UNIVERSIDADE ESTADUAL DE MARINGÁ - UEM
CENTRO DE TECNOLOGIA - CTC
DEPARTAMENTO DE INFORMÁTICA - DIN

MATEUS SCARPIN RIBEIRO
SERGIO DE ALMEIDA CEZAR

RELATÓRIO: AVALIAÇÃO COMPUTACIONAL DE PRIM VS. KRUSKAL

Algoritmos em Grafos

1. Introdução e Objetivo

Este trabalho tem como objetivo avaliar e comparar o desempenho computacional (tempo de execução e consumo de memória) dos algoritmos de Prim e Kruskal para a obtenção de Árvores Geradoras Mínimas (MST). A análise foca no comportamento dos algoritmos frente ao crescimento do número de vértices e arestas, utilizando instâncias de grafos baseadas em coordenadas euclidianas.

2. Metodologia

A implementação foi realizada em Python 3, utilizando apenas bibliotecas padrão para estruturas de dados (como heapq para fila de prioridade). Foram implementadas quatro variações de algoritmos para análise comparativa:

1. Prim (Heap): Utiliza um *Binary Heap* para extração do vértice de menor custo, com complexidade esperada de O(E log V).
2. Prim (Quadrático): Implementação ingênua com busca linear pelo menor vértice, complexidade O(V^2).
3. Kruskal (Otimizado): Utiliza a estrutura *Union-Find* com as otimizações de *Path Compression* (compressão de caminho) e *Union by Rank* (união por altura).
4. Kruskal (Simples): Utiliza *Union-Find* sem as otimizações citadas, para fins de controle.

Os testes foram executados em 6 instâncias de grafos de tamanhos variados. O tempo limite (*timeout*) estabelecido para execução foi de 300 segundos. Este valor encontra-se configurado no módulo `metricas.py` e pode ser ajustado.

Grafo	Vértices (V)	Arestas (E)	Custo MST
Grafo 1	8.980	12.629	3015372,43
Grafo 2	5.152	7.086	2641212,29
Grafo 3	37.128	54.867	3061889,58
Grafo 4	12.368	16.965	4794641,15
Grafo 5	212.341	307.292	16391318,64
Grafo 6	154.938	214.893	12342510,35

3. Resultados Obtidos

Nesta seção, são apresentados os dados quantitativos coletados durante a execução dos algoritmos sobre as seis instâncias de teste. Os resultados foram obtidos a partir da média de execuções consecutivas para garantir a estabilidade das medições.

A análise divide-se em duas métricas principais: o tempo de processamento (eficiência temporal) e o consumo de memória RAM (eficiência espacial). É importante notar que,

conforme definido na metodologia, as execuções que excederam o limite de 300 segundos foram interrompidas e constam como *"Timeout"* nos resultados.

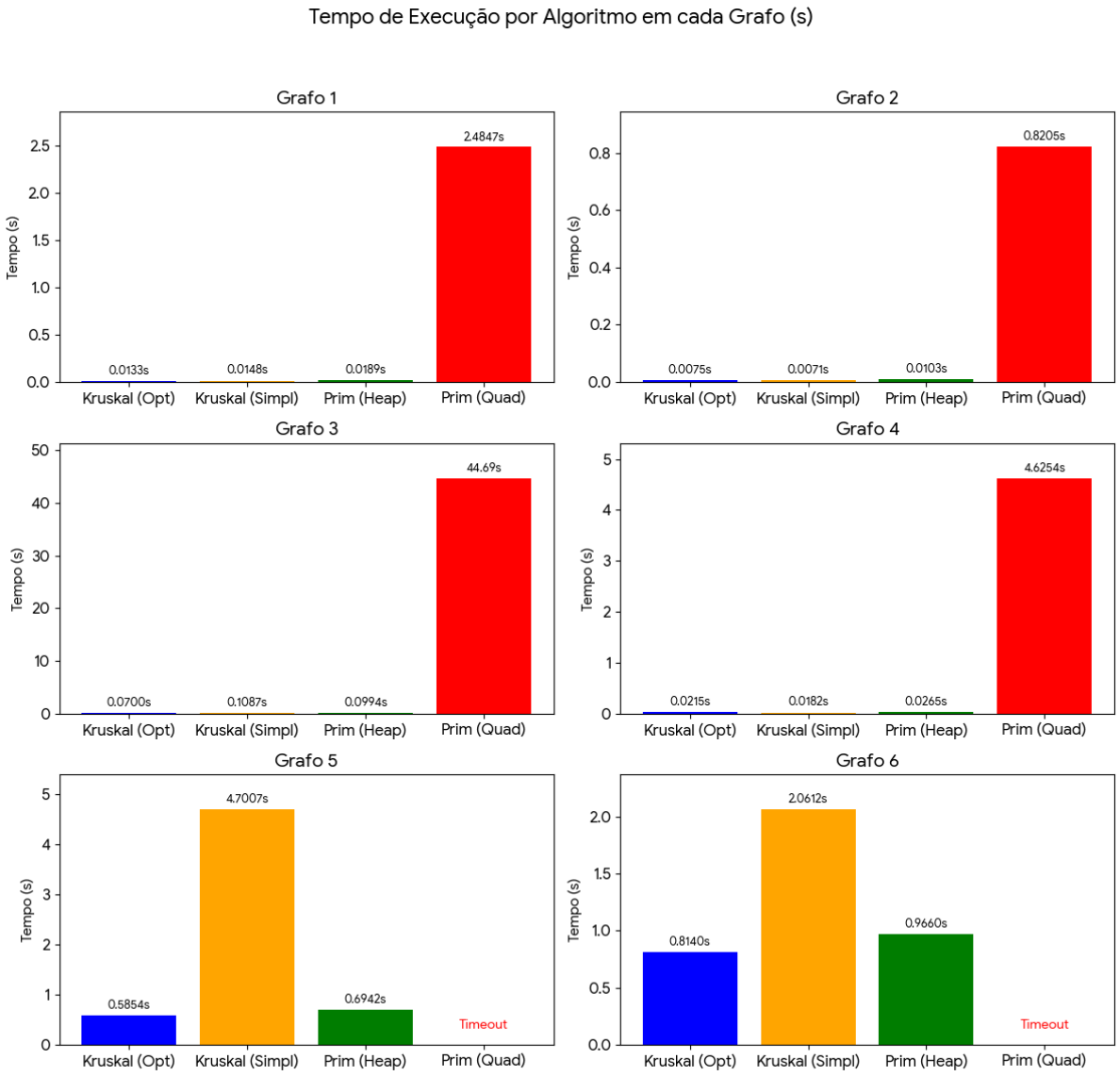
3.1 Desempenho de Tempo

A Tabela 1 apresenta os tempos médios de execução em segundos. Os dados evidenciam a escalabilidade das diferentes abordagens conforme o número de vértices (V) e arestas (E) aumenta.

Observa-se que, enquanto os algoritmos com complexidade próxima a linear-logarítmica (O(E log V)) mantiveram tempos abaixo de 1 segundo mesmo para as maiores instâncias, a abordagem quadrática tornou-se inviável para os Grafos 5 e 6.

Instância	Kruskal (Opt) Tempo (s)	Kruskal (Simples) Tempo (s)	Prim (Heap) Tempo (s)	Prim (Quad) Tempo (s)
Grafo 1	0,0133	0,0148	0,0189	2,4847
Grafo 2	0,0075	0,0071	0,0103	0,8205
Grafo 3	0,0700	0,1087	0,0994	44,6860
Grafo 4	0,0215	0,0182	0,0265	4,6254
Grafo 5	0,5854	4,7007	0,6942	*Timeout*
Grafo 6	0,8140	2,0612	0,9660	*Timeout*

Tabela 1: Tempo de Execução (segundos)



3.2 Consumo de Memória

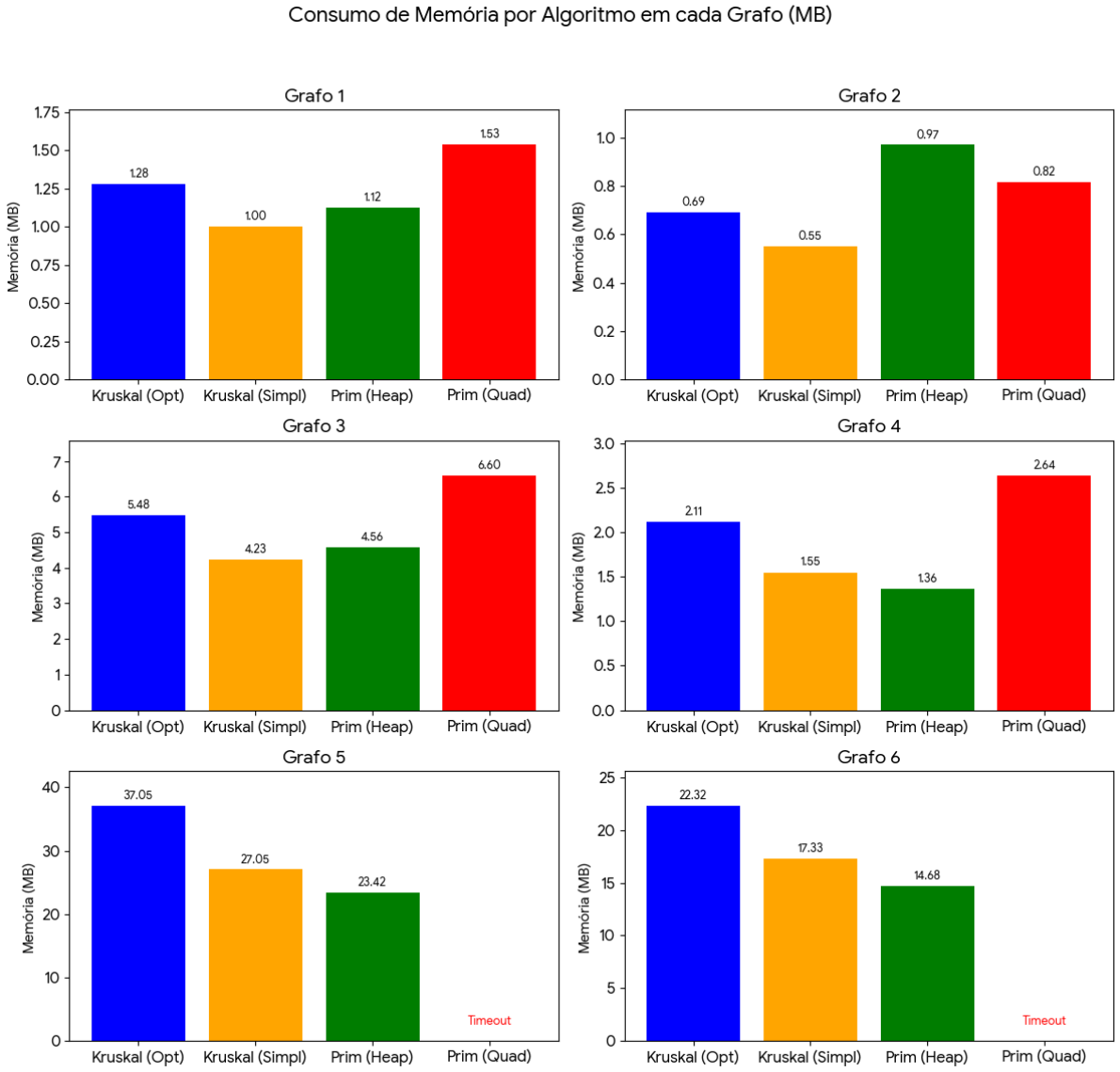
A Tabela 2 detalha o pico de memória (em Megabytes) monitorado pelo módulo tracemalloc durante a construção da MST.

Estes dados revelam o custo espacial das estruturas auxiliares utilizadas, como a pilha de recursão do Union-Find no algoritmo de Kruskal e a estrutura de Heap no algoritmo de Prim.

Instância	**Kruskal (Opt) [MB]**	**Kruskal (Simples) [MB]**	**Prim (Heap) [MB]**	**Prim (Quad) [MB]**

Grafo 1	1,279	0,998	1,121	1,535
Grafo 2	0,690	0,549	0,971	0,816
Grafo 3	5,483	4,233	4,564	6,598
Grafo 4	2,113	1,550	1,359	2,640
Grafo 5	37,047	27,047	23,423	*Timeout*
Grafo 6	22,320	17,326	14,684	*Timeout*

Tabela 2: Custo de Memória (MB)



4. Análise e Discussão

Esta seção discute comparativamente o desempenho dos algoritmos implementados: Prim (Heap), Prim (Quadrático), Kruskal (Otimizado) e Kruskal (Simples), relacionando resultados experimentais com suas complexidades teóricas, bem como o impacto direto das escolhas de estrutura de dados refletidas no código.

4.1. Prim Heap (O(E log V)) vs. Prim Quadrático (O(V^2))

A diferença entre essas duas versões decorre diretamente das estruturas de dados utilizadas. No código:

- Prim Heap usa heapq, uma priority queue (heap binário), o que garante extração do mínimo em O(log V).
- Prim Quadrático realiza a busca linear do vértice mínimo em cada iteração, resultando em O(V) por extração.

Versão	Estrutura	Complexidade
Prim Heap	Heap Binário	O (E log V)
Prim Quadrático	Busca linear	O (V^2)

4.1.1 Análise de Tempo

A diferença de desempenho entre as duas versões do Prim é evidente e cresce exponencialmente conforme o número de vértices aumenta.

- No Grafo 2 (aprox. 5 mil vértices), a versão quadrática foi cerca de 80 vezes mais lenta que a versão Heap (0.82s vs 0.01s).
- No Grafo 3 (aprox. 37 mil vértices), essa diferença saltou para 440 vezes mais lenta (44.68s vs 0.099s).
- Nas instâncias maiores (Grafo 5 e 6), a versão quadrática atingiu timeout (300s) e não conseguiu concluir. Já a versão Heap concluiu em 0.69 o Grafo 5 e em 0.96 o Grafo 6.

Isso confirma a teoria: para grafos esparsos (como estes, onde E ≈ 1.5V), a complexidade O(V^2) é proibitiva. A implementação com Heap, mantendo a complexidade próxima de linear-logarítmica, provou-se altamente escalável.

4.1.2 Análise de Memória

Em termos de consumo de memória, ambas as abordagens mantiveram-se estáveis e próximas nos grafos pequenos. No entanto, nota-se que o Prim Heap é eficiente espacialmente. Mesmo no maior grafo (Grafo 5, 212k vértices), ele consumiu apenas 23,4 MB. O *overhead* de manter a estrutura de heap é compensado pela eficiência em não precisar de matrizes de adjacência (usando listas), mantendo o consumo linear em relação ao tamanho do grafo (O(V+E)).

4.2. Prim (Heap) vs. Kruskal (Otimizado)

Para grafos esparsos, caso das instâncias deste trabalho, tanto Prim Heap quanto Kruskal possuem complexidades próximas:

Versão	Estrutura	Complexidade
Prim Heap	Heap Binário	O (E log V)
Kruskal (Otimizado)	Sort + Union Find com path compression e rank	O (E log V)

4.2.1 Análise de Tempo

Mesmo com complexidades semelhantes, Kruskal Otimizado foi consistentemente mais rápido que Prim Heap em todas as instâncias.

- Exemplo (Grafo 5): Kruskal (0.58s) foi cerca de 15% mais rápido que o Prim Heap (0.69s).

Essa vantagem do Kruskal pode ser atribuída à:

1) Baixa densidade dos grafos (E próximo de V)
   → Ordenação é barata.
2) Timsort detecta padrões parcialmente ordenados
   → comum em listas de distâncias euclidianas.
3) Union-Find otimizado reduz quase a O(1) os testes de ciclo.

Assim, mesmo com teorias equivalentes, o custo constante de Kruskal é menor, e a implementação em Python favorece operações sobre listas (sorting) em relação a push/pop intensivas no heap.

4.2.2 Análise de Memória

O Kruskal Otimizado apresentou consumo de memória maior que o Prim Heap, especialmente nos grafos de maior porte. Isso ocorre porque o Kruskal precisa manter a lista completa de arestas em memória e realizar a ordenação dessa estrutura, que gera buffers temporários adicionais. Além disso, a estrutura auxiliar usada para detectar ciclos, embora eficiente em tempo, também contribui para o aumento do uso de memória.

Já o Prim Heap opera de forma incremental, trabalhando apenas com a lista de adjacência e com um heap cujo tamanho cresce conforme a expansão da árvore. Como não precisa armazenar todas as arestas simultaneamente nem ordenar estruturas grandes, seu uso de memória permanece significativamente mais baixo, especialmente em grafos grandes.

Esse comportamento explica por que, nos grafos 5 e 6, o Prim Heap foi o algoritmo mais econômico em termos de RAM, mesmo não sendo o mais rápido.

4.3. Kruskal (Otimizado) vs. Kruskal (Simples)

A comparação entre o Kruskal Otimizado e o Simples (sem *path compression* e *rank*) destaca a importância das estruturas de dados eficientes.

Versão	Find/Union	Complexidade
Kruskal Simples	O(V)	O (E log V)
Kruskal (Otimizado)	O(α(V)) ≈ O(1)	O (E log V + $E \bullet V$)

4.3.1 Análise de Tempo

Efeitos nos resultados:

- Para grafos pequenos, as diferenças são pequenas.
- Para grafos grandes, o custo explode, veja:

1) Exemplo (Grafo 5 – 212k vértices):
    - Kruskal Otimizado: 0,58s
    - Kruskal Simples: 4,70s
    - → 8× mais lento

2) Exemplo (Grafo 6 – 154k vértices):
    - Otimizado: 0,81s
    - Simples: 2,06s
    - → > 2,5× mais lento

Por que isso acontece?
No Union-Find sem otimização, árvores internas podem se tornar caminhos longos (degeneradas). Cada chamada de find() precisa percorrer toda a cadeia, chegando perto de O(V).
Com otimização, porém:

- path compression "achata" as árvores
- rank previne árvores profundas
- profundidade efetiva fica próxima de 1
- find/union se tornam quase O(1)

Portanto, mesmo usando o mesmo algoritmo base, as versões têm desempenho drasticamente diferente, e o custo constante do Kruskal Simples torna-o proibitivo para grafos grandes.

4.3.2 Análise de Memória

A comparação entre o Kruskal Otimizado e o Kruskal Simples mostra que a versão otimizada, embora mais rápida, tende a usar mais memória. Isso acontece porque o método de detecção de ciclos utilizado nessa versão depende de estruturas auxiliares mais completas, o que aumenta o custo espacial.

O Kruskal Simples, por outro lado, utiliza uma estrutura auxiliar mais leve, o que o torna mais econômico em memória, especialmente nos grafos maiores. Como ambos os algoritmos manipulam a mesma lista de arestas e executam a mesma ordenação, toda a diferença de consumo de RAM decorre exclusivamente dessas estruturas auxiliares.

Por esse motivo, em praticamente todas as instâncias, e principalmente no Grafo 6, o Kruskal Simples apresentou menor uso de memória, ainda que com desempenho temporal inferior.

5. Conclusão

Os experimentos demonstraram que, para as instâncias de grafos euclidianos fornecidas (caracteristicamente esparsas), tanto o Kruskal (Otimizado) quanto o Prim (Heap) são soluções eficientes e viáveis.

O Prim Quadrático deve ser evitado em qualquer cenário que não envolva grafos densos e pequenos. O Kruskal mostrou-se o campeão de desempenho nestes testes específicos, beneficiando-se da baixa densidade de arestas e da eficiência da ordenação nativa da linguagem utilizada. A consistência dos resultados foi validada, garantindo que todas as implementações geraram MSTs com o mesmo peso total.