Algoritmos de Kruskal e Prim para busca de árvores geradoras de tamanho mínimo: relatório final da disciplina GA-026

Thiago da Mota Souza
thiagoms@posgrad.lncc.br

Resumo

Grafos fornecem modelos úteis para abordagem de problemas em diversas áreas do conhecimento e da industria: de cadeias de logística à pesquisa de neurociência, de forma que a solução de problemas abstratos em grafos podem transladar em soluções para problemas importantes. Em particular, o problema de se encontrar AGTM (Árvores Geradoras de Tamanho Mínimo) em grafos era de fundamental importância para a indústria de telecomunicações que precisava fornecer telefonia fixa para consumidores dispersos geograficamente, pois a sua solução era fundamental para minimizar os custos com o cabeamento. Neste contexto, Joseph Kruskal e Robert Prim, dois engenheiros trabalhando para a Bell Labs nos anos 50 publicaram artigos fundamentais para a ciência da computação [1][2] que são estudados em cursos de algoritmos até os dias de hoje. Neste artigo, apresentado como parte da avaliação da disciplina GA-026 do Laboratório Nacional de Computação Científica, serão conduzidos experimentos computacionais a fim de verificar as previsões teóricas quanto a ordem de crescimento do tempo de processamento dos algoritmos propostos por estes autores.

1 Introdução

Grafos são estruturas definidas por dois conjunto, nós (V) e arestas (E), em que as arestas representam relações entre os nós. Atribuindo-se atributos a arestas e nós, essas estruturas tornam-se poderosos modelos para representar: redes sociais, redes neurais, cadeias de logística, workflows de processamento científico, circuitos elétricos, cadeias metabólicas em células, dentre outros. Em particular, a industria de telefonia utilizou a modelagem por grafos para minimizar o custo total do cabeamento necessário para fornecer redes que ligassem todos seus clientes. Representando consumidores particulares por nós e as distâncias entre eles por arestas com pesos, cujos valores representam o custo de ligar esses clientes a rede. O custo de instalação de uma determinada rede poderia ser computada como a soma dos pesos das arestas dessa rede. O número de clientes atendidos; o número de nós dessa rede. Assim os subgrafos que:

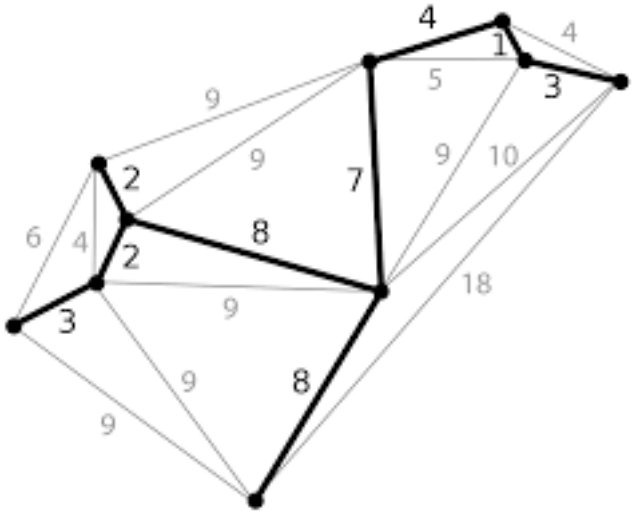

Figura 1: Exemplo de uma AGTM, em negrito, para um grafo e uma função de pesos, representados na arestas do grafo.

1. contivesse todos os nós do grafo original
2. contivesse um conjunto de arestas suficientes para ligar com pelo menos um caminho todos os nós dessa rede. Figrua retirada de [4]

Representam soluções para o problema enfrentado de como se construir uma rede capaz de levar a telefonia fixa a todos os clientes dispersos geograficamente. Dessas soluções, aquelas cuja soma dos pesos nas arestas é mínima eram as mais desejáveis por serem as mais baratas.

Os subgrafos que atendem as duas propriedades listadas acima são chamados de Árvores Geradoras (AG) das quais as que tem a soma das arestas mínimas são chamadas Árvores Geradoras de Tamanho Mínimo(AGTM), ou, simplesmente Árvores Geradoras Mínimas (AGM). Matematicamente definimos, dado um grafo $G = (V, E)$: e uma função que mapeia arestas em pesos $w(a)$, o grafo M de vértices V e arestas A

$$M = (V, A) : A = \arg\min_{A \in E} \sum_{a \in A} w(a) \qquad (1)$$

Esse problema pode ser resolvido por força bruta

uma vez que o conjunto de arestas do grafo G é finito, mas essa solução não é aceitável do ponto de vista computacional já que o conjunto das partes do conjunto E tem $2^{|E|}$ elementos. $|E|$ representando o número de elementos do conjunto E.

Nas seções seguintes deste relatório: apresentam-se dois algoritmos eficientes para encontrar as AGTMs em 2; seguindo-se traz-se notas dignas da implementação dos algoritmos estudados que foi feita para fins desse trabalho em 2.3 e 2.4; em 3 apresentam-se os experimentos conduzidos para medir empiricamente a complexidade desses mesmos algoritmos; a seção 4 traz os resultados obtidos nos experimentos que foram executados e por fim a seção 5 discute esses resultados e concluí o relatório. Os apêndices trazem detalhes das implementações que podem ser relevantes ao leitor do relatório.

2 Algoritmos eficientes para encontrar as AGTMs

Após suas publicações na década de 50, os algoritmos propostos por Joseph Kruskal [2] e Robert Prim [1] tornaram-se canônicos e presentes em livros base de cursos [3] de algoritmos e grafos. Detalhes extensos da implantação desses algoritmos podem ser encontrados nas referências fornecidas até aqui. Alguns comentários sobre esses algoritmos serão dados nas seções seguintes a fim de embasar as discussões deste artigo.

2.1 Meta Algoritmo

No capítulo que dedicou aos algoritmos de Kruskal e Prim, capítulo 21 [3], Cormen nos apresenta com um "meta-algoritmo"para encontrar as AGTMs. Uma abstração dos algoritmos de Kruskal e Prim que nos ajuda a compreender o cerne de ambos.

```
defina acha_AGTM(G):
    agtm = grafo_vazio()
    enquanto é_arvore_geradora(agtm) == falso:
        aresta = próxima_aresta_segura()
        agtm.adicione_aresta(aresta)

    retorne agtm
```

Através dessa abstração podemos compreender que:

- Os algoritmos fazem "crescer"uma AGTM uma aresta por vez, até que ela de fato seja uma AG para o grafo G.
- o Cerne dos algoritmos giram em torno do conceito de arestas seguras: que são aquelas que, dada uma subárvores de uma AGTM para o grafo, se adicionadas geram um novo grafo que é também um subconjunto da mesma AGTM.

As propriedades listadas em 2.1 são comuns a ambos algoritmos que serão discutidos em 2.3 e 2.4 e podem ser utilizadas para estabelecer invariantes que demonstram a corretude de ambos e que permitem classificá-los como algoritmos gulosos.

A condição de parada é quase trivial, uma vez que ao adicionar apenas as arestas seguras garantes-se que o grafo gerado é uma árvore, ou uma floresta, restando apenas testar que o grafo gerado contenha todos os vértices do grafo G.

O *loop* de adição de arestas seguras será executado no máximo $|V| - 1$ vezes no cado do grafo G ser conexo. Os algoritmos vão diferir no caso em que G não é conexo como será discutido nas seções 2.3 e 2.4.

2.2 Aresta segura

Uma aresta segura pode ser definida utilizando-se justamente a propriedade 2 em 2.1. O procedimento que as encontra faz uso do teorema, retirado de 1 [3].

Theorem 1 *Dado um Corte $S = (V', E')$ do grafo $G = (V, E)$ e a aresta $(u, v) \in E$ que liga os nós u e v, que cruza o corte, isto é, $u \in S$ e $v \in V - S$, tal que*

$$w(u, v) \leq w(u', v') \ \forall \ (u', v') \ \textit{cruzando o corte}$$

então (u, v) é uma aresta segura.

Esse teorema possuí corolário que vale para florestas geradoras que pode ser encontrado nas referências, bem como uma prova, ambos serão omitidos aqui por questões de concisão.

Assim o procedimento *próxima_aresta_segura* do meta-algoritmo, basicamente procurar pela aresta de menor peso cruzando o corte estabelecido pela árvore, ou floresta, *agtm*.

2.3 Algoritmo de Kruskal

Como citado em 1, na década de 50, o engenheiro Joseph Kruskal publica, [2], seu algoritmo para achar uma $AGTM$. Neste trabalho implementou-se esse algoritmo utilizando-se linguagem Python [5], conforme A

1. O algoritmo de Kruskal começa com uma floresta que contém todos os vértices do grafo G e nenhuma aresta.
2. a seguir as arestas de G são ordenadas na ordem crescente dos pesos w.
3. As arestas são percorridas e: caso a ligue dois componentes disjuntos da árvore, ela é adicionada a floresta.

Caso a o grafo G não seja conexo, o algoritmo retornará uma floresta com componentes que são árvores

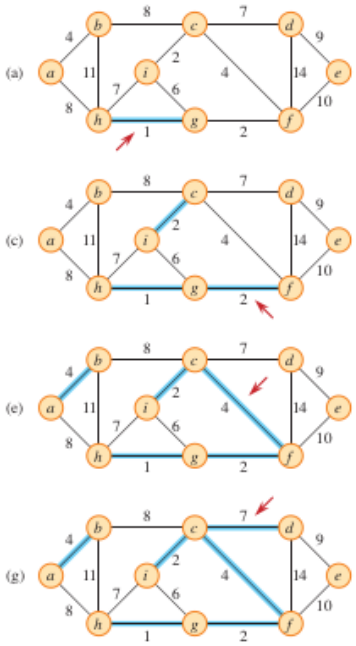

Figura 2: Exemplo de uma execução do algoritmo de Kruskal. A cada passo uma aresta segura é adicionada a floresta. [3]

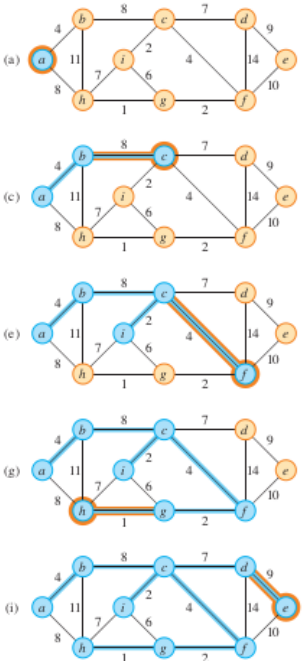

Figura 3: Exemplo de uma execução do algoritmo de Prim. A cada passo uma aresta segura é adicionada a árvore. [3]

de tamanho mínimo para todos os componentes conexos do grafo.

O caso mais trivial: para o algoritmo é o caso em que G é uma floresta sem arestas, o que implica que apenas a cópia dos vértices do grafo original deve ser feita, logo o tempo do algoritmo é de complexidade $O(|V|)$.

O pior caso: é representado por um grafo G completo. O que obrigará a o algoritmo a executar o maior número possível de verificações no *loop* interno.

No caso médio: [3] fornece uma análise da complexidade do algoritmo para grafos conexos que é $o(|E|log|V|)$ supondo que a implementação do algoritmo de ordenação usado é $O(|E|log|E|$

Da implantação feita para esse trabalho destaca-se que, foi utilizado um *dictionary* que permite descobrir qual o índice do componente de um nó em tempo $O(1)$ utilizando uma *função hash*.

2.4 Algoritmo de Prim

O algoritmo de Prim segue a mesma estrutura que o de Kruskal definida em 2.1. No entanto:

1. Prim inicializa a partir de um nós raiz uma árvore contendo apenas esse nó e nenhuma aresta

2. um corte Q é definido contendo todos os nós que não são o nó raiz.

3. Enquanto houver nós em Q:

   (a) se a resta (u, v) é segura dado o corte Q, $u \in Q$, acrescente essa aresta a árvore.

   (b) retire o nó u do corte Q

Assim, caso o grafo G não seja conexo, o algoritmo retornará a AGTM do componente que contém o nó raiz.

O pior e o melhor caso são os mesmos descritos em 2.3 por razões similares. Um grafo G completo aumenta o número de candidatas a aresta segura a cada passagem do *loop* interno.

A complexidade algorítmica do **caso médio** segundo [3] é a mesma do algoritmo de Kruskal $|E|log|V|$ para grafos conexos.

Da implementação feita em Python B destacamos que a inexistência de um método que nos permita remover um objeto de uma heap a partir da busca por um valor de uma de suas propriedades, obriga a recriação da heap que guarda os nós no corte Q a cada *loop*, o que não é o ideal, mas que trouxe pouco impacto como se v erá na Seção 4.

3 Mediação empírica da complexidade dos algoritmos

A fim de verificar a ordem de complexidade do algoritmo, foram gerados grafos artificialmente através da escolha, com distribuição uniforme, de todas as possíveis arestas de um grafo de $|V|$ nós. A fim de ser gerar grafos com: 0%, 5%, 10%, até 100% das arestas

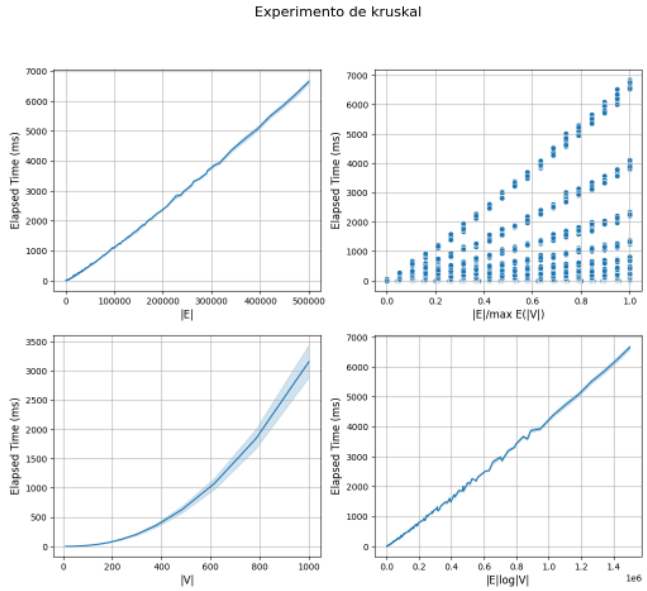

Figura 4: Crescimento do tempo de execução do algoritmo de Kruskal.

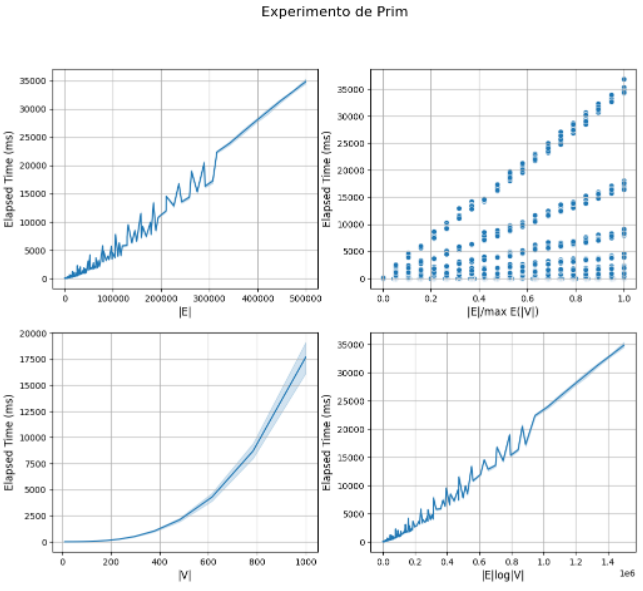

Figura 5: Crescimento do tempo de execução do algoritmo de Prim.

em escala linear e grafos com nós variando de 10 a 1000 nós em escala logarítmica (20 pontos foram intercalados). Para cada combinação de número de nós e número de arestas, das 400 consideradas, foram executados 10 vezes cada um dos algoritmos. Tomou-se o cuidado de variar o nó raiz na execução do algoritmo de Prim afim de evitar qualquer viés que isso por ventura pudesse introduzir nas medições. Os dados coletados totalizam assim 4000 pontos para cada um dos algoritmos.

O método utilizado para gerar os grafos muito provavelmente gera muitos grafos desconexos, o que poderia fazer com que o tempo esperado para o caso médio não se verifica-se empiricamente. Isso não aconteceu com será discutido nas Seções 4 e 5.

Os experimentos foram executados em um *container Docker* rodando numa máquina servidora *Acer/Nitro 5* com 16 GB de RAM, SSD de 256GB rodando sistema Ubuntu 22.04.01. O *container* foi utilizado para encapsular o ambiente utilizado para os experimentos e sua imagem base *mcr.microsoft.com/devcontainers/python:1-3.11-bullseye* que fornece um executor Python 3.11.5 conforme disponibilizado em [5], foi baixada do Docker Hub. A dependências utilizadas no projeto foram listadas no arquivo de *requirements.txt* no repositório deste trabalho. Os dados foram analizados utilizando-se um Notebook Jupyter igualmente presente no repositório do projeto. A versão do *Docker* utilizada foi 24.0.7.

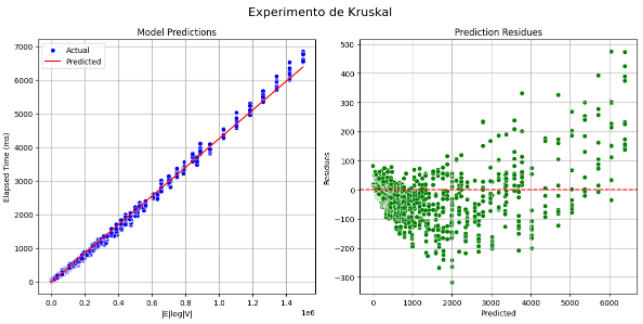

Figura 6: Regressão linear executada sobre os dados obtidos nos experimentos para o algoritmo de Kruskal.

4 Resultados

Os resultados dos experimentos são apresentados em 4 e 5 onde fica clara a relação linear do tempo de crescimento com o aumento do valor de $|E|log|V|$. A seguir foi executada uma regressão linear para cada um dos conjuntos de dados 6 e 7.

Algoritmo	*parâmetros*		
	c_0	c_1	R^2
Kruskal	-15.55492	0.00427	0.9974
Prim	-268.13894	0.02187	0.9785

A regressão linear aproxima bem a relação de tempo de execução e o crescimento de $|E|log|V|$ conforme pode-se observar pelos valores do coeficiente de determinação (R) que são muito próximos a 1.

Apesar do valor alto de R, os resíduos dos modelos não se distribuem de forma uniforme em torno de 0, isso se explica em parte porque o crescimento do

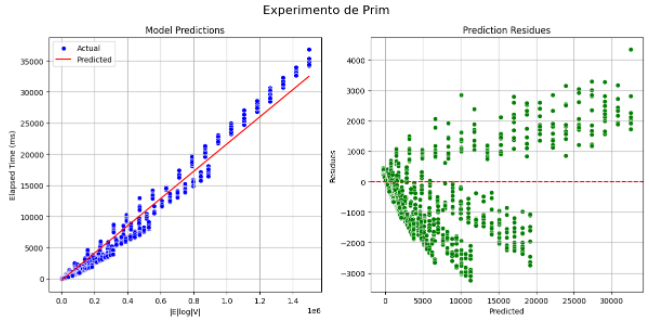

Figura 7: Regressão linear executada sobre os dados obtidos nos experimentos para o algoritmo de Prim.

temo de execução é assintoticamente aproximado pela função linear de $|E|log|V|$ no pior caso, assim espera-se que hajam constantes que façam com o que o tempo de execução seja sempre menor que o valor predito pela regressão linear. O fato dos resíduos ainda guardarem alguma correlação com os valores de $|E|log|V|$ mereceria mais investigação, que será postergada por questões de tempo.

5 Conclusão

Neste artigo foram apresentados experimentos feitos com grafos artificialmente gerados que nos permitiram a comprovação empírica do teoricamente esperado para a complexidade do tempo de crescimento dos algoritmos de Kruskal e Prim. Implementações desses algoritmos foram feitas em Python e estão disponíveis no Github. Alguns fenômenos observados quanto a distribuição dos resíduos da regressão linear merecem investigações futuras, o que possivelmente poderiam levar a novos ajustes na implementação.

De forma geral, esse trabalho contribuiu para a melhor compressão de algoritmos fundamentais a teoria dos grafos com aplicações vastas em diversas áreas do conhecimento. Os resultados aqui listados serão apresentados a turma de Algoritmos I do LNCC em Dezembro de 2023 e junto com esse relatório cão compor parte da avaliação do seus autor.

Referências

[1] Prim, R. C. "Shortest Connection Networks and Some Generalizations."Bell System Technical Journal 36.6 (1957): 1389-401. Web.

[2] Kruskal, Joseph B. "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem."Proceedings of the American Mathematical Society 7.1 (1956): 48-50. Web.

[3] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms, fourth edition

[4] Wikipedia, the free encyclopedia. Minimum spanning tree. Online: accessed December 2, 2023

[5] Thiago da Mota Souza Implementação acadêmica dos algoritmos de kruskal e Prim.

A Implementação do algoritmo de Kruskal em Python

Aqui foi destacada apenas a função *kruskal_mst*, que é um método da classe *TGraph*. Para o código completo, favor referir-se a [5]

```
    def kruskal_mst(self, roo_id: int = 0, weight_property = "weight") -> TGraph:

        mst = TGraph(num_vertices=self.get_num_vertices())
        mst.copy_nodes_properties(self)

        node_component_map = {v._id: i for i, v in enumerate(mst._V)}
        forest = [set([v._id]) for v in mst._V]

        edges = self.get_list_of_edges()

        sorted_edges = sorted(edges, key=lambda x: x[2][1][weight_property])

        for u_id, e_id, edge in sorted_edges:
            component_u = node_component_map[u_id]
            component_v = node_component_map[e_id]
            if component_u != component_v:
                mst.add_edge(u_id, e_id, edge[1])
                forest[component_u] = forest[component_u].union(forest[component_v])
                for v_id in forest[component_v]:
                    node_component_map[v_id] = component_u
                forest[component_v] = set()

        return mst
```

B Implementação do algoritmo de Prim em Python

Aqui foi destacada apenas a função *prims_mst*, que é um método da classe *TGraph*. Para o código completo, favor referir-se a [5]

```
    def prims_mst(self, root_id: int = 0, weight_property = "weight") -> TGraph:

        mst = TGraph(num_vertices=self.get_num_vertices())
        mst.copy_nodes_properties(self)
        for v in mst._V:
            mst.add_node_property(v._id, 'pi', None)
            mst.add_node_property(v._id, 'key', sys.float_info.max)

        mst.add_node_property(root_id, 'key', 0)
        mst.add_node_property(root_id, 'pi', None)
        Q = [n for n in mst._V]

        while len(Q) > 0:

            # get node with min edge weight crossing the cut
            u = min(Q, key=lambda x: x['key'])

            # add safe edge to tree, must guard against root on first loop
            if u['pi'] is not None:
                edge_propeties = self.get_edge(u['pi'], u._id)[1]
                mst.add_edge(u['pi'], u._id, edge_propeties)

            # update edges weights == keys crossing the cut via u
            for edge in self._E[u._id]:
                v_id = edge[0]._id
                v = mst.get_node(v_id)
                q_ids = [n._id for n in Q]

                if v._id in q_ids and edge.get_property(weight_property) < v['key']:
```

```
            v['pi'] = u._id
            v['key'] = edge.get_property(weight_property)

    # remove u from Q
    Q = [n for n in Q if n._id != u._id]

return mst
```