Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Link-cut trees e aplicações em estruturas de dados retroativas

Felipe Castro de Noronha

Monografia Final

mac 499 — Trabalho de Formatura Supervisionado

Supervisora: Profª. Drª. Cristina Gomes Fernandes

São Paulo
2022

Agradecimentos

Eu sou quem sou porque estou aqui.
— Paul Atreides

Primeiramente, gostaria de agradecer a professora Cris por todo o tempo dedicado em me orientar na escrita deste trabalho, assim como por suas minuciosas revisões, que aumentaram bastante a qualidade deste texto. Além disso, menciono aqui os professores Carlinhos e Coelho, que inspiraram e trouxeram meu interesse para a área de teoria.

Gostaria também de agradecer a meus colegas de curso, em especial aos amigos do BCCombi, que foram uma ótima companhia durante a graduação e agora fazem parte da minha vida. Ademais, agradeço a todos os membros do MaratonUSP, o grupo de extensão que contribuiu tanto para minha formação quanto o restante das matérias.

Por último, mas não menos importante, agradeço a meus pais, por todos os sacrifícios realizados e pelo apoio fornecido para que eu pudesse chegar até aqui.

Resumo

Felipe Castro de Noronha. **Link-cut trees e aplicações em estruturas de dados retroativas**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2022.

Estruturas de dados retroativas permitem a realização de operações que afetam não somente o estado atual da estrutura, mas também qualquer um de seus estados passados. Além disso, uma link-cut tree é uma estrutura de dados que permite a manutenção de uma floresta de árvores enraizadas com peso nas arestas, e onde os nós de cada árvore possuem um número arbitrário de filhos. Tal estrutura é muito utilizada como base para o desenvolvimento de estruturas de dados retroativas, e neste trabalho estudaremos as versões retroativas dos problemas de union-find e floresta geradora mínima. Para isso, implementamos essas estruturas em `C++` e descrevemos as ideias por trás de seus funcionamentos. Ademais, apresentamos uma melhoria da solução originalmente apresentada para a floresta geradora mínima retroativa, que retira limitações sem piorar sua performance.

Palavras-chave: link-cut tree. estrutura de dados retroativa. union-find. floresta geradora mínima.

Abstract

Felipe Castro de Noronha. **Link-cut trees and applications on retroactive data structures**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2022.

Retroactive data structures allow operations to be performed not only in the current state of the structure, but also in any of its past states. Moreover, a link-cut tree is a data structure that allows the maintenance of a forest of rooted trees with weighted edges, and where the nodes of each tree have an arbitrary number of children. Such structure is widely used as a basis for the development of retroactive data structures, and in this work we will study the retroactive versions of the union-find and minimum spanning forest problems. To do so, we implement these structures in `C++` and describe the ideas on how they work. Furthermore, we present an improvement of a solution originally presented for the retroactive minimum spanning forest, which removes limitations without worsening its performance.

Keywords: link-cut tree. retroactive data structure. union-find. minimum spanning forest.

Lista de Programas

Sumário

Capítulo 1

Introdução

De acordo com Cormen *et al.* (2009), *estruturas de dados* são uma maneira de guardar e organizar dados de forma a facilitar o acesso e modificação destes. Além disso, cada estrutura de dados serve a um proposito, ou seja, um contexto em que sua aplicação faz mais sentido.

Em geral, estamos preocupados em fazer com que a estrutura represente e modifique os dados sempre em um único estado, o presente. Porém, em muitos casos, essa premissa faz com que modificações e consultas sejam difíceis de serem aplicados quando queremos realizá-las no passado.

Por exemplo, uma das estruturas de dados mais simples, a *pilha*, é capaz de realizar duas rotinas básicas: `empilha` e `desempilha`, junto com a consulta `topo`. Caso tenhamos uma sequência de operações: `empilha(1)`, `empilha(2)`, `empilha(3)` e `desempilha()`, a consulta `topo()` deve retornar 2. Entretanto, caso esqueçamos de realizar a operação `empilha(2)`, nossa consulta agora retornaria 1, e a única maneira de corrigir o nosso erro seria desfazer todas as operações feitas após `empilha(1)`, e então refazê-las corretamente.

Visando solucionar este tipo de problema, Demaine, Iacono *et al.* (2007) apresentaram a ideia de *estruturas de dados retroativas*. Com elas, cada operação possui um instante de tempo associado, o que permite que elas sejam realizadas em qualquer momento. Em outras palavras, podemos agora realizar operações em qualquer estado passado da estrutura. Além disso, é possível remover uma operação que aconteceu em um certo instante de tempo, fazendo com que seus efeitos desapareçam da estrutura.

Em particular, temos duas categorias de retroatividade: a *retroatividade parcial*, que permite apenas que modificações ou remoções sejam realizadas no passado, enquanto todas as consultas devem ser realizadas no estado presente da estrutura; e a *retroatividade total*, que, além das modificações e remoções, permite também a realização de consultas em qualquer instante de tempo.

Neste trabalho, realizaremos um estudo das versões retroativas de dois problemas bastante famosos em ciência da computação: o *union-find* e a *floresta geradora mínima*. Para isso, será necessária a apresentação de uma estrutura de dados chamada *link-cut tree*, que servirá como base para as soluções de ambos os problemas citados acima.

Vale notar que a retroatividade não é a única alternativa para permitir que estruturas de dados lidem com o tempo de maneira diferente. Apresentada por Sarnak (1986), as *estruturas de dados persistentes* criam uma nova versão da estrutura toda vez que uma modificação é realizada, fazendo com que seja possível realizar consultas em qualquer uma das versões da estrutura. Ademais, assim como na retroatividade, temos duas categorias de persistência: a *persistência parcial*, que permite modificações apenas na versão atual da estrutura; e a *persistência total*, onde qualquer versão, tanto no passado quanto no presente, possa ser modificada.

Capítulo 2

Link-Cut Trees

Neste capítulo, apresentaremos as *link-cut trees*, introduzida por Sleator e Tarjan (1981). Esta estrutura de dados serve como base para as estruturas retroativas apresentadas nos próximos capítulos.

2.1 Ideia

As *link-cut trees* são uma estrutura de dados que nos permite manter uma floresta de árvores enraizadas com peso nas arestas, onde os vértices de cada árvore possuem um número arbitrário de filhos. Ademais, a floresta armazenada por essa estrutura não é orientada — isto é, suas arestas não possuem uma direção — e devido à maneira que ela é usada para nas implementações a seguir, sua raiz é constantemente redefinida, de modo que perdemos o arranjo original das árvores. Com isso, essa estrutura nos fornece o seguinte conjunto de operações:

- `make_root(u)`: enraíza no vértice u a árvore que o contém.
- `link(u, v, w)`: dado que os vértices u e v estão em árvores separadas, transforma v em raiz de sua árvore e o liga como filho de u, colocando peso w na nova aresta criada.
- `cut(u, v)`: retira da floresta a aresta com pontas em u e v, quebrando a árvore que continha estes vértices em duas novas árvores.
- `is_connected(u, v)`: retorna `verdadeiro` caso u e v pertençam à mesma árvore, `falso` caso contrário.

Por último, as *link-cut trees* possuem a capacidade de realizar operações agregadas nos vértices, isto é, consultas acerca de propriedades de uma sub-árvore ou de um caminho entre dois vértices. Em particular, estamos interessados na rotina `maximum_edge(u, v)`, que nos informa o peso máximo de uma aresta no caminho entre os vértices u e v.

Todas essas operações consomem tempo $O(\log n)$ amortizado, onde n é o número de vértices na floresta.

2.2 Definições

Primeiramente, precisamos apresentar algumas definições acerca da estrutura que vamos estudar.

Chamamos de *árvores representadas* as componentes da floresta armazenada nas *link-cut trees*. Para a representação que as *link-cut trees* utilizam internamente, dividimos uma árvore representada em caminhos vértice-disjuntos, os chamados *caminhos preferidos*. Todo caminho preferido vai de um vértice a um ancestral deste vértice na árvore representada. Por conveniência, definimos o início de um caminho preferido como o vértice mais profundo contido nele.

Se uma aresta faz parte de um caminho preferido, a chamamos de *aresta preferida*. Ademais, mantemos a propriedade de que um vértice pode ter no máximo uma aresta preferida com a outra ponta em algum de seus filhos. Caso tal aresta exista, ela liga um vértice a seu *filho preferido*.

Finalmente, para cada caminho preferido, elegemos um *vértice identificador*. A manutenção deste vértice será importante para a estrutura auxiliar que utilizaremos para manter os caminhos preferidos, dado que o vértice identificador de um caminho preferido será responsável por guardar um ponteiro para um vértice da árvore imediatamente acima do caminho preferido.

Ademais, para armazenar os pesos das arestas da floresta, a estrutura usada terá nós para vértices e para arestas da floresta. O nó correspondente à aresta uv tem o vértice u como seu pai e v como seu único filho, e armazena o peso de uv.

2.3 Operações

Nessa seção, apresentaremos o código por trás das operações que estamos interessados em implementar nas *link-cut trees*. Em um primeiro momento, assumiremos que já sabemos como implementar alguns métodos que lidam com os caminhos preferidos. Desta forma, a implementação dos métodos abaixo fica reservada para a próxima seção.

- `make_identifier(u)`: transforma um vértice u em identificador de seu caminho preferido.
- `split(u)`: separa o caminho preferido em que o vértice u é identificador em dois, quebrando a conexão entre u e seu filho preferido, caso exista. Ao final, tanto u quanto o seu filho preferido inicial serão os identificadores de seus caminhos preferidos.
- `join(u, v)`: recebe dois vértices, u e v — identificadores de seus caminhos e sendo v um filho de u na árvore representada — e concatena os respectivos caminhos preferidos, transformando uv em aresta preferida de u. Com isso, separa u da parte mais profunda de seu caminho preferido inicial, deixando o identificador de tal caminho com um ponteiro para u. Ao final da operação, u será o identificador do novo caminho criado.
- `reverse_path(u)`: recebe u, o identificador de um caminho preferido, e inverte a

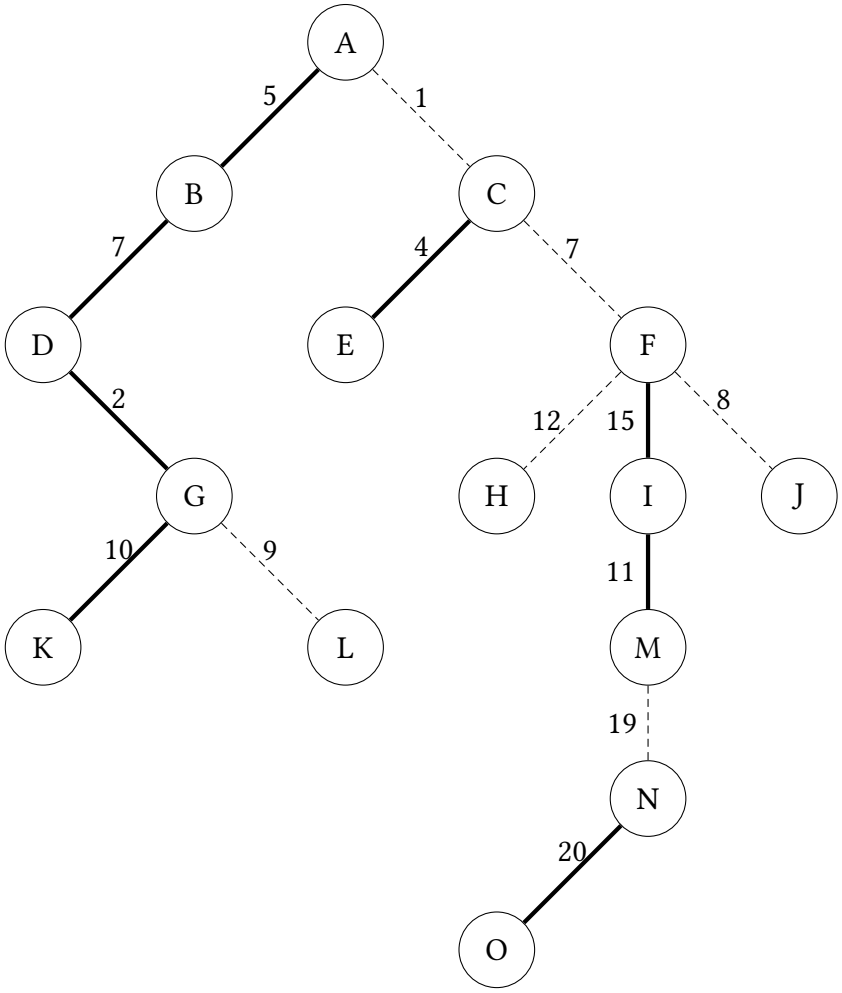


Figura 2.1: *Árvore representada e seus caminhos preferidos. Na figura acima, as arestas escuras representam caminhos preferidos, com isso, temos o seguinte conjunto de caminhos vértice-disjuntos* $\{\langle K, G, D, B, A\rangle, \langle E, C\rangle, \langle M, I, F\rangle, \langle L\rangle, \langle H\rangle, \langle J\rangle, \langle O, N\rangle\}$.

orientação desse caminho na árvore representada, isto é, o fim se transforma no começo e o começo no fim.

- `get_path_end_node(u)`: retorna o vértice menos profundo do caminho preferido de u, em outras palavras, o vértice no fim do caminho preferido que contém u. Na árvore da Figura 2.1, a chamada `get_path_end_node(G)` retorna o vértice `A`.
- `get_parent_path_node(u)`: retorna o vértice na floresta imediatamente acima do fim do caminho preferido que contém u; caso o caminho preferido de u contenha a raiz da árvore representada, este método retorna `null`. Na árvore da Figura 2.1, `get_parent_path_node(M)` retorna o vértice `C` e `get_parent_path_node(L)` retorna `G`.
- `get_maximum_path_value(u)`: recebe u, o identificador de um caminho preferido, e retorna o maior valor de uma aresta neste caminho.

Além disso, de modo a determinarmos a complexidade dos métodos das *link-cut trees*, temos que `split`, `join`, `reverse_path` e `get_maximum_path_value` consomem tempo

constante, enquanto `make_identifier`, `get_path_end_node` e `get_parent_path_node` gastam tempo a O(log n) amortizado, onde n é o número de vértices nos caminhos preferidos.

2.3.1 Rotina Access

Uma rotina utilizada por todos os métodos das *link-cut trees* que vamos implementar é a `access`. A partir dela conseguimos reorganizar a estrutura interna de uma árvore representada a nosso favor. Basicamente, a operação `access(u)` cria um caminho preferido que parte de u e vai até a raiz da árvore representada de u. Com isso, todas as arestas preferidas que tinham somente uma das pontas fazendo parte deste novo caminho deixam de ser preferidas e u termina sem nenhum filho preferido.

Para isso, começamos uma sequência de iterações, que vão crescendo um caminho preferido desde u até que tal caminho contemple a raiz da árvore representada. Inicialmente, separamos a parte superior do caminho preferido de u da parte inferior, fazendo com que este caminho comece pelo vértice u.

Com isso, a rotina agora realiza uma série de iterações, mantendo a seguinte invariante: u é identificador de um caminho preferido que começa nele e vai até um dos filhos do vértice `above_path`. Com essa invariante, conseguimos concatenar o caminho que começa em u com o caminho preferido imediatamente acima dele, o qual é representado pelo vértice `above_path`. Desta maneira, sabemos que atingimos a raiz da árvore quando o caminho imediatamente acima é vazio, ou seja, `above_path` é nulo.

Programa 2.1 Rotina Access

```
 1: function ACCESS(u)
 2:     make_identifier(u)
 3:     join(u, NULL)
 4:     above_path ← get_parent_path_node(u)
 5:     ▷ concatena todos os caminhos preferidos de u até a raiz da árvore representada
 6:     while above_path ≠ NULL do
 7:         make_identifier(above_path)
 8:         ▷ concatena a parte superior do caminho de above_path ao caminho de u
 9:         join(above_path, u)
10:         make_identifier(u)
11:         above_path ← get_parent_path_node(u)
12:     end while
13: end function
```

Logo, fica claro que o consumo de tempo dessa rotina é algo proporcional ao número de iterações realizadas vezes o custo das chamadas que operam sobre os caminhos preferidos. Em particular, Demaine, Holmgren *et al.* (2012) usam a técnica de *heavy-light decomposition* para mostrar que, em uma sequência de m operações `access`, são realizadas O(m log n) iterações. Ademais, eles mostram que as chamadas realizadas a cada iteração custam tempo O(1) amortizado. Portanto, a rotina `access` consome tempo O(log n) amortizado.

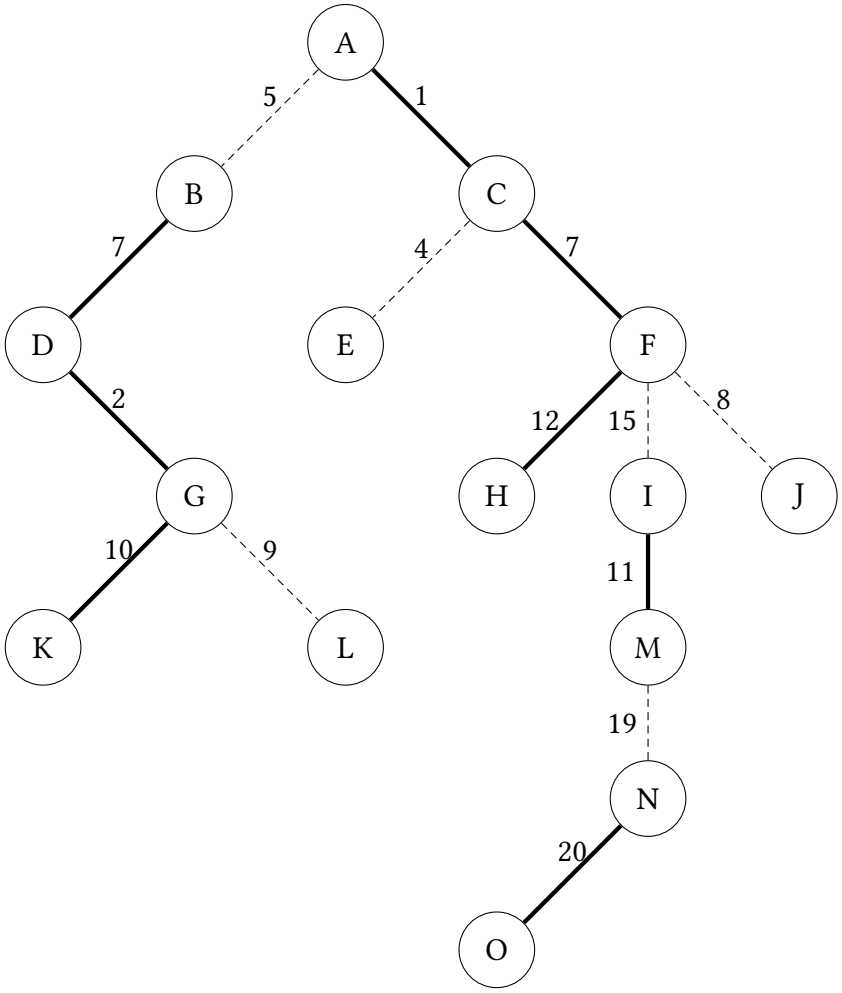


Figura 2.2: *Caminhos preferidos na árvore da Figura 2.1 após uma operação de* `access` *no vértice H. Com isso temos o novo conjunto de caminhos vértice-disjuntos* $\{\langle H, F, C, A\rangle, \langle K, G, D, B\rangle, \langle M, I\rangle, \langle E\rangle, \langle J\rangle, \langle L\rangle, \langle O, N\rangle\}$.

2.3.2 Rotinas Make Root, Link e Cut

Em seguida, temos a função `make_root(u)`, que enraíza em u a árvore representada que o contém. Para isso, criamos um caminho preferencial que vai de u até a raiz dessa árvore, utilizando `access(u)`. Em seguida, utilizamos a rotina `reverse_path(u)`, que inverte a orientação deste caminho preferido recém-criado. Tal inversão coloca u como o vértice de menor profundidade da sua árvore representada, o que se traduz neste sendo a nova raiz.

Programa 2.2 Rotina Make Root

```
1: function MAKE_ROOT(u)
2:     access(u)
3:     reverse_path(u)
4: end function
```

Como rotinas que dão nome à estrutura, temos `link(u, v, w)` e `cut(u, v)`.

A primeira delas recebe dois nós u e v que estão em árvores distintas, e cria uma aresta de peso w, conectando-os. Primeiramente, devemos lembrar que as arestas da floresta representada viram nós em nossa representação interna. Com isso, o primeiro passo é criar um nó que tem seu valor definido como w; vamos chamá-lo `uv_edge`. Dessa forma, criaremos as seguintes conexões: u torna-se o pai de `uv_edge` e `uv_edge` o pai de v.

Inicialmente, colocamos v como raiz de sua árvore representada, e criamos um caminho preferido que só possui este vértice como integrante. Com isso, conseguimos concatenar este caminho preferido de tamanho unitário com o caminho que `uv_edge` constitui. A seguir, aplicamos a mesma ideia, criando um caminho unitário que contém u em sua árvore e o concatenamos com um caminho que possui v e `uv_edge`.

Programa 2.3 Rotina Link

```
Require: u e v em árvores distintas
 1: function LINK(u, v, w)
 2:     uv_edge ← new Node(w) ▷ cria nó com peso w, representando a aresta
 3:     ▷ ligando (v) - (uv_edge)
 4:     make_root(v)
 5:     access(v)
 6:     join(v, uv_edge)
 7:     ▷ ligando (uv_edge)-(u)
 8:     make_root(u)
 9:     access(u)
10:     access(uv_edge)
11:     join(uv_edge, u)
12: end function
```

Já a operação `cut(u, v)`, que separa dois vértices ligados por uma aresta, é um pouco mais simples. Note que temos que separar as conexões entre u e `uv_edge`, assim como entre `uv_edge` e v. O processo de separação é igual para as duas partes, por isso vamos explicar somente a separação de u e `uv_edge`.

Programa 2.4 Rotina Cut

```
Require: u e v na mesma árvore e uv uma aresta da floresta representada
 1: function CUT(u, v)
 2:     ▷ cortando (u) - (uv_edge)
 3:     make_root(u)
 4:     access(uv_edge)
 5:     split(uv_edge)
 6:     ▷ cortando (v) - (uv_edge)
 7:     make_root(v)
 8:     access(uv_edge)
 9:     split(uv_edge)
10: end function
```

A ideia é colocarmos u como raiz de nossa árvore representada. Com isso, podemos criar um caminho preferido que vai de `uv_edge` até u . Agora, basta usarmos nossa opera-

ção `split(uv_edge)`, que separa `uv_edge` da parte superior de seu caminho preferido, efetivamente quebrando sua conexão com u.

2.3.3 Consultas Is Connected e Maximum Edge

A função `is_connected(u, v)`, que nos informa se u e v pertencem à mesma árvore, funciona da seguinte maneira. Primeiro acessamos u, criando um caminho deste até a raiz da sua árvore. Em seguida, guardamos o vértice que está no fim desse caminho, isto é, guardamos a raiz da árvore que contém u. A seguir, repetimos o mesmo processo com o vértice v. Agora, basta compararmos se ambos os valores que guardamos são iguais.

Programa 2.5 Consulta Is Connected

```
1: function IS_CONNECTED(u, v)
2:     access(u)
3:     u_tree_root ← get_path_end_node(u)
4:     access(v)
5:     v_tree_root ← get_path_end_node(v)
6:     return (u_tree_root = v_tree_root)
7: end function
```

Por último, temos a função `maximum_edge(u, v)`, que supõe que u e v estão na mesma árvore da floresta representada e retorna o peso da maior aresta no caminho simples entre u e v. Como transformamos as arestas em vértices na nossa representação interna, precisamos procurar o maior valor de um vértice no caminho preferido entre u e v. Para isso, transformamos v na raiz de nossa árvore e acessamos u. Com isso, podemos utilizar `get_maximum_path_value(u)` para obter o maior valor contido neste caminho preferido.

Programa 2.6 Consulta Maximum Edge

```
Require: u e v na mesma árvore
1: function MAXIMUM_EDGE(u, v)
2:     make_root(v)
3:     access(u)
4:     return get_maximum_path_value(u)
5: end function
```

Assim, encerramos a explicação da implementação dos métodos das *link-cut trees*. Além disso, podemos perceber que todos os métodos executam simplesmente chamadas para os modificadores de caminhos preferidos e para a rotina `access`. Portanto, temos que as funções apresentadas até agora consomem tempo amortizado $O(\log n)$.

2.4 Splay Trees

No artigo original, Sleator e Tarjan (1981) propuseram a utilização de uma árvore binária enviesada como estrutura para os caminhos preferidos. Porém, quatro anos depois, eles apresentaram a *splay tree* — Sleator e Tarjan (1985), que possibilita realizarmos as

operações necessárias para a manipulação dos caminhos preferidos em tempo $O(\log n)$ amortizado, onde n é o numero de nós da floresta representada, com uma implementação muito mais elegante do que a da versão original. Portanto, usaremos as splay trees para armazenar os caminhos preferidos.

Uma splay tree é uma árvore binária de busca auto-balanceável. Estas árvores utilizam rotações para se auto-balancear, através de uma operação chamada `splay`. A operação `splay` traz um nó para a raiz da árvore através de sucessivas rotações. Mas antes de nos aprofundarmos neste método, examinaremos como os caminhos preferidos são representados aqui.

Primeiramente, em nosso uso, a ordenação dos nós na splay tree é dada pela profundidade destes nas link-cut trees. Note que, para garantir a eficiência, não guardamos explicitamente esses valores. Em vez disso, utilizamos a ideia de chave implícita, isto é, só nos preocupamos em manter a ordem relativa dos nós após as operações de separação e união das árvores, apresentadas a seguir.

Ademais, para implementar eficientemente a operação `get_maximum_path_value`, mantemos o peso máximo dos nós em cada sub-árvore de uma splay tree.

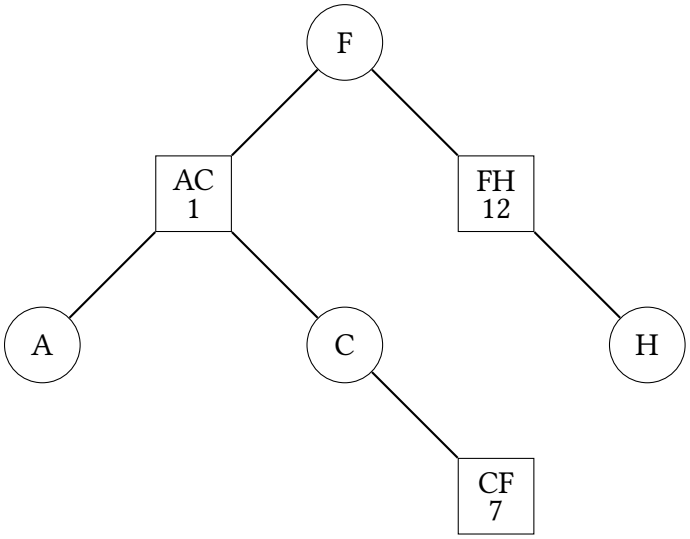


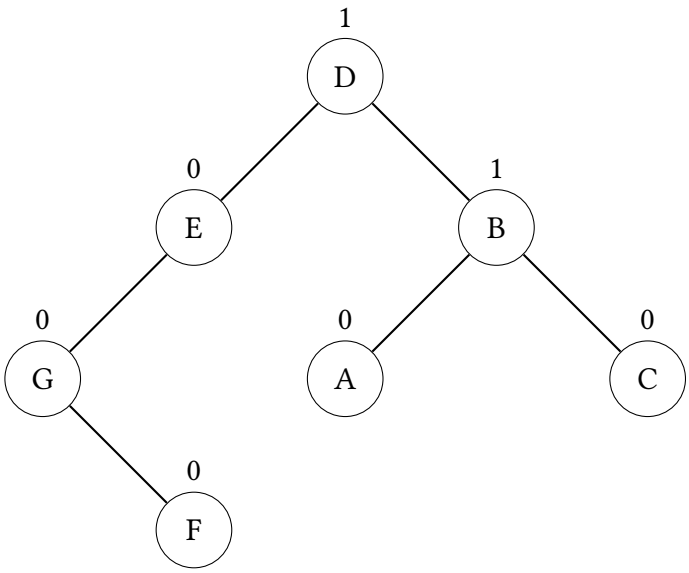
Figura 2.3: *Uma possível configuração da splay tree que armazena o caminho preferido $\langle H, F, C, A\rangle$ da Figura 2.2, onde F é o identificador do caminho. Os nós em formato retangular mostram as arestas da árvore representada, com o peso de tal aresta na parte inferior.*

Além disso, como usamos a profundidade dos nós na árvore representada como chave para a splay tree, temos que todos os nós na sub-árvore esquerda da raiz têm uma profundidade menor que a raiz, enquanto os nós à direita têm uma profundidade maior. Contudo, ao realizamos uma operação `make_root(u)`, fazemos com que todos os nós que estavam acima de u na árvore representada se tornem parte de sua sub-árvore. Para isso, incluímos na splay tree um mecanismo para inverter a ordem de todos os seus nós, efetivamente invertendo a orientação de um caminho preferido.

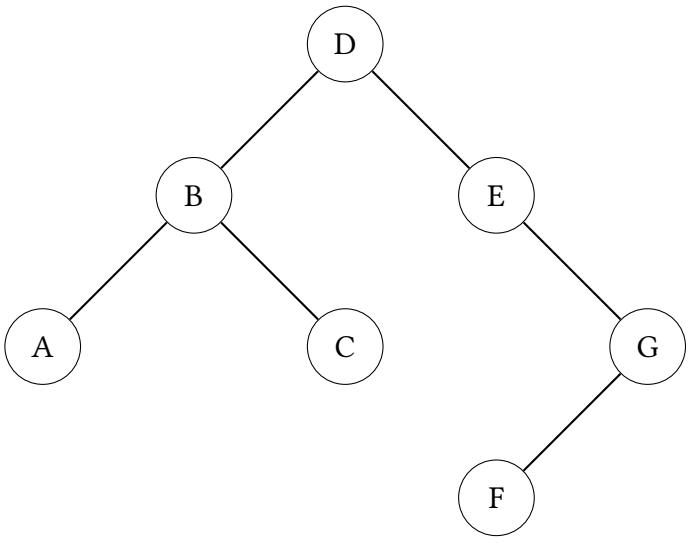
Com isso, os nós da splay tree têm os seguintes campos:

- `parent`: apontador para o pai na splay tree.

- `left_child` e `right_child`: apontadores para os filhos esquerdo e direito de um nó na splay tree.
- `value`: se o nó representa uma aresta da árvore representada guarda o peso desta aresta, senão guarda 0.
- `max_subtree_value`: guarda o valor máximo armazenado na sub-árvore do nó.
- `is_reversed`: valor booleano para sinalizar se a sub-árvore do nó está com sua ordem invertida ou não, isto é, se todas as posições de filhos esquerdos e direitos estão invertidas nessa sub-árvore. Seu funcionamento é ilustrado na Figura 2.4.



(a) *splay tree com valores* `is_reversed`



(b) *splay tree após a propagação de todos os valores* `is_reversed`

Figura 2.4: *A árvore em (a) mostra os valores do booleano* `is_reversed` *acima do rótulo de cada nó. Já a árvore em (b) mostra a splay tree resultante após todas as inversões serem resolvidas.*

Ademais, caso o nó seja a raiz da splay tree, o campo `parent` armazena um ponteiro

para o nó que está logo acima do fim deste caminho preferido na árvore representada. Ou seja, a raiz de uma splay tree aponta para um nó de outra splay tree, aquela que contém o vértice a que seu caminho preferencial se liga na sua árvore representada.

2.4.1 Rotina Splay

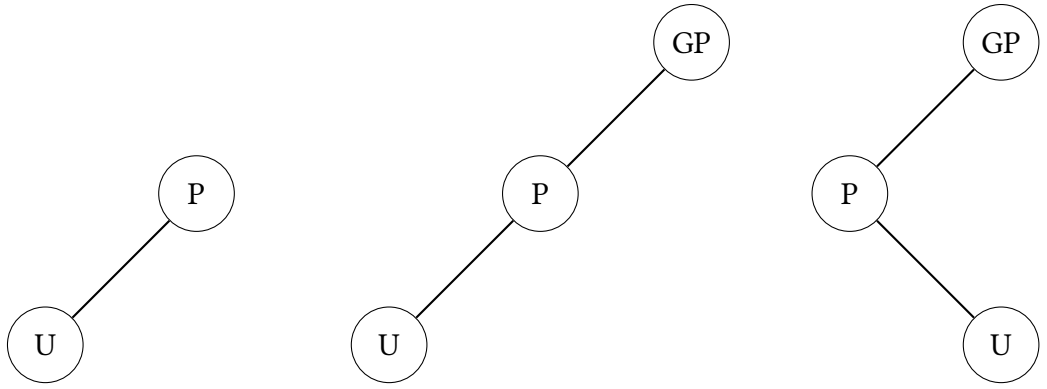
A rotina `splay` é o que garante o auto-balanceamento de uma splay tree. Como já mencionamos, seu efeito é trazer um dado nó para a raiz da árvore por meio de uma série de rotações. Para que o custo total de m operações em uma splay com n nós seja $O(m \log n)$, resultando num custo $O(\log n)$ amortizado por operação, a implementação deve garantir que o método `splay` seja sempre acionado no nó acessado mais profundo da splay tree, em toda operação. Ademais, as rotações que trazem o nó para a raiz da árvore devem seguir uma ordem particular, descrita através dos *passos de splay*, que aplicam rotações duplas ou unitárias, até que o nó em que estamos aplicando a operação chegue à raiz. Por último, devido ao bit que indica a inversão da sub-árvore de cada nó, temos que tomar alguns cuidados extras na nossa implementação dos passos de splay.

Em particular, podemos dizer que esta operação é responsável por transformar um nó em identificador de seu caminho, ou seja, entendemos como sinônimos os métodos `make_identifier` e `splay`.

De modo a facilitarmos nossa explicação detalhada, chamamos de `parent` o pai de um nó u e de `grandparent` o pai de `parent`. Como dissemos, uma operação de `splay` consiste em realizamos diversos *passos de splay*, que trazem u cada vez mais próximo à raiz da árvore, isto é, em cada um desses passos, realizamos uma ou duas rotações que diminuem a profundidade de u. Porém, ao realizar estes passos, temos que nos preocupar com dois fatores:

- A propagação do valor booleano *is_reversed* de `grandparent` e em seguida o de `parent`, fazendo as devidas reversões caso necessário. Isso nos fornece a invariante de que iremos fazer comparações entre os filhos corretos para determinar qual rotação fazer.

- A orientação em que `grandparent`, `parent` e u se encontram, isto é, se estão em uma orientação que Sleator e Tarjan chamaram de *zig*, *zig-zig* ou *zig-zag*, como exemplificadas na Figura 2.5. Dependendo da orientação, fazemos uma rotação em u ou em `parent`, sempre com a ideia de diminuirmos em 1 a profundidade de u.



(a) *zig* **(b)** *zig-zig* **(c)** *zig-zag*

Figura 2.5: *Orientações zig, zig-zig e zig-zag na splay tree. Aqui, `P` e `GP` abreviam `parent` e `grandparent`, respectivamente. As orientações zag, zag-zag e zag-zig são simétricas a estas.*

Ao sair da função `splay`, o nó *u* estará na raiz da splay tree que o contém. Além disso, seu valor booleano *is_reversed* estará nulo, pois as reversões já terão sido propagadas aos seus filhos, e seu *max_subtree_value* estará atualizado, contendo o maior valor presente na splay tree.

Programa 2.7 Rotina Splay

```
1:  function Splay(u)
2:      while not u.is_root() ▷ u não é raiz da Splay
3:          parent ← u.parent
4:          grandparent ← parent.parent
5:          if not parent.is_root() then
6:              ▷ caso o seja necessário, inverte os filhos do nó e propaga a inversão do bit
7:              grandparent.push_reversed_bit()
8:              parent.push_reversed_bit()
9:              if (grandparent.r_child = parent) = (parent.r_child = u) then
10:                 rotate(parent) ▷ zig-zig ou zag-zag
11:             else
12:                 rotate(u) ▷ zig-zag
13:             end if
14:         end if
15:         rotate(u)
16:     end while
17:     u.push_reversed_bit()
18: end function
```

Assim como a operação acima, o restante da nossa implementação de uma splay tree é bastante tradicional. Com isso, nossos únicos cuidados extras são a manutenção do bit `is_reversed`, do valor máximo das sub-árvores e da consistência das chaves implícitas. Por exemplo, no método `rotate(u)`, temos como primeiro passo a propagação do bit `is_reversed` de `grandparent` até *u* e como última etapa o cálculo dos novos valores de `max_subtree_value`.

2.4.2 Rotinas Split e Join

Temos também dois métodos importantes das splay trees usados na manutenção dos caminhos preferidos: `split` e `join`, responsáveis por separar e concatenar caminhos preferidos, respectivamente.

Primeiramente, falaremos do método `split(u)`, que recebe um nó u, identificador de seu caminho preferido, e separa o caminho preferido que o contém em dois: um com os nós menos profundos que u em seu caminho, e outro com u e os nós mais profundos que u. Para isso, o método simplesmente separa a sub-árvore esquerda de u. Vale notar que, este método é destrutivo, removendo tanto o ponteiro para o filho preferido de u quanto o ponteiro `parent` que tal filho possui para u. Logo, usamos essa rotina apenas para o `cut()` das *link-cut trees*.

Programa 2.8 Rotina Split

```
Require: u é identificador de seu caminho preferido
1: function SPLIT(u)
2:     if u.l_child ≠ NULL then
3:         u.l_child.parent ← NULL
4:     end if
5:     u.l_child ← NULL
6: end function
```

De maneira complementar, temos a rotina `join(u, v)`, que recebe dois nós e concatena a parte superior do caminho preferido de u ao caminho preferido de v. Para isso, assume-se que v seja filho de u na árvore representada que os contém, além de que ambos os nós sejam identificadores de seus caminhos preferidos, ou seja, que eles sejam as raízes de suas splay trees. Com isso, simplesmente colocamos a splay tree em que v é raiz como a sub-árvore direita de u, atualizando os respetivos apontadores e recalculando o valor máximo na splay tree de u.

Programa 2.9 Rotina Join

```
Require: u e v são identificadores de seus caminhos preferidos
1: function JOIN(u, v)
2:     if v ≠ NULL then
3:         v.parent ← u
4:     end if
5:     u.r_child ← v
6:     ▷ atualiza max_subtree_value com o máximo entre o value dos dois filhos de u
7:     u.recalculate_max_subtree_value()
8: end function
```

Note que o vértice u poderia ter uma sub-árvore direita, correspondendo à parte mais profunda de seu caminho preferido. Após a operação `join`, esse trecho de seu velho caminho preferido torna-se um novo caminho preferido que aponta — através do ponteiro `parent` — para o novo caminho preferido de u, que acabou de ser concatenado ao de v.

Como ambas as rotinas simplesmente leem e alteram variáveis, definimos seus respectivos custos como O(1).

2.4.3 Métodos auxiliares

Para finalizar, nossa splay tree possui quatro métodos auxiliares: o `reverse_path`, `get_path_end_node`, `get_parent_path_node` e `get_maximum_path_value`.

Primeiramente, o `reverse_path(u)` recebe o identificador u de um caminho e inverte a orientação desse caminho. Tal tarefa é realizada invertendo o valor do bit `is_reversed` de u. Com isso, nas próximas operações realizadas neste nó, seus filhos serão trocados de posição e o bit será propagado nas suas sub-árvores. Além disso, podemos perceber que seu custo é O(1).

Programa 2.10 Rotina Revese Path

```
Require: u identificador de seu caminho preferido
1: function REVESE_PATH(u)
2:     u.is_reversed ← not u.is_reversed
3: end function
```

A seguir, os métodos `get_path_end_node(u)` e `get_parent_path_node(u)` são usados para acessar o fim e o pai do caminho preferido que contém u. Em particular, a primeira rotina retorna o nó menos profundo do caminho preferido de u, fazendo isso ao acessar o nó mais à esquerda na sua splay tree. Já o segundo método é responsável por retornar o nó imediatamente acima do fim do caminho preferido que contém u. Caso tal caminho contenha a raiz da árvore representada, este método retorna `null`. Para fazer isso, efetuamos uma operação `splay` em u e retornamos o valor de seu `parent`.

Programa 2.11 Consulta Get Path End Node

```
1: function GET_PATH_END_NODE(u)
2:     splay(u)
3:     smallest_value ← u
4:     while smallest_value.l_child ≠ NULL do
5:         smallest_value ← smallest_value.l_child
6:     end while
7:     splay(smallest_value)
8:     return smallest_value
9: end function
```

Programa 2.12 Consulta Get Parent Path Node

```
1: function GET_PARENT_PATH_NODE(u)
2:     splay(u)
3:     return u.parent
4: end function
```

Como as duas rotinas acima utilizam o método `splay`, podemos afirma que os respectivos custos são de $O(\log n)$ amortizado.

Por último, temos a função `get_maximum_path_value(u)`, que recebe um nó u identificador de caminho e retorna o maior valor de uma aresta no caminho preferencial de u. Em termos práticos, retorna o valor de `max_subtree_value`. Claramente, temos que o custo de tal função e $O(1)$.

Programa 2.13 Consulta Get Maximum Path Value

```
Require: u identificador de seu caminho preferido
1: function GET_MAXIMUM_PATH_VALUE(u)
2:     return u.max_subtree_value
3: end function
```

Com isso, temos todas as ferramentas necessárias para manipularmos a splay tree em seu uso representando os caminhos preferidos nas *link-cut trees*.

Capítulo 3

Union-Find Retroativo

Neste capítulo falaremos do union-find retroativo, introduzido por Demaine, Iacono *et al.* (2007). Ele será a primeira estrutura retroativa que vamos implementar usando as *link-cut trees*.

3.1 Ideia

O union-find é uma estrutura de dados utilizada para manter uma coleção de conjuntos disjuntos, isto é, conjuntos que não se intersectam, sujeita a duas operações:

- `same_set(a, b)`: retorna `verdadeiro` caso a e b estejam no mesmo conjunto, `falso` caso contrário.
- `union(a, b)`: se a e b estão em conjuntos distintos, realiza a união destes conjuntos.

A primeira versão do union-find foi apresentada por Galler e Fisher (1964). Posteriormente, Tarjan e Leeuwen (1984) utilizaram a técnica de compressão de caminhos para mostrar uma implementação com complexidade $O(\alpha(n))$ amortizada por operação, onde n é o número total de elementos nos conjuntos que estamos representando e α é o inverso da função de Ackermann.

Como já dissemos, na versão retroativa, estamos interessados em realizar as operações em uma linha de tempo, isto é, conseguirmos adicionar ou remover operações do tipo `union` em certos instantes de tempo. Ademais, queremos conseguir checar se dois elementos pertencem a um mesmo conjunto num instante arbitrário.

Para isso, vamos trocar a operação `union(a, b)` da estrutura original por duas novas rotinas, `create_union(a, b, t)` e `delete_union(t)`. A primeira delas é responsável por adicionar uma união dos conjuntos que contém a e b no instante de tempo t, enquanto a segunda desfaz a união realizada em t. Além disso, colocamos um terceiro parâmetro t na operação `same_set`, para com isso conseguirmos consultar se dois elementos pertenciam ao mesmo conjunto neste dado instante t.

Por exemplo, a Figura 3.1 mostra o estado de uma coleção de conjuntos disjuntos após quatro operações serem aplicadas. Antes da operação `delete_union(3)`, as consul-

tas `same_set(a, b, 3)` e `same_set(c, d, 3)` retornam `verdadeiro`, por outro lado `same_set(a, d, 3)` e `same_set(c, d, 3)` retornam `falso` após a chamada da função `delete_union(3)`.

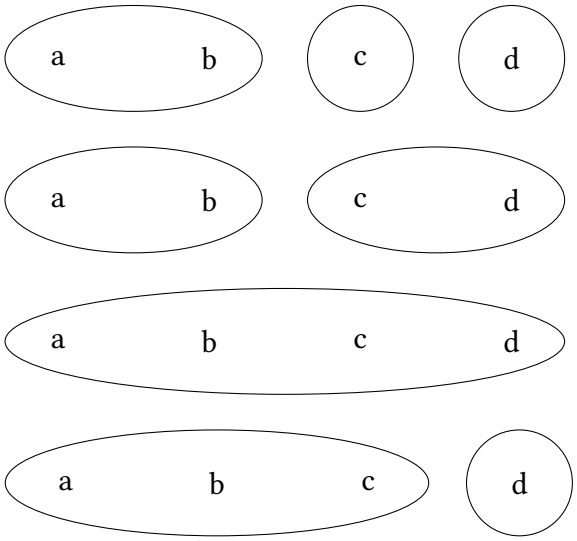


Figura 3.1: *Representação dos conjuntos com os elementos $\{a, b, c, d\}$ após a seguinte sequência de operações:* `create_union(a, b, 2)`, `create_union(c, d, 3)`, `create_union(b, c, 4)` *e* `delete_union(3)`. *Cada linha mostra o estado atual da coleção imediatamente após uma operação.*

Assumiremos que em nenhum momento podemos fazer uma operação que seria inválida em algum instante de tempo. Em outras palavras, não podemos remover uma união que não aconteceu, assim como não podemos criar uma união em dois elementos que já pertencem ao mesmo conjunto, sempre visando que a sequência de uniões possa ser traduzida em uma floresta.

Por exemplo, a sequência de operações `create_union(a, b, 1)`, `create_union(b, c, 2)`, `create_union(d, c, 3)` e `create_union(a, d, 5)` seria inválida, pois a última operação tenta criar uma união entre elementos que já pertencem ao mesmo conjunto. Entretanto, caso a operação `delete_union(1)` fosse adicionada imediatamente antes de `create_union(a, d, 5)`, a sequência se tornaria válida.

3.2 Estrutura interna

Para implementarmos o union-find retroativo, vamos utilizar as *link-cut trees* como estrutura interna. Para isso, fazemos com que os elementos dos conjuntos sejam vértices na floresta mantida pelas *link-cut trees* e que cada aresta das *link-cut trees* represente uma operação de `union`. Com isso, cada conjunto de nossa coleção será uma árvore na floresta. Note que, essa simples ideia já pode ser utilizada para implementar uma versão não retroativa do union-find, visto que a operação de `union` pode ser traduzida para uma chamada de `link`, assim como `same_set` para `is_connected`.

Para introduzirmos o caráter retroativo da estrutura, vamos utilizar o atributo `value` nas arestas das *link-cut trees*. Este campo será usado para guardar o tempo em que uma operação de `union` aconteceu, isto é, uma chamada `create_union(a, b, 3)` cria uma

aresta de valor 3 entre os vértices a e b da *link-cut tree*. Este valor poderá então ser utilizado para checar se dois elementos já pertenciam a um certo conjunto em um dado instante de tempo.

Ademais, como estamos simplesmente usando métodos já implementados pelas *link-cut trees*, basicamente sem nenhuma computação adicional, podemos perceber que o union-find retroativo tem uma complexidade amortizada de $O(\log n)$ por operação, tanto em consultas quanto em atualizações, onde n é o número total de elementos nos conjuntos da coleção.

A seguir, mostramos mais detalhadamente como essas operações são realizadas.

3.3 Consultas Same Set

Primeiramente, para checarmos se dois elementos a e b, no instante de tempo t, estão em um mesmo conjunto de nossa coleção, temos que conferir se eles estão na mesma árvore da floresta representada pelas *link-cut trees*. Para essa verificação inicial, podemos usar a consulta `is_connected`. Caso esta consulta retorne `verdadeiro`, prosseguimos para checar se eles já pertenciam ao mesmo conjunto no instante t.

Para isso, devemos lembrar que: cada aresta das *link-cut trees* representa uma operação de `union`; e que existe apenas um único caminho entre dois vértices quaisquer de uma árvore. Logo, todas as arestas que compõem o caminho entre os vértices que representam os elementos a e b se traduzem na sequência de uniões que resultaram no conjunto que contém estes vértices. Portanto, caso alguma dessas uniões tenha acontecido em um instante maior que t, os elementos a e b ainda não fariam parte do mesmo conjunto no tempo t. Finalmente, para realizar esta checagem, basta usarmos o método `maximum_edge` para obter o valor da maior aresta entre a e b, e com isso checar se a união mais recente aconteceu em um instante menor ou igual a t.

Programa 3.1 Consulta Same Set

```
1: function SAME_SET(a, b, t)
2:     if not linkCutTree.is_connected(a, b) then
3:         return false
4:     end if
5:     return linkCutTree.maximum_edge(a, b) ≤ t
6: end function
```

A partir do custo da chamada `linkCutTree.is_connected`, podemos perceber que o consumo de tempo desta consulta é $O(\log n)$ amortizado, onde n é o número de conjuntos em nossa coleção.

3.4 Rotinas Create Union e Delete Union

Por último, temos as rotinas de inserção e deleção de uniões. Aqui, as implementações são bem diretas, uma vez que essas operações se traduzem na inserção e deleção de uma

aresta nas *link-cut trees*, respectivamente. Com isso, temos apenas que nos preocupar com dois detalhes extras.

O primeiro deles é a transformação de elementos dos conjuntos em nossa coleção para vértices da *link-cut tree*. No pseudo-código abaixo, a função `create_node(x)` cria um vértice para o elemento x se e somente se ele ainda não possui um vértice correspondente na árvore. Essa rotina corresponde à rotina (make_set) da implementação tradicional do union-find. Ademais, para dar suporte à deleção de uma união criada em um instante t, precisamos criar um mapeamento que guarda o par de elementos unidos tendo como chave o instante em que a união ocorreu. No pseudo-código esse mapeamento é realizado pela estrutura `edges_by_time`, que, caso seja uma *hash table*, não muda a complexidade da rotina.

Programa 3.2 Rotina Create Union

```
1: function CREATE_UNION(a, b, t)
2:     linkCutTree.create_node(a)
3:     linkCutTree.create_node(b)
4:     linkCutTree.link(a,b,t)
5:     edges_by_time[t] ← (a, b)
6: end function
```

Programa 3.3 Rotina Delete Union

```
1: function DELETE_UNION(t)
2:     (u,v) ← edges_by_time[t]
3:     linkCutTree.cut(u,v)
4:     edges_by_time.erase(t)
5: end function
```

Utilizando o custo das chamadas `linkCutTree.link` e `linkCutTree.cut`, percebemos que o custo destas rotinas é $O(\log n)$ amortizado.

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. 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 $O(|E| \log |V|)$, onde V é o conjunto de vértices e E 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, as arestas são inseridas, cada uma 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 dá suporta à seguinte interface:

- `add_edge(u, v, w)`: adiciona no grafo a aresta com pontas em u e v, e 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 uma respectiva floresta maximal de peso mínimo. Em particular, a rotina `add_edge` consumirá tempo $O(\log n)$ amortizado por operação — onde n é o número de vértices do grafo, a rotina `get_msf` consumirá tempo $O(\log n)$ 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, possamos 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 numa 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 uma tabela de *hash* `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 MSF 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 pela tabela `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`.

Programa 4.1 Consulta Get MSF

```
1: function GET_MSF
2:     msf ← []
3:     for each id in current_msf do
4:         msf.append(edges_by_id[id])
5:     end for
6:     return msf
7: end function
```

Programa 4.2 Consulta Get MSF Weight

```
1: function GET_MSF_WEIGHT
2:     return current_msf_weight
3: end function
```

Com isso, a consulta `get_msf` tem um custo proporcional a O($min(m, n)$), onde n é o número de vértices e m é o número de arestas no grafo. Já a consulta `get_msf_weight` tem um custo O(1).

4.4 Rotina Add Edge

Como a parte mais importante da estrutura, a rotina `add_edge(u, v, w)` é responsável por adicionar uma nova aresta ao grafo, com extremos em u e v e peso w, possivelmente tendo que atualizar a floresta maximal de peso mínimo. Este processo pode ser divido em dois casos.

O primeiro deles ocorre quando u e v pertencem a componentes distintas do grafo. Neste caso, simplesmente adicionamos a aresta uv à floresta maximal de peso mínimo, ou seja, à floresta representada pelas *link-cut trees*.

O segundo caso ocorre quando u e v fazem parte da mesma componente do grafo. Neste caso, devemos decidir se essa aresta uv deve ou não substituir alguma aresta na árvore geradora mínima dessa componente. Note que, se adicionarmos essa nova aresta na árvore, criaremos um ciclo, que consiste em todas as arestas no caminho de u até v na árvore, mais a nova aresta uv. Ademais, a adição da aresta uv somente faz sentido caso ela diminua o custo total da árvore, em outras palavras, caso ela não seja a maior aresta deste ciclo. Dessa forma podemos simplesmente excluir a aresta com maior peso do ciclo, que pode ser uv ou alguma outra, preservando a estrutura de árvore e possivelmente contribuindo para uma diminuição de seu custo total. Note que, este ciclo nunca vai ser criado na estrutura, apenas consultamos o valor da maior aresta entre u e v, caso eles estejam conectados, e fazemos a devida comparação entre este valor e o da aresta uv.

Com isso, como esta rotina usa apenas os métodos fornecidos pelas *link-cut trees*, podemos concluir que ela consome tempo amortizado proporcional a O($\log n$), onde n é o número de vértices do grafo representado.

Programa 4.3 Rotina Add Edge

```
1:  function ADD_EDGE(u, v, w)
2:      edge_id ← create_unique_edge_id()
3:      edges_by_id[edge_id] ← new edge(u, v, w, edge_id)
4:      if not linkCutTree.is_connected(u, v) then
5:          linkCutTree.link(u, v, w, edge_id)
6:          current_msf.append(edge_id)
7:          current_msf_weight ← current_msf_weight + w
8:      else if linkCutTree.maximum_edge(u, v) > w then
9:          maximum_edge_id ← linkCutTree.maximum_edge_id(u, v)
10:         maximum_edge ← edges_by_id[maximum_edge_id]
11:         linkCutTree.cut(maximum_edge.u, maximum_edge.v)
12:         current_msf.erase(maximum_edge.id)
13:         current_msf_weight ← current_msf_weight - maximum_edge.w
14:         linkCutTree.link(u, v, w, edge_id)
15:         current_msf.append(edge_id)
16:         current_msf_weight ← current_msf_weight + w
17:     end if
18: end function
```

4.5 Versão dinâmica

Além da versão incremental do problema, que apresentamos neste capítulo, existe uma versão dinâmica, onde é permitida também a remoção de arestas do grafo. O trabalho de Hanauer *et al.* (2021) cita algumas soluções para essa versão do problema. Em particular, Holm *et al.* (2001) propõem uma solução com custo amortizado $O(\log^4 n)$, onde n é o número de vértices do grafo. Porém, essa solução é bastante sofisticada e seu estudo fugiria do escopo deste trabalho.

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.

Ademais, 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 (2020) 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.

5.1 Square-root decomposition

Inicialmente, vamos conhecer a técnica de *square-root decomposition*, utilizada para transformar soluções que consomem tempo $O(n)$ por operação — onde n é o número de elementos no problema em questão — em soluções com custo $O(\sqrt{n})$ por operação. Para nossa explicação, vamos utilizar o seguinte problema como exemplo: dada uma lista de inteiros $[a_1, a_2, a_3, \ldots, a_n]$, queremos efetuar as duas operações a seguir:

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

Este problema possui duas soluções *ingênuas*, cada uma favorecendo uma das operações. A primeira, e mais simples, consiste em utilizar um laço 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 $\Sigma_{j=1}^{i} a_j$ — para respondermos às consultas `find_sum` em tempo constante, porém, acarretando na reconstrução de `prefix_sum` em toda chamada de `update_value`, o que consome $O(n)$.

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 $n = db$. 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 do bloco.

Com isso, podemos explicar como adaptamos as operações para funcionarem utilizando esta divisão em blocos. Apesar de estarmos focados em resolver o problema da 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 laço. Esta tarefa está representada na Figura 5.1 e podemos perceber que a resposta para a consulta é simplesmente a soma $x + y + z$.

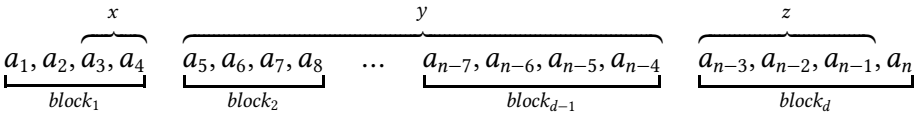


Figura 5.1: *Divisão de uma lista de tamanho n em d blocos de tamanho $b = 4$, mostrando que a soma de x, y e z responde à consulta feita por* `find_sum(3, n-1)`.

Já a rotina `update_value(i, x)` é um pouco mais simples. Ao atualizarmos o valor da posição i, temos simplesmente que atualizar o valor pré-calculado do bloco que a 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 laço sobre estes elementos, e como no pior caso podemos acabar percorrendo $b - 1$ elementos, esta etapa tem custo $O(b)$. Já para encontrar y, iteramos sobre os blocos em si, portanto, 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 torne b e d tão próximos quanto possível. Para isso, podemos fazer:

$$b = d \Rightarrow b = \frac{n}{b} \Rightarrow b^2 = n \Rightarrow b = \pm\sqrt{n} \tag{5.1}$$

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

5.2 Rotinas extras para a versão incremental

Antes de seguirmos adiante com a explicação, temos que apresentar duas funções extras adicionadas à 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. Para isso, elas realizam o que chamamos de *rollback*, isto é, após a realização de todas as adições e da respectiva consulta, desfazemos estas operações, com a ideia de trazer a estrutura de volta ao seu estado inicial.

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 nas *link-cut trees* — `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 em 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 das *link-cut trees* durante este processo, precisamos percorrer esta lista de trás para frente, revertendo uma operação de cada vez.

Finalmente, com estes métodos em mãos, podemos implementar as rotinas extras em que estávamos interessados. Vamos mostrar a seguir somente o pseudo-código da rotina `get_msf_after_operations`, dado que a única diferença entre ela e a outra consulta seria a chamada na terceira linha do Programa 5.2.

Além disso, podemos perceber que a complexidade destes métodos é $O(q \log n)$ amortizado, onde q é o número de arestas na lista `edges[]` e n é o número de vértices do grafo.

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 ← 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 ← current_msf_weight + edge.w
12:         end if
13:     end for
14: end function
```

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
```

5.3 Ideia

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. Agora, com todas as peças necessárias em mãos, podemos partir para a explicação da solução.

Em particular, queremos ser capazes de adicionar uma aresta ao grafo em um certo instante de tempo, assim como realizar consultas acerca de uma 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, no instante t, uma aresta com pontas u e v e peso w;
- `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 (2020), que fornece os métodos acima com um custo de amortizado $O(\sqrt{m} \log n)$ 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, assim como o intervalo de tempo em que estas operações podem acontecer. Já a segunda solução corresponde a uma melhoria nossa da versão original, onde eliminamos as restrições e oferecemos um custo amortizado de $O(\sqrt{m} \log n)$ por operação.

5.3.1 Versão original

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 (2020) 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 os instantes de tempo das operações sejam inteiros distintos 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, para simplificar a explicação, 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* $c_1, c_2, \ldots, c_{\sqrt{m}}$, que correspondem aos instantes de tempo no fim de cada bloco. Assim temos que c_i corresponde ao instante de tempo da operação $i\sqrt{m}$. Em seguida, atribuímos uma floresta geradora mínima 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

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 $c_i \leq t$;
- em seguida, vamos *aumentar* este bloco até que ele possua todas as operações de inserção até o instante t, isto é, com a estrutura t_i em mãos, a incrementamos com todas as arestas adicionadas entre os instantes $c_i + 1$ e t;
- finalmente, basta retornarmos a consulta de interesse, isto é, as arestas que compõem a floresta 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.

Agora, para executamos 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 $t < c_i$ 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 $j \in [i, \sqrt{m}]$.

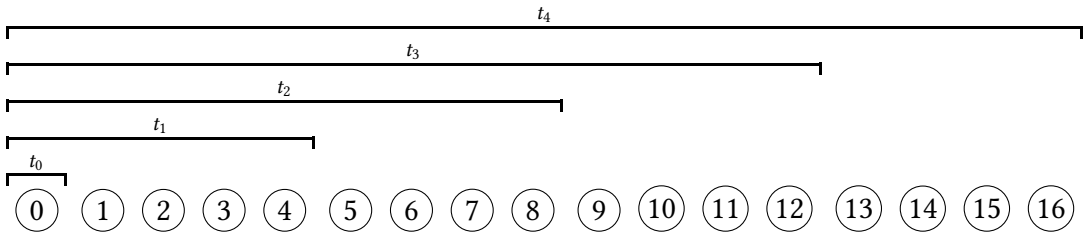


Figura 5.2: *Representação da lista* `edges_by_time` *com* m *igual a* 16. *Neste caso, cada bloco tem tamanho* 4 *e os instantes* $0, 4, 8, 12$ *e* 16 *são* c_0, c_1, c_2, c_3 *e* c_4, *respectivamente. Assim, por exemplo, a estrutura* t_3 *contém todas as arestas adicionadas desde o instante* 1 *até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

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 $O(\sqrt{m} \log n)$.

5.3.2 Versão melhorada

Como podemos ver acima, a versão original de Andrade Júnior e Duarte Seabra (2020) 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.* (2007), 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 aresta 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 por 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 $t_0, t_1, \ldots, t_{\sqrt{m}}$. 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:

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

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 cada t_i a partir do zero, e que talvez exista uma maneira de

reaproveitar 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 $c_0^*, c_1^*, \ldots, c_{\sqrt{m^*}}^*$ e $t_0^*, t_1^*, \ldots, t_{\sqrt{m^*}}^*$ 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, considerando $c_{-2} = c_{-1} = 0$, 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^*.

Na próxima seção, provaremos que $c_{i-2} \leq c_i^*$, de modo que o último passo resulta numa versão correta de t_i^*, além de demonstrar que esse passo é eficiente.

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.

5.3.3 Correção e Complexidade

Antes de adentrarmos nos detalhes de como são as implementações dos métodos da nossa estrutura, vamos analisar a correção e 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.

Lema 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 $(\sqrt{m}+1)^2$. Logo, podemos calcular a diferença entre estes dois valores:

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

Portanto, temos que $2\sqrt{m} + 1$ operações serão realizadas até a próxima reconstrução. □

Em seguida, vamos constatar que o deslocamento de t_i^* até c_i^* sempre consiste na adição de zero ou mais arestas, isto é, que $c_i \leq c_i^*$.

Teorema 5.1. *Durante uma reconstrução, a transformação de* t_{i-2} *em* t_i^* *sempre consiste na adição de novas arestas, em outras palavras, todas as arestas em* t_{i-2} *devem estar* t_i^*.

Demonstração. Primeiramente, temos que cuidar de t_0^* e t_1^*, criadas durante a fase inicial da reconstrução. Neste caso, ambas as estruturas começam representando um grafo vazio, e apenas t_1^* recebe novas arestas.

Vamos chamar p_i a posição da operação que define c_i na lista `edges_by_time` durante a última reconstrução, ou seja, $p_i = i\sqrt{m}$ para $i \in [0, \sqrt{m}]$. Analogamente, definimos $p_i^* = i\sqrt{m^*}$, para $i \in [0, \sqrt{m^*}]$.

Estamos interessados em descobrir o intervalo que teremos que deslocar t_i^*, buscando determinar qual o número mínimo de arestas a serem adicionadas — o que é atingindo quando todas as novas arestas são inseridas antes de c_i.

Logo, 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. □

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 5.2. *Durante uma reconstrução, no máximo* $3\sqrt{m^*}$ *arestas são adicionadas a cada* t_i^*.

Demonstração. Sejam p_i e p_i^* as posições definidas no Teorema 5.1. Da mesma maneira, começamos analisando a criação de t_0^* e t_1^*. 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, o que se encontra dentro do custo apresentado. Agora vamos cuidar do restante das t_i^*.

Aqui, estamos interessados em determinar qual o número máximo de arestas a serem adicionadas a cada t_i^* — o que acontece quando todas as novas arestas são inseridas após c_i, em outras palavras, quando nenhuma nova aresta é adicionada ao i-ésimo bloco ou a algum de seus antecessores.

Em particular, podemos medir o deslocamento de t_i^* neste caso fazendo, para $i \in [0, \sqrt{m}]$:

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

Desta forma, podemos conferir que no máximo $3\sqrt{m^*}$ arestas são adicionadas a cada t_i^*. □

Finalmente, devido ao deslocamento que será realizado em cada t_i^* e utilizando os resultados acima, podemos ver que o custo total 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 por cada inserção de aresta é de $O(\sqrt{m^*} \log n)$.

5.4 Rotina Build Decomposition

A primeira rotina que vamos fornecer uma explicação mais detalhada é a responsável por reconstruir a decomposição, a `build_decomposition`. Este método vai ser acionado pela rotina `add_edge` toda vez que m for um quadrado perfeito.

Para facilitarmos o código, criamos uma rotina auxiliar `move_imsf_checkpoint(t, a, b)`, que recebe uma floresta geradora mínima incremental t e a desloca, adicionando todas arestas no intervalo de tempo $(a, b]$, com custo amortizado $O((b-a)\log n)$, onde n é o numero de vértices no grafo.

Com isso, a implementação segue diretamente da explicação nas Seção 5.3.3, com a criação de novas listas c^* e t^*, para armazenar os *checkpoints* e as florestas geradoras mínimas incrementais, respectivamente. Além disso, utilizamos um laço para preencher t^* e fazer com que cada uma das t_i^* seja deslocada para o seu respectivo c_i^*.

Por último, o custo do laço na linha 6 é de $O(m \log m)$ — porque estamos percorrendo uma lista ordenada, que possui um custo logarítmico para consultas e modificações — e o custo de cada chamada `move_imsf_checkpoint` é de $O(\sqrt{m} \log n)$ — pois como demonstrado no Teorema 5.2, são adicionadas no máximo $3\sqrt{m}$ arestas, com cada adição tendo um custo amortizado de $O(\log n)$. Portanto, podemos conferir que a rotina gasta tempo amortizado $O(m \log n)$, onde m é o número de arestas na estrutura e n a quantidade de vértices no grafo.

Programa 5.3 Rotina Build Decomposition

```
1:  function BUILD_DECOMPOSITION()
2:      block_size ← block_size + 1
3:      n_blocks ← block_size + 1
4:      position ← 0
5:      c* ← [0]
6:      for each [edge, time] in edges_by_time do
7:          position gets position + 1
8:          if position % block_size = 0 then
9:              c*.append(time)
10:         end if
11:     end for
12:     t* ← [new IncrementalMSF(), new IncrementalMSF()]
13:     move_imsf_checkpoint(t*[1], 0, c*[1])
14:     for i ∈ [2, n_blocks) do
15:         t*.append(t[i-2])
16:         move_imsf_checkpoint(t*[i], c[i-2], c*[i])
17:     end for
18:     c ← c*
19:     t ← t*
20: end function
```

5.5 Consultas Get MSF e Get MST Weight

Primeiramente, para falarmos sobre as consultas, precisamos lembrar a ideia por trás delas. Dado um instante de tempo t, precisamos encontrar o maior *checkpoint* tal que $c_i \leq t$. Depois disso, aumentamos a estrutura t_i do respectivo bloco, adicionando todas as arestas no intervalo de tempo $(c_i, t]$. Por último, podemos retornar a consulta propriamente dita, isto é, a árvore geradora mínima ou o seu respetivo peso.

Entretanto, não podemos simplesmente realizar uma cópia de t_i para então aumentá-la, pois isso teria um custo $O(m)$ no pior caso. Para contornar este problema, usamos os métodos apresentados na Seção 5.2, que nos permitem adicionar as arestas no intervalo e depois desfazer essas alterações.

Ademais, criamos mais duas rotinas auxiliares para ajudar na implementação das consultas. A primeira delas é a `find_left_checkpoint_index(t)`, que percorre a lista de *checkpoints* e retorna o maior inteiro i tal que $c_i \leq t$. Em seguida, temos a rotina `get_delta_edge_operations(i, t)`, que retorna uma lista com todas as arestas no intervalo $(c_i, t]$. Os custos destas rotinas são $O(\sqrt{m})$ e $O(\sqrt{m} \log m)$, respectivamente.

Deste modo, como a lista `delta_operations` tem no máximo $\sqrt{m}$ arestas, e como o custo para adicionar e remover cada uma destas arestas é de $O(\log n)$ amortizado, temos que a rotina `get_msf_weight` possui um custo amortizado de $O(\sqrt{m} \log n)$. Além disso, a rotina `get_msf`, que teve seu pseudo-código omitido devido à similaridade com o Programa 5.4, possui um custo amortizado de $O(m)$.

Programa 5.4 Consulta Get MSF Weight

```
1: function GET_MSF_WEIGHT(t)
2:     checkpoint_index ← find_left_checkpoint_index(t)
3:     delta_operations ← get_delta_edge_operations(last_checkpoint_index, t)
4:     return t[checkpoint_index].get_msf_weight_after_operations(delta_operations)
5: end function
```

5.6 Rotina Add Edge

Finalmente, chegamos a rotina responsável por adicionar novas arestas a estrutura. Sua responsabilidade consiste em adicionar a aresta (u, v, w) na lista `edges_by_time`, além de inserir a aresta na t_i de todos os blocos tal que $t < c_i$. Ademais, também atribuímos a está rotina a função de acionar a reconstrução da estrutura, caso necessário.

Programa 5.5 Rotina Add Edge

```
1: function GET_MSF(u,v,w,t)
2:     edges_by_time[t] ← Edge(u,v,w)
3:     for i ∈ (find_left_checkpoint_index(t), n_blocks) do
4:         t[i].add_edge(u, v, w)
5:     end for
6:     if (block_size + 1)^2 = edges_by_time.size() then
7:         rebuild_decomposition()
8:     end if
9: end function
```

Com isso, temos que o custo amortizado dessa rotina é de $O(\sqrt{m} \log n)$, o custo de inserção em cada uma das t_i, além do custo amortizado da reconstrução.

Capítulo 6

Implementação

Todas as estruturas apresentadas nesse trabalho foram implementadas e testadas. Neste capítulo, discutiremos acerca dos detalhes de implementação, tais como aspectos de linguagem, testagem e organização dos códigos.

6.1 Organização

Todas as implementações foram feitas utilizando `C++17` como linguagem de programação. Além disso, criamos um repositório no GitHub[1] para versionar o desenvolvimento dos programas assim como a escrita da monografia. Este repositório pode ser acessado em www.github.com/fcnoronha/mac0499.

Dentro do repositório, o diretório `implementations` contém todos os códigos e arquivos necessários para a execução das estruturas. A Figura 6.1 mostra detalhadamente a estrutura deste diretório.

Em particular, a implementação de cada estrutura contem três arquivos principais: um arquivo *header* `.hpp`, contendo o protótipo para todos os métodos e uma breve documentação acerca do funcionamento de cada um; um arquivo de código fonte `.cpp`, que contém a implementação dos métodos e a lógica por trás de cada estrutura; um arquivo de teste, com o sufixo `Test.cpp`, que contém as rotinas utilizadas para testar cada uma das estruturas. Discutiremos mais sobre estes testes na próxima seção.

6.2 Construção e testes

Durante a etapa de desenvolvimento, foi necessário encontrar uma maneira fácil e eficiente de compilar as estruturas e testes. Com isso em mente, utilizamos a ferramenta de construção e testes Bazel[2]. A partir dessa ferramenta, concentramos toda a lógica necessária para construir executáveis e executar testes no arquivo `BUILD`.

[1] www.github.com

[2] www.bazel.build

```
implementations
├── BUILD
├── splayTree.cpp
├── splayTree.hpp
├── linkCutTree.cpp
├── linkCutTree.hpp
├── linkCutTreeTest.cpp
├── retroactiveUnionFind
│   ├── retroactiveUnionFind.cpp
│   ├── retroactiveUnionFind.hpp
│   └── retroactiveUnionFindTest.cpp
├── incrementalMSF
│   ├── incrementalMSF.cpp
│   ├── incrementalMSF.hpp
│   └── incrementalMSFTest.cpp
└── retroactiveMSF
    ├── retroactiveMSF.cpp
    ├── retroactiveMSF.hpp
    └── retroactiveMSFTest.cpp
```

Figura 6.1: *Árvore do diretório que contém as implementações das estruturas apresentadas no trabalho.*

Além disso, a execução de testes automatizados é uma parte fundamental para o desenvolvimento de programas. Por causa disso, decidimos utilizar a biblioteca Google Test[3], que permite a realização de testes unitários em `C++`. Para cada uma das estruturas, escrevemos uma suíte de testes com o intuito de simular os casos de borda e verificar a correção da implementação.

Em especial, tivemos um pouco mais de cuidado para testar a implementação da floresta geradora mínima semi-retroativa, realizando um teste de estresse. Para isso, criamos um grafo com mil vértices e utilizamos um gerador de números aleatórios para criar suas duas mil arestas. Em seguida, associando um instante de tempo distinto para cada uma das arestas, comparamos a floresta geradora maximal de peso mínimo gerada pela nossa implementação com a floresta geradora maximal de peso mínimo encontrada pela execução do algorítimo de Kruskal. Esta checagem foi realizada diversas vezes durante a inserção de arestas na estrutura, a fim de garantir que a resposta gerada estava correta.

[3] https://google.github.io/googletest

Capítulo 7

Conclusão

Inicialmente, o objetivo deste trabalho era estudar a implementação retroativa de uma árvore binária de busca e de uma tabela *hash*, citadas no artigo de Agarwal e Panwaria (2012). Entretanto, como não conseguimos encontrar material acerca de tais estruturas, decidimos considerar outras opções dentro do tópico de retroatividade.

Após realizarmos uma busca na literatura por outros temas, decidimos focar o trabalho em um estudo sobre as versões retroativas do union-find e da floresta geradora mínima. Este estudo se resumiria em compreender o funcionamento de cada uma das estruturas, assim como implementá-las. Em particular, a parte de implementação se mostrou tão difícil quanto a escrita dessa dissertação. Porém, esta tarefa foi fundamental para consolidar o entendimento sobre as estruturas, e somente depois dela estivemos confortáveis para explicar o funcionamento das soluções.

Ademais, ficamos bastante contentes com a melhoria realizada no trabalho de Andrade Júnior e Duarte Seabra (2020). Desde o início, ficamos incomodados com as limitações que tal solução apresentava, e buscamos diversas formas para contorná-las, até que chegamos à ideia aqui apresentada. Além disso, deixamos em aberto dois problemas para serem abordados em trabalhos futuros.

O primeiro deles diz respeito a uma outra maneira de reconstruir a decomposição utilizada pela floresta geradora maximal de peso mínimo semi-retroativa. Talvez, ao dividirmos em dois, os maiores dois blocos da decomposição, consigamos garantir que o número de blocos e seus respetivos tamanhos serão proporcionais a $\sqrt{m}$. Especificamente, esta divisão consiste em adicionar uma nova floresta geradora maximal de peso mínimo incremental no meio de cada bloco, ou seja, entre dois *checkpoints*. Essa ideia foi considerada no lugar da versão apresentada neste trabalho, porém, não conseguimos elaborar uma prova acerca de seu consumo de tempo.

Já o segundo problema consiste em buscar uma forma de adaptar a solução do problema da floresta geradora maximal de peso mínimo para suportar a remoção de inserções de arestas, o que tornaria a estrutura, de fato, totalmente retroativa.

Referências

[Agarwal e Panwaria 2012] Suneeta Agarwal e Prakhar Panwaria. "Implementation, analysis and application of retroactive data structures". Em: *International Conference on Advances in Computer Science and Electronics Engineering* (fev. de 2012), pgs. 88–92. doi: 10.15224/978-981-07-1403-1-247 (citado na pg. 39).

[CP-Algorithms 2022] CP-Algorithms. *Square root decomposition*. 2022. url: https://cp-algorithms.com/data_structures/sqrt_decomposition.html (acesso em 09/07/2022).

[Andrade Júnior e Duarte Seabra 2020] José Wagner de Andrade Júnior e Rodrigo Duarte Seabra. "Fully Retroactive Minimum Spanning Tree Problem". Em: *The Computer Journal* 65.4 (dez. de 2020), pgs. 973–982. issn: 0010-4620. doi: 10.1093/comjnl/bxaa135. eprint: https://academic.oup.com/comjnl/article-pdf/65/4/973/43377476/bxaa135.pdf. url: https://doi.org/10.1093/comjnl/bxaa135 (citado nas pgs. 21, 25, 28, 29, 31, 39).

[Cormen *et al.* 2009] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest e Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 2009. isbn: 0262033844 (citado na pg. 1).

[Demaine, Holmgren *et al.* 2012] Erik D. Demaine, Justin Holmgren, Jing Jian, Maksim Stepanenko e Mashhood Ishaque. *6.851: Advanced Data Structures Spring 2012 - Lecture 19*. 2012. url: https://courses.csail.mit.edu/6.851/spring12/scribe/L19.pdf (citado na pg. 6).

[Demaine, Iacono *et al.* 2007] Erik D. Demaine, John Iacono e Stefan Langerman. "Retroactive data structures". Em: *ACM Trans. Algorithms* (2007). issn: 1549-6325. doi: 10.1145/1240233.1240236. url: https://doi.org/10.1145/1240233.1240236 (citado nas pgs. 1, 17, 31).

[Demaine, Langerman *et al.* 2008] Erik D. Demaine, Stefan Langerman e Eric Price. "Confluently persistent tries for efficient version control". Em: *Scandinavian Workshop on Algorithm Theory (SWAT)*. Ed. por Joachim Gudmundsson. Springer, 2008, pgs. 160–172. isbn: 978-3-540-69903-3 (citado na pg. 31).

[Frederickson 1985] Greg N. Frederickson. "Data structures for on-line updating of minimum spanning trees, with applications". Em: *SIAM Journal on Computing* 14.4 (1985), pgs. 781–798 (citado na pg. 21).

[Galler e Fisher 1964] Bernard A. Galler e Michael J. Fisher. "An improved equivalence algorithm". Em: *Commun. ACM* 7.5 (1964), pgs. 301–303. ISSN: 0001-0782. DOI: 10.1145/364099.364331. URL: https://doi.org/10.1145/364099.364331 (citado na pg. 17).

[Hanauer *et al.* 2021] Kathrin Hanauer, Monika Henzinger e Christian Schulz. "Recent advances in fully dynamic graph algorithms". Em: *arXiv preprint arXiv:2102.11169* (2021) (citado na pg. 24).

[Holm *et al.* 2001] Jacob Holm, Kristian de Lichtenberg e Mikkel Thorup. "Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity". Em: *J. ACM* 48.4 (jul. de 2001), pgs. 723–760. ISSN: 0004-5411. DOI: 10.1145/502090.502095. URL: https://doi.org/10.1145/502090.502095 (citado na pg. 24).

[Kruskal 1956] Joseph B. Kruskal. "On the shortest spanning subtree of a graph and the traveling salesman problem". Em: *Proceedings of the American Mathematical Society* 7.1 (1956), pgs. 48–50 (citado na pg. 21).

[Prim 1957] R. C. Prim. "Shortest connection networks and some generalizations". Em: *The Bell System Technical Journal* 36.6 (1957), pgs. 1389–1401. DOI: 10.1002/j.1538-7305.1957.tb01515.x (citado na pg. 21).

[Sarnak 1986] Neil I. Sarnak. *Persistent data structures*. New York University, 1986 (citado na pg. 2).

[Sleator e Tarjan 1981] Daniel D. Sleator e Robert E. Tarjan. "A data structure for dynamic trees". Em: *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing (STOC)*. New York, NY, USA, 1981, pgs. 114–122. ISBN: 9781450373920. DOI: 10.1145/800076.802464. URL: https://doi.org/10.1145/800076.802464 (citado nas pgs. 3, 9).

[Sleator e Tarjan 1985] Daniel D. Sleator e Robert E. Tarjan. "Self-adjusting binary search trees". Em: *J. ACM* 32.3 (1985), pgs. 652–686. ISSN: 0004-5411. DOI: 10.1145/3828.3835. URL: https://doi.org/10.1145/3828.3835 (citado na pg. 9).

[Tarjan e Leeuwen 1984] Robert E. Tarjan e Jan van Leeuwen. "Worst-case analysis of set union algorithms". Em: *J. ACM* 31.2 (1984), pgs. 245–281. ISSN: 0004-5411. DOI: 10.1145/62.2160. URL: https://doi.org/10.1145/62.2160 (citado na pg. 17).