Complexidade do Algoritmo de Kruskal em Diferentes Estruturas de Grafo

Victor S. Melo

Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Curso de Ciência da Computação – Faculdade de Informática (FACIN)
Porto Alegre – RS – Brazil
victor.melo.001@acad.pucrs.br

Resumo

Este trabalho lida com a dependência da complexidade de algoritmos com a estrutura de dados nos quais eles são executados, mostrando que a escolha da estrutura pode afetar o desempenho do algoritmo. Neste trabalho, o Algoritmo de Kruskal é utilizado para demonstrar tal afirmação, considerando duas implementações de grafos: a primeira baseada em listas encadeadas e a segunda em tabelas hash. É explorado qual estrutura é a mais adequada para este algoritmo, mostrando se há uma estrutura mais adequada e quais considerações devem ser tomadas na escolha de uma estrutura de dados ao implementar um algoritmo.

1 Introdução

Ao analisar-se algoritmos, são buscados algoritmos que executam em menor tempo ou que utilizam menos memória durante sua execução. *Notação assintótica* é uma maneira útil de analisar a complexidade de um algoritmo, definindo o tempo de execução em função da quantidade de valores de entrada. Assim, independente se o algoritmo será executado em um computador com maior ou menor capacidade de processamento, a variação de seu tempo de execução em função do valor de entrada será o mesmo, demonstrando o comportamento do algoritmo à medida que aumenta o valor de entrada.

O Algoritmo de Kruskal é um algoritmo guloso[1] que encontra a árvore geradora mínima (em inglês, *minimal spanning tree*) de um grafo [1, 2].

Neste trabalho será feita uma análise de complexidade do tempo de clock da execução de um algoritmo em função da entrada. Através do uso do Algoritmo de Kruskal e duas estruturas diferentes de grafo, será demonstrado que a estrutura teoricamente melhor, na prática pode acabar não sendo a mais adequada, e que é necessário estar atento para os requisitos do algoritmo em relação à estrutura, não adiantando implementar uma estrutura otimizada para uma ação específica, sendo que o algoritmo não executa tal ação.

As seções abaixo são divididas da seguinte forma: Na seção 2, são apresentados alguns fundamentos necessários para a compreensão do resto deste trabalho; Na seção 3, é apresentado o algoritmo, contextualizando-o e mostrando seu pseudocódigo; Na seção 4 é apresentado o método de análise utilizado no experimento; Na seção 5, é mostrado como este algoritmo é implementado utilizando-se listas encadeadas, e sua complexidade de tempo em tal estrutura; Na seção 6 é mostrado como o algoritmo é implementado utilizando-se tabelas hash, e sua complexidade de tempo nesta estrutura; Na seção 7 é feita uma comparação da complexidade do algoritmo em ambas estruturas e é buscada uma razão que justifique os resultados obtidos. Por fim, a seção 8 apresenta a importância destes resultados e como a consideração destas informações podem auxiliar no desenvolvimento de sistemas, na computação, melhores.

[1] Algoritmo que encontra a melhor solução no momento. Por conta disto, muitas vezes a solução encontrada não é a solução ótima.

2 Fundamentos

Para facilitar o entendimento deste trabalho, é necessário definir alguns termos e conceitos que serão utilizados ao longo do texto. Isto é feito nesta seção.

2.1 Limite Superior Assintótico

Considerando que um algoritmo possui uma quantidade de entrada n, teremos uma função *f(n)* que retorna a quantidade de operações executadas pelo algoritmo.

A notação O considera esta função $f(n)$, mas despreza constantes, mantendo apenas variáveis que dependam da entrada. Por exemplo, para um algoritmo que executa $n^2 + 5$ operações, temos a função $f(n) = n^2 + 5$ e um limite superior $O(n^2)$. Também é possível definir um limite superior assintótico pelo tempo de clock da execução de um algoritmo em função da entrada. Este último será o método utilizado neste trabalho (o método é melhor explicado na seção 4). Formalmente, a notação O representa um conjunto de funções, porém, para o objetivo deste trabalho, a descrição informal acima é suficiente. Uma melhor descrição sobre notação assintótica e O pode ser encontrada em [3].

2.2 Tabela Hash

Uma tabela hash é uma estrutura de dados que associa chaves a seus valores. Possui o objetivo de, a partir de uma chave simples, realizar uma busca rápida e obter o valor desejado. A busca e inserção de dados com tabelas hash tem complexidade $O(1)$.

Como exemplo, caso sejam armazenados dados que representam uma pessoa, uma possível chave seria o nome.

2.3 Árvore Geradora Mínima

Dado um grafo não direcionado, uma árvore geradora mínima é um subgrafo o qual é uma árvore que conecta todos os vértices, formada buscando-se as arestas com menor valor. Um exemplo pode ser visto na Figura [**?**].

A hipótese desenvolvida é que a complexidade do algoritmo é dependente da estrutura em que este é executado, e que a busca pela estrutura adequada depende das operações realizadas pelo algoritmo.

3 Algoritmo de Kruskal

O Algoritmo de Kruskal retorna uma árvore geradora mínima de um grafo G. Como este é um algoritmo guloso, ele retorna uma árvore geradora mínima possível, não necessariamente a melhor. Como exemplo, considere o grafo representado na Figura 1

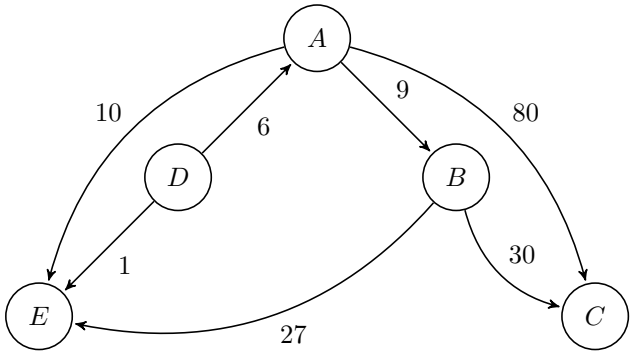

Figura 1: Grafo onde será aplicado o algoritmo de exemplo

Ao aplicarmos o algoritmo neste grafo, obtemos a árvore geradora mínima representada na Figura 2.

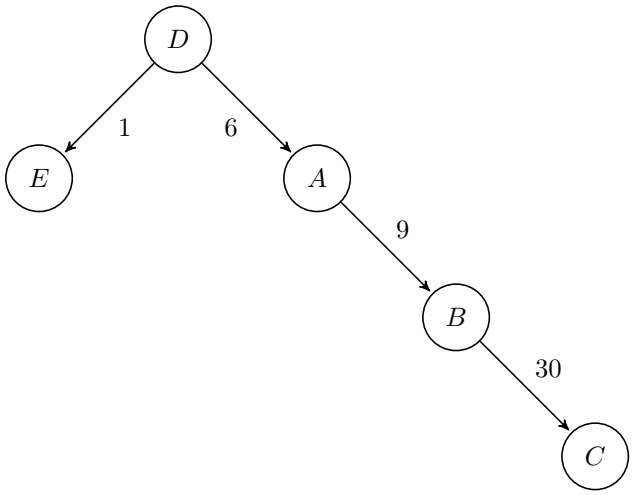

Figura 2: Árvore geradora mínima obtida a partir do grafo da Figura 1

O algoritmo, aplicado a um grafo G, pode ser visto em Algoritmo 1.

Algoritmo 1 Algoritmo de Kruskal em linguagem natural

1: Criar uma floresta F, onde cada vértice do grafo G é uma árvore separada.
2: Criar conjunto S com arestas do grafo G ordenadas do menor valor para o maior.
3: Para cada aresta (u,v) removida ordenadamente de S:
4: Se (u,v) conecta duas árvores diferentes, adicione-a à floresta F, juntando as árvores do nodo u à do nodo v.
5: Retorna a floresta F

Agora que já foi apresentada uma breve descrição do comportamento do algoritmo, é possível compreender melhor um pseudocódigo de sua implementação. Este pode ser visto no Algoritmo 2, que retorna uma árvore geradora mínima de um grafo G. No pseudocódigo, são utilizados os seguintes métodos:

- *MAKE-SET(v):* cria um conjunto com um único elemento v;
- *FIND-SET(v):* retorna o único conjunto que contém v;
- *UNION(u,v):* une os conjuntos que contêm u e v.

Algoritmo 2 Algoritmo de Kruskal em pseudocódigo

1: **para** cada vértice v de G
2: **faça** *MAKE-SET(v)*
3: cria conjunto S de arestas de G
4: ordena as arestas de S por valor não decrescente
5: **para** cada aresta (u, v) de S em ordem de peso não decrescente
6: **faça** se *FIND-SET(u)* $\neq$ *FIND-SET(v)* **então**
7: A $\leftarrow$ A $\cup$ $\{(u, v)\}$
8: *UNION(u,v)*
9: **retorna** A

4 Metodologia de Análise

O algoritmo foi executado em duas estruturas de grafos. A primeira foi implementada utilizando listas (*ArrayList* em Java) e a segunda utilizando tabelas hash (*HashTable* em Java), através de uma abordagem experimental,

utilizando do método hipotético-dedutivo. O universo dos grafos fora limitado para grafos não direcionados que contenham n nodos e $(n-1)$ arestas.

Duas possibilidades para a análise da complexidade foram consideradas: a análise da quantidade de operações em função da entrada, e a análise do tempo de clock em função da entrada. O segundo método foi escolhido. Visto que todos os experimentos foram realizados em uma mesma máquina e, como estamos interessados no comportamento do algoritmo, não há problema em fazer esta análise em função do tempo de clock, pois este comportamento será preservado de forma independente da máquina.

É buscada a classe da complexidade do algoritmo. Desta forma, se está interessado em saber se os algoritmos são executados em tempo exponencial ou polinomial. Para determinar a classe sem a necessidade de contar o número de operações executadas pelo algoritmo (o que seria uma análise exaustiva do algoritmo para saber seu comportamento interno), um método mais interessante é utilizado:

Seja uma função $f(x)$ da variação do tempo de clock (tempo de execução) em função da quantidade de nodos e arestas de um grafo. Considere que $f(x)$ seja uma função exponencial descrita na Equação 1, considerando c uma constante qualquer:

$$exp(x) = c^x. \tag{1}$$

Desta forma, ao ser aplicado $log(f(x))$, teremos $log(f(x)) = x$ caso a função seja exponencial, ou seja, $log(f(x))$ será linear. Caso contrário, a função poderá ser polinomial, e para verificar isto, é necessário aplicar a escala logarítmica tanto no eixo x quanto no eixo y, fazendo $log(f(log(x)))$. Caso a representação desta função seja linear, a função $f(x)$ é uma função polinomial.

Abaixo segue uma explicação informal de cada estrutura e a análise do tempo de execução do algoritmo em função do tamanho n do grafo (n nodos, $n-1$ arestas).

5 Grafo com listas encadeadas

Um grafo implementado com lista de adjacência possui um vetor n, onde cada elemento de n representa um nodo e possui uma lista encadeada e de nodos. Simplificando, cada nodo é representado por uma posição de n, e para cada aresta partindo de um nodo na posição n_i da lista n, o nodo destino é inserido na lista encadeada da posição n_i de n. Uma representação pode ser vista na Figura 3, onde um grafo e sua representação com listas é apresentado.

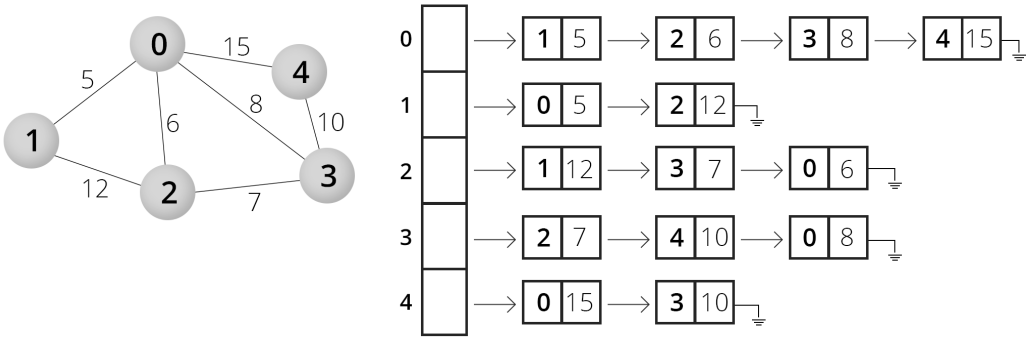

Figura 3: Implementação de grafo com listas encadeadas.

5.1 Análise do Algoritmo

Foi analisado o tempo de execução para grafos de 0 a 10.000 nodos, com um intervalo de 100 a 100. Desta forma, os dados coletados são para grafo com $0, 100, 200, ..., 9900$ e 10.000 nodos. O comportamento do algoritmo pode ser observado no gráfico da Figura 4, onde temos o tempo de execução (eixo das ordenadas) variando de acordo com a quantidade de nodos do grafo (eixo das abscissas).

Conforme comentado na seção 4, este gráfico representa uma função $f(x)$ do tempo de clock variando em função da entrada do algoritmo, que é um grafo com uma quantidade x de nodos. Assim, aplicando o método

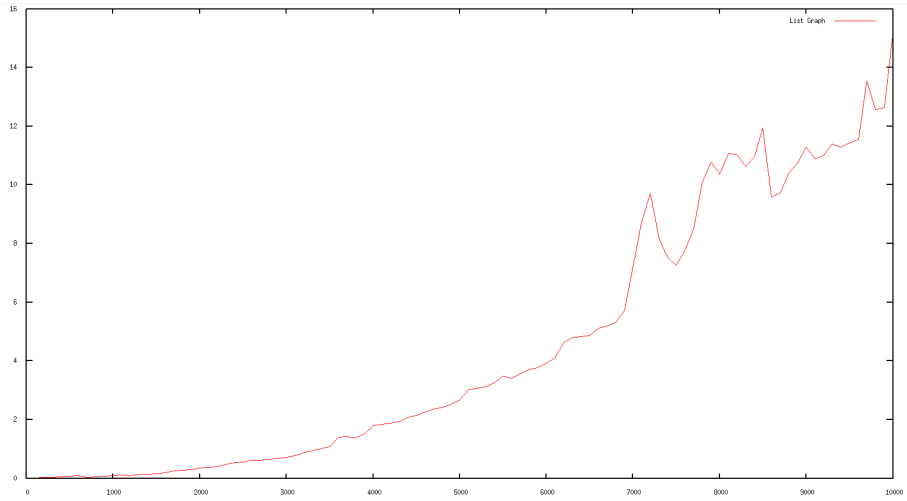

Figura 4: Tempo de clock da execução do algoritmo em função da quantidade de nodos de um grafos com listas encadeadas.

apresentado, primeiro será testado se a função da Figura 4 é uma função exponencial. Aplicando $log(f(x))$, obtem-se a função representada na Figura 5.

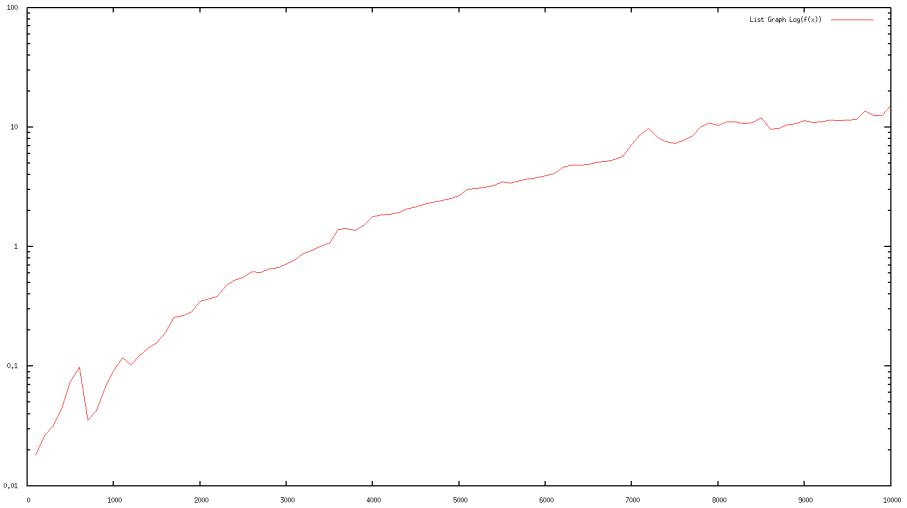

Figura 5: Aplicação de $log(f(x))$ na função f da Figura 4.

Como $log(f(x))$ não é uma função linear, então a função $f(x)$ deve ser uma função polinomial. Verificamos tal afirmação aplicando $log(f(log(x))$, obtendo o gráfico da Figura 6. Neste gráfico é obtida uma aproximação de uma função linear, abstraindo algumas variações que são comuns ao analisar-se funções a partir de dados. Assim sendo, pode-se dizer que nesta estrutura, o algoritmo possui comportamento aproximadamente polinomial.

6 Grafo com tabelas hash

A implementação de grafo que utiliza tabelas hash, possui uma tabela hash $nodos(k_n, n)$, onde a chave k_n é o valor do nodo n. Cada objeto nodo possui internamente outras duas tabelas hash: a primeira, $adjacentes(k_{ed}, e_a)$ possui todas as arestas que partem do nodo n, onde a chave k_{ed} é o valor do nodo destino da aresta e_a. A segunda tabela hash $incidentes(k_{eo}, e_i)$ possui todas as arestas que possuem n como destino, onde a chave k_{eo} é o valor do nodo origem da aresta e_i. Esta distinção entre incidentes e adjacentes é válido para grafos direcionados, porém este não é nosso caso. Ainda assim, a estrutura utilizada foi implementada desta forma, logo é um fator importante a ser comentado. A Figura 8 mostra uma representação desta estrutura.

Com esta estrutura baseada em tabelas hash, adição e remoção de nodos e arestas se tornam operações com custo $O(1)$.

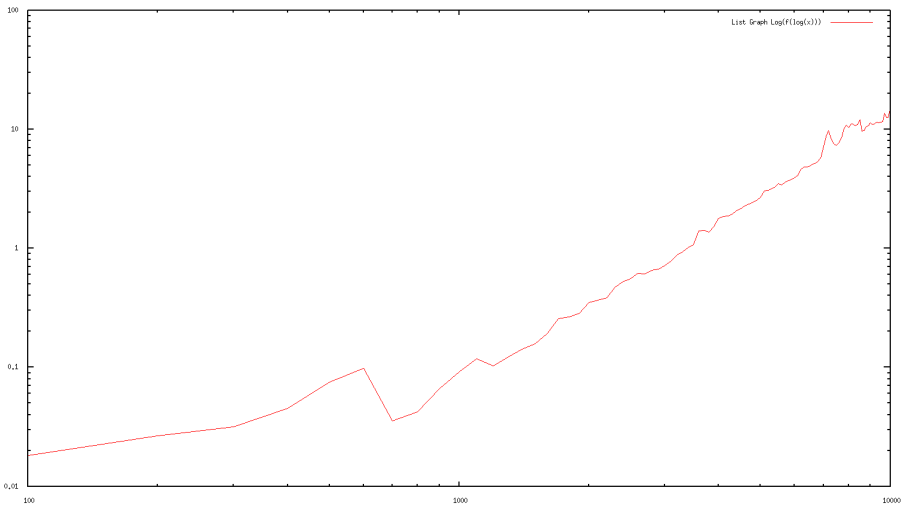

Figura 6: Aplicação de $log(f(log(x)))$ na função f da Figura 4.

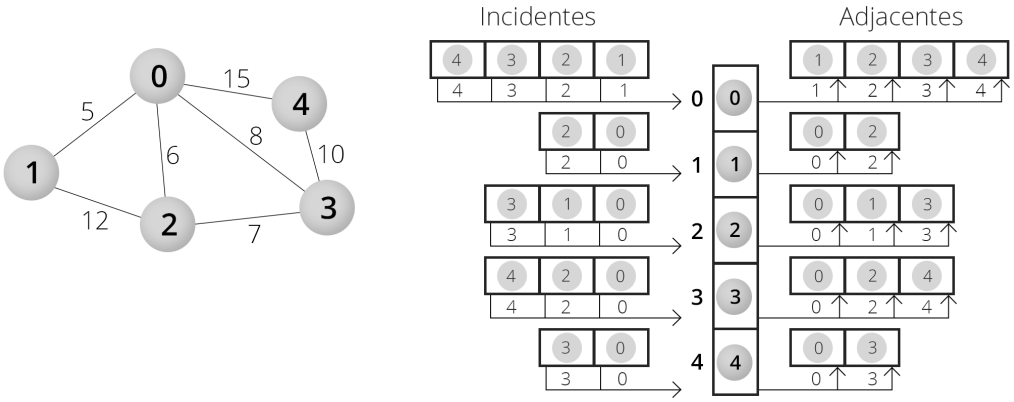

Figura 7: Implementação de grafo com tabelas hash.

6.1 Análise do Algoritmo

Assim como para a estrutura baseada em listas encadeadas, foi analisado o tempo de execução para grafos de 0 a 10.000 nodos, com um intervalo de 100 a 100. O comportamento do algoritmo pode ser visto no gráfico da Figura 8, onde temos o tempo de execução (eixo das ordenadas) variando de acordo com a quantidade de nodos do grafo (eixo das abscissas).

Novamente, devemos encontrar a classe da função $f(x)$ do tempo de clock em função da quantidade de nodos x do grafo de entrada do algoritmo. Conforme mostrado na Figura 9, o algoritmo não é exponencial ao ser aplicado em uma estrutura com tabelas hash, visto que $log(f(x))$ não é uma função linear.

Na Figura 10, é aplicado $log(f(log(x))$, que resulta em uma função aproximadamente linear, mostrando que em implementações de grafos com tabelas hash, o algoritmo possui tempo de execução aproximadamente polinomial.

7 Comparação do algoritmo nas estruturas

Conforme visto nas seções 5 e 6, em ambas implementações de grafos, o algoritmo possui um comportamento polinomial. A comparação dos comportamentos pode ser vista na Figura 11.

Pode-se pensar que, no final, não importa a estrutura, o algoritmo terá o mesmo comportamento em ambas estruturas. Porém, esta não é uma conclusão verdadeira, visto que o experimento nos diz informações mais sutis. Primeiro, o experimento mostra que o algoritmo executado em ambas estruturas possui comportamento polinomial, então podemos imaginar que, independente da estrutura, a função do comportamento do algoritmo continuará pertencendo à mesma classe de funções (no caso, polinomial). Porém, dentro da classe de funções polinomiais, há, por exemplo, $x^2, x^3, ...x^n$, para qualquer $n \in \mathbb{R}$. Desta forma, ainda que o comportamento

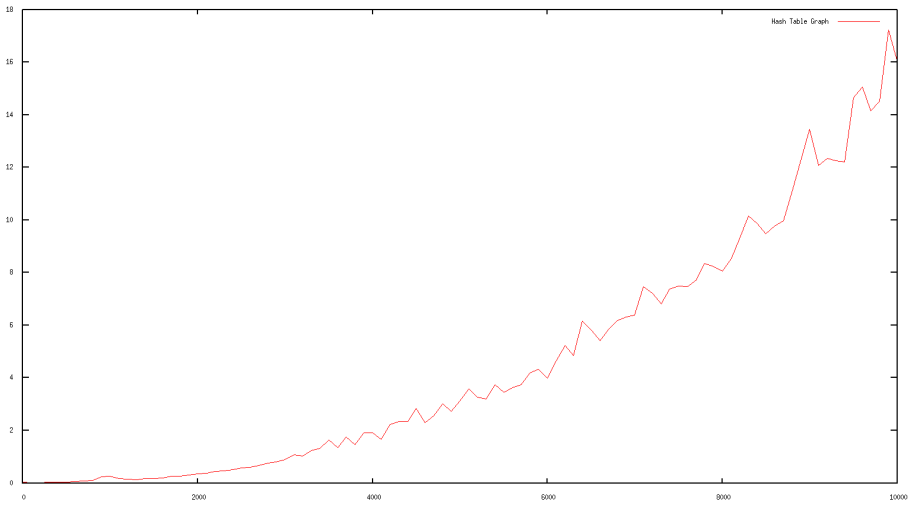

Figura 8: Tempo de clock da execução do algoritmo em função da quantidade de nodos de um grafo com tabelas hash.

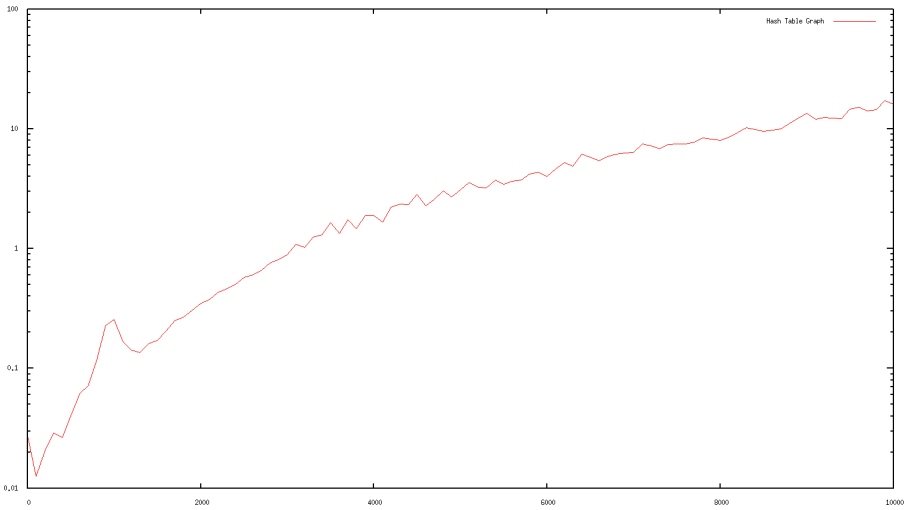

Figura 9: Aplicação de $log(f(x))$ na função f da Figura 8.

do algoritmo seja polinomial em ambas estruturas, um pode, na prática, acabar executando mais rápido que o outro. Para validar tal afirmação, foi testado o algoritmo para um grafo com 100.000 nodos, onde foram obtidos os valores apresentados na Tabela 1.

Assim, a diferença de tempo entre um grafo com listas encadeadas e outro com tabelas hash é de aproximadamente 2 minutos. Para um algoritmo que roda em 50 minutos, talvez não seja uma diferença tão significativa. Porém, à medida que a entrada de dados aumenta, esta diferença aumenta, conforme mostrado na Figura 11, onde a diferença entre uma execução e a outra é maior à medida em que a quantidade de nodos dos grafos é maior.

Outra informação importante que esta análise nos da é que cada estrutura de dados possui suas particularidades e são melhores para determinadas situações. A estrutura com tabelas hash foi feita para acesso e inserção rápida de valores, enquanto em estruturas de listas normalmente é necessário percorrer a lista a fim de encontrar um

Estrutura	**Tempo Total**
Listas encadeadas	≈48'06"
Tabelas Hash	≈50'30"

Tabela 1: Tempo de execução do algoritmo para grafos com 100.000 nodos

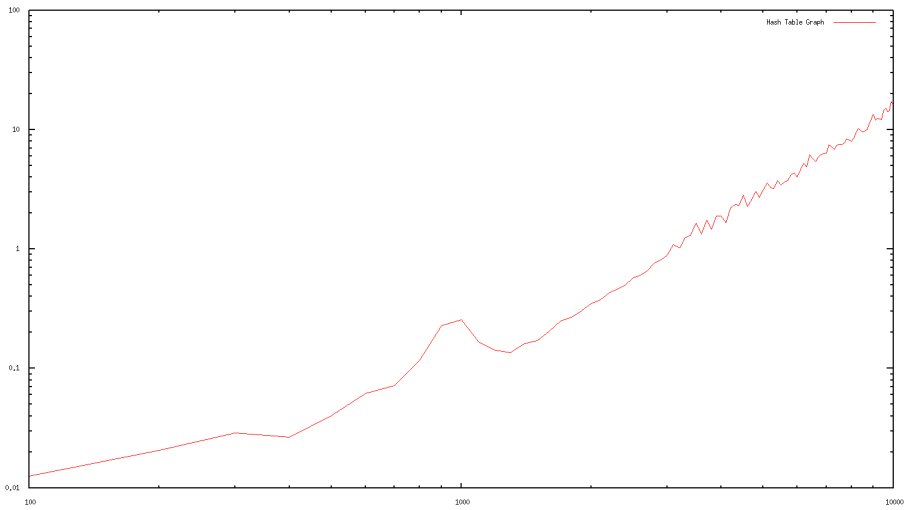

Figura 10: Aplicação de $log(f(log(x)))$ na função f da Figura 8

valor. Assim, deve-se considerar que o algoritmo de Kruskal não aproveita a vantagem das estruturas com tabelas hash, visto que este não insere ou remove nodos (aproveitando a operação $O(1)$ de tabelas hash), mas percorre todos os nodos de qualquer forma. Isto explica a similaridade do tempo de clock entre as implementações, mostrando que a estrutura de lista é suficiente para este algoritmo, podendo até ser melhor, já que o valor do nodo é armazenado diretamente na lista de nodos através dos índices, diferente da implementação com tabelas hash que armazena um objeto nodo, que deve ser acessado para obter-se o seu valor.

8 Conclusões

A partir dos experimentos realizados, foi feita uma análise da complexidade do tempo de clock do Algoritmo de Kruskal executado em diferentes estruturas, e a partir dos dados obtidos foi possível concluir que, ao escolher a estrutura adequada para um algoritmo, deve-se considerar se as vantagens da estrutura serão aproveitadas pelo algoritmo. No caso do algoritmo de Kruskal, a implementação de tabelas hash não foi vantajosa, visto que sua vantagem (acesso rápido à elementos específicos) não é utilizada no algoritmo. Talvez para um algoritmo que necessite de acesso direto a elementos específicos, a estrutura com tabelas hash seja a mais adequada. Porém, para o algoritmo de Kruskal, a melhor opção é a estrutura com listas, visto que esta é mais simples e aparentemente executa um pouco mais rápido, o que acaba sendo uma diferença significativa para valores muito grandes.

Por fim, percebe-se que, em teoria, a implementação com tabelas hash deveria ser melhor, ou ao menos igual a implementação com listas encadeadas. Porém, dado algumas características das implementações, esta mostrou-se menos interessante para o algoritmo em questão, visto que o uso de tabelas hash aumenta a complexidade da estrutura e, aparentemente, por armazenar elementos nodo e não apenas valores (como no caso da lista), esta estrutura realiza mais acessos à memória, tornando o processo um pouco mais demorado, conforme mostrado na Tabela 1. Assim, é possível perceber que muitas vezes a solução ótima na prática é diferente da que teoricamente deveria ser a melhor, mostrando que nem sempre a prática confirma a teoria.

Referências

[1] Kruskal's algorithm. `http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/GraphAlgor/kruskalAlgor.htm`. Acessado em: 09/06/2016.

[2] Kruskal's mst algorithm. `http://www.comp.dit.ie/rlawlor/Alg_DS/MST/Kruskal.pdf`. Acessado em: 09/06/2016.

[3] João B. Oliveira. Notação assintótica. `http://www.inf.pucrs.br/~oliveira/alestI/notacao.pdf`. Acessado em: 14/06/2016.

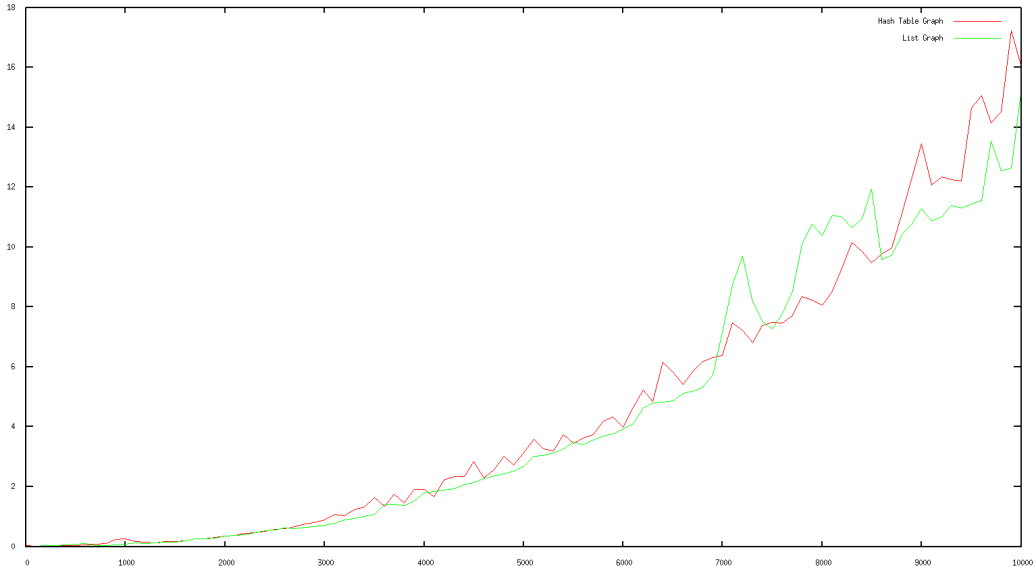

Figura 11: Comparação do algoritmo em ambas estruturas. A cor verde representa a execução na estrutura com listas e a cor vermelha na estrutura com tabelas hash.