UNIVERSIDADE FEDERAL DO PIAUÍ – UFPI
CENTRO DE CIÊNCIAS DA NATUREZA – CCN
DEPARTAMENTO DE COMPUTAÇÃO – DC
DOCENTE: ANTÔNIO COSTA DE OLIVEIRA
DISCIPLINA: TEORIA E APLICAÇÕES EM GRAFOS

Aplicação de Árvore Geradora Mínima em Grafos

Gabriel Lopes Bastos
José Victor Vieira de Oliveira
Pedro Emanuel Moreira Carvalho
Thalys Yago Silva Nascimento

Teresina
2024

INTRODUÇÃO

Uma Árvore Geradora Mínima (MST - Minimum Spanning Tree) é um subgrafo de um grafo conexo e ponderado (as arestas possuem peso) que conecta todos os vértices do grafo original com o menor custo possível, sem a formação de ciclos. Em outras palavras, é uma árvore que inclui todos os vértices do grafo e um subconjunto das arestas, tal que a soma dos pesos das arestas na árvore é mínima.

A MST tem diversas aplicações práticas em áreas como redes de computadores, design de circuitos, clustering de dados e muito mais. A determinação de uma MST pode ser resolvida por diferentes algoritmos, entre os quais destacam-se os algoritmos de **Prim** e **Kruskal**.

O algoritmo de Prim constroi a MST de maneira iterativa, começando de um vértice arbitrário e expandindo a árvore, adicionando a aresta de menor peso que conecta um vértice da árvore a um vértice fora dela. O processo continua até que todos os vértices estejam incluídos na árvore geradora mínima.

O algoritmo de Kruskal, por outro lado, constroi a MST selecionando arestas em ordem crescente de peso, garantindo que nenhuma aresta adicionada forme um ciclo à árvore em construção. A abordagem de Kruskal se baseia no conceito de conjuntos disjuntos (*Union-Find*) para gerenciar os componentes conectados do grafo e verificar se a adição de uma nova aresta formará um ciclo.

METODOLOGIA

Utilizou-se a linguagem Python para a realização do trabalho. Inicialmente implementamos os algoritmos de uma forma simplória apenas para verificar a funcionalidade, logo após isso, aplicou-se a modularização para organização do código em uma classe e em métodos, com o desenvolvimento de uma classe chamada “Grafo”, com os devidos algoritmos de Prim e de Kruskal embarcados em métodos da mesma, utilizando como estrutura primária uma lista de adjacência. Observa-se que a modularização é importante, uma vez que a interface necessita consumir essa classe para o seu funcionamento, logo, o código deve estar bem estruturado.

Para a implementação do algoritmo de Prim, utilizamos uma função que percorre todas as arestas de todos os vértices “verificados” (que já fazem parte da MST) e adiciona o menor deles, sempre se atentando para caso forme ciclos.

Para a implementação do algoritmo de Kruskal, por culpa da técnica de *Union-Find*, além da função para construção da MST por Kruskal, tem-se uma classe dedicada a realizar a *union-find*. No caso de Kruskal, desde o começo já se tem, em ordem crescente, todas as arestas do grafo original que apenas são adicionadas da menor para a maior, sempre se atentando para caso forme ciclos.

Para a criação da interface, foi criada uma classe especial chamada *grafo_gui*, responsável por tal tarefa. Foi feita uma janela com o canvas, que tem a função de mostrar graficamente os vértices e as arestas, e cinco botões, sendo eles: “Adicionar”, que fazia o input de novos elementos para a classe *grafo_gui*; “Resetar” que removia tudo que já estava adicionado, reiniciando o grafo de *grafo_gui*; “Grafo Aleatório”, que pergunta a quantidade de vértices e se o grafo é completo ou não para formar um grafo aleatório; e por fim “Prim” e “Kruskal” que fazem a chamada do respectivo algoritmo e em seguida permite o canvas a expor a ordem de inserção dos elementos da Árvore Geradora Mínima. Os algoritmos feitos fora da classe foram consumidos por meio da modularização, que para ser utilizada primeiro precisava que as estruturas de dados fossem “traduzidas”, já que a classe *grafo_gui* utiliza a representação do grafo por meio de listas, que são traduzidas para lista de adjacência e depois tem seu retorno traduzido novamente para definir as arestas que serão selecionadas.

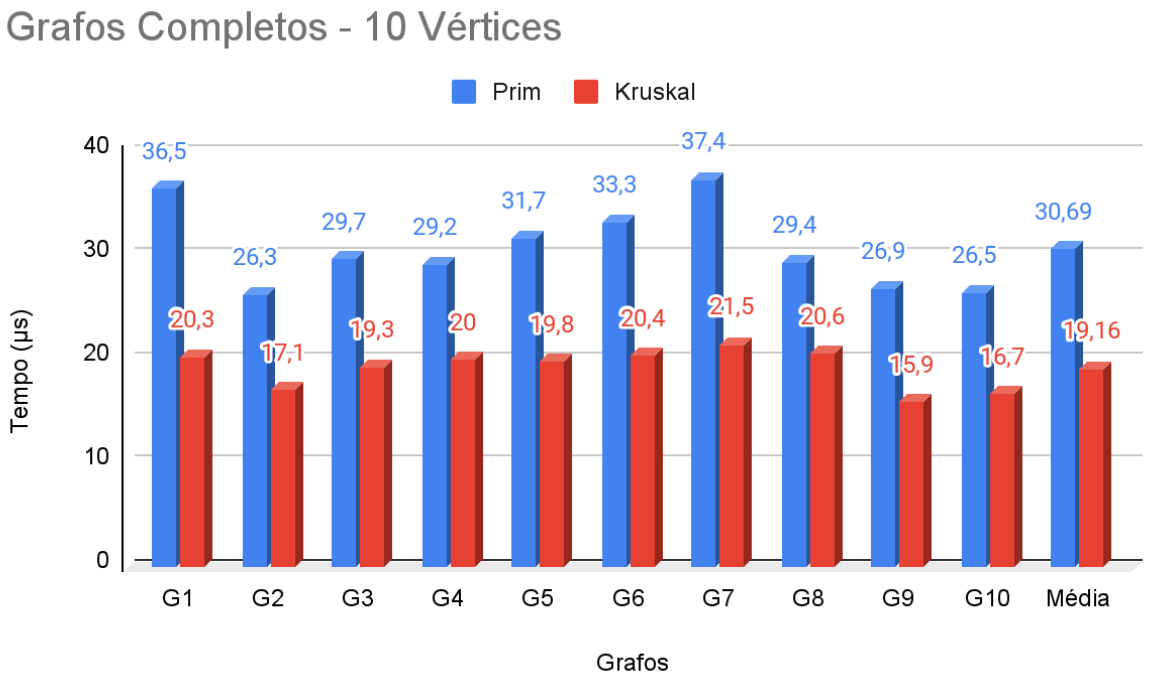
Para a realização dos testes foram utilizados grafos completos e incompletos com dez, quinze e vinte vértices, sendo dez grafos de cada categoria. O tempo médio de execução dos algoritmos para cada grafo foi de dez execuções em um Macbook Air M1 e a unidade de medida de tempo foi o microssegundo (μs).

RESULTADOS

Para os resultados, utilizamos os valores adquiridos nos testes e colocamos em tabelas e geramos gráficos para cada “grupo” de grafos.

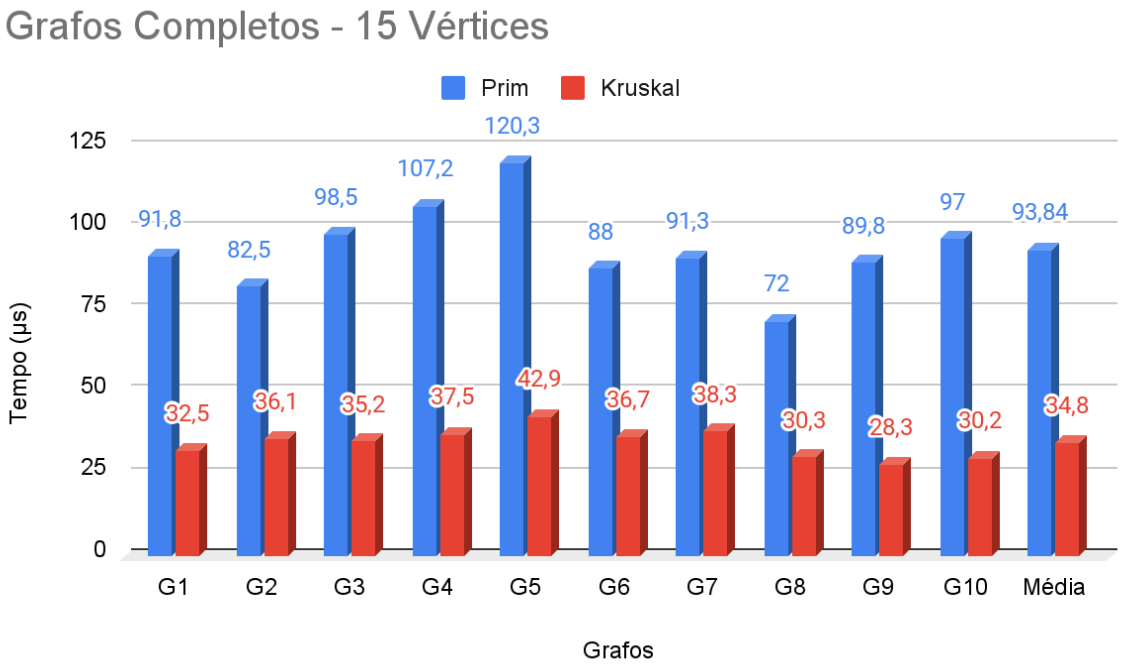
Grafos 10-Completo

	G1	G2	G3	G4	G5	G6	G7	G8	G9	G10	**Média**
Prim	36,5	26,3	29,7	29,2	31,7	33,3	37,4	29,4	26,9	26,5	30,69
Kruskal	20,3	17,1	19,3	20	19,8	20,4	21,5	20,6	15,9	16,7	19,16



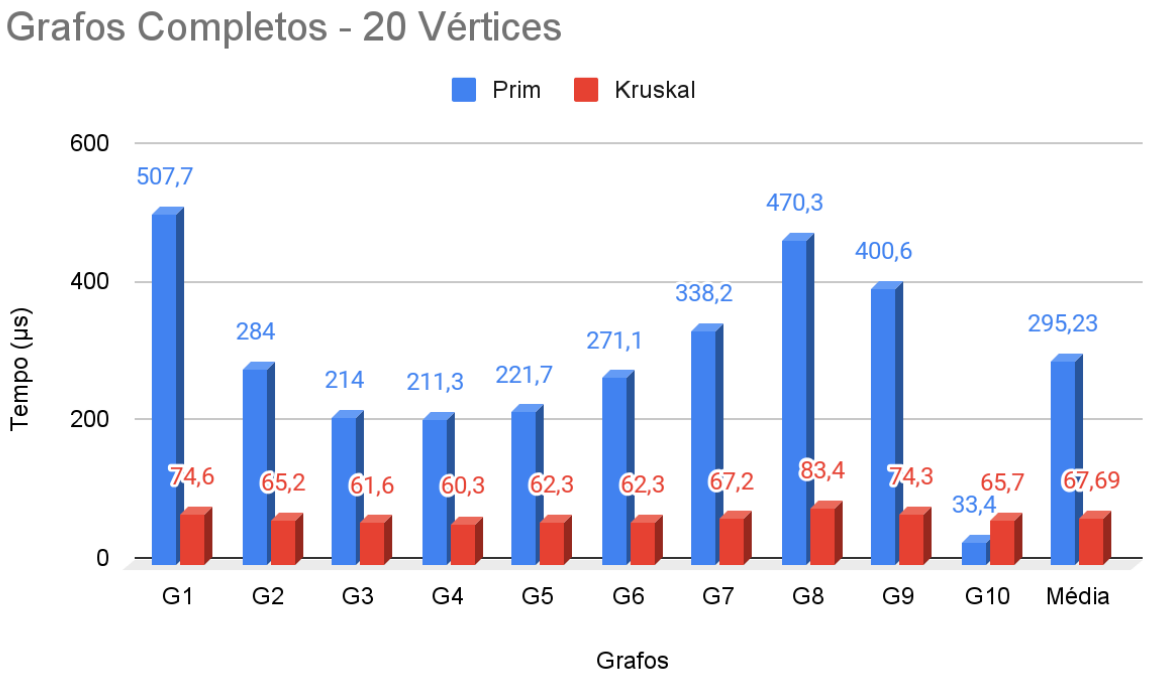
Grafos 15-Completo

	G1	G2	G3	G4	G5	G6	G7	G8	G9	G10	**Média**
Prim	91,8	82,5	98,5	107,2	120,3	88	91,3	72	89,8	97	93,84
Kruskal	32,5	36,1	35,2	37,5	42,9	36,7	38,3	30,3	28,3	30,2	34,8



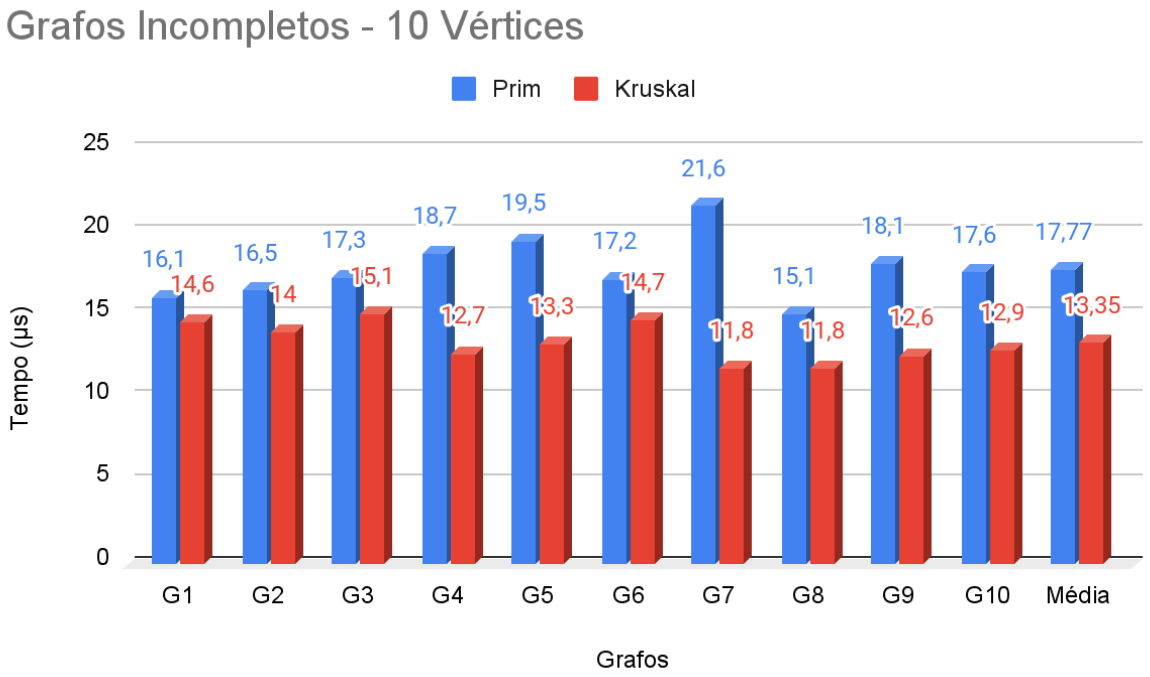
Grafos 20-Completo

	G1	G2	G3	G4	G5	G6	G7	G8	G9	G10	**Média**
Prim	507,7	284	214	211,3	221,7	271,1	338,2	470,3	400,6	33,4	295,23
Kruskal	74,6	65,2	61,6	60,3	62,3	62,3	67,2	83,4	74,3	65,7	67,69



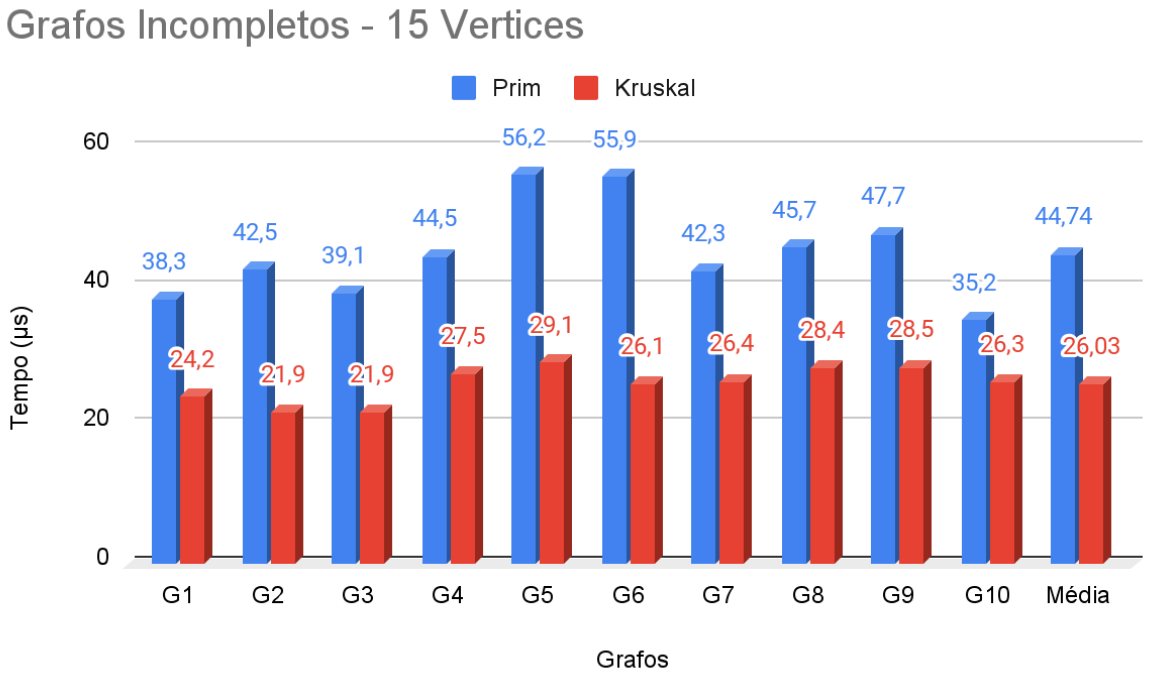
Grafos 10-Incompleto

	G1	G2	G3	G4	G5	G6	G7	G8	G9	G10	**Média**
Prim	16,1	16,5	17,3	18,7	19,5	17,2	21,6	15,1	18,1	17,6	17,77
Kruskal	14,6	14	15,1	12,7	13,3	14,7	11,8	11,8	12,6	12,9	13,35



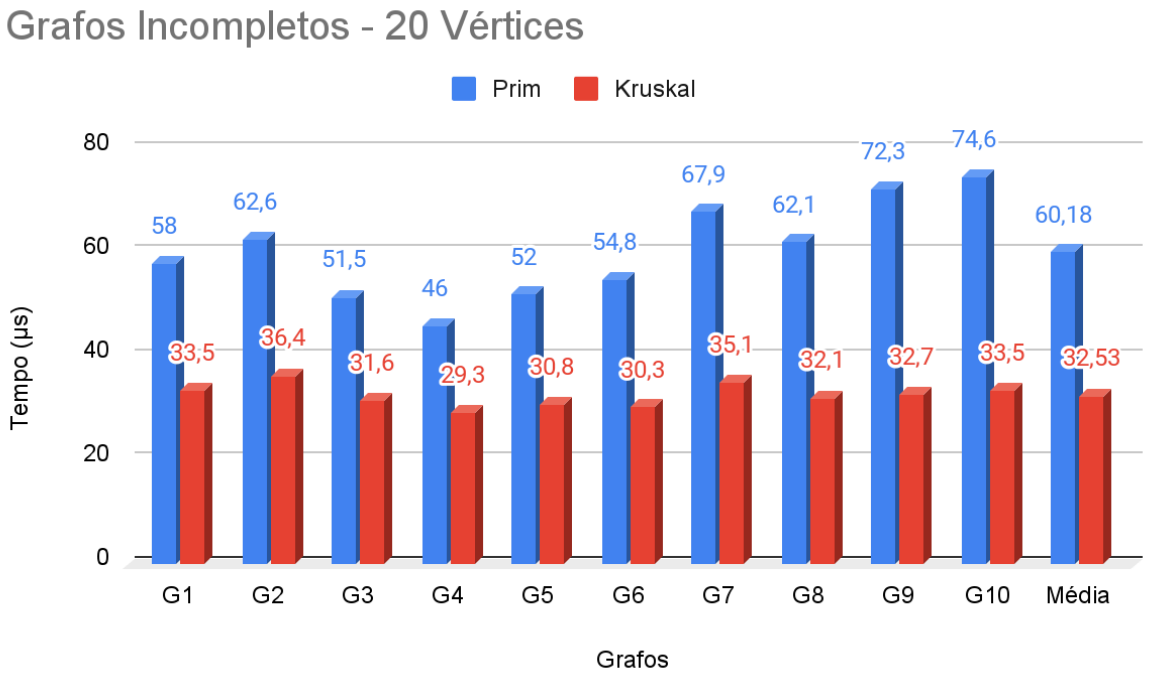
Grafos 15-Incompleto

	G1	G2	G3	G4	G5	G6	G7	G8	G9	G10	**Média**
Prim	38,3	42,5	39,1	44,5	56,2	55,9	42,3	45,7	47,7	35,2	44,74
Kruskal	24,2	21,9	21,9	27,5	29,1	26,1	26,4	28,4	28,5	26,3	26,03



Grafos 20-Incompleto

	G1	G2	G3	G4	G5	G6	G7	G8	G9	G10	**Média**
Prim	58	62,6	51,5	46	52	54,8	67,9	62,1	72,3	74,6	60,18
Kruskal	33,5	36,4	31,6	29,3	30,8	30,3	35,1	32,1	32,7	33,5	32,53



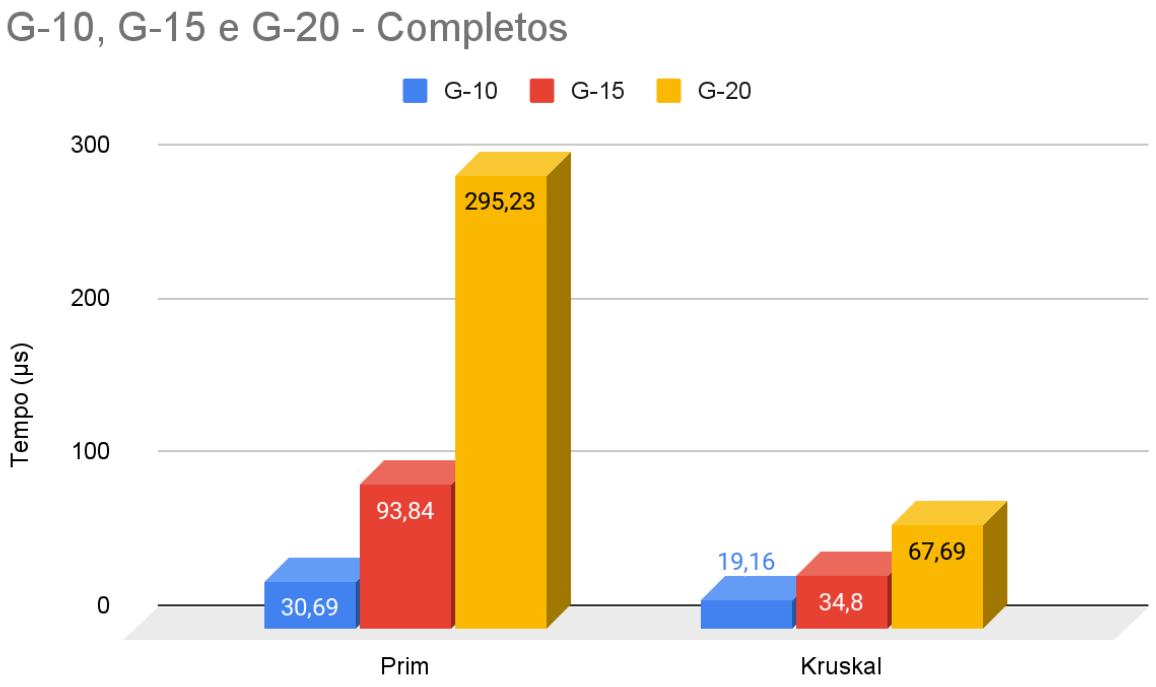
Grafos com 100 vértices

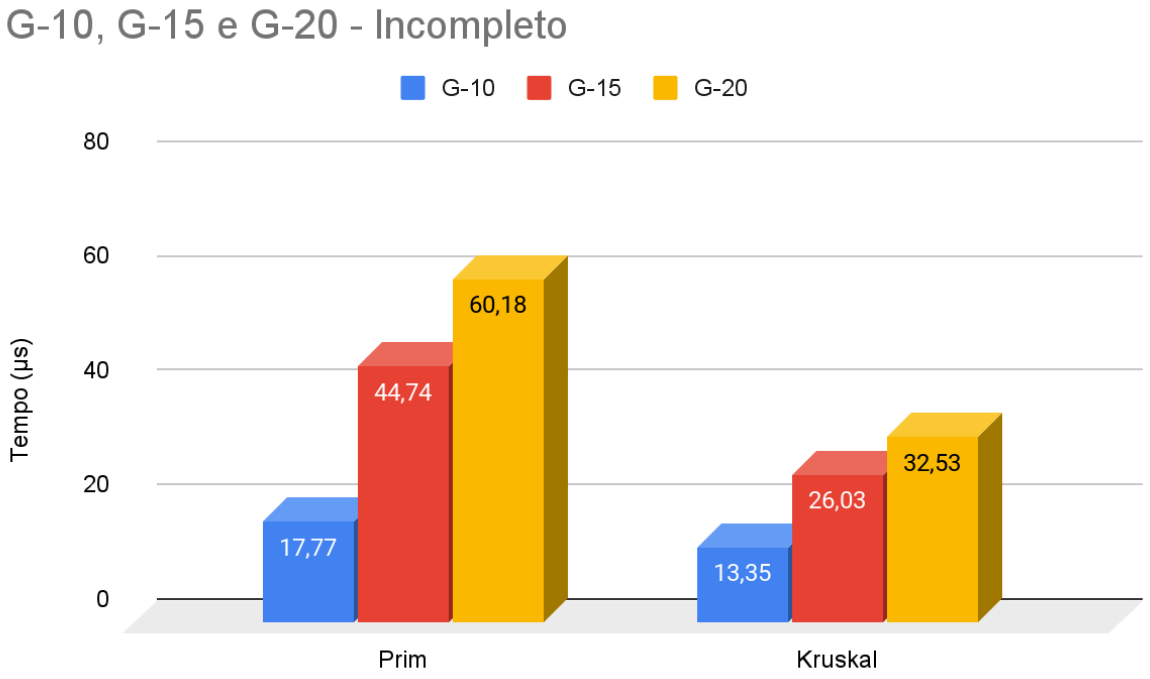
Também foram feitos testes com grafos maiores, de 100 vértices. Para o grafo incompleto, o tempo de execução do algoritmo de Prim teve tempo médio de execução de 251,77 µs, já o algoritmo de Kruskal teve tempo de execução médio de 111,40 µs. Já o grafo 100-completo levou em média 41.711,1 µs com o algoritmo de Prim e 940,7 µs com o método de Kruskal.

DISCUSSÃO

A partir da análise dos resultados, foi possível observar de maneira clara que a quantidade de arestas em cada grafo influencia diretamente no tempo de performance. Os grafos incompletos tiveram execuções mais rápidas para os dois algoritmos em comparação aos grafos completos, mostrando que uma quantidade maior de arestas aumenta consideravelmente a performance do programa, podendo ser observado principalmente nos grafos maiores, com 20 e 100 vértices.

Além disso, é notório que o algoritmo de Kruskal obteve melhor performance do que Prim em todas as execuções, para todos os grafos, completos e incompletos, enquanto essa diferença de desempenho ampliou ainda mais com o aumento da quantidade de vértices, indicando que para descobrir a árvore geradora mínima de um grafo, o algoritmo de Kruskal se mostra mais performático em relação ao método de Prim.




G-10, G-15 e G-20 - Incompleto
G-10
G-15
G-20
Tempo (µs)
80
60
40
20
0
17,77
44,74
60,18
13,35
26,03
32,53
Prim
Kruskal