

Capítulo 4

Floresta Geradora Mínima Incremental

Neste capítulo, falaremos do problema da floresta geradora mínima incremental — *incremental minimum spanning forest*, em inglês. A solução deste problema é utilizada por [ANDRADE JÚNIOR e DUARTE SEABRA \(2020\)](#) para implementar uma versão semi-retroativa da floresta geradora mínima, que estudaremos no próximo capítulo.

4.1 Ideia

O problema da árvore geradora mínima consiste no seguinte: dado um grafo conexo G , com um peso associado a cada aresta, determinar uma árvore geradora de peso mínimo. Note que, este problema admite tanto pesos positivos quanto negativos nas arestas, mas ao admitir pesos negativos, é necessário exigir que o grafo em questão seja acíclico. Caso o grafo de entrada seja desconexo, buscamos uma floresta maximal de peso mínimo, que consiste em uma árvore geradora mínima de cada componente conexa do grafo.

 Algoritmos como o de [PRIM \(1957\)](#) ou o de [KRUSKAL \(1956\)](#) são famosos por resolver este problema de maneira eficiente, ambos com complexidade de $O(|E| \log |E|)$, onde E é o conjunto de arestas do grafo. 

Já na versão incremental do problema, inicialmente sabemos apenas o número de vértices do grafo, que começa sem nenhuma aresta. Em seguida, uma a uma das arestas são inseridas, com um dado peso. Devemos, sempre que for requisitado, fornecer eficientemente uma floresta maximal de peso mínimo do grafo corrente.

 Uma solução ingênua para esta versão seria acionar o algoritmo de Prim ou o de Kruskal a cada consulta. Porém, essa alternativa seria muita cara, pois ela não considera que entre uma consulta e outra o grafo pode ter mudado muito pouco. A ideia então é utilizar a versão apresentada por [FREDERICKSON \(1985\)](#) para mantermos informações sobre  o grafo, de modo a conseguirmos responder às consultas de maneira eficiente. 

Desta forma, a estrutura de dados que vamos apresentar da  suporta à seguinte interface:

- `add_edge(u, v, w)`: adiciona no grafo a aresta com pontas em u e v com peso w .
- `get_msf()`: retorna a lista com todas as arestas que compõem uma floresta maximal de peso mínimo do grafo corrente.
- `get_msf_weight()`: retorna o custo de uma floresta maximal de peso mínimo do grafo corrente.

A partir destes métodos, é possível construir um grafo de maneira incremental, isto é, adicionando aresta por aresta, com o advento de termos sempre em mãos a sua respectiva floresta geradora mínima. Em particular, a rotina `add_edge` consumirá tempo ~~proporcional~~ ~~O(log n)~~ amortizado por operação — onde n é o número de vértices do grafo, a rotina `get_msf` consumirá tempo ~~proporcional a~~ ~~O(log n)~~ e `get_msf_weight` será executada em tempo constante.

4.2 Estrutura interna

Assim como no union-find retroativo, vamos utilizar as *link-cut trees* como estrutura interna da solução deste problema. Para isso, queremos que as *link-cut trees* sejam utilizadas para manter uma floresta maximal de peso mínimo do grafo corrente, de modo que, ao adicionarmos uma nova aresta, com peso w e pontas em u e v , ao grafo, podemos usar as rotinas `is_connected(u, v)` e `maximum_edge(u, v)` para decidir se incluímos ou não a aresta à floresta maximal de peso mínimo.

Um detalhe importante é que, para essa implementação, necessitamos de uma maneira de consultar qual a aresta com maior peso no caminho entre dois vértices ~~na~~^{nova} árvore, não apenas o seu peso. Para isso, modificamos a implementação das *link-cut trees* para incluir um novo parâmetro opcional `id` na rotina `link`, além de um novo método `maximum_edge_id`, que retorna o `id` de uma aresta de peso máximo no caminho entre dois vértices. Este `id` será definido por nossa estrutura, e a partir dele, utilizando um mapa `edges_by_id`, conseguimos recuperar em quais vértices tal aresta incide.

Finalmente, mantemos uma lista `current_msf` de `id`'s das arestas que compõem uma floresta maximal de peso mínimo, assim como um inteiro `current_msf_weight`, que armazena o seu custo. Estes atributos nos permitem responder de maneira eficiente às consultas, como mostraremos a seguir.

4.3 Consultas Get MSF e Get MST Weight

Primeiramente, para realizarmos a consulta acerca da composição de uma floresta maximal de peso mínimo, simplesmente percorremos a lista dos `id`'s das arestas que compõem a floresta armazenadas nas *link-cut trees* e criamos uma nova lista com as arestas em si, utilizando o mapeamento fornecido pelo `edges_by_id`.

Já a consulta sobre o custo de uma floresta maximal de peso mínimo pode ser facilmente respondida retornando o inteiro `current_msf_weight`, mantido pela rotina `add_edge`.

Capítulo 5

Floresta Geradora Mínima Semi-Retroativa

Neste capítulo, descreveremos uma versão aprimorada da solução apresentada por ANDRADE JÚNIOR e DUARTE SEABRA (2020) para o problema da floresta geradora mínima retroativa — *retroactive minimum spanning forest*, em inglês. Esta versão utiliza a técnica de *square-root decomposition* junto com a estrutura do Capítulo 4 para solucionar o problema.

5.1 Nomenclatura

Tendo em vista as definições acerca de estruturas de dados parcialmente e totalmente retroativas, mostradas no Capítulo 1, decidimos nos referir tanto à solução de Andrade Júnior e Duarte Seabra quanto à versão aqui apresentada como *semi-retroativas*. Isto se dá pelo fato de ambas suportarem inserções e consultas em qualquer instante de tempo, porém nenhuma delas suporta a remoção de uma operação, o que as exclui de qualquer uma das duas definições.

Eu entendo que isso como totalmente retroativo.

Vale notar que HENZINGER e WU (2019) apresentam ainda outra definição: a de *estrutura de dados incremental totalmente retroativa*. Neste contexto, uma operação retroativa pode criar ou cancelar uma inserção, fazendo com que, no segundo caso, a operação cancelada seja totalmente removida da estrutura de dados. Entretanto, não se pode criar uma remoção, isto é, não existe suporte para que uma operação seja cancelada somente após um certo instante de tempo. Portanto, apesar de um nome sugestivo, essa definição também não contempla a solução que vamos estudar neste capítulo.

Acho que tá na hora de escrever o Capítulo 1. Inclusive pode ser que lá possamos incluir essa discussão.

5.2 Square-root decomposition

a solução não-retroativa do Capítulo 4,

Inicialmente, vamos conhecer a técnica de *square-root decomposition*, utilizada para transformar soluções que consomem tempo $O(\log n)$ — onde n é o número de elementos no problema em questão — em soluções com custo $O(\sqrt{n})$. Para nossa explicação, vamos

Para ganhar o quê?

retroativas?

por operações

Este consumo de tempo não bate com o do exemplo a seguir.

utilizar o seguinte problema como exemplo: dado uma lista de inteiros $[a_1, a_2, a_3, \dots, a_n]$, queremos conseguir efetuar as duas operações a seguir:

- `find_sum(l, r)`: determina a soma de todos os valores no intervalo $[l, r]$;
- `update_value(i, x)`: atualiza para x o valor do elemento na posição i .

Este problema possui duas soluções *ingênuas*, cada uma favorecendo uma das operações. A primeira, e mais simples, consiste em utilizar um *loop* para responder consultas `find_sum`, o que acaba custando $O(n)$, e apenas atualizando a respectiva posição para a operação `update_value`, o que consome tempo $O(1)$.

Já a segunda solução se resume a utilizarmos um vetor de soma de prefixos — isto é, um vetor tal que `prefix_sum[i]` equivale a $\sum_{j=1}^i a_j$ — para respondermos as consultas `find_sum` em tempo constante, porém, acarretando na reconstrução de `prefix_sum` em toda chamada de `update_value`, o que consome $O(n)$.

$$n = db$$

Todavia, utilizando a *square-root decomposition*, podemos responder consultas do primeiro tipo em tempo $O(\sqrt{n})$ e executar rotinas do segundo tipo em tempo constante, um bom meio termo. O cerne desta técnica consiste em duas etapas. Primeiramente, dividimos a estrutura de interesse — neste caso, a lista de inteiros — em d blocos de tamanho b . Sem perda de generalidade, assumimos que n , o tamanho da lista, é um múltiplo de b , com $d = \frac{n}{b}$. Em seguida, para cada um dos blocos, pré-calculamos alguma informação auxiliar. No problema utilizado como exemplo, isso se traduz em pré-calcular a soma de todos os elementos dentro de um bloco.

Com isso, podemos explicar como adaptamos as operações para funcionarem utilizando esta divisão em blocos. Apesar de estarmos focados em resolver o problema de soma em um intervalo, a *receita* por trás dessa adaptação pode ser facilmente utilizada em outros contextos, como veremos na próxima seção.

Para respondermos consultas do tipo `find_sum(l, r)`, utilizaremos o pré-cálculo realizado nos blocos para eliminar a necessidade de percorrer todos os elementos no intervalo entre l e r . Primeiramente, iteramos sob todos os blocos completamente contidos no intervalo e acumulamos a respectiva soma em uma variável y . Com isso em mãos, podemos nos concentrar para calcular a soma x e z das *pontas* do intervalo, isto é, os pedaços que fazem parte de um bloco não totalmente contido no intervalo, utilizando um simples *loop*. Esta tarefa está representada na Figura 5.1 e podemos perceber que a resposta para a consulta é simplesmente a soma $x + y + z$.

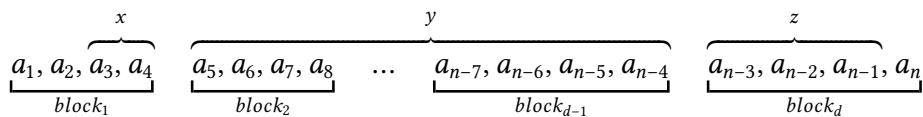


Figura 5.1: Divisão de uma lista de tamanho n em d blocos de tamanho b , mostrando que a soma de x , y e z responde à consulta feita por `find_sum(3, n-1)`.

Já a rotina `update_value(i, x)` é um pouco mais simples. Ao atualizamos o valor da posição i , temos simplesmente que atualizar o valor pré-calculado do bloco que o contém,

e isso é o suficiente.

Note que a segunda operação tem um custo constante, dado que apenas atualizamos um único valor, porém a primeira operação requer uma análise mais cuidadosa. Para encontrar os valores das *pontas*, x e z , somos obrigados a realizar um *loop* sob estes elementos, e como no pior caso podemos acabar percorrendo $b - 1$ elementos, esta etapa tem custo $O(b)$. Já para encontrar y , iteramos sob os blocos em si, portanto, no pior caso, gastamos $O(d)$ para o seu cálculo. Com isso, temos que a consulta `find_sum` tem um custo final $O(\max(b, d))$.

Com o intuito de maximizarmos a eficiência desta função, queremos encontrar um tamanho de bloco b ótimo que minimize o valor de $\max(b, d)$, isto é, que tornem b e d ~~mais~~ próximos possível. Para isso, podemos fazer:

~~tão~~ ^{quanto}

$$b = d \Rightarrow b = \frac{n}{b} \Rightarrow b^2 = n \Rightarrow b = \pm\sqrt{n} \quad (5.1)$$

Portanto, \sqrt{n} é o tamanho ótimo para um bloco, o que implica que a nossa lista será dividida em \sqrt{n} blocos, daí o nome da técnica. Finalmente, temos agora que a consulta `find_sum` consome tempo $O(\sqrt{n})$, com update-value consumindo $O(1)$.

5.3 Rotinas extras para a versão incremental

Antes de seguirmos adiante com a explicação ~~versão do problema~~, temos que apresentar duas funções extras adicionadas na nossa solução para a versão incremental do problema. Em particular, ambas possuem o mesmo objetivo: possibilitar consultas acerca da floresta maximal de peso mínimo após a adição de um conjunto de arestas sem que tais modificações persistam na estrutura original. Em outras palavras, elas simulam o que poderia ser consultado caso fizéssemos estas adições de arestas em uma cópia da estrutura, porém, sem o custo adicional que tal cópia implica.

Essas rotinas são as `get_msf_after_operations(edges[])` e `get_msf_weight_after_operations(edges[])`, que recebem uma lista de arestas e retornam, respectivamente, as arestas que fazem parte de uma floresta maximal de peso mínimo e seu peso caso as arestas da lista fornecida fossem adicionadas ao grafo. Desta forma, a execução destes métodos consiste em três etapas: adição das arestas na estrutura; realização da consulta que estamos interessados; reversão da estrutura para o seu estado inicial.

Para a realização da primeira etapa, criamos o método `apply_add_edge_operations`, que recebe uma lista de arestas, e realiza a adição delas na estrutura, de maneira muito similar ao que acontece na rotina `add_edge`. Entretanto, este método retorna uma lista de pares `{operação, aresta}`, indicando quais operações foram realizadas na *link-cut tree* — `link` ou `cut` — assim como as arestas envolvidas em cada uma delas. Como este método é muito semelhante à rotina `add_edge`, não mostraremos seu pseudo-código.

Logo, após realizarmos as consultas que estamos interessados, precisamos reverter as operações realizadas na *link-cut tree*. Para isso, criamos o método `apply_rollback`, que recebe a lista criada pela rotina acima e desfaz as operações. Note que, para mantermos a

consistência da *link-cut tree* durante este processo, precisamos percorrer esta lista de trás para frente, revertendo uma operação de cada vez.

Programa 5.1 Rotina Apply Rollback

```

1: function APPLY_ROLLBACK(operations_list[])
2:   revert(operations_list)
3:   for each (operation, edge) in operations_list do
4:     if operation = link then
5:       linkCutTree.cut(edge.u, edge.v)
6:       current_msf.erase(edge.id)
7:       current_msf_weight -= edge.w
8:     else
9:       linkCutTree.link(edge.u, edge.v, edge.w, edge.id)
10:      current_msf.append(edge.id)
11:      current_msf_weight += edge.w
12:    end while
13:   end for
14: end function

```

Finalmente, com estes métodos em mãos, podemos implementar as rotinas extras que estávamos interessados. Vamos mostrar somente o pseudo-código da rotina *get_msf_after_operations*, dado que a única diferença entre as duas implementações seria a chamada na terceira linha.

Programa 5.2 Rotina Get MSF After Operations

```

1: function GET_MSF_AFTER_OPERATIONS(edges[])
2:   rollback_operations ← apply_add_edge_operations(edges)
3:   msf ← get_msf()
4:   apply_rollback(rollback_operations)
5:   return msf
6: end function

```

Além disso, podemos perceber que a complexidade destes métodos é ~~proporcional à~~ $O(q \log m)$, onde q é o número de arestas na lista *edges*[].

quem é m mesmo?

5.4 Ideia

Agora, com todas as peças necessárias em mãos, podemos partir para a explicação da solução. Assim como no Capítulo 4, estamos interessados em resolver o *problema da floresta geradora maximal de peso mínimo*, porém agora em sua versão semi-retroativa.

Em particular, queremos ser capazes de adicionar uma aresta ao grafo em certo ~~estante~~ ^{im} de tempo, assim como realizar consultas acerca da floresta geradora maximal de peso mínimo em algum momento do presente ou do passado. Para isso, a estrutura deve conseguir dar suporte a seguinte interface:

- `add_edge(u, v, w, t)`: adiciona no grafo a aresta com pontas ~~em~~ u e v ~~com~~² peso w ~~no instante t~~ ,
- `get_msf(t)`: retorna a lista com todas as arestas que compõem uma floresta maximal de peso mínimo do grafo no instante t ;
- `get_msf_weight(t)`: retorna o custo de uma floresta maximal de peso mínimo do grafo no instante t .

A seguir, vamos apresentar duas soluções para este problema: a primeira delas é a versão original de Andrade Júnior e Duarte Seabra, que fornece os métodos acima com um custo de $O(\sqrt{m} \log n)$ por operação — onde m é o número de operações realizadas até o instante atual e n é o número de vértices do grafo.[?] Porém essa abordagem apresenta uma restrição em relação à quantidade de operações que podem ser realizadas na estrutura. Já a segunda solução corresponde a uma melhoria da versão original, onde eliminamos a restrição acerca da quantidade de operações, oferecendo um custo amortizado de $O(\sqrt{m} \log n)$ por operação.

5.4.1 Versão original

~~Na verdade a restrição delas é maior ainda, pois exigem que as m operações tenham instantes de 1 a m , ou seja, quando uma operação chega, já sobrem sua posição na lista final de operações. Não é?~~

Inicialmente, vamos pensar em como resolver este problema de uma maneira ingênua, isto é, sem usar as técnicas mais sofisticadas que vimos até agora. Para isso, podemos manter uma lista ordenada `edges_by_time`, onde cada posição corresponde a uma operação `add_edge(u, v, w, t)`, com a aresta (u, v, w) sendo armazenada como valor e t sendo usado como chave de ordenação para a lista. Assim, podemos responder às consultas da seguinte maneira: separamos todas as arestas inseridas até o instante de tempo t fornecido para a consulta e executamos, por exemplo, o algoritmo de Kruskal para determinar a floresta geradora maximal de custo mínimo. Dessa maneira, o consumo de tempo da rotina `add_edge` é $O(\log m)$ — sendo m o número de inserções realizadas, e o consumo de tempo das consultas `get_msf` e `get_msf_weight` é de $O(m \log m)$, pois no pior caso executamos o algoritmo de Kruskal para todas as arestas na lista.

Como podemos perceber, a solução acima gasta muito tempo construindo a resposta do zero para cada uma das consultas. Para melhorar isso, Andrade Júnior e Duarte Seabra sugerem uma maneira de acelerar esta etapa de construção de resposta, mas comprometendo um pouco o custo da rotina de inserção de novas arestas, através do uso da técnica de *square root decomposition*.

Para procedermos com a explicação, precisamos assumir duas coisas: que m é um inteiro conhecido de antemão, representando o número total de arestas a serem inseridas no grafo; e que ~~o instante de tempo fornecido a toda operação esteja~~ no intervalo $[1, m]$. Esses dois detalhes representam uma grande restrição para a versão original, e buscamos eliminá-los na versão melhorada da solução. Além disso, ~~sem perda de generalidade~~, vamos assumir que m é um quadrado perfeito.

Primeiramente, utilizando a ideia de *square root decomposition*, vamos dividir a lista `edges_by_time` em \sqrt{m} blocos. Além disso, definimos os *checkpoints* $c_1, c_2, \dots, c_{\sqrt{m}}$, que correspondem aos instantes de tempo no fim de cada bloco.[?] Assim temos que c_i corresponde ao instante de tempo $i\sqrt{m}$. Em seguida, atribuímos uma floresta geradora mínima

da operação?

incremental t_i a cada *checkpoint*, onde t_i é incrementada com todas as arestas inseridas em um instante de tempo menor ou igual a c_i .

Em outras palavras, podemos descrever esta construção da seguinte maneira: dividimos a lista de inserções em \sqrt{m} blocos de tamanho \sqrt{m} , onde cada bloco possui uma estrutura para resolver o problema da floresta geradora mínima incremental. Dessa forma, fazemos com que a estrutura em cada bloco possua todas as arestas inseridas desde o instante de tempo inicial até o instante de tempo máximo contido naquele bloco

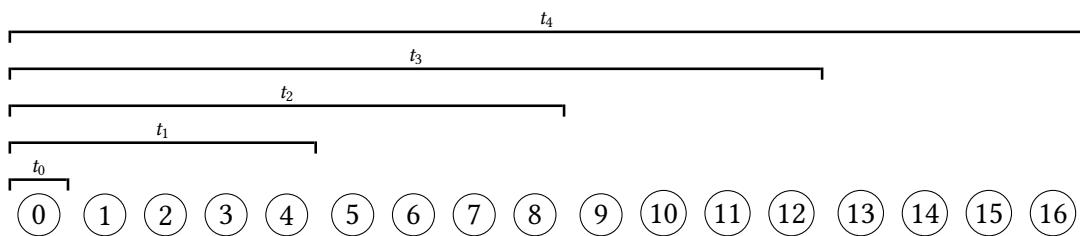


Figura 5.2: Representação da lista `edges_by_time` com m igual a 16. Neste caso, cada bloco tem tamanho 4 e os instantes 0, 4, 8, 12 e 16 são c_0, c_1, c_2, c_3 e c_4 , respectivamente. Assim, por exemplo, a estrutura t_3 contém todas as arestas adicionadas desde o instante 1 até o instante 12.

A partir dessa construção, podemos responder uma consulta acerca do estado da floresta geradora maximal de peso mínimo no instante de tempo t utilizando a seguinte abordagem:

- para começar, precisamos encontrar o último bloco da decomposição que não possui a aresta adicionada no instante de tempo t , ou o bloco que a possui em seu extremo. Mais formalmente, isso corresponde a encontrarmos o maior i tal que $c_i \leq t$ ~~e~~; —
- em seguida, vamos *aumentar* este bloco até que ele possua ~~o~~ todas as operações de inserção até o instante t , isto é, com a estrutura t_i em mãos, incrementamos ~~o~~ com todas as arestas adicionadas entre os instantes $c_i + 1$ e t ; —
- finalmente, basta retornar a consulta de interesse, isto é, as arestas que compõem a floresta geradora ou o seu respectivo peso. —

Vale notar que, caso a consulta aconteça no primeiro bloco, não existe uma estrutura inicial a qual podemos utilizar para incrementar a resposta final. Por isso, vamos definir o *checkpoint* $c_0 = 0$ e sua respectiva estrutura t_0 , uma floresta geradora mínima incremental de um grafo vazio. *Mo já queceu na Fig 5.2. Deve ser feito antes ou removido de Fig 5.2.*

Agora, para executarmos uma rotina `add_edge(u, v, w, t)` fazemos o seguinte:

- inicialmente, encontramos o primeiro bloco em que a aresta adicionada no instante t deve ser passada para a estrutura incremental, ou seja, o menor i tal que $t \leq c_i$ é verdade;
- por último, basta adicionarmos esta aresta nas estruturas de cada bloco daqui para frente, o que se traduz em realizarmos uma operação de `add_edge(u, v, w)` em todas as t_j , com $j \in [i, \sqrt{m}]$.

Finalmente, podemos analisar a complexidade desta solução, onde m é o número de operações realizadas e n é o número de vértices do grafo.

Para as consultas, podemos perceber que o primeiro passo tem custo $O(\sqrt{m})$, dado que temos que percorrer todos os blocos até encontrarmos o i de interesse. Já o segundo passo implica um custo amortizado de $O(\sqrt{m} \log n)$, dado que, no pior caso, teremos que adicionar quase todas as arestas de um bloco na estrutura incremental. Assim, a consulta `get_msf` fica com um custo amortizado de $O(m + \sqrt{m} \log n) = O(m)$ e a consulta `get_msf_weight` fica com custo amortizado de $O(1 + \sqrt{m} \log n) = O(\sqrt{m} \log n)$.

Além disso, na rotina `add_edge`, podemos ter que adicionar a aresta na estrutura incremental de quase todos os blocos da decomposição, com isso, seu custo amortizado também será de $O(\sqrt{m} \log n)$.

5.4.2 Versão melhorada

Como podemos ver acima, a versão original de Andrade Júnior e Duarte Seabra oferece uma solução para o problema que estamos interessados em resolver, porém, as restrições que a acompanham acabam se tornando um grande inconveniente. Logo, ao refletirmos sobre maneiras de melhorar a ideia apresentada, rapidamente notamos que a construção inicial da decomposição acaba se tornando um limitante para a estrutura.

Em particular, a construção realizada na versão original se baseia no artigo proponente da ideia de estruturas de dados retroativas, de Demaine, Iacono et al., que mostra uma receita para transformar estruturas parcialmente retroativas em estruturas totalmente retroativas — traduzindo para o nosso caso, uma maneira para transformar a floresta geradora mínima incremental em uma retroativa. Entretanto, na abordagem sugerida pelo artigo, são realizadas diversas reconstruções da decomposição, conforme novas arestas vão sendo adicionadas. Mais especificamente, os autores sugerem uma reconstrução a cada $\frac{\sqrt{m}}{2}$ operações, de modo a garantir que nenhum bloco tenha tamanho maior que $\frac{3\sqrt{m}}{2}$.

Além disso, assumindo que a reconstrução possui um custo de $O(m \log n)$, podemos distribuir este gasto de maneira amortizada em cada operação, fazendo com que o custo amortizado de cada uma continue sendo $O(\sqrt{m} \log n)$. Todavia, para que a reconstrução tenha este custo, é necessário que a estrutura parcialmente retroativa tenha uma versão persistente, de modo a podermos utilizar uma única cópia para representar $t_0, t_1, \dots, t_{\sqrt{m}}$. Por sua vez, uma versão persistente da floresta geradora mínima incremental requer a implementação de uma *link-cut tree* persistente, como a apresentada por DEMAINE, LANGERMAN et al. (2008). Porém, essa implementação é bastante sofisticada, e seu estudo fugiria do escopo deste trabalho.

Logo, estaremos interessados em resolver este problema utilizando uma versão não persistente da floresta geradora mínima incremental, criando uma cópia da estrutura para cada t_i . Desse modo, durante uma reconstrução, as operações do último bloco serão inseridas em $t_{\sqrt{m}}$, as operações do penúltimo bloco serão inseridas em $t_{\sqrt{m}-1}$ e $t_{\sqrt{m}}$, e assim por diante. Portanto, podemos calcular o custo dessa reconstrução da seguinte maneira:

$$\sum_{i=1}^{\sqrt{m}} (i\sqrt{m} \log n) = \frac{m \log n (\sqrt{m} + 1)}{2} \Rightarrow O(m \log n \sqrt{m}). \quad (5.2)$$

Ou seja, com o custo da reconstrução apresentado acima, cada operação teria agora um custo amortizado de $O(m \log n)$, o que está longe do ideal. Neste ponto, nossa abordagem para solucionar o problema fica bastante criativa, com a apresentação de uma maneira totalmente nova de realizar tal tarefa.

O principal ponto a ser notado é que, entre uma reconstrução e outra, gastamos muito tempo para reconstruir t_i a partir do zero, e que talvez exista uma maneira de aproveitar as estruturas da decomposição anterior durante a reconstrução da nova. Para facilitar as coisas, vamos diferenciar a notação relativa à nova decomposição em relação à antiga. Deste modo, definimos m^* e $\sqrt{m^*}$ como o número de operações e o tamanho dos blocos na nova versão, além de $c_0^*, c_1^*, \dots, c_{\sqrt{m^*}}^*$ e $t_0^*, t_1^*, \dots, t_{\sqrt{m^*}}^*$ como as novas listas de *checkpoints* e florestas geradoras mínimas incrementais.

Assim, nossa versão melhorada funciona da seguinte maneira:

- primeiramente, uma reconstrução vai ser realizada toda vez que m for um quadrado perfeito, fazendo com que $\sqrt{m^*}$ seja igual a $\sqrt{m} + 1$;
- em cada reconstrução, faremos com que t_0^* e t_1^* sejam uma floresta geradora incremental de um grafo vazio; além disso, definimos $t_i^* = t_{i-2}$, para $i \in [2, \sqrt{m^*}]$;
- por último, deslocamos cada t_i^* para o seu respectivo c_i^* , isto é, incluímos todas as arestas adicionadas desde o instante c_{i-2} até o instante c_i — caso $i - 2$ seja menor que 0 incluímos todas as arestas desde o início da lista de operações.

A partir dessa versão, a reconstrução consome o mesmo tempo da reconstrução sugerida por Demaine, Iacono et al., porém agora sem a necessidade de uma estrutura persistente. Além disso, o funcionamento das outras rotinas continuam iguais ao da versão original.

Na próxima seção, provaremos que $c_i^ \geq c_{i-2}$, de modo que o último passo resulta numa versão correta da t_i^* .*

5.4.3 Complexidade

Antes de adentrarmos nos detalhes de como são as implementações dos métodos da nossa estrutura, vamos analisar a complexidade da versão aqui proposta. Primeiramente, precisamos obter um resultado relativamente simples, porém muito útil para nossa análise: o número de operações realizadas entre duas reconstruções.

Corolário 5.1. Seja m , a quantidade atual de operações realizadas, um quadrado perfeito. Então o número de operações a serem realizadas até a próxima reconstrução é de $2\sqrt{m} + 1$.

Demonstração. Como sabemos, uma nova reconstrução acontece quando m for um quadrado perfeito. Desse modo, na próxima reconstrução teremos um novo m^* igual a $(\sqrt{m} + 1)^2$.

Corolário é um resultado que é uma consequência relativamente simples de um teorema anterior.

Logo, podemos calcular a diferença entre estes dois valores ~~como~~:

$$\begin{aligned} m^* - m &= (\sqrt{m} + 1)^2 - m \\ &= (m + 2\sqrt{m} + 1) - m \\ &= 2\sqrt{m} + 1 \end{aligned}$$

Portanto, temos que $2\sqrt{m} + 1$ operações serão realizadas entre duas reconstruções.

Acho que é melhor separar em dois resultados. Um tam a ver com a correta da estratégia: mostram que $c_i \geq c_{i-2}$. Outro, que todos as arestas que estão em t_{i-2} devem estar em t_i . O outro tam a ver com o consumo de tempo: quantas arestas temos que adicionas no maximo.

Agora, precisamos descobrir qual o número máximo de posições que cada uma das t_i^* pode ser deslocada, ou seja, o número máximo de arestas que podem existir entre um instante c_{i-2} e c_i^* , para $i \in [2, \sqrt{m}]$.

Teorema 5.1. Durante uma reconstrução, a transformação de t_{i-2} em t_i^* sempre consiste na adição de novas arestas, e no máximo $3\sqrt{m}^*$ arestas são adicionadas a cada t_i^* .

Demonstração. Primeiramente, temos que cuidar de t_0^* e t_1^* , criadas durante a fase inicial da reconstrução. Como $c_0^* = 0$, nenhuma aresta é adicionada em t_0^* . Já para t_1^* , é necessário que todas as \sqrt{m}^* arestas até c_1^* sejam adicionadas, porém isso ainda se encontra dentro do custo apresentado. Agora vamos cuidar do restante das t_i^* .

de operações que define

Vamos chamar p_i a posição de c_i na lista edges_by_time durante a última reconstrução, ~~ou seja,~~ em particular, sabemos que $p_i = i\sqrt{m}$ para $i \in [0, \sqrt{m}]$. Analogamente, definimos $p_i^* = i\sqrt{m}^*$, para $i \in [0, \sqrt{m}]$.

determinar

Estamos interessados em descobrir o intervalo que teremos que deslocar t_i^* , buscando entender qual o número mínimo de arestas a serem adicionadas — o que é atingindo quando todas as *operações* são inseridas antes de c_i — e também o respectivo número máximo — o que acontece quando todas as *operações* são inseridas após c_i .

No primeiro caso, para $i \in [0, \sqrt{m}]$, temos:

$$\begin{aligned} p_i &= i\sqrt{m} \\ &< i\sqrt{m} + 2\sqrt{m} + 1 \\ &= (i+2)\sqrt{m} \\ &< (i+2)\sqrt{m}^* \\ &= p_{i+2}^* \end{aligned}$$

Portanto, a transformação de t_{i-2} em t_i^* funciona, pois, o deslocamento de c_i para c_{i+2}^* sempre adiciona mais arestas a estrutura.

Por último, podemos medir o deslocamento no pior caso, onde nenhuma aresta ~~será~~ adicionada ~~no~~ *a* *nova é* *bloco* ou *em* algum de seus antecessores. Assim, para $i \in [0, \sqrt{m}]$,

④ Acho que isso pode ser o algoritmo 5.2. Isso prova que $c_i^* \geq c_{i-2}$, ou seja, que todos os arestas de t_{i-2} devem estar em t_i^* .
 Disso é fácil concluir também que o número de arestas a serem adicionadas a t_i^* para obtermos $t_i^* < 3\sqrt{m^*}$, logo $O(\sqrt{m^*})$.

temos:

$$\begin{aligned} p_i^* - p_{i-2} &= i\sqrt{m^*} - (i-2)\sqrt{m} \\ &= i(\sqrt{m} + 1) - (i-2)\sqrt{m} \\ &= i\sqrt{m} + i - i\sqrt{m} + 2\sqrt{m} \\ &= i + 2\sqrt{m} \\ &\leq \sqrt{m} + 2\sqrt{m} \\ &< 3\sqrt{m^*} \end{aligned}$$

Acho que fui eu que falei para colocar em amortizado aqui, por causa dos splay trees, mas tá estranho. Acho que só vamos usar a análise dos splay trees para uma sequência de operações, e não amortizado por operação

Para cada i , esse é o número de adições de arestas que precisaremos fazer sempre.

□

Finalmente, devido ao deslocamento que será realizado em cada t_i^* e utilizando os resultados acima, podemos ver que o custo total amortizado de uma reconstrução é de $O(m^* \log n)$ — onde n é o número de vértices do grafo. Além disso, podemos amortizar este custo em cada uma das $2\sqrt{m} + 1$ operações que levaram a essa reconstrução, assim o custo amortizado em cada uma delas é de $O(\sqrt{m^*} \log n)$.

por inserção de aresta

no que?

5.5 Rotina Build Decomposition

5.6 Consultas Get MSF e Get MST Weight

5.7 Rotina Add Edge