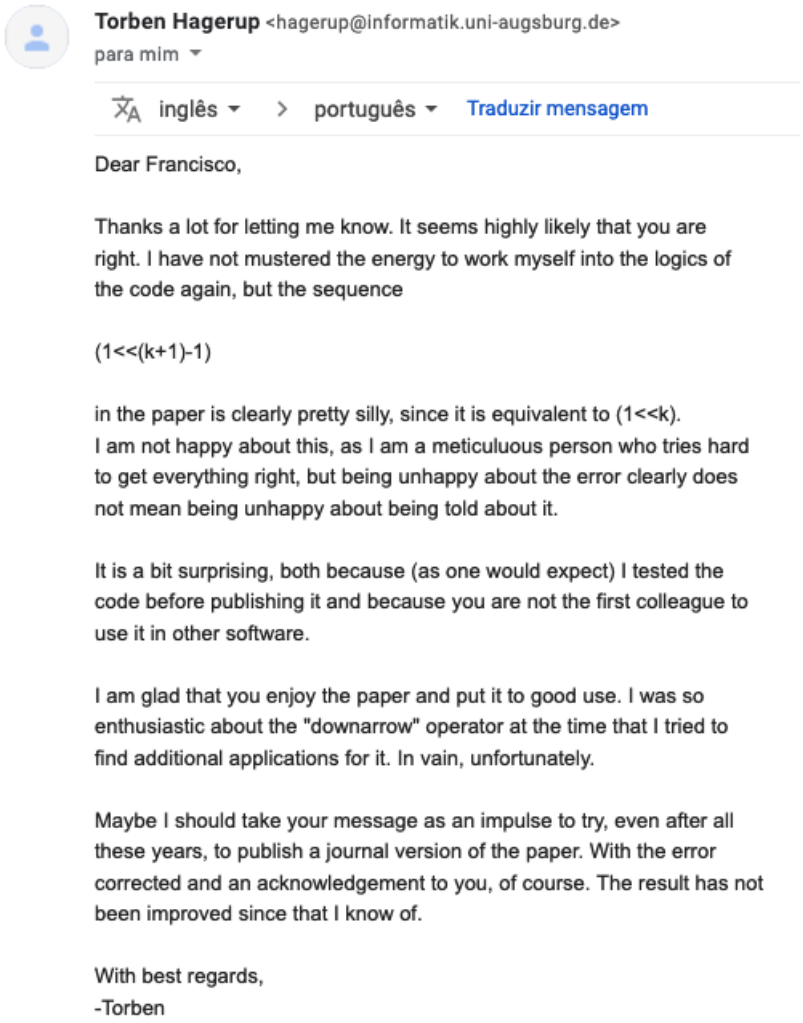
Seguindo a linha do pseudocódigo disponível na seção sobre o algoritmo KKT, em determinado momento do código precisamos filtrar apenas as LightEdges de um grafo G'', com relação a uma FGM F_1. Essa tarefa é equivalente a remover as arestas de G'' que são F_1-Heavy. Note que uma aresta de $G'' = (u, v)$ com custo w é F_1-Heavy se existe um caminho em F_1 que liga u a v no qual a aresta de maior custo do caminho tem custo $< w$.

Esse problema é estudado na literatura como o problema de verificação de árvores geradora mínimas, uma vez que se temos um palpite de árvore geradora mínima $T \in G$ e queremos garantir sua minimalidade, basta conferir que todas as arestas de G são $T - Light$. A primeira solução em tempo linear para esse problema foi descrita pela primeira vez em Dixon-Rauch-Tarjan em 1992 (Dixon, Rauch e Robert E. Tarjan 1992). Em 1995, King encontrou uma versão mais simples de resolver esse problema com a mesma complexidade e publicou sob o título de "A Simpler Minimum Spanning Tree Verification Algorithm"(King 1997), mas que ainda assim era razoavelmente complexo. Em 2010, Hagerup publicou uma versão consideravelmente mais simples, em que um grafo qualquer G era primeiro reduzido para uma

Full Branching Tree (árvore em que todos os nós internos tem ao menos 2 filhos, e todas as folhas estão na mesma altura) em tempo linear para depois efetuar a verificação, seu trabalho foi publicado com o nome de "An Even Simpler Linear-Time Algorithm for Verifying Minimum Spanning Trees"(Hagerup 2009), fazendo alusão ao trabalho anterior publicado por King.

Para esse projeto, optamos por utilizar a versão baseada no trabalho de Hagerup. Um fator que pesou bastante para essa decisão é que o autor disponibilizou uma implementação de seu algoritmo na linguagem de programação D (Hagerup 2009), que serviu de referência para a implementação em C++. A implementação da verificação é razoavelmente complexa e utiliza de várias ideias interessantes que não serão discutidas em detalhe, mas não deixam de ensinar ideias e conceitos úteis, como o algoritmo de Farach-Colton e Bender para responder consultas de menor ancestral comum em uma árvore em $O(1)$, com preprocessamento em $O(n)$ (Bender e Farach-Colton 2000).

Uma curiosidade é que durante a implementação do algoritmo KKT, o sistema de testes exaustivos retornava um erro da ordem de $\approx 1\%$ em algumas instâncias, sendo que as principais partes do projeto já tinham sido validadas extensivamente. Após muito tempo, descobriu-se que o erro estava no código da verificação fornecido por Hagerup (que serviu de inspiração para a implementação do projeto), em uma das operações sobre conjuntos. Houve uma troca de e-mails com o próprio Hagerup, que ficou surpresou ao saber que seu código de 10 anos de idade provavelmente tinha um erro, mesmo após ter sido utilizado múltiplas vezes no passado.



Torben Hagerup <hagerup@informatik.uni-augsburg.de>
para mim

inglês > português Traduzir mensagem

Dear Francisco,

Thanks a lot for letting me know. It seems highly likely that you are right. I have not mustered the energy to work myself into the logics of the code again, but the sequence

(1<<(k+1)-1)

in the paper is clearly pretty silly, since it is equivalent to (1<<k).
I am not happy about this, as I am a meticuluous person who tries hard to get everything right, but being unhappy about the error clearly does not mean being unhappy about being told about it.

It is a bit surprising, both because (as one would expect) I tested the code before publishing it and because you are not the first colleague to use it in other software.

I am glad that you enjoy the paper and put it to good use. I was so enthusiastic about the "downarrow" operator at the time that I tried to find additional applications for it. In vain, unfortunately.

Maybe I should take your message as an impulse to try, even after all these years, to publish a journal version of the paper. With the error corrected and an acknowledgement to you, of course. The result has not been improved since that I know of.

With best regards,
-Torben

Figura 28: Resposta de Hagerup ao ser contatado sobre o erro em questão.

6.1.4.4 Grafo de dependências

Para ter uma ideia de dependências da implementação de KKT neste projeto, podemos ver abaixo o próprio grafo de dependências.

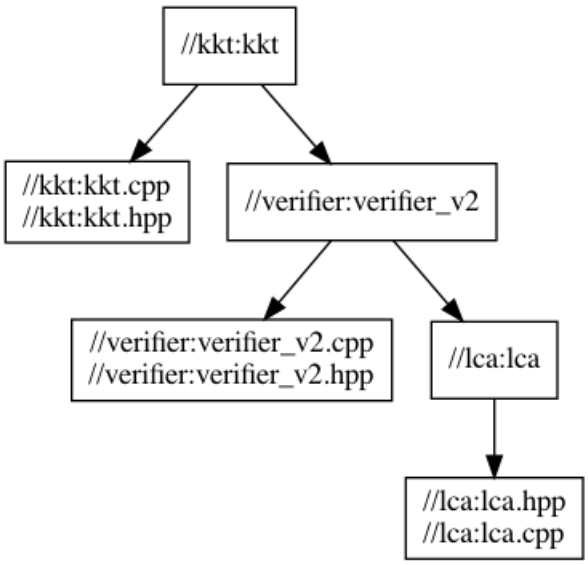


Figura 29: Dependências da implementação de KKT. Aqui, lca significa lowest common ancestor (menor ancestral comum) e o verifier é o módulo responsável pela verificação das arestas F-Heavy, discutido acima.

6.1.4.5 Resumo

Por fim, a implementação do KKT combina os 3 itens discutidos acima de forma similar a descrita no pseudocódigo do KKT, discutido na seção de Teoria. Contando com os includes e macros utilizados no código, sua implementação tem um total de 365 linhas. Algumas das funções utilizadas pelo KKT foram importados de outros módulos do projeto, mas se tudo fosse condensado em um único arquivo, acreditamos que 1000 linhas seria uma estimativa razoável de tamanho. Isso ilustra quão mais complexa a ideia do KKT com relação aos outros algoritmos de árvores geradoras mínimas, que geralmente não precisam de mais de 50 linhas para serem implementados em C++.

6.2 Testes

Acreditamos que testes são um aspecto muito importante para qualquer projeto de software, principalmente quando se está implementando um algoritmo especialmente complexo/complicado. Uma ênfase maior foi dada para garantir que a saída produzida pelos algoritmos era correta para diversas instâncias aleatórias. Os algoritmos implementados foram testados e validados para grafos conexos de diferentes tamanhos e densidade de arestas. Um indicador de quão forte foram os testes elaborados é que estes permitiram a descoberta um erro no código de um artigo de 10

anos de idade, que já foi utilizado outras vezes.

Por questões de familiaridade foi utilizada a biblioteca GoogleTest que é distribuída pelo Google com uma licença de código aberto BSD-3 desde 2008 (Google 2019). Essa biblioteca é utilizada internamente na empresa para o teste de código escrito em C++ e fornece diversas diretivas e macros para facilitar o desenvolvimento de testes e permite com que os testes sejam sempre executados quando uma mudança é feita em algum dos módulos da biblioteca, sem grande esforço adicional.

Cód. 5: Exemplo de teste seguindo a sintaxe do GoogleTest

```
1  TEST(positive_verification, random_graph)
2  {
3          auto random_graph = build_random_connected_graph(100, 500);
4          auto kruskal_mst = kruskal(random_graph, 100);
5          ASSERT_TRUE(verify_mst(random_graph, kruskal_mst, 100));
6          auto boruvka_mst = boruvka(random_graph, 100);
7          ASSERT_TRUE(verify_mst(random_graph, boruvka_mst, 100));
8          auto prim_mst = prim(random_graph, 100);
9          ASSERT_TRUE(verify_mst(random_graph, prim_mst, 100));
10 }
```

Durante o projeto, foram feitos uma grande quantidade de testes para garantir o funcionamento dos algoritmos para encontrar AGM e também foi dada ênfase em garantir que o algoritmo de verificação de AGM/FGM estava funcionando corretamente.

6.3 Resultados

Nessa seção, vamos comparar e analisar o desempenho e complexidade experimental dos algoritmos implementados ao longo do projeto.

6.3.1 Desempenho

Para avaliar as implementações de algoritmos de AGM, decidimos utilizar grafos de 10000 vértices e diferentes densidades de arestas, gerados de forma aleatória. Para discutir o desempenho dos algoritmos, vamos utilizar como base os tempos obtidos que estão exibidos na tabela abaixo.

	Kruskal	**Prim**	**Borůvka**	**KKT**
1250000	145.90ms	170.79ms	107.03ms	2349.63ms
2500000	305.69ms	284.41ms	335.25ms	5402.63ms
5000000	621.93ms	541.86ms	385.90ms	11672.84ms
10000000	1364.03ms	1117.15ms	1065.54ms	22823.18ms
20000000	2878.81ms	2813.01ms	2206.00ms	45424.80ms

Tabela 2: Tempos de execução dos algoritmos de MST implementados. No eixo vertical está exibida a quantidade de arestas do grafo correspondente a cada instância de teste.

Nota-se que o desempenho dos 3 algoritmos clássicos é similar, sendo o algoritmo de Borůvka um pouco mais rápido que os outros. Já o algoritmo de KKT é ao menos uma ordem de magnitude mais lento do que os outros algoritmos.

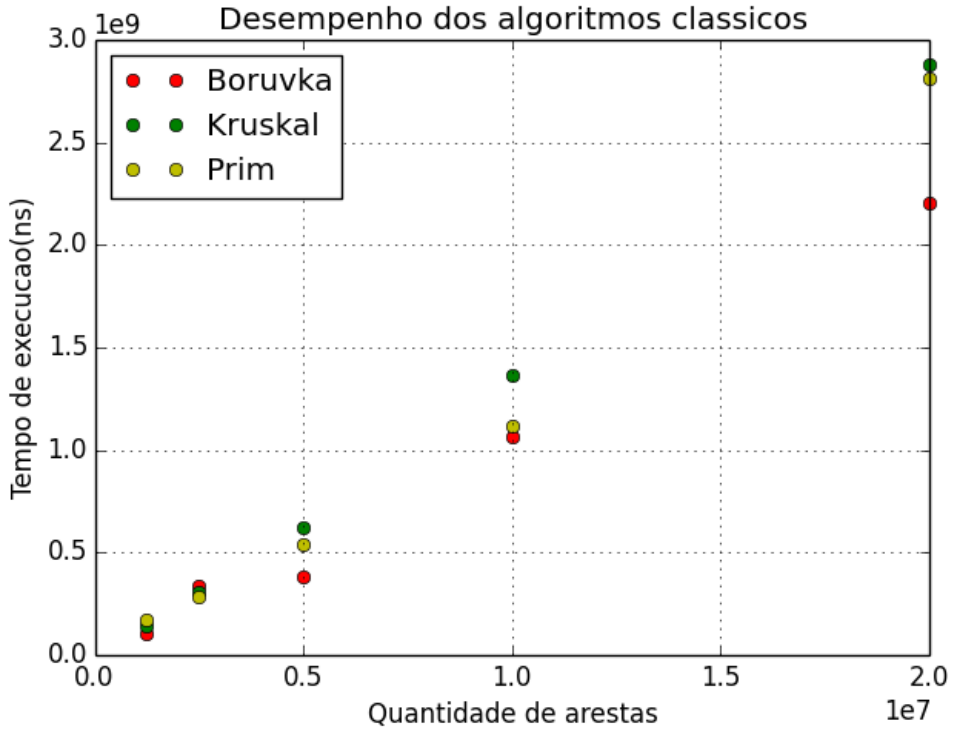


Figura 30: Tempos dos 3 algoritmos clássicos.

Na seção de Teoria foi discutida a complexidade teórica de cada um dos algoritmos implementados, mas não necessariamente observa-se essa mesma complexidade na prática. Para analisar a complexidade da implementação prática nas instâncias de teste, foi utilizada a biblioteca benchmark do Google que também distribuída com uma licença de código aberto (Google 2020a).

De forma bem resumida, a biblioteca mede o tempo de execução do código dentro do loop que menciona o *state*, garante que o código crítico cujo desempenho está tendo medido não é otimizado pelo compilador e permite com que o argumento n para a análise de complexidade assintótica seja manualmente configurado. Abaixo segue um exemplo do trecho de benchmark utilizado para o KKT. A funcionalidade mais surpreendente é que a própria biblioteca testa várias funções de complexidades comuns e indica a que melhor se encaixou, levando em conta a raiz do erro quadrático médio e também exibe a constante da complexidade. A biblioteca também decide em tempo de execução quantas iterações serão feitas de cada algoritmo, para poder determinar de forma consistente o tempo tomado pelo programa.

Cód. 6: Exemplo de código de benchmark usando a biblioteca do Google.

```
1 static void bm_kkt(benchmark::State& state)
2 {
3         auto G = build_random_connected_graph(state.range(0), state.range(1));
4         auto P = problem(state.range(0), G);
5         for (auto _: state) { benchmark::DoNotOptimize(kkt(P)); }
6         state.SetComplexityN(state.range(0) + state.range(1));
7 }
8
9 BENCHMARK(bm_kkt)->Apply(CustomArguments)->Complexity();
```

Cód. 7: Saída produzida pela biblioteca de benchmark para os algoritmos implementados

```
Run on (4 X 1800 MHz CPU s)
CPU Caches:
  L1 Data 32 KiB (x2)
  L1 Instruction 32 KiB (x2)
  L2 Unified 256 KiB (x2)
  L3 Unified 3072 KiB (x1)
Load Average: 2.67, 3.04, 3.43
-------------------------------------------------------------------------------
Benchmark                                     Time             CPU   Iterations
-------------------------------------------------------------------------------
bm_boruvka/10000/1250000    112812997 ns     107026667 ns            6
bm_boruvka/10000/2500000    476104841 ns     335252500 ns            2
bm_boruvka/10000/5000000    386507383 ns     385897500 ns            2
bm_boruvka/10000/10000000  1093499487 ns    1065545000 ns            1
bm_boruvka/10000/20000000  2267219934 ns    2205997000 ns            1
bm_boruvka_BigO                  4.67 NlgN        4.52 NlgN
bm_boruvka_RMS                     14 %             8 %
bm_kkt/10000/1250000       2390859693 ns    2349635000 ns            1
bm_kkt/10000/2500000       6121601931 ns    5402627000 ns            1
bm_kkt/10000/5000000      14439639388 ns   11672843000 ns            1
bm_kkt/10000/10000000     24511124937 ns   22823183000 ns            1
bm_kkt/10000/20000000     46079279988 ns   45423793000 ns            1
bm_kkt_BigO                   2357.85 N        2272.15 N
bm_kkt_RMS                         7 %              2 %
bm_kruskal/10000/1250000    148066775 ns     145902800 ns            5
bm_kruskal/10000/2500000    311809275 ns     305691000 ns            2
bm_kruskal/10000/5000000    625207382 ns     621930000 ns            1
bm_kruskal/10000/10000000  1369298067 ns    1364027000 ns            1
bm_kruskal/10000/20000000  2895922720 ns    2878812000 ns            1
bm_kruskal_BigO                  5.94 NlgN        5.90 NlgN
bm_kruskal_RMS                     2 %              2 %
bm_prim/10000/1250000       172899655 ns     170785750 ns            4
bm_prim/10000/2500000       285907010 ns     284413000 ns            3
bm_prim/10000/5000000       548358336 ns     541855000 ns            1
bm_prim/10000/10000000     1143797262 ns    1117153000 ns            1
bm_prim/10000/20000000     3112637341 ns    2813013000 ns            1
bm_prim_BigO                     6.08 NlgN        5.58 NlgN
bm_prim_RMS                       15 %             10 %
```

O resultado do benchmark indica que a implementação do KKT condiz com a complexidade teórica de $O(n + m)$ nas instâncias de teste, mas com uma constante bem maior do que a dos outros algoritmos. O gráfico abaixo mostra que uma simples ligação dos pontos medidos se aproxima bastante de uma reta.

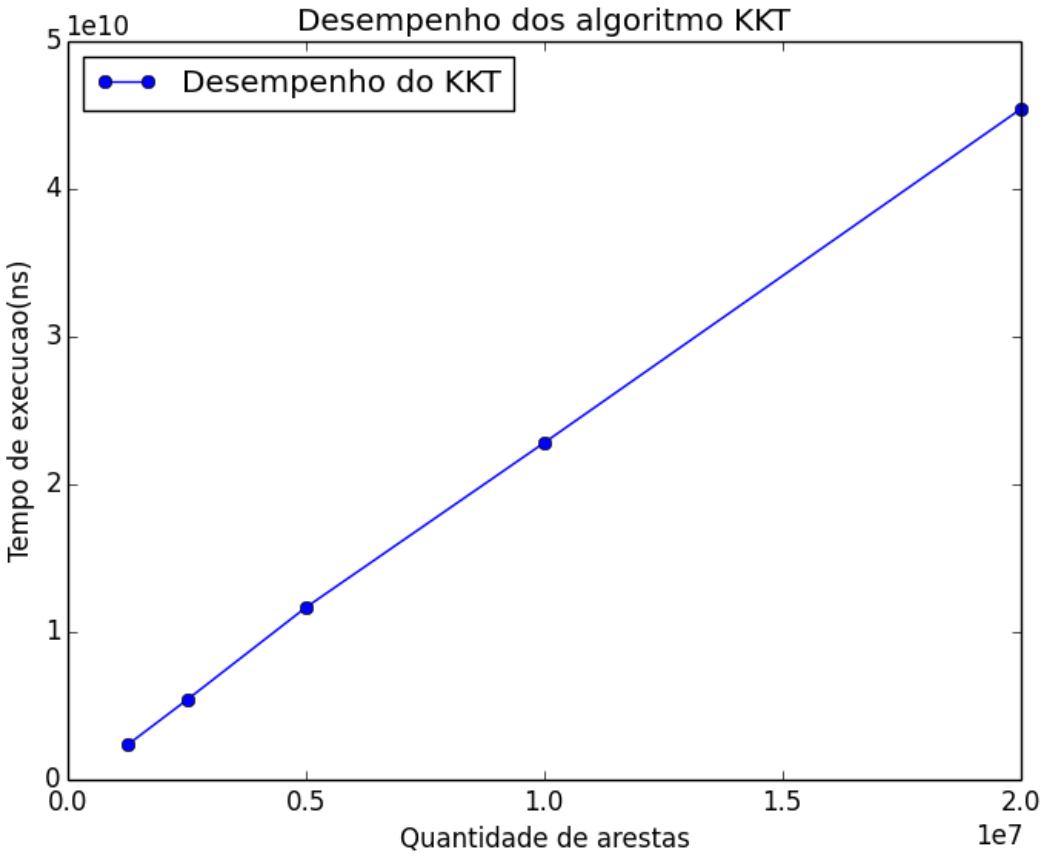


Figura 31: Tempo do KKT para as instâncias de teste.

É possível concluir trivialmente que por mais que o algoritmo de KKT tenha uma complexidade assintótica "melhor"que os outros algoritmos, isso acaba não sendo tão relevante nas instâncias de teste, pelo fato da implementação do projeto possuir uma constante bem grande na complexidade. Entretanto, podemos utilizar a constante fornecida pela biblioteca de benchmark para estimar a partir de que tamanho de instâncias o algoritmo de KKT tomaria menos tempo que o algoritmo de Borůvka, o mais rápido dos algoritmos clássicos. Usando as estimativas produzidas pelo benchmark, basta resolver a seguinte inequação:

$$2357.85 * N < 4.67 * N * \log_2(N)$$

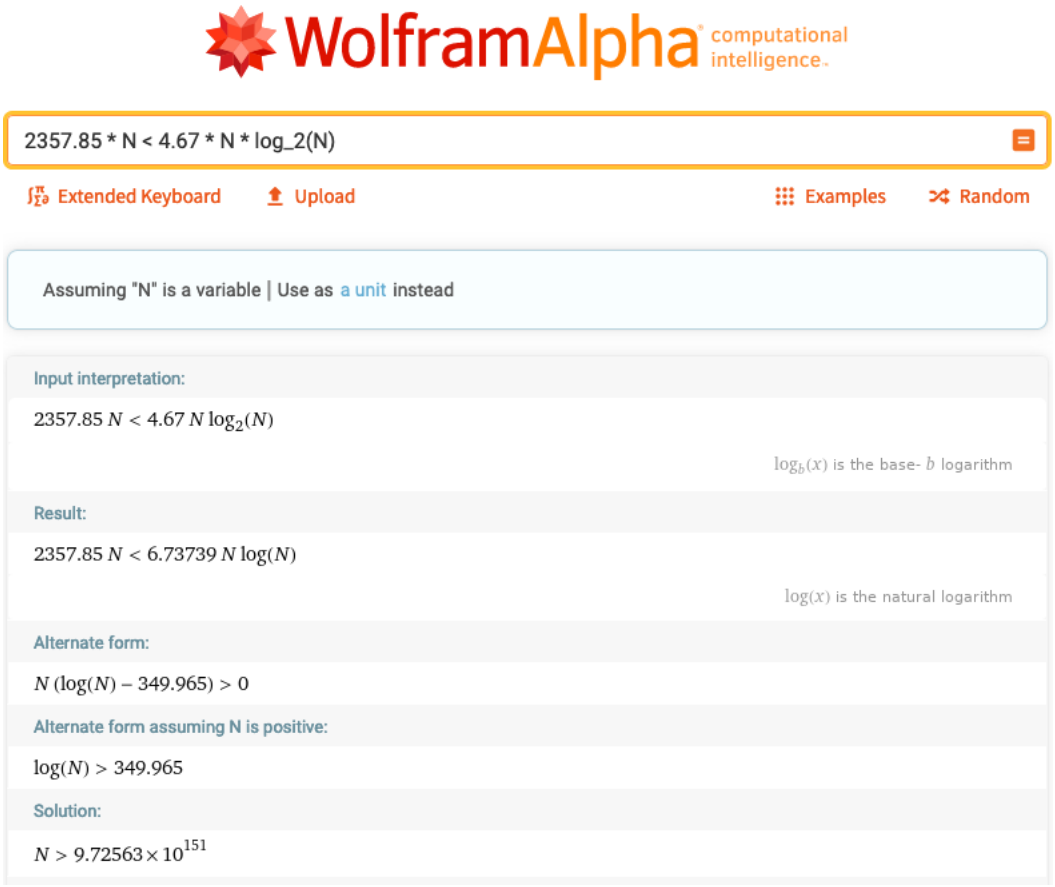


Figura 32: Solução da inequação acima, obtida no website www.wolframalpha.com Research 2009

A solução da inequação nos diz que, dada a implementação do projeto do algoritmo KKT e do algoritmo de Borůvka e a estimativa de complexidade dos dois algoritmos, o algoritmo de KKT seria mais rápido para grafos $G(n, m)$, onde $n+m > 9.7*10^{151}$. Estimativas de astronomia moderna baseadas em uma das equações de Friedmann Friedmann 1920 colocam o número de átomos no universo observável em $\approx 10^{80}$, o menor grafo que daria vantagem ao KKT teria uma soma de vértices e arestas maior do que a quantidade de átomos no universo observável. Além disso, resultados publicados em 2007 estimam a capacidade total de armazenamento da humanidade em $\approx 2.9^{20}$, o que significa que estamos muito longe de conseguir armazenar um grafo desse tamanho para passar como entrada para o algoritmo. Isso é um excelente exemplo de que não necessariamente um algoritmo com complexidade assintótica melhor vai performar melhor que outro em instâncias de tamanho prático.

6.4 Tecnologias utilizadas

Nessa subseção, vamos abordar brevemente as tecnologias que foram utilizadas durante o desenvolvimento da parte prática deste projeto.

6.4.1 GIT

Utilizamos o git para realizar o controle de versão durante a implementação do projeto. O repositório do projeto está hospedado no github.com Microsoft 2008 e será aberto ao público e disponibilizado com uma licença MIT de código aberto.

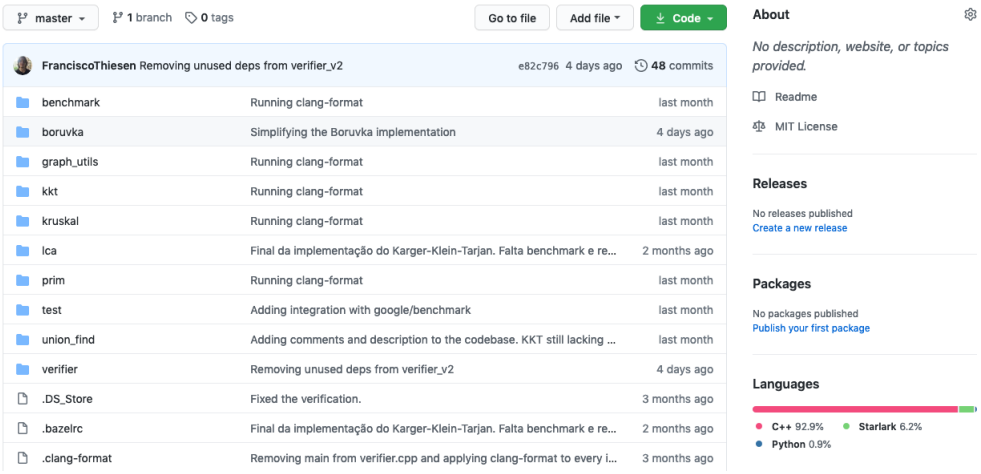


Figura 33: Imagem do repositório e sua organização.

6.4.2 Overleaf + LaTeX

Este relatório foi escrito utilizando o site overleaf.com (Overleaf 2012), que permite com que o desenvolvimento de arquivos em LaTeX seja feito no navegador de internet (Lamport 2001). Isso permite que a escrita seja feita em praticamente qualquer dispositivo, sem preocupações com instalação de LaTeX e compatibilidade com sistemas operacionais.

6.4.3 Clang-Format

O clang-format é uma ferramenta que permite a formatação de código em C, C++, Java, JavaScript, Objective-C, Protobuf e C# de acordo com um arquivo de configuração (Team 2020). Foi utilizado ao longo do projeto para deixar a base de código com um estilo uniforme. O arquivo de configuração do clang-format foi construído usando o site zed0.co.uk que permite construir o arquivo de forma dinâmica, vendo como a mudança de parâmetros altera o código.

6.4.4 Bazel

Ferramenta com licença de código-aberto desenvolvida pelo Google para permitir a automação de construção de software e testes (Source 2020). Ela foi feita baseada na ferramenta interna do Google, chamada Blaze. É compatível com projetos em C++, Java, Android, Go, Python e Objective-C. Várias grandes empresas fazem uso desse sistema de build, como: Pinterest, Dropbox, Google, Huawei e Stripe (**Bazel**). Alguns dos benefícios que essa ferramenta oferece são:

- Análise de dependência otimizada, para que sejam recompilados apenas os módulos necessários. Melhorando assim o tempo para construir o projeto.
- Possibilidade do uso de cache local e distribuído.
- Compatível com diversas linguagens, mas com a mesma sintaxe. Contribuindo dessa forma para uma base mais uniforme.
- Altamente escalável para lidar com bases de dados de diferentes tamanhos. O fato de ser inspirada na ferramenta própria do Google diz muito sobre esse aspecto.

6.4.5 Google Test

Biblioteca com licença de código aberto, também desenvolvida pelo Google. É basicamente um arcabouço para testes de projetos em C++, utilizado em diversos projetos famosos de código aberto, como: Chromium, OpenCV e o compilador LLVM. É bem conveniente de ser integrado ao sistema de automação Bazel. (Google 2019)

Cód. 8: Exemplo de teste presente na documentação da Google Test.

```
1  // Tests Dequeue().
2  TEST_F(QueueTest, Dequeue) {
3    int* n = q0_.Dequeue();
4    EXPECT_TRUE(n == nullptr);
5
6    n = q1_.Dequeue();
7    EXPECT_TRUE(n != nullptr);
8    EXPECT_EQ(1, *n);
9    EXPECT_EQ(0u, q1_.Size());
10   delete n;
11
12   n = q2_.Dequeue();
13   EXPECT_TRUE(n != nullptr);
14   EXPECT_EQ(2, *n);
15   EXPECT_EQ(1u, q2_.Size());
16   delete n;
17 }
```

6.4.6 Google Benchmark

Biblioteca com licença de código aberto, desenvolvida pelo Google. Ela visa facilitar a elaboração de benchmarks e microbenchmarks para projetos de C++. Possui diversas funcionalidades interessantes, como a determinação da quantidade de iterações que cada teste é efetuado de forma dinâmica e a possibilidade de estimar a complexidade assintótica de uma determinada implementação. (Google 2020a)

Cód. 9: Exemplo de benchmark fornecido na documentação da biblioteca Google Benchmark.

```
#include <benchmark/benchmark.h>

static void BM_SomeFunction(benchmark::State& state) {
  // Perform setup here
  for (auto _ : state) {
    // This code gets timed
    SomeFunction();
  }
}
// Register the function as a benchmark
BENCHMARK(BM_SomeFunction);
// Run the benchmark
BENCHMARK_MAIN();
```

7 Considerações Finais

Nesse projeto apresentamos a teoria e implementação dos algoritmos clássicos de árvore geradora mínima e também do algoritmo de Karger-Klein-Tarjan, que possui a melhor complexidade esperada para grafos gerais da atualidade ($O(n+m)$). As instâncias de teste selecionadas acabaram revelando que por mais que o algoritmo de Karger-Klein-Tarjan possua uma complexidade assintótica melhor, a implementação dele teve um desempenho pior do que os algoritmos clássicos da literatura para grafos de qualquer tamanho que seja passível de ser armazenado em um computador atual.

As implementações do projeto foram constantemente testadas, de modo a garantir o funcionamento dos algoritmos em questão. Inclusive, os testes rigorosos permitiram com que encontrássemos um erro em um artigo de 10 anos de idade, erro esse que foi reconhecido por autor via e-mail. Também foi feita uma estimativa da complexidade de cada uma das implementações, e as estimativas estavam sempre alinhadas com a complexidade assintótica teórica, o que de certa forma ajuda a validar que o algoritmo foi implementado de maneira razoável e correta.

O algoritmo de Karger-Klein-Tarjan ensina a valiosa lição de que utilizar passos aleatórios no algoritmo, combinando bem os subproblemas pode melhorar a complexidade de algoritmos. Alguns outros exemplos de algoritmos que usam aleatoriedade para obter uma complexidade melhor são:

- Algoritmo de Welzl. Esse algoritmo busca encontrar o menor círculo que engloba um conjunto de pontos no plano. A solução tem complexidade $O(n)$, onde n é o número de pontos. (Welzl 1991)
- Algoritmo de Karger-Stein, que busca encontrar o corte mínimo de um grafo. (D. Karger 1993)

Após o final do semestre, esse código será disponibilizado publicamente com uma licença permissiva de código aberto, para que outros possam estudá-lo e/ou utilizá-lo como referência. Vale lembrar que no segundo semestre de 2020 não havia nenhuma implementação do algoritmo de Karger-Klein-Tarjan facilmente disponível na web, então essa implementação é pioneira nesse aspecto.

7.1 Próximos passos

Após a publicação em plataforma de código aberto das implementações desse projeto, outro desafio interessante é implementar a melhor versão determinística conhecida, proposta por Bernard Chazelle (Chazelle 2000) que funciona em $O(m * \alpha(n, m))$.

8 Referências

Antoš, Karel (2016). "The Use of Minimal Spanning Tree for Optimizing Ship Transportation". Em: *Naše more, Special Issue* 63.3, pp. 81–85.

Asano, T., B. Bhattacharya e M. Keil (1988). "Clustering algorithms based on minimum and maximum spanning trees." Em: *Fourth Annual Symposium on Computational Geometry* 1, pp. 252–257.

Assunção, R. M., M. C. Neves, G. Câmara e C. Da Costa Freitas (2006). "Efficient regionalization techniques for socio-economic geographical units using minimum spanning trees". Em: *International Journal of Geographical Information Science* 20.7, pp. 797–811.

Bender, Michael A. e Martin Farach-Colton (2000). "The LCA problem revisited". Em: *Proceedings, Lecture Notes in Computer Science* 1776, pp. 88–94.

Boost Graph Library (2001). Versão 1.74.9. URL: `https://www.boost.org/doc/libs/1_74_0/libs/graph/doc/index.html`.

Borůvka, Otakar (1926a). "O jistém problému minimálním [About a certain minimal problem]". Em: *Práce Moravské přírodovědecké společnosti (in Czech and German)* 3, pp. 37–58.

— (1926b). "Příspěvek k řešení otázky ekonomické stavby elektrovodních sítí (Contribution to the solution of a problem of economical construction of electrical networks)". Em: *Elektronický Obzor (in Czech)* 15, pp. 153–154.

C++ implementation of set (2020). URL: `http://www.cplusplus.com/reference/set/set/`.

Chazelle, Bernard (2000). "A minimum spanning tree algorithm with inverse-Ackermann type complexity". Em: *Journal of the Association for Computing Machinery* 47.6, pp. 1028–1047.

Christofides, Nico (1976). *Worst-case analysis of a new heuristic for the travelling salesman problem.*

Cytoscape (2002). Versão 3.8.2. URL: `https://cytoscape.org/`.

Dahihaus, E, D. S Johnson, C. H PApadimitriou, P. D Seymour e M Yannakakis (1994). "The complexity of multiterminal cuts". Em: *SIAM Journal on Computing* 23.4, pp. 864–894.

Dalal, Yogen K. e Robert M. Metcalfe (1978). "Reverse path forwarding of broadcast packets". Em: *Communications of the ACM* 21.12, pp. 1040–1048.

Devillers, J. e J.C. Dore (1989). “Heuristic potency of the minimum spanning tree (MST) method in toxicology”. Em: *Ecotoxicology and Environmental Safety* 17.2, pp. 227–235.

Dijkstra, E. W. (1959). “A note on two problems in connexion with graphs"”. Em: *Numerische Mathematik* 1.1, pp. 269–271.

Dixon, Brandon, Monika Rauch e Robert E. Tarjan (1992). “Verification and Sensitivity Analysis of Minimum Spanning Trees in Linear Time”. Em: *SIAM Journal on Computing* 21.6, p. 1184.

Felzenszwalb, P. e D. Huttenlocher (2000). “Efficient Graph-Based Image Segmentation”. Em: *International Conference on Image Processing* 1, pp. 481–484.

Filliben, James J., Karen Kafadar e Douglas R. Shier (1983). “Testing for homogeneity of two-dimensional surfaces”. Em: *Mathematical Modelling* 4.2, pp. 167–189.

Fredman, M. L. e R. E. Tarjan (1987). “Fibonacci heaps and their uses in improved network optimization algorithms”. Em: *Journal of the ACM* 34.3.

Friedmann, Alexander (1920). *Friedmann Equation*. URL: `http://hyperphysics.phy-astr.gsu.edu/hbase/Astro/fried.html`.

Google (2019). *Google Test*. Versão 1.10.0. URL: `https://github.com/google/googletest`.

— (2020a). *Google Benchmark*. Versão 1.5.2. URL: `https://github.com/google/benchmark`.

— (2020b). *Google Trends*. URL: `https://trends.google.com/trends/`.

Gower, J. C. e G. J. S Roos (1969). “Minimum Spanning Trees and Single Linkage Cluster Analysis”. Em: *Journal of the Royal Statistical Society. C (Applied Statistics* 18.1, pp. 54–64.

Hagberg, Aric, Pieter Swart e Dan Schult (2005). *NetworkX*. URL: `https://networkx.org/`.

Hagerup, T. (2009). “An Even Simpler Linear-Time Algorithm for Verifying Minimum Spanning Trees”. Em: *Graph-Theoretic Concepts in Computer Science* 5911, pp. 178–189.

http://e-maxx.ru/ (2014). *Prim Implementation*. URL: `https://cp-algorithms.com/graph/mst_prim.html`.

Jarník, V. (1930). “O jistém problému minimálním"[About a certain minimal problem]”. Em: *Práce Moravské Přírodovědecké Společnosti (in Czech)* 6.4, pp. 57–63.

JGraphT (2000). Versão 1.5.0. URL: `https://jgrapht.org/`.

Kadiva, Mehdi e Mohammad E. Shiri (2011). “An adaptive MST-based topology connectivity control algorithm for wireless ad-hoc networks”. Em: *International Journal of Communication Networks and Distributed Systems* 6.1, pp. 79–96.

Kalaba, Robert E. (1963). *Graph Theory and Automatic Control*.

Karger, David (1993). “Global Min-cuts in RNC and Other Ramifications of a Simple Mincut Algorithm”. Em: *ACM-SIAM Symposium on Discrete Algorithms* 4.

Karger, David R., Philip N. Klein e Robert E. Tarjan (1995). “A randomized linear-time algorithm to find minimum spanning trees”. Em: *Journal of the Association for Computing Machinery* 42.2, pp. 321–328.

King, Valerie (1997). “A Simpler Verification Spanning Tree Verification Algorithm”. Em: *Algorithmica* 18, pp. 263–270.

Kruskal, J. B. (1956). “On the shortest spanning subtree of a graph and the traveling salesman problem”. Em: *Proceedings of the American Mathematical Society* 7.1, pp. 48–50.

Lamport, Leslie (2001). *Latex*. URL: `https://pt.wikipedia.org/wiki/LaTeX`.

Li, Ning, J. C. Hou e L. Sha (2005). “Design and analysis of an MST-based topology control algorithm”. Em: *IEEE Transactions on Wireless Communications* 4.5, pp. 1195–1206.

Ma, B., A. Hero, J. Gorman e O. Michel (2000). “Image registration with minimum spanning tree algorithm”. Em: *International Conference on Image Processing* 1, pp. 481–484.

Microsoft (2008). *github*. URL: `https://www.github.com`.

Mori, H. e S. Tsuzuki (1991). “A fast method for topological observability analysis using a minimum spanning tree technique”. Em: *IEEE Transactions on Power Systems* 6.2, pp. 491–500.

Ohlsson, H. (2004). “RImplementation of low complexity FIR filters using a minimum spanning tree.” Em: *12th IEEE Mediterranean Electrotechnical Conference* 1, pp. 261–264.

Overleaf (2012). *Overleaf*. URL: `https://github.com/overleaf/overleaf`.

Päivinen, Niina (2005). "Clustering with a minimum spanning tree of scale-free-like struture". Em: *Pattern Recognition Letters* 26.7, pp. 921–930.

Prim, R. C. (1957). "Shortest connection networks And some generalizations". Em: *Bell Systems Technical Journal* 36.6, pp. 1389–1401.

Research, Wolfram (2009). *wolframalpha.com*. URL: `https://www.wolframalpha.com/`.

Santhi, S.V.S e P. Padmaja (2016). "An efficient and refined minimum spanning tree approach for large water distribution system". Em: *International Journal of Communication Networks and Distributed Systems* 2.1,2,3, pp. 181–206.

Sneath, P. H. A (1957). "The Application of Computers to Taxonomy". Em: *Journal of General Microbiology* 17.1, pp. 201–226.

Source, Google Open (2020). *Bazel*. Versão 3.7.0. URL: `https://github.com/bazelbuild/bazel`.

Suk, Minsoo e Ohyoung Song (1984). "Curvilinear feature extraction using minimum spanning trees". Em: *Computer Vision, Graphics, and Image Processing* 26.3, pp. 400–411.

Supowit, Kenneth J, David A Plaisted e Edward M. Reingold (1980). "Heuristics for weighted perfect matching". Em: *12th Annual ACM symposium on Theory of Computing*, pp. 398–419.

Tapia, Ernesto e Raúl Rojas (2004). "Recognition of On-line Handwritten Mathematical Expressions Using a Minimum Spanning Tree Construction and Symbol Dominance". Em: *Graphics Recognition. Recent Advances and Perspectives. Lecture Notes in Computer Science* 3088, pp. 329–340.

Tarjan, R.E. (1975). "Efficiency of a Good But Not Linear Set Union Algorithm". Em: *Journal of the ACM* 22.2, pp. 215–225.

Team, The Clang (2020). *ClangFormat*. Versão 12. URL: `http://clang.llvm.org/docs/ClangFormat.html`.

Understanding Multiple Spanning Tree Protocol (802.1s) (2020). URL: `https://www.cisco.com/c/en/us/support/docs/lan-switching/spanning-tree-protocol/24248-147.html`.

Wayne, Kevin (2003). *Slide com os tempos dos algoritmos de árvores geradoras mínimas*. URL: `https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pearson/04MinimumSpanningTrees-2x2.pdf`.

Welzl, Emo (1991). "Smallest enclosing disks (balls and ellipsoids)". Em: *New Results and New Trends in Computer Science, Lecture Notes in Computer Science* 555, pp. 359–370.

Wikipedia, the free encyclopedia (2001a). *Borůvka*. URL: https://pt.wikipedia.org/wiki/Algoritmo_de_Bor%C5%AFvka.

— (2001b). *Borůvka Step*. URL: https://en.wikipedia.org/wiki/Expected_linear_time_MST_algorithm#Bor%C5%AFvka_Step.

— (2001c). *Busca em Largura*. URL: https://pt.wikipedia.org/wiki/Busca_em_largura.

— (2001d). *Busca em Profundidade*. URL: https://pt.wikipedia.org/wiki/Busca_em_profundidade.

— (2001e). *Cut Property*. URL: https://en.wikipedia.org/wiki/Minimum_spanning_tree#Cut_property.

— (2001f). *Cycle Property*. URL: https://en.wikipedia.org/wiki/Minimum_spanning_tree#Cycle_property.

— (2001g). *Filter*. URL: https://en.wikipedia.org/wiki/Expected_linear_time_MST_algorithm#F-heavy_and_F-light_edges.

— (2001h). *Imagem de árvore geradora mínima de um grafo*. URL: https://en.wikipedia.org/wiki/Minimum_spanning_tree#/media/File:Minimum_spanning_tree.svg.

— (2001i). *Inverse Ackermann Bound*. URL: https://en.wikipedia.org/wiki/Ackermann_function#Inverse.

— (2001j). *Kruskal*. URL: https://pt.wikipedia.org/wiki/Algoritmo_de_Kruskal.

— (2001k). *Lema da Amostragem Aleatória*. URL: https://en.wikipedia.org/wiki/Expected_linear_time_MST_algorithm#Random_sampling_lemma.

— (2001l). *Union-Find information*. URL: https://en.wikipedia.org/wiki/Disjoint-set_data_structure.

Xu, Y, V. Olamn e D. Xu (2002). "Clustering gene expression data using a graph-theoretic approach: an application of minimum spanning trees". Em: *Bioinformatics* 18.4, pp. 536–545.