

**Raphael Henrique Ribas**

*Aplicação da teoria da discrepância para computar a  
árvore geradora mínima*

Curitiba

**Raphael Henrique Ribas**

*Aplicação da teoria da discrepância para computar a  
árvore geradora mínima*

UNIVERSIDADE FEDERAL DO PARANÁ

Curitiba

# *Sumário*

<b>1</b>	<b>Introdução</b>	p. 4
<b>2</b>	<b>O problema</b>	p. 5
<b>3</b>	<b>Trabalhos anteriores</b>	p. 7
3.1	Boruvka . . . . .	p. 7
3.2	Kruskal . . . . .	p. 8
3.3	Prim . . . . .	p. 8
3.4	Fredman e Tarjan . . . . .	p. 8
3.5	Um algoritmo probabilístico . . . . .	p. 9
3.5.1	Verificação linear . . . . .	p. 9
3.5.2	Motivação . . . . .	p. 10
3.5.3	O algoritmo . . . . .	p. 10
3.5.4	Corretude . . . . .	p. 11
3.5.5	Complexidade . . . . .	p. 11
3.5.6	Entendendo o algoritmo . . . . .	p. 12
<b>4</b>	<b>O algoritmo de Chazelle</b>	p. 14
4.1	Soft Heap . . . . .	p. 14
4.2	Motivação . . . . .	p. 15
4.3	O algoritmo . . . . .	p. 16
4.4	Efeito da corrupção . . . . .	p. 17
4.5	Corretude do algoritmo . . . . .	p. 18

4.5.1	Contratibilidade . . . . .	p. 18
4.5.2	Invariantes que garantem a contratibilidade . . . . .	p. 18
4.6	Complexidade . . . . .	p. 19
4.6.1	Controle de corrupção . . . . .	p. 19
4.6.2	Invariantes que controlam a quantidade de corrupção . . . . .	p. 20
4.6.3	Quantidade de corrupção . . . . .	p. 21
4.6.4	Complexidade da construção da partição . . . . .	p. 22
4.6.5	Complexidade do algoritmo todo . . . . .	p. 23
<b>5</b>	<b>Conclusão</b>	<b>p. 26</b>
	<b>Referências Bibliográficas</b>	<b>p. 27</b>

# *1 Introdução*

O problema da árvore geradora mínima já é bastante antigo e foi provavelmente enunciado pela primeira vez por Boruvka [8] em 1926. A sua motivação veio de uma pergunta feita por uma companhia de energia elétrica, eles estavam interessados em saber como construir uma rede de distribuição de energia de forma que o seu custo fosse o menor possível. Esse é exatamente o problema de encontrar a árvore geradora mínima. Então Boruvka apresentou um algoritmo polinomial para o problema.

Depois de então, vários outros algoritmos foram propostos. No entanto o problema ainda continua em aberto. Não se sabe qual exatamente é a quota inferior para resolver o problema. Nesse texto faremos um estudo de um algoritmo proposto por Chazelle [2] que é baseado na teoria da discrepância. Este algoritmo apresenta complexidade  $O(m\alpha(m, n))$  onde  $\alpha$  é um inverso da função de Ackermann. Este algoritmo é o melhor conhecido até o momento.

## 2 *O problema*

No decorrer do texto, algumas convenções serão adotadas. O grafo será representado da forma usual, isto é, um grafo  $G$  é um par de conjuntos  $(V, E)$ , onde  $V$  é o conjunto de vértices e  $E$  é um conjunto de arestas. Todo grafo mencionado será não dirigido. As letras  $n$  e  $m$  representarão o número de vértices e de arestas respectivamente. Por simplicidade, assumimos que não existem duas arestas distintas com o mesmo custo. A árvore geradora mínima de um grafo  $G$  será representada por  $AGM(G)$ . Uma aresta que pertence à  $AGM$  é chamada de aresta leve, caso contrário, é uma aresta pesada.

O problema da árvore geradora mínima, ou PAGM, pode ser formalizado da seguinte forma. É dado um grafo  $G = (V, E)$  e um custo associado a cada aresta, também chamado de peso da aresta, representado por uma função  $c : E \rightarrow \mathbb{R}$ . A saída é a  $T = AGM(G)$ . Isto é,  $T$  é uma árvore geradora de  $G$ , isso significa que  $T$  é grafo conexo acíclico cujo conjunto de vértices é  $V$  e o conjunto de arestas é um subconjunto de  $E$ . E  $T$  apresenta custo mínimo, ou seja, a soma dos custos de suas arestas é a menor de todas as árvores geradoras de  $G$ .

Uma  $AGM$  possui duas propriedades bem interessantes que são a base de todos os algoritmos para o problema. A primeira delas é que a aresta de menor custo de um corte do grafo original pertence à  $AGM$ . A segunda é que a aresta de maior custo de qualquer circuito não pertence à  $AGM$ . O fato é que qualquer uma dessas duas propriedades é suficiente para computar o PAGM. E o interessante disso é que, essas duas propriedades não fazem menção à soma de pesos e única a operação necessário para sua verificação é a comparação entre os pesos das arestas. Isto é, em relação aos custos das arestas, só importa a ordem dos mesmos.

Vale notar também que essas regras não restringem de forma alguma qual corte ou circuito devem ser escolhidos. Então, essencialmente, a diferença dos algoritmos presentes na literatura se diferem pela forma que esses são escolhidos.

Existe um algoritmo que computa a  $AGM$  em tempo linear com a restrição de que os custos das arestas sejam números inteiros. Esse algoritmo depende de algumas operações com os valores dos custos além de apenas comparações. Mas aqui não colocamos nenhuma restrição

nos custos das arestas e nos limitamos a apenas fazer comparações. O problema foi enunciado com os custos sendo números reais, mas na verdade isso também não é necessário, pois estamos apenas interessados na ordem dos custos. Então o único requisito é que os pesos das arestas sejam elementos de um conjunto totalmente ordenado.

## 3 *Trabalhos anteriores*

Neste capítulo serão apresentados alguns algoritmos propostos anteriormente que serão de interesse na apresentação do algoritmo de Chazelle.

### 3.1 **Boruvka**

O algoritmo de Boruvka computa a AGM por construção e se baseia na primeira propriedade. O conjunto de arestas incidentes em um dado vértice é um corte do grafo, logo, dado um vértice, a aresta de menor custo incidente nele pertence à AGM. Então, para cada vértice, é escolhida a aresta de menor custo incidente nele, essa aresta é inserida da AGM, tomando cuidado com repetições. Depois disso é feita a contração das arestas escolhidas, isto é, é feita a união dos vértices incidentes nas arestas escolhidas, as auto arestas são removidas. O processo é repetido até que o grafo contenha apenas um vértice. De certa forma o que acontece é que o algoritmo vai crescendo a AGM a partir de várias árvores desconexas que corresponde a uma subárvore da AGM. A cada iteração as árvores são conectadas entre si pelas arestas de menor custo do seu corte até que no final resta apenas uma árvore que é a AGM.

A corretude desse algoritmo pode ser facilmente verificada. Na primeira iteração as arestas escolhidas de fato pertencem à AGM pela primeira propriedade. Para compreender porque o processo de contração das arestas funciona, basta observar o que representa a vizinhança de um vértice após feita a contração. Vamos supor agora que durante a contração a aresta  $e$  que incide nos vértices  $u$  e  $v$  foi contraída e deu origem ao vértice  $w$ . A vizinhança de  $w$  possui todas as arestas incidentes em  $u$  e  $v$  com exceção das arestas  $uv$ , observe que podem aparecer arestas múltiplas durante a contração e a remoção das mesmas não é necessário. Logo, as arestas de  $w$  correspondem as arestas que incidem em  $u$  ou em  $v$  e em algum outro vértice fora  $u$  e  $v$ . Isso mostra que a vizinhança de  $w$  é um corte que separa  $u$  e  $v$  do restante do grafo. Então a primeira regra pode ser novamente aplicada na vizinhança de  $w$ . Portanto, todas as arestas escolhidas são de fato arestas que pertencem à AGM. E todas as arestas da AGM são eventualmente escolhidas, para cada contração o número de vértices reduz em uma unidade,

então o algoritmo precisa fazer  $n - 1$  contrações para obter um grafo com apenas um vértice, ou seja,  $n - 1$  arestas serão escolhidas.

Para cada iteração do algoritmo, é escolhido a aresta de menor custo de cada vértice, o que pode ser feito em tempo  $O(m)$ . Depois as contrações são feitas, o que também pode ser feito em  $O(m)$ . Em cada iteração serão escolhidas pelo menos  $n/2$  arestas distintas, logo, após a contração serão removidos pelo menos  $n/2$  vértices. Com isso pode se concluir que o número de iterações é  $O(\log n)$  o que resulta em uma complexidade igual a  $O(m \log n)$ .

## 3.2 Kruskal

Posteriormente surgiu o algoritmo de Kruskal [7] e do Prim [9]. No Kruskal a escolha das arestas é feita tomando a menor das arestas que ainda não foram considerada, tal aresta, se não forma um circuito com as arestas que já foram incluídas na AGM, certamente é a menor de algum corte e pode ser incluída na AGM. A verificação de circuito pode ser feita de forma eficiente usando a estrutura union-find, no entanto as arestas precisam ser ordenadas pelo custo e isso se torna o gargalo do algoritmo, portanto a complexidade é  $O(m \log m)$ .

## 3.3 Prim

No Prim, diferente do Boruvka, é construída apenas uma árvore. Em cada iteração, são consideradas as arestas do corte que separa a árvore que já foi construída do restante do grafo, pela primeira regra, a menor delas está na AGM. Então a árvore é estendida com essa aresta e o vértice incidente. O algoritmo pára quando a árvore cobre o grafo inteiro. Com o uso da Fibonacci heap [4] para selecionar a menor aresta do corte e mantendo apenas uma aresta do corte para cada vértice que ainda não entrou na árvore, esse algoritmo tem complexidade  $O(n \log n + m)$ . Notamos que se o grafo for denso o suficiente, nesse caso isso quer dizer que tem  $\Omega(n \log n)$  arestas, o algoritmo leva tempo linear para computar a AGM, então os algoritmos mais sofisticados atacam os casos em que o grafo é mais esparso.

## 3.4 Fredman e Tarjan

Fredman e Tarjan [4] publicaram um algoritmo baseado no algoritmo de Prim que computa a AGM em tempo  $O(m\beta(n, m))$ , onde  $\beta(n, m) = O(\log * n)$ . Analisando a complexidade do Prim, notamos que o gargalo está no  $n \log n$  que se dá pelo fato de que a heap uma quantidade

grande de elementos e as suas operações se tornam caras. A idéia aqui é limitar o tamanho dessa heap. E para isso foi necessário adotar uma estratégia parecida com o algoritmo do Boruvka, isto é, crescer várias árvores separadamente. Para cada iteração é escolhida uma certa contante  $k$ , executa-se o Prim a partir de algum vértice até que a heap atinja tamanho  $k$  ou até que a árvore incorpore um vértice que já esteja em outra árvore. Depois, escolhe-se um vértice que não esteja em nenhuma árvore. Esse processo é repetido até que todos os vértices estejam em alguma árvore. Nesse momento existem várias árvores separadas, então é feita a contração de todas as arestas que foram escolhidas para alguma árvore. Feita a contração, cada árvore se torna um vértice e o grafo resultante possui um número menor de arestas. Essa iteração é repetida até que o grafo resultante possua apenas um vértice. O valor de  $k$  é escolhido de forma que cada iteração leva um tempo linear em  $m$  e tal valor cresce exponencialmente conforme a densidade do grafo  $e$ , que por sua vez, cresce a cada iteração. A última iteração ocorre quando o valor de  $k$  atinge o número de vértice do grafo corrente, o que acontece com um número pequeno de iterações. Uma explicação mais detalhada desse algoritmo está além do escopo desse trabalho, o leitor interessado poderá consultar as referências. O importante aqui é observar que a técnica de construir a árvore de forma separada foi aplicada com sucesso. Ela pode ser vista como uma divisão e conquista bottom-up com o detalhe que a divisão não é clara até que todas as subárvores estejam prontas. Essa técnica será levada mais adiante no algoritmo de Chazelle.

## 3.5 Um algoritmo probabilístico

Mais recentemente, Karger, Klein e Tarjan [5] descobriram um algoritmo probabilístico para computar a AGM em tempo esperado linear. Este foi baseado em um algoritmo para o problema de verificar uma AGM. Este foi resultado de vários autores e sua versão final foi apresentada por King [6]. Faremos uma breve discussão sobre o algoritmo de verificação pois uma explicação detalhada foge do contexto desse texto.

### 3.5.1 Verificação linear

O problema de verificar uma AGM pode ser formalizado da seguinte forma. É dado um grafo  $G$  e uma árvore geradora  $T$ . A entrada deve ser aceita se  $T = AGM(G)$ .

O algoritmo é uma aplicação da segunda propriedade. Para cada aresta que está fora da árvore verifica-se no circuito que ela forma com as arestas da árvore qual é a aresta de maior peso. Para que a árvore seja mínima a aresta de maior peso do circuito deverá ser a aresta que está fora da árvore se isso não acontecer a árvore dada não é mínima e o algoritmo rejeita. Essa

verificação é feita para todas as arestas. A árvore só é dita mínima se todas as arestas passarem na verificação. A utilidade desse algoritmo não é simplesmente a verificação, mas sim o próprio algoritmo. Ele fornece uma forma eficiente de descartar uma grande quantidade de arestas que não estão na AGM. Note que até agora, a construção da árvore se dava pela inclusão de arestas apenas, agora vamos construir pela exclusão de arestas.

### 3.5.2 Motivação

A idéia desse algoritmo vem da observação de que a partir de uma árvore geradora que é quase mínima, o algoritmo de verificação nessa árvore, vai encontrar algumas arestas que está de fora mas que não são a maior do circuito fundamental, o que prova que a árvore dada não é mínima. Mas para a maioria das arestas que estão de fora, será verificado que de fato essa aresta é a maior do seu circuito fundamental e pela segunda propriedade essa aresta não pertence à AGM então pode ser descartada. Infelizmente o número de arestas descartadas em um grafo muito esparso é pequena, para isso será algumas iterações do algoritmo de Boruvka que reduz o número de vértices e torna o grafo mais denso, com isso permitindo que o descarte de arestas seja eficiente. A grosso modo o algoritmo funciona em duas etapas, uma que reduz o número de arestas e outra que reduz o número de vértices. E cada iteração reduz o número de vértices e arestas para menos da metade, o que nos dá um algoritmo linear.

A parte probabilística entra na construção da árvore quase mínima e a proposta do algoritmo é, na verdade, bastante simples. Sorteia-se metade das arestas e, recursivamente, computa-se a AGM usando apenas essa metade. A árvore encontrada a partir das arestas sorteadas é próximo o suficiente da AGM e o esperado é que, após o processo de descarte de arestas, reste apenas  $2n$  arestas. O número de arestas que vão restar dependem das arestas sorteadas no passo anterior, pode ser até que as arestas sorteadas formem um grafo desconexo, mas, felizmente, isso não acarreta nenhum problema muito sérios. Vamos mostrar que a chance de o número de arestas após o descarte ser muito maior que o  $2n$  esperado é pequena. Quando tivermos um grafo desconexo, será computado a floresta geradora mínima e no descarte de arestas, qualquer aresta que liga duas árvores separadas é considerada leve e não é descartada.

### 3.5.3 O algoritmo

Então o algoritmo final pode ser escrito da seguinte forma.

1. Execute duas iterações do algoritmo do Boruvka.

2. Faça o sorteio das arestas. Chamamos o grafo com somente as arestas sorteadas de  $G'$
3. Compute recursivamente  $T' = AGM(G')$ .
4. Execute o algoritmo de King para descartar arestas pesadas das arestas que não foram sorteadas. Chamamos de  $E'$  o conjunto de arestas não descartadas.
5. Compute recursivamente  $AGM(T' \cup E')$ .

### 3.5.4 Corretude

A corretude do algoritmo é bastante direta, só precisamos nos preocupar mais com o fato de se o algoritmo realmente termina. A expectativa do número de arestas após a filtragem vem da análise do que ocorre com a AGM das arestas sorteadas. O sorteio ocorre de forma um pouco diferente do que foi comentado anteriormente, cada aresta é sorteada com probabilidade meio de forma independente. Supondo que o sorteio acontece da aresta mais leve para a mais pesada, toda vez que decidimos não sortear uma aresta leve, alguma aresta pesada se torna leve, a aresta não sorteada será considerada leve no filtro e não será descartada. Aqui vou me referir a arestas leves e pesadas em relação à AGM das arestas que foram sorteadas até o momento mais as arestas que ainda não foram consideradas, assim, tal árvore pode mudar cada vez que uma aresta não é sorteada. Isso quer dizer que as arestas que não são filtradas são somente as arestas que eram leves no momento do seu sorteio mas não foram sorteadas. A chance de não conseguirmos sortear  $n - 1$  arestas após  $k$  sorteio de arestas leves se comporta de acordo com a função cumulativa da distribuição binomial negativa que decresce de forma exponencial a partir de  $n - 1$  tentativas e o valor esperado é  $n - 1$ . Juntando essas arestas não filtradas com as arestas que estão na árvore quase mínima, temos os  $2n$  esperados de arestas após a filtragem.

### 3.5.5 Complexidade

A complexidade de cada iteração, desconsiderando as chamadas recursivas, é linear no tamanho do grafo. O passo 1 é linear como já vimos anteriormente. O passo 2 também, porque consiste apenas de  $m$  sorteios. O passo 4 depende apenas do algoritmo de King que também vimos que é linear. Então em cada iteração a complexidade é  $O(n + m)$ , é importante colocar o  $n$  porque em qualquer momento o grafo em questão pode ser bem esparsa e desconexo, isso torna o número de vértices relevante.

Os demais passos são chamadas recursivas. Considere que as iterações do Boruvka ocorre antes de cada chamada recursiva, o efeito é o mesmo mas facilitará a explicação. O tamanho

de um grafo será representado por um par  $(n, m)$ . Na primeira recursão, a expectativa é de sejam sorteadas metade das arestas, e após as iterações do Boruvka o número de vértices é reduzido para no máximo um quarto. Então o tamanho do grafo na primeira recursão será  $(n/4, m/2)$ . Na segunda recursão o grafo terá apenas as arestas não descartadas e a expectativa é que sobrem  $2n$  arestas, então o tamanho do grafo esperado é  $(n/4, 2n)$ . Então podemos escrever a complexidade com a seguinte recursão  $F(n, m) = O(n + m) + F(n/4, m/2) + F(n/4, 2n)$  que é  $O(n + m)$ .

### 3.5.6 Entendendo o algoritmo

Agora vamos parar um pouco e refletir sobre o que acontece com esse algoritmo. As idéias apresentadas são a base para a construção do algoritmo de Chazelle. Tal algoritmo foi publicado junto com um livro sobre técnicas relacionadas à teoria da discrepância. Essa teoria nasceu de uma pergunta feita por van der Corput em 1935. Quão uniforme pode ser uma seqüência finita de números em  $[0, 1]$ ? Em outras palavras, quanto podemos aproximar uma seqüência contínua com um seqüência discreta?

Do ponto de vista computacional, podemos esquecer os detalhes e vamos nos preocupar com outra pergunta semelhante. Quão bem um subconjunto representa o conjunto original? Para mostrar isso de forma simples, tome o problema da seleção linear, que consiste em determinar o  $k^{\text{th}}$  menor elemento de um conjunto. O interesse nesse problema é o algoritmo que o resolve em tempo linear.

A idéia fundamental do algoritmo é por partição do conjunto, escolha um elemento  $x$  do conjunto, conte quantos elementos menores do que  $x$  existem, se esse valor for maior do que  $k$ , então o  $k^{\text{th}}$  elemento é menor que  $x$ , logo todos os elementos maiores que  $x$  podem ser descartados. Caso contrário, os elementos menores que  $x$  são descartados. E o processo se repete até que sobre apenas um elemento. A quantidade de elementos descartados depende da escolha de  $x$ . Para garantir que essa quantidade seja grande o suficiente, temos que garantir que as quantidades de elementos maiores e menores que  $x$  não sejam muito diferentes. Assim qualquer que seja o resultado da comparação com  $k$ , a quantidade de elementos descartados vai ser aproximadamente a mesma. Isso é o mesmo que garantir que  $x$  será um elemento próximo da mediana.

Uma estratégia é fazer um sorteio de um quarto dos elementos, computa-se a mediana desses elementos sorteados com uma chamada recursiva. A expectativa é de que essa mediana seja bem próxima da mediana do conjunto original e que seja uma boa escolha para  $x$ . Uma outra forma determinística é particionar o conjunto em partes de tamanho cinco, tomar a mediana de

cada parte, a mediana dessas medianas estará, no máximo, há uma distância de um quarto dos elementos da mediana original. Essa escolha garante que será descartada pelo menos um quarto dos elementos o que é suficiente para garantir comportamento linear.

As duas estratégias apresentadas, corresponde a uma amostragem do conjunto original. Tal amostra é representativa o suficiente para que com a sua mediana nós possamos inferir algo útil sobre o conjunto original. E essa é a questão principal da teoria da discrepância. O conjunto das medianas das partes corresponde a um conjunto de baixa discrepância em relação ao conjunto original.

Voltando ao algoritmo probabilístico para encontrar a AGM, podemos observar que o que sorteia metade das arestas, na verdade, está construindo um grafo que possui baixa discrepância em relação ao grafo original. A árvore mínima desse subgrafo nos dá informações suficientes para inferirmos a ausência de várias arestas da AGM. A pergunta agora é outra. Quão bem esse subgrafo representa o grafo? E a resposta já está bem respondida com o que foi apresentado anteriormente.

O algoritmo probabilístico funciona bem, mas ainda estamos curiosos para entender completamente o PAGM. Enquanto esse algoritmo é revelador ele não mostrou tudo ainda. Ainda não se sabe ainda, por exemplo, qual é a quota inferior para o problema. Para isso uma versão determinística desse algoritmo seria muito interessante. E a busca pelo mesmo é o objetivo principal dos estudos publicados no livro do Chazelle, lá ele mostra a aplicação de várias técnicas relacionadas a teoria da discrepância aplicadas em diversos problemas. Primeiramente a aplicação mais ingênua resulta em um algoritmo probabilístico, mas com um estudo detalhado, versões determinísticas também foram apresentadas.

## 4 *O algoritmo de Chazelle*

A pergunta agora é como determinar um subgrafo que seja um bom representante do grafo de forma determinística. Em comparação com o problema da seleção linear, em ambos, uma estratégia probabilística funciona igualmente bem. Será que o mesmo pode ser dito sobre uma estratégia determinística? A dificuldade no grafo é que, além de a amostra precisar que ter uma boa representativa numérica, ou seja, represente bem os pesos das arestas, ela também precisa ter uma boa representatividade combinatória, ou seja, represente bem a estrutura do grafo. Uma simples amostragem aleatória se mostrou bem eficaz nesses dois requisitos.

Lembrando que nos algoritmos determinísticos, de certa forma, eles eram todos limitados pela procura das arestas de menor custo. Geralmente era usada uma heap para determinar a menor aresta de um corte. O custo elevado das operações da heap motivou a forma dos algoritmos até então. Seguindo a tendência, novamente a heap é atacada, como só queremos algo que é quase mínimo, podemos relaxar a forma heap e permitir que ela cometa erros em prol do desempenho. Então a proposta é o uso da soft heap [3].

### 4.1 **Soft Heap**

O princípio da soft heap é aumentar o valor das chaves dos elementos a fim de controlar o número de chaves distintas, um elemento cuja chave foi modificada será considerado corrompido. A chave só aumenta, então uma vez que um elemento foi corrompido, ele continuará corrompido. Para simplificar as discussões futuras, a chave usada pela soft heap será chamada de chave corrente que pode ser maior que a chave original se o elemento estiver corrompido. Para o problema da AGM as chaves serão os custos das arestas e nesse caso falaremos em custo original e custo corrente.

O número de elementos corrompidos é controlado por um parâmetro  $0 < \varepsilon < 1/2$ . A garantia é que após  $n$  inserções, a heap deverá conter no máximo  $\varepsilon n$  elementos corrompidos a qualquer instante. Note que  $n$  não representa o número de elementos presentes na heap como

normalmente acontece. Com base no que foi dito, as operações da heap podem ser feitas em tempo constante com exceção da remoção da menor chave que é feita com tempo  $O(\log 1/\varepsilon)$ . No entanto, o valor de  $\varepsilon$  será uma única constante durante toda execução do algoritmo, então podemos considerar que todas as operações dessa heap vão levar um tempo constante.

A soft heap foi desenvolvida com base na heap binomial, uma vantagem é que esta permite união de duas heaps de forma eficiente. Outra vantagem é o seu formato mais flexível que permite que a inserção e a união ocorram em tempo constante, é a mesma idéia que foi aplicada na Fibonacci heap. E só estamos interessados em três operações da heap, que são a inserção, remoção e união.

## 4.2 Motivação

Infelizmente, não é possível simplesmente usar a soft heap em algum algoritmo visto anteriormente, a quantidade de chaves corrompidas seria enorme e a árvore produzida seria inútil. A maior dificuldade é controlar a quantidade e o efeito da corrupção. Para isso será usado uma estratégia semelhante ao que foi aplicado no algoritmo de Fredman e Tarjan, que era construir várias árvores separadas a fim de reduzir o tamanho da heap e ganhar desempenho. Aqui esta técnica ganha outra justificativa, quanto menor a heap, menor é o número de chaves corrompidas.

No algoritmo de Fredman e Tarjan, os conjuntos de vértices das árvores construídas em cada iteração correspondem a uma partição do conjunto de vértices do grafo. Tal partição só é conhecida ao final da iteração. Mas aqui será feito diferente, a partição será construída antes de computar a AGM. A vantagem óbvia é que podemos aplicar uma divisão e conquista efetiva, isto é, podemos computar a AGM de cada parte separadamente. A outra vantagem não tão óbvia é que a construção da partição é mais tolerante a corrupção gerada pela soft heap. As arestas serão guardadas na soft heap, e quando a soft heap corromper a chave dessa arestas, dizemos que essa aresta se tornou corrompida. Arestas corrompidas que incidem em dois vértices da mesma parte não influenciam no resultado.

Podemos organizar as partições como uma hierarquia  $T$ . As folhas correspondem aos vértices do grafo inicial. Cada nó interno  $z$  corresponde a um vértice que foi criado pela contração do subgrafo  $C_z$  que é composto pelos vértices correspondentes aos filhos de  $z$ .

Até agora só nos referimos à contração de arestas. Vamos estender essa noção para um subgrafo. Contrair um subgrafo é o mesmo que contrair todas as suas arestas. De certa forma, essa ação já foi utilizada no algoritmo de Tarjan, onde cada árvore era contraída. Dizemos que

um subgrafo  $C$  de  $G$  pode ser contraído, isto é, ele é contraível quando a intersecção de  $C$  com a  $AGM(G)$  é conexa. É fácil perceber que isso nos garante que a  $AGM(C)$  também é um subgrafo da  $AGM(G)$ . Assim, para computar a  $AGM$  do grafo, basta computar a  $AGM(C)$  e do grafo resultante da contração de  $C$ .

O efeito das arestas corrompidas é que as partições computadas não serão corretas. Algumas arestas corrompidas vão atrapalhar, essas arestas são chamadas de arestas ruins e serão temporariamente descartadas para serem reprocessadas posteriormente. Mas as partições estarão corretas em relação as arestas boas. Ela nos fornecerá um meio de encontrarmos uma  $AGM$  mínima das arestas boas que será uma árvore geradora quase mínima do grafo original. Qualquer aresta boa fora desta árvore pode ser definitivamente descartada. Em seguida, as arestas ruins são colocadas de volta no grafo para computar e então a verdadeira  $AGM$  é computada.

### 4.3 O algoritmo

O algoritmo pode ser descrito pelos seguintes passos.

1. Se o grafo for menor que uma constante, compute a  $AGM$  de forma direta.
2. Execute  $k$  iterações do algoritmo de Boruvka.
3. Construa a hierarquia  $T$  e o conjunto  $B$  de arestas ruins.
4. Compute  $T' \leftarrow \bigcup_{z \in T} AGM(C_z B)$
5. Devolva  $AGM(T' \cup B)$

Podemos notar a semelhança com o algoritmo probabilístico, o dois primeiros passos são praticamente os mesmos e desempenha o mesmo papel. O terceiro e quarto passos eliminam arestas que não estão na  $AGM$ . E o último passo é a recursão com as arestas que sobraram.

O primeiro passo trata do caso base da recursão, se o grafo se tornou pequeno o suficiente, pode aplicar qualquer algoritmo visto anteriormente. O segundo passo serve para reduzir o número de vértices, isso é importante para garantir que o número de arestas descartadas nos passos seguintes seja grande o suficiente para garantir o andamento do algoritmo.

O terceiro passo é parte mais interessante de todo o processo, a implementação dos demais passos é bastante direta. Ao contrário do que acontece nos outros algoritmos, a hierarquia é

construída de cima para baixo. A construção é feita com duas operações. A extensão, que a escolha de uma vértice novo para a criação de um novo nó na hierarquia. E a retração, que é feita quando um subgrafo atinge o seu tamanho esperado, ele é contraído e o vértice resultante é inserido no subgrafo pai.

Para explicar esse processo vamos considerar o algoritmo já em execução, a construção da hierarquia se dá como uma busca em profundidade. Então sejam os nós  $z_0, \dots, z_k$  correspondentes ao caminho ativo, os subgrafos  $C_{z_0}, \dots, C_{z_k}$  estão sendo construídos.

A retração ocorre sempre no subgrafo do último nó do caminho ativo, isto é, em  $C_{z_k}$ . O vértice resultante da contração de  $C_{z_k}$  é inserido em  $C_{z_{k-1}}$ . Após a retração, o caminho ativo deixa de incluir o nó  $z_k$ . A retração em si é bem trivial, a dificuldade maior é manter as estruturas de dados atualizadas. Essas estruturas são utilizadas na escolha do vértice para extensão e será vista em detalhes em seguida.

## 4.4 Efeito da corrupção

Vamos chamar de arestas de borda as arestas que incidem exatamente em um vértice que já foi incluído na hierarquia. Para extensão, podemos escolher o vértice que incide na aresta de borda de menor custo. Pela propriedade do corte, sabemos que tal aresta pertence a AGM portanto é uma escolha segura para manter cada subgrafo contraível. No entanto determinar esta aresta é custoso e queremos fugir disso. Então usaremos a menor de acordo com a soft heap, isto é, será escolhido a aresta de menor custo corrente.

O efeito da corrupção é que as arestas corrompidas podem se "esconder", isto é, eles deveriam ter sido escolhidas em algum momento como a menor, mas, por causa do seu custo aumentado, não foram. Essas arestas ainda podem ser escolhidas antes que o seu subgrafo incidente seja contraído, ou seja, o erro pode ser arrumado até a contração se eventualmente essa aresta for trazida para dentro do subgrafo. Por esse motivo, o erro gerado pela corrupção de uma aresta só é definitivo quando o subgrafo incidente da aresta é contraído, mas somente se ela for de borda. Nesse caso, fingimos que as arestas de borda corrompidas incidentes no subgrafos que está sendo contraído possuem como custo original o mesmo custo corrente e essas arestas são consideradas ruins. Para isso definimos o custo de trabalho, que é igual ao custo original se a aresta é boa, caso contrário, é o custo corrente. O resultado disso é que as partições estarão corretas em relação ao custo de trabalho. O que acontece é que toda vez que uma aresta atrapalha, ela é marcada como ruim e tudo volta a ficar certo de acordo com o custo de trabalho.

## 4.5 Corretude do algoritmo

### 4.5.1 Contratibilidade

Apenas marcar as arestas ruins não é suficiente, é preciso um certo cuidado extra para garantir que cada subgrafo seja contraível. Uma forma é garantir que os subgrafos sejam fortemente contraível, isto é, para todo par de arestas de borda de  $C$ ,  $e$  e  $f$ , existe um caminho em  $C$  que conecta  $e$  e  $f$  mas que não possui nenhuma aresta que excede os custos de  $e$  e  $f$ . Provamos por contradição que se um subgrafo  $C$  é fortemente contraível, então ele é contraível. Assuma que tal subgrafo não é contraível, por definição a sua intersecção com a AGM deve possuir mais de uma componente conexa. Agora considere o caminho  $\pi$  na  $AGM(G)$  que conecta duas componentes distintas de  $C$ . Esse caminho não possui nenhuma aresta em  $C$ , caso contrário não seria o menor caminho, e possui mais de uma aresta, se tivesse apenas uma aresta, essa aresta estaria em  $C$  pois este é um subgrafo induzido pelos vértices. Então, suas arestas de extremidades são distintas e cada uma incide em exatamente um vértice de  $C$ . Como  $C$  é conexo,  $\pi$  forma um circuito com algum caminho em  $C$ , então pela propriedade da AGM a maior das arestas desse circuito não pertence à AGM, mas qualquer aresta em  $C$  possui custo menor que as extremidades de  $\pi$ , logo a aresta mais pesada não está em  $C$ , mas está na AGM o que é uma contradição e isto prova a hipótese.

### 4.5.2 Invariantes que garantem a contratibilidade

Para garantir que os subgrafos sejam fortemente contraíveis são mantidas algumas invariantes. A invariante 1 é: para todo  $i < k$ , é mantido uma aresta chamada de aresta de cadeia que conecta  $C_{z_i}$  e  $C_{z_{i+1}}$ . O custo corrente dessa aresta (i) não é maior que o custo corrente de nenhuma aresta de borda incidente em  $C_{z_1} \cup \dots \cup C_{z_i}$  e (ii) menor do que o custo de trabalho de qualquer aresta que conecta dois  $C_{z_j}$  ( $j \leq i$ ) distintos.

A retração não viola essas invariantes, pois a aresta de cadeia do subgrafo que está sendo retraído passa a ser uma aresta interna do subgrafo pai e não precisamos mas nos preocupar com ela, ou seja, ela deixa de ser uma aresta de cadeia. O subgrafo que ganha o vértice resultante da contração, pode ganhar algumas arestas de borda novas, mas estas não geram restrições nas arestas de cadeia que aparecem acima no caminho ativo.

A parte mais difícil de manter essas invariantes é durante a extensão. A aresta  $(u, v)$  de borda de menor custo corrente será a aresta de cadeia. Seja  $v$  o vértice que está de fora durante a extensão. Um novo subgrafo  $C_{z_{i+1}}$  será criado e conterá apenas o vértice  $v$ . Agora preci-

samos garantir as invariantes descritas a cima. A invariante 1.i já é mantida pela escolha da aresta, que é a menor aresta de borda. Mas pode ser que o custo dessa nova aresta de cadeia não obedeça a invariante 1.ii, pode até ser que tal aresta não incida no último  $C_z$  ativo e teríamos uma ramificação do caminho ativo. Se for o caso, é feita a fusão, que a contração dos últimos subgrafos da cadeia até que a primeira restrição seja satisfeita. Mais especificamente, é escolhida uma aresta  $(a, b)$  que conectam dois  $C_z$  distintos cujo custo de trabalho é no máximo o custo corrente de  $(u, v)$  e incide no  $C_{z_i}$  mais próximo da raiz do caminho ativo. Considere que  $a \in C_{z_i}$ . Então é feita a contração dos  $C_{z_{i+1}} \cup \dots \cup C_{z_k}$  e depois é feita a contração da aresta  $(a, b)$ .

Depois da fusão, a extensão pode ser feita. Pela escolha de  $(a, b)$ , se o vértice  $u$  não pertencia ao último  $C_z$  da cadeia, agora pertence por consequência das duas restrições. O subgrafo  $C_{z_{i+1}}$  é colocado no final da cadeia. Para uma escolha eficiente da aresta  $(a, b)$ , é também mantido, para cada par  $C_z$  da cadeia, a menor aresta que os conectam. Essas arestas podem ser atualizadas de forma trivial em cada retração e extensão.

A fusão é um processo necessário mas tem um efeito meio confuso. Vários subgrafos em construção são contraídos mesmo que não tenham atingidos o seus tamanhos desejados. Mas nada de muito dramático acontece, os subgrafos “incompletos” são considerados prontos. A verdade das fusões é que elas são boas do ponto de vista computacional. Se ocorresse fusão em toda extensão, a altura da cadeia não passaria de dois nós e sua construção seria mais rápida, porque a segunda restrição e a manutenção das heaps penalizam os caminhos ativos muito longos.

## 4.6 Complexidade

Até agora só nos preocupamos com a corretude do algoritmo, agora focamos a atenção no seu desempenho. Até o momento, alguns detalhes importantes foram omitidos, são eles o tamanho máximo que cada  $C_z$  pode atingir. Outro detalhe é que só foi comentado como tratar arestas corrompidas, mas nada foi dito como manter a quantidade destas sob controle. Esses aspectos não afetam a corretude do algoritmo.

### 4.6.1 Controle de corrupção

Usar apenas uma soft heap para guardar todas as aresta não é suficiente. A quantidade de corrupção ainda seria grande demais. Então as arestas de borda são distribuídas em várias heaps seguindo o seguinte critério que será a invariante 2. Antes de entrar em detalhes, vamos lembrar

as garantias da soft heap. O número de arestas corrompidas depende do número de inserções naquela heap. Isso quer dizer que depois de muitas inserções e remoções na mesma heap, o número de chaves corrompidas na heap pode ser muito grande, isto é, se uma heap se torna muito “velha” ela se torna inútil. A estratégia será manter as arestas em várias heaps, e durante o processo heaps serão criadas e destruídas, assim podemos manter a quantidade de corrupção sob controle.

#### 4.6.2 Invariantes que controlam a quantidade de corrupção

Para todo  $j$ , toda aresta de borda  $(u, v)$ , tal que  $u \in C_{z_j}$ , é colocada ou na heap  $H(j)$  ou na heap  $H(i, j)$ , onde  $0 \leq i < j$ . Toda aresta de borda se encontra em exatamente uma heap. As arestas presentes em  $H(i, j)$  são arestas cujo vértice  $v$  é vizinho de  $C_{z_i}$  mas não é vizinho a nenhum  $C_{z_l}$  tal que  $i < l < j$ . Se existem mais de uma aresta, sejam elas  $(u', v)$  e  $(u, v)$ , em que ambos  $u', u \in C_{z_j}$ , logo ambas são candidatas a mesma heap  $H(i, j)$ , então uma delas pode ser colocada na heap  $H(j)$ . Nota que  $i$  pode ser 0 o que quer dizer que  $v$  não é adjacente a nenhum  $C_{z_l}$  onde  $l < j$ . Todas as soft heaps utilizam o mesmo valor para  $\varepsilon$ .

Depois de uma retração em  $C_{z_k}$ , as heaps  $H(k)$  e  $H(k-1, k)$  são destruídas, as suas arestas corrompidas se tornam ruins e são descartadas. As arestas boas são separadas em conjuntos, um para cada vértice externo. Para cada conjunto é selecionada a aresta de menor custo e as demais são descartadas. Esse descarte é possível porque estamos tratando de arestas que incidem em um mesmo vértice externo e em um subgrafo que é contraível, no máximo uma de cada conjunto vai estar na AGM. As arestas que sobraram são colocadas nas heaps adequadas, note que as arestas de borda de  $C_{z_k}$  agora são de  $C_{z_{k-1}}$ . Para cada aresta  $(r, s)$ , se ela vem de  $H(k)$  e existe uma aresta em  $H(k-1, k)$  incidente em  $s$ , ou se ela vem de  $H(k-1, k)$  então já existe uma aresta em algum  $H(i, k-1)$  incidente em  $s$ , portanto essa pode ser inserida em  $H(k-1)$ . Caso contrário, essa aresta veio de  $H(k)$  então existe uma aresta em algum  $H(i, k)$  para  $i < k-1$  que incide em  $s$ , logo a aresta deve ser inserida em  $H(i, k)$ . Vale notar que nesse último caso, também poderíamos inserir a aresta em  $H(k-1)$  obedecendo a invariante 2. Mas queremos adiar o reprocessamento dessas arestas, isto é, quando  $C_{z_{k-1}}$  for contraído, essas arestas terão que ser inseridas novamente em outra heap e isso seria caro demais. Ao colocá-las em  $H(i, k)$ , só vamos nos preocupar com essa aresta novamente mais tarde. E por fim, toda heap  $H(i, k)$  para  $i < k-1$  é unida com a heap  $H(i, k-1)$ .

Após uma extensão novas heaps são criadas. Seja  $(u, v)$  a aresta de extensão, o novo subgrafo  $C_{z_k}$  criado terá como arestas de borda, as arestas vizinhas de  $v$  que não incidem em nenhum outro  $C_{z_i}$ . Essas arestas são separadas em conjuntos, um para cada vértice externo, e só ficamos

com a menor de cada conjunto. Cada aresta é inserida no seu  $H(i, j)$  apropriado. A atualização das heaps na fusão é feita da mesma forma do que na retração, só tomando o cuidado que aqui ocorre a contração de vários subgrafos, mas é fácil adaptar para essa diferença. Outro detalhe da extensão, é que para determinar a menor aresta, devemos consultar todas as heaps e escolher a menor aresta de todas as heaps.

Essa organização das heaps servem para dois propósitos. As heaps da forma  $H(i)$  servem para controlar a quantidade de arestas ruins, somente elas já seriam o suficiente para este propósito. O problema que surge é que na retração, temos que destruir uma heap, e nesse caso a quantidade de arestas na heap final pode se tornar muito grande. O número de arestas que seriam processadas a cada seria muito grande. A vantagem das heaps  $H(i, j)$  é que elas não precisam ser inteiramente processadas quando o  $C_{z_j}$  está sendo contraído, uma simples união é o suficiente para manter a organização desejada. Basicamente, quando uma aresta entra em algum  $H(i)$  é porque haverá descarte dessa ou de alguma outra aresta presente em  $H(i-1, i)$  quando o  $C_{z_i}$  for contraído. Assim temos certeza que o número de arestas que serão levadas para cima é pequeno.

### 4.6.3 Quantidade de corrupção

O resultado disso é que o número de inserções em heaps é no máximo  $4m$  durante a construção de  $T$ . Para provar isso, considere que cada aresta possui quatro créditos. Para cada vez que essa aresta é inserida em uma heap, ela perde um crédito, quando ela for descartada os seus créditos podem ser transferidos para outra aresta. Vamos mostrar que para qualquer  $j$ , toda aresta de  $H(j)$  tem pelo menos dois créditos e para todo  $i, j$  e todo vértice  $v$  fora da cadeia, as  $q$  arestas de  $H(i, j)$  somam pelo menos  $q + 2$  créditos. As arestas são inseridas pela primeira vez em alguma heap em uma extensão. Como não mantemos múltiplas arestas no grafo, as arestas começam em algum  $H(i, j)$  com três créditos. Agora considere a aresta  $(r, s)$  que é reinserida na retração. Se esta aresta foi inserida em  $H(k-1)$ , então as arestas incidentes em  $s$  que estavam em  $H(k-1, k)$  liberaram pelo menos três créditos. Um deles cobre a reinserção dessa aresta e os outros dois cobrem os créditos extra para  $H(k-1)$  pela aresta extra. As demais arestas incidentes em  $s$  que estavam em  $H(k-1, k)$  foram descartadas. Caso contrário, se a aresta foi inserida em  $H(i, k)$ , serão liberados pelo menos dois créditos porque essa aresta veio de  $H(k)$ , um crédito cobre a reinserção e o outro cobre o crédito extra na heap  $H(i, j)$ , esta já possui pelo menos um vértice incidente em  $s$  então só é necessário um crédito extra. A mesma análise pode ser aplicada para o processo de fusão.

Então seja o parâmetro da soft heap  $\varepsilon = 1/c$  onde  $c$  é uma constante grande o suficiente.

Então a quantidade de arestas corrompidas será  $m/c$  mais o número de remoções da menor aresta das heaps, o que ocorre  $n - 1$  vezes. Lembrando que a garantia da quantidade de chaves corrompidas só leva em conta as arestas que estão na heap, então para cada aresta corrompida que for removida, uma nova aresta poderá ser corrompida. Isso nos leva a um total de  $m/c + n - 1$  arestas corrompidas. Portanto o número de arestas ruins será menor que  $m/c + n$ .

#### 4.6.4 Complexidade da construção da partição

O tempo para construir a partição do grafo depende principalmente das operações da heap. Na extensão, é preciso consultar todas as heaps para determinar a aresta de menor custo. Sendo  $d$  a altura máxima que a hierarquia pode atingir, o número de heaps existentes em qualquer momento será  $O(d^2)$ . Toda extensão traz um vértice nova para a partição, então ocorrerá exatamente  $n - 1$  extensões. Com cuidado necessário a garantia das invariantes pode ser feita com uma complexidade que não excede o total das operações da heap. Então as extensões custarão ao todo  $O(d^2n)$ . Nas retrações, o limitante é o número de inserções feitas nas heaps e como foi visto anteriormente esse valor não excede  $4m$ . Logo a construção da partição levará tempo  $O(m + d^2n)$ .

Por fim, resta decidir o tamanho de cada  $C_z$  da partição e a altura de  $T$ . Vimos que quanto maior a altura mais custoso se torna a construção da partição, então queremos manter a altura baixa. Uma escolha razoável para  $d$  é  $c \lceil \sqrt[3]{m/n} \rceil$  em que  $c$  é uma constante grande o suficiente. Assim o tempo de construção da partição se torna linear. O tamanho dos  $C_z$  influencia o tempo gasto nas recursões. Quanto menores melhor, mas se forem pequenos demais não será possível garantir a altura de  $T$ . Para isso usaremos uma função semelhante a função de Ackermann, a diferença é pequena e só serve para encaixá-la melhor no algoritmo.

$$S(i, j) = \begin{cases} 2j & \text{se } i = 1 \\ 2 & \text{se } j = 1 \\ S(i, j - 1)S(i - 1, S(i, j - 1)) & \text{se } i, j > 1 \end{cases}$$

Seja  $C_z$  a altura de  $z$  em  $T$ , sendo que as folhas possuem altura 0. Uma boa escolha para o grau  $n_z$  de  $z$  é

$$n_z = \begin{cases} S(t, 1)^3 & \text{se } d_z = 1 \\ S(t - 1, S(t, d_z - 1)) & \text{se } d_z > 1. \end{cases}$$

A variável fica como parâmetro do algoritmo, inicialmente será escolhido como sendo o

menor valor possível para obedecer a altura de  $T$  desejada, isto é,  $t$  é o menor inteiro positivo tal que  $n \leq S(t, d)^3$ . Agora vale a pena retomarmos os passos do algoritmo que foram apresentados anteriormente com as modificações necessárias para o parâmetro  $t$ . O algoritmo recebe dois parâmetros, um grafo e o valor de  $t$ .

1. Se  $t = 1$  ou se  $n = O(1)$ , compute  $AGM(G)$  diretamente.
2. Execute  $c$  iterações do algoritmo de Boruvka.
3. Construa a hierarquia  $T$  e o conjunto  $B$  de arestas ruins.
4. Compute  $T' \leftarrow \bigcup_{z \in T} AGM(C_z B, t - 1)$
5. Devolva  $AGM(T' \cup B, t)$

#### 4.6.5 Complexidade do algoritmo todo

Agora vamos mostrar, por indução em  $t$  e  $n$ , que executar  $AGM(G, t)$  leva tempo no máximo  $bt(m + d^3(n - 1))$ , em que  $b$  é uma constante grande o suficiente. Para o caso base em que  $t = 1$ , podemos computar a árvore em  $O(n^2)$ , mas  $N \leq S(1, d)^3 = 8d^3$ , então  $O(n^2) = O(d^3 n) \leq b(m + d^3(n - 1))$ . Ou se  $n = O(1)$ , podemos computar em  $O(m) \leq b(m + d^3(n - 1))$ .

No passo 2, as iterações do Boruvka levam tempo  $O(n + m)$  e transformam o grafo  $G$  em  $G_0$  com  $n_0 \leq n/2^c$  vértices e  $m_0 \leq m$  arestas. O passo 3 leva tempo  $O(m + d^3 n)$  como visto anteriormente. No passo 4 temos a recursão do algoritmo. Vamos desconsiderar a ocorrência das fusões por enquanto, ela será tratada separadamente em seguida. Dizemos que um nó  $z$  está completo quando este atinge a quantidade desejada de filhos. A razão pela qual um nó não se torna completo pode ser simplesmente porque o algoritmo terminou, isto é, não há mais nenhum vértice para fazer extensão, ou por razão de uma fusão, mas vamos deixar a fusão para depois. Agora seja  $N_z$  a quantidade total de vértices do grafo original que aparecem no nó  $z$ , isto é, a quantidade de folhas que estão abaixo de este nó. Se todos os nós abaixo de  $z$  estão completos, o valor de  $N_z$  é  $S(t, d_z)^3$ . Esse resultado pode ser verificado por indução seguindo a seguinte identidade

$$S(t, d_z - 1)^3 n_z = S(t, d_z - 1)^3 S(t - 1, S(t, d_z - 1))^3 = S(t, d_z)^3.$$

A quantidade de nós incompletos pode ser no máximo um, afinal o algoritmo acaba apenas uma vez. Então sabemos que  $N_z \geq (n_z - 1)S(t, d_z - 1)^3$ , para todo  $d_z > 1$ . Isso é equivalente

a deixar de contar um dos filhos. Aplicando a hipótese da indução do tempo de execução de  $AGM(C_z B, t - 1)$  temos

$$b(t - 1)(m_z + S(t, d_z - 1)^3(n_z - 1)) \leq b(t - 1)(m_z + N_z),$$

onde  $m_z$  é número de arestas em  $C_z B$ . Se contarmos os nós que foram podados pela fusão separadamente, esse limite também se aplica. Fazendo soma para todos os  $C_z$  temos

$$\sum_z b(t - 1)(m_z + N_z) = b(t - 1)(\sum_z m_z + \sum_z N_z).$$

Nenhuma aresta pode aparecer em mais de um  $C_z$  logo  $\sum_z m_z \leq m_0 - |B|$ . Cada vértice só pode ser contado no máximo em  $d$ 's  $N_z$  diferentes então  $\sum_z N_z \leq dn_0$ . Com isso temos que

$$b(t - 1)(\sum_z m_z + \sum_z N_z) \leq b(t - 1)(m_0 - |B| + dn_0).$$

E por fim, aplicando a hipótese para a recursão do passo 5, o seu tempo é no máximo  $bt(n_0 - 1 + |B| + d^3(n_0 - 1))$ . Somando o tempo de todos os passos e levando em conta a quantidade de corrupção concluímos que o tempo total é  $bt(m + d^3(n - 1))$  o que prova a indução.

Pela escolha do valor de  $d$  temos que  $d^3n = O(m)$  portanto a complexidade do algoritmo pode ser expressa como  $O(tm)$ . E  $t$  se comporta como um inverso da função  $S$  que cresce mais rápido que a função de Ackermann, mas não muito mais rápido, então é razoável assumir que  $t$  é limitado pela seguinte inversa da função de Ackermann

$$\alpha(m, n) = \min_{i \geq 1} : A(i, 4 \lceil m/n \rceil) > \log n.$$

Para provar essa afirmativa vamos colocar a definição da função de Ackermann.

$$A(i, j) = \begin{cases} 2j & \text{se } i = 0 \\ 0 & \text{se } j = 0 \text{ e } i \geq 1 \\ A(i - 1, S(i, j - 1)) & \text{se } i \geq 1 \text{ e } j \geq 2 \end{cases}$$

Para todo  $i \geq 1$  e  $j \geq 4$ ,

$$A(3i, j) = A(3i - 1, A(3i, j - 1)) > 2^{A(3i, j - 2)} = 2^{A(3i - 1, A(3i, j - 2))}$$

Como  $A$  é sempre crescente, como  $A(3i, j-2) \geq j$ , concluímos que  $A(3i, j) \geq 2^{A(i,j)}$ . Pode ser mostrado facilmente por indução que para qualquer  $u \geq 2, v \geq 3, A(u, v) \geq 2^{v+1}$ , então

$$A(3i, j) = A(3i-1, A(3i, j-1)) \geq A(3i-1, 2^j) \geq A(i, 2^j).$$

Pela definição, pode ser verificado facilmente que  $A(u-1, v) \leq S(u, v)$ , para qualquer  $u, v \geq 1$ , o que implica em  $S(9\alpha(m, n) + 1, d) \geq A(9\alpha(m, n), d)$ . E portanto, para  $d \geq 4$ ,

$$S(9\alpha(m, n) + 1, d) > 2^{A(\alpha(m, n), 2^d)} \geq 2^{A(\alpha(m, n), 4^{\lceil m/n \rceil})} > n,$$

e então o menor  $t$  tal que  $n \leq S(t, d)^3$  satisfaz  $t \leq 9\alpha(m, n) + 1$ .

## 5 *Conclusão*

Foi estudado com detalhe o algoritmo que computa a árvore geradora mínima e que faz uso da teoria da discrepância. Essa idéia se mostrou bastante eficaz nesse problema e em outros como o fecho convexo [1]. Isso mostra que o resultado obtido com esse algoritmo vai além do problema da árvore geradora mínima. E certamente essa técnica poderá ser aplicada em outros problemas que ainda estão em aberto.

## *Referências Bibliográficas*

- [1] Bernard Chazelle. An optimal convex hull algorithm and new results on cuttings (extended abstract). In *SFCS '91: Proceedings of the 32nd annual symposium on Foundations of computer science*, pages 29–38, Washington, DC, USA, 1991. IEEE Computer Society.
- [2] Bernard Chazelle. A minimum spanning tree algorithm with inverse-ackermann type complexity. *J. ACM*, 47(6):1028–1047, 2000.
- [3] Bernard Chazelle. The soft heap: an approximate priority queue with optimal error rate. *J. ACM*, 47(6):1012–1027, 2000.
- [4] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.
- [5] David R. Karger, Philip N. Klein, and Robert E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *J. ACM*, 42(2):321–328, 1995.
- [6] Valerie King. A simpler minimum spanning tree verification algorithm. *Algorithmica*, 18:440–448, 1997.
- [7] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7:48–50, 1956.
- [8] Nesetril, Milkova, and Nesetrilova. Otakar boruvka on minimum spanning tree problem: Translation of both the 1926 papers, comments, history. *DMATH: Discrete Mathematics*, 233, 2001.
- [9] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.