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

Estruturas de dados retroativas

Um estudo sobre Union-Find e ...

Felipe Castro de Noronha

Monografia Final

mac 499 — Trabalho de
Formatura Supervisionado

Supervisora: Profª. Drª. Cristina Gomes Fernandes

São Paulo
2022

Dedico este trabalho a meus pais e todos aqueles que me ajudaram durante esta caminhada.

Agradecimentos

Eu sou quem sou porque estou aqui.

— Paul Atreides

Texto texto. Texto opcional.

Resumo

Felipe Castro de Noronha. **Estruturas de dados retroativas: *Um estudo sobre Union-Find e …***. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2022.

Elemento obrigatório, constituído de uma sequência de frases concisas e objetivas, em forma de texto. Deve apresentar os objetivos, métodos empregados, resultados e conclusões. O resumo deve ser redigido em parágrafo único, conter no máximo 500 palavras e ser seguido dos termos representativos do conteúdo do trabalho (palavras-chave). Deve ser precedido da referência do documento. Texto texto. Texto texto.

Palavras-chave: Palavra-chave1. Palavra-chave2. Palavra-chave3.

Abstract

Felipe Castro de Noronha. **Retroactive data structures: *A study about Union-Find and***. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2022.

Elemento obrigatório, elaborado com as mesmas características do resumo em língua portuguesa. De acordo com o Regimento da Pós-Graduação da USP (Artigo 99), deve ser redigido em inglês para fins de divulgação. É uma boa ideia usar o sítio www.grammarly.com na preparação de textos em inglês. Text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text. Text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text.

Keywords: Keyword1. Keyword2. Keyword3.

Lista de Programas

2.1 Rotina Access 6
2.2 Rotina Make Root 7
2.3 Rotina Link 8
2.4 Rotina Cut 8
2.5 Consulta Is Connected 9
2.6 Consulta Maximum Edge 9
2.7 Rotina Splay 12
2.8 Rotina Split 13
2.9 Rotina Join 13
2.10 Rotina Revese Path 14
2.11 Consulta Get Path End Node 14
2.12 Consulta Get Parent Path Node 14
2.13 Consulta Get Maximum Path Value 15
3.1 Consulta Same Set 19
3.2 Rotina Create Union 20
3.3 Rotina Delete Union 20
4.1 Consulta Get MSF 22
4.2 Consulta Get MSF Cost 23
4.3 Rotina Add Edge 23
4.4 Rotina Apply Rollback 25
4.5 Rotina Get MSF After Operations 25

Sumário

1 Introdução **1**
1.1 Persistência Parcial . . . 1
1.2 Persistência Total . . . 1
1.3 Retroatividade Parcial . . . 1
1.4 Retroatividade Total . . . 1

2 Link-Cut Tree **3**
2.1 Ideia . . . 3
2.2 Definições . . . 4
2.3 Operações . . . 4
2.3.1 Rotina Access . . . 6
2.3.2 Rotinas Make Root, Link e Cut . . . 6
2.3.3 Consultas Is Connected e Maximum Edge . . . 9
2.4 Splay Tree . . . 9
2.4.1 Rotina Splay . . . 11
2.4.2 Rotinas Split e Join . . . 13
2.4.3 Métodos auxiliares . . . 14

3 Union-Find Retroativo **17**
3.1 Ideia . . . 17
3.2 Estrutura interna . . . 18
3.3 Consultas Same Set . . . 19
3.4 Rotinas Create Union e Delete Union . . . 19

4 Floresta Geradora Mínima incremental **21**
4.1 Ideia . . . 21
4.2 Estrutura interna . . . 22
4.3 Consultas Get MSF e Get MST Cost . . . 22
4.4 Rotina Add Edge . . . 23

4.5 Rotinas Extras 24

5 Floresta Geradora Mínima retroativa **27**

5.1 Square-root decomposition 27

5.2 Ideia 28

5.3 Consultas Get MSF e Get MST Cost 28

5.4 Rotina Add Edge 28

5.5 Complexidade 28

Referências **29**

Capítulo 1

Introdução

Estruturas de dados retroativas bla bla bla

1.1 Persistência Parcial

1.2 Persistência Total

1.3 Retroatividade Parcial

1.4 Retroatividade Total

Capítulo 2

Link-Cut Tree

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

2.1 Ideia

A link-cut tree é uma estrutura de dados que nos permite manter uma floresta de árvores enraizadas com peso nas arestas, onde os nós de cada árvore possuem um número arbitrário de filhos. Ademais, 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 árvore a aresta com pontas em u e v, efetivamente separando estes vértices e resultando duas novas árvores.
- `is_connected(u, v)`: retorna `verdadeiro` caso u e v pertençam à mesma árvore, `falso` caso contrario.

Por último, a link-cut tree possui 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 fazer algumas definições acerca da estrutura que vamos estudar.

Chamamos de *árvores representadas* as componentes da floresta armazenada na link-cut tree. Para a representação que a link-cut tree utiliza, 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 tais vértices serão responsáveis por guardar um ponteiro para o vértice do caminho preferido imediatamente acima do caminho que o contém.

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 nó u como seu pai e v como seu único filho.

2.3 Operações

Nessa seção, apresentaremos o código por trás das operações que estamos interessados em implementar na link-cut tree. 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)`: recebe um nó u e separa o caminho preferido que contém este nó 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.
- `join(u, v)`: recebe dois nós, 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. 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 orientação desse caminho, isto é, o fim se transforma no começo e o começo no fim.

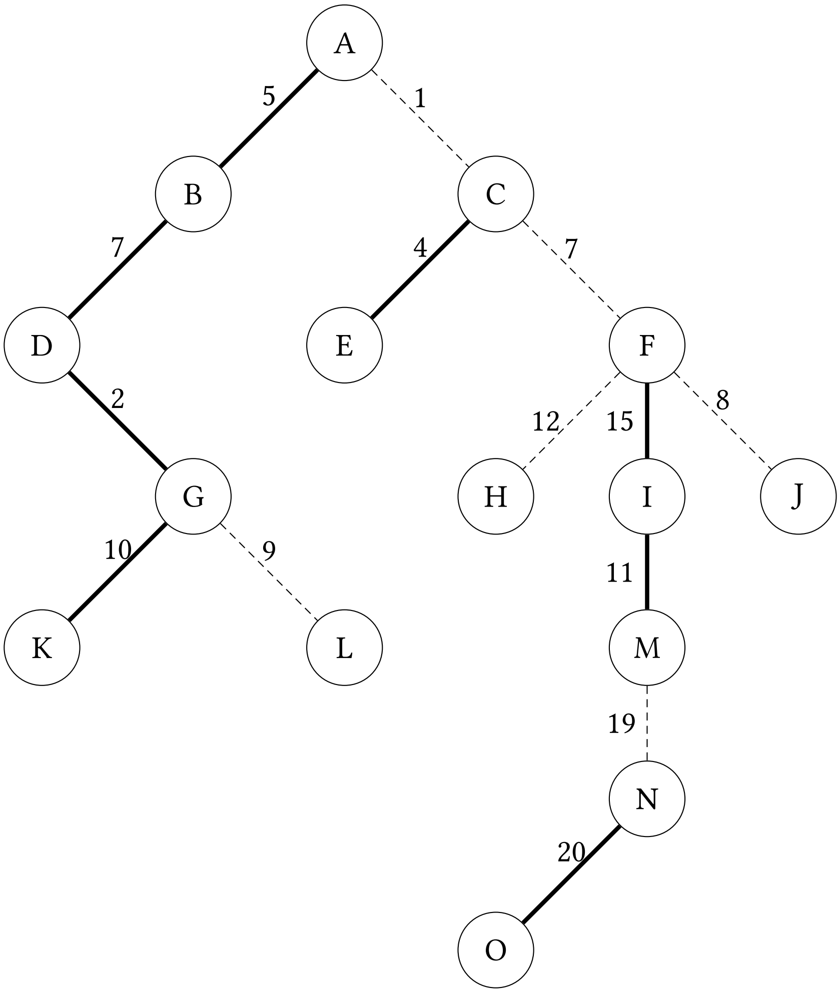


Figura 2.1: *Árvore representada e seus caminhos preferidos. Na figura acima, as arestas escuras representam caminhos preferidos, com isso, temos o seguinte conjunto de caminhos vértice-disjuntos* $\{\langle K, G, D, B, A\rangle, \langle E, C\rangle, \langle M, I, F\rangle, \langle L\rangle, \langle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

- `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 ma floresta imediatamente acima do fim do caminho preferido que contém u; caso tal caminho contenha a raiz da árvore representada, este método retorna `null`. Aqui, na árvore da figura 2.1, `get_parent_path_node(M)` retorna o vértice `C`.

- `get_maximum_path_value(u)`: recebe u, o identificador de um caminho preferido, e retorna o maior valor de uma aresta neste caminho.

Com tal conjunto de funções, podemos avançar para os métodos da link-cut tree.

2.3.1 Rotina Access

Uma rotina utilizada por todos os métodos da link-cut tree que vamos implementar é a `access(u)`. A partir dela conseguimos reorganizar a estrutura interna da 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. Com isso, todas as arestas preferidas que tinham somente uma das pontas fazendo parte deste novo caminho são destruídas 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. A cada iteração, fazemos com que uma variável `current_root`, que inicialmente corresponde ao vértice u, vire o identificador de seu caminho preferido. Além disso, mantemos uma variável `last`, que corresponde a `current_root` da iteração anterior, no início com valor igual a `null`.

Com estes valores em mãos, podemos ir criando um caminho preferido através de sucessivas concatenações, unindo o caminho que `current_root` identifica à parte superior do caminho mantido por `last`. Ao final dessa concatenação, temos que `current_root` é o identificador deste caminho que está sendo construído, e após guardarmos seu valor em `last`, podemos prosseguir para o próximo passo, onde `current_root` agora corresponde ao nó imediatamente em cima do caminho preferido que estamos construindo.

Programa 2.1 Rotina Access

```
function ACCESS(u)
    last ⟵ NULL
    current_root ⟵ u
    ▷ concatena todos os caminhos preferidos de u até a raiz da árvore representada
    while current_root ≠ NULL do
        make_identifier(current_root)
        ▷ concatena um novo pedaço de caminho preferido ao caminho em que last é identificador
        join(current_root, last)
        last ⟵ current_root
        current_root ⟵ get_parent_path_node(current_root)
        make_identifier(u)
    end while
end function
```

Ao final da iteração, colocamos o vértice u como identificador deste novo caminho preferencial, simplificando as operações a seguir.

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 da raiz dessa árvore até u, 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

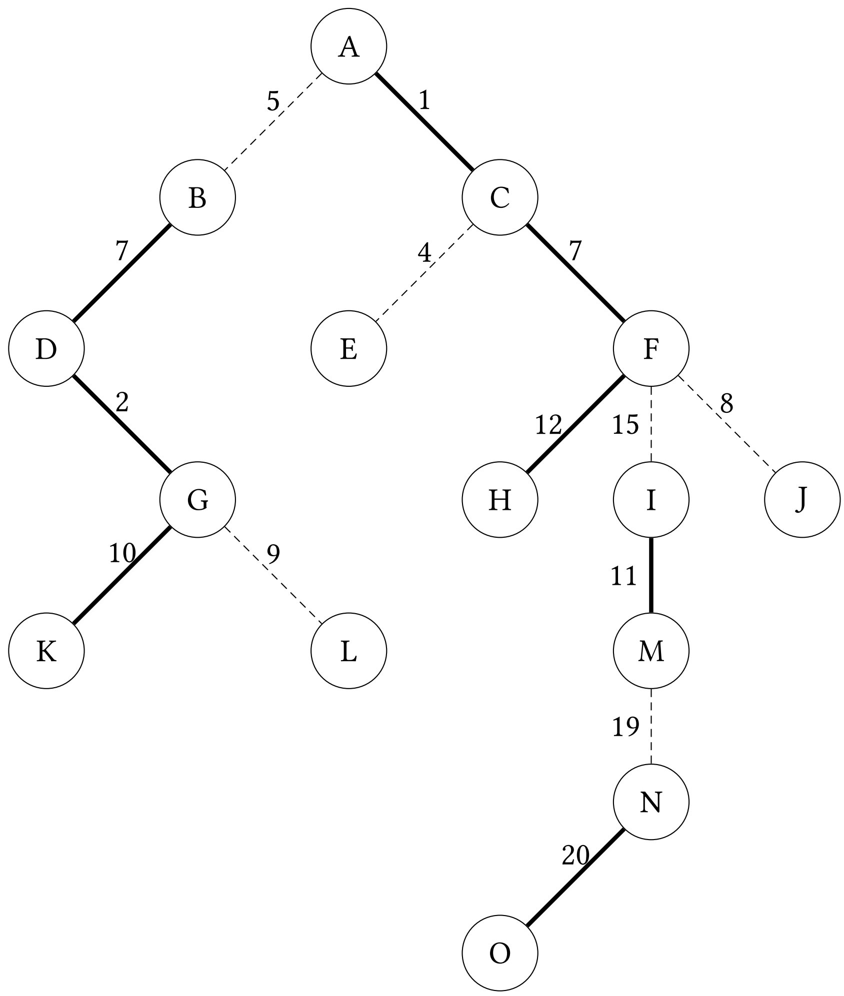


Figura 2.2: *Caminhos preferidos na árvore da figura 2.1 após uma operação de* `access` *no nó H. Com isso temos o novo conjunto de caminhos vértice-disjuntos* $\{\langle H, F, C, A\rangle, \langle K, G, D, B\rangle, \langle M, I\rangle, \langle E\rangle, \langle J\rangle, \langle L\rangle, \langle O,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

de menor profundidade da sua árvore representada, o que se traduz neste sendo a nova raiz.

Programa 2.2 Rotina Make Root

```
function MAKE_ROOT(u)
    access(u)
    reverse_path(u)
end function
```

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

A primeira delas recebe dois vértices 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 vértices em nossa representação interna. Com isso, o primeiro passo é criar um vértice 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
  function LINK(*u, v, w*)
    uv_edge ⟵ *new Node(w)* ▷ cria nó com peso w, representando a aresta
    ▷ ligando (v) – (uv_edge)
    make_root(v)
    access(v)
    join(v, uv_edge)
    ▷ ligando (uv_edge) – (u)
    make_root(u)
    access(u)
    access(uv_edge)
    join(uv_edge, u)
  end function

Já a operação `cut(u, v)`, que separa dois nós 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`.

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

Programa 2.4 Rotina Cut

Require: u e v na mesma árvore
  function CUT(*u, v*)
    ▷ cortando (u) – (uv_edge)
    make_root(u)
    access(uv_edge)
    split(uv_edge)
    ▷ cortando (v) – (uv_edge)
    make_root(v)
    access(uv_edge)
    split(uv_edge)
  end function

2.3.3 Consultas Is Connected e Maximum Edge

A função `is_connected(u, v)`, que nos informa se u e v pertencem a mesma árvore, funciona da seguinte maneira. Primeiro acessamos u, criando um caminho deste até a raiz da árvore. Em seguida, guardamos o vértice que esta 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

```
function IS_CONNECTED(u, v)
    access(u)
    u_tree_root ← get_path_end_node(u)
    access(v)
    v_tree_root ← get_path_end_node(v)
    return (u_tree_root = v_tree_root)
end function
```

Por último, temos a função `maximum_edge(u, v)`, que 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
  function MAXIMUM_EDGE(u, v)
      make_root(v)
      access(u)
      return get_maximum_path_value(u)
  end function
```

Assim, encerramos a explicação da implementação dos métodos da link-cut tree.

2.4 Splay Tree

No artigo original, Sleator e Tarjan 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 na link-cut tree. 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. Em contrapartida, com este método, perdemos a capacidade de realizarmos buscas por chave na splay tree, porém não necessitamos dessa operação.

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

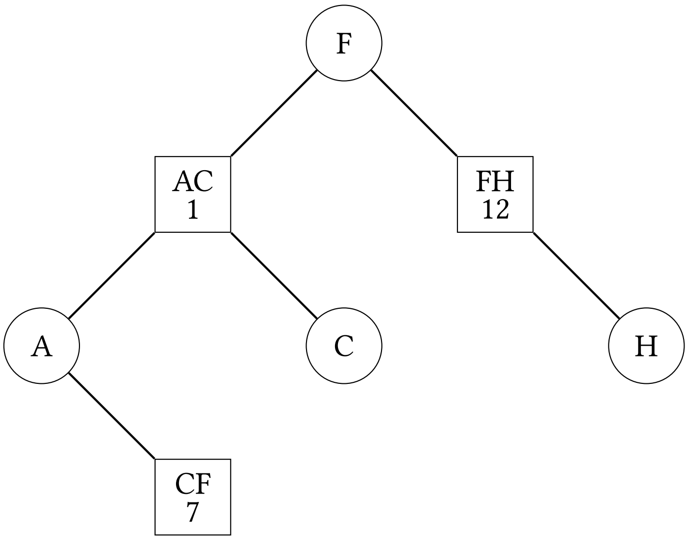


Figura 2.3: *Uma possível configuração da splay tree que armazena o caminho preferido* $\langle H, F, C, A\rangle$ *da figura 2.2, onde* F *é identificador do caminho. Os nós em formato retangular mostram as arestas da árvore representada, com o peso de tal aresta na parte inferior.*

Além disso, como usamos a profundidade dos nós na árvore representada como chave para a árvore auxiliar, temos que todos os nós na sub-árvore esquerda da raiz de uma splay tree 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 nós de uma árvore auxiliar, efetivamente invertendo a orientação de um caminho preferido.

Com isso, os nós da árvore auxiliar 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.

Em particular, caso o nó seja a raiz da árvore auxiliar, o campo `parent` armazena um ponteiro para o vértice que está logo acima do fim deste caminho preferido na árvore representada. Ou seja, a raiz de uma árvore auxiliar aponta para um nó de outra árvore auxiliar, aquela que contém o vértice a que seu caminho preferencial se liga na sua árvore representada.

2.4.1 Rotina Splay

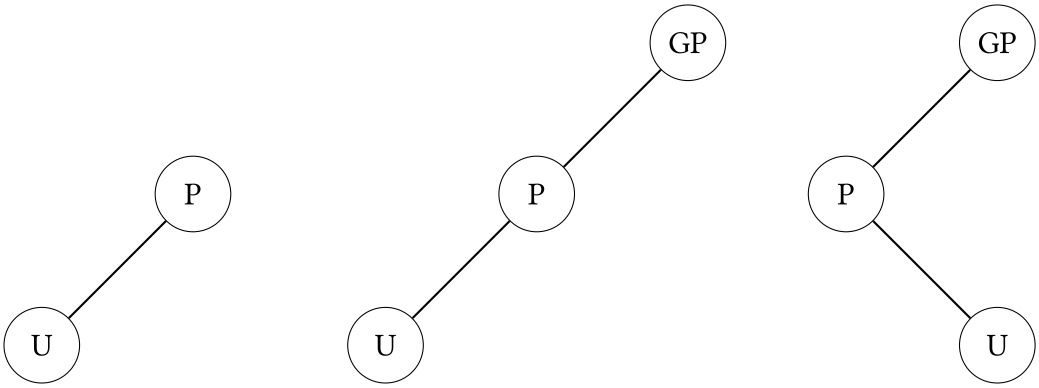
A rotina `splay` é o que garante o auto-balanceamento de uma splay tree. Como já mencionamos, seu efeito é trazer um dado nó para a raiz da árvore por meio de uma série de rotações. Para que o custo de m operações em uma splay seja $O(m \log n)$ amortizado, a implementação deve garantir que o método `splay` é 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 as rotações duplas ou unitárias, até que o nó 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 vértice em identificador de seu caminho, ou seja, entendemos como sinônimos os métodos `make_identifier` e `splay`.

De modo a facilitarmos nossa explicação detalhada, chamamos `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 que `grandparent`, `parent` e u se encontram, isto é, se estão em uma orientação de *zig*, *zig-zig* ou *zig-zag*, como exemplificadas na figura 2.4. Dependendo da orientação, fazemos uma rotação em u ou em `parent`, sempre com a ideia de diminuirmos em 1 a profundidade de u.

Ao sair da função `splay`, o nó u estará na raiz de sua árvore auxiliar. Além disso, seu



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

Figura 2.4: *Orientações zig, zig-zig e zig-zag na splay tree. Aqui,* `P` *e* `GP` *abreviam* `parent` *e* `grand-parent`*, respectivamente.*

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

```
function SPLAY(u)
    while ¬u.is_root() do ▷ u não ser raiz da LCT e nem da Splay
        parent ⟵ u.parent
        grandparent ⟵ parent.parent
        if ¬parent.is_root() then
            grandparent.push_reversed_bit()
            parent.push_reversed_bit()
            if (grandparent.r_child = parent) = (parent.r_child = u) then
                rotate(parent) ▷ zig-zig ou zag-zag
            else
                rotate(u) ▷ zig-zag
            end if
        end if
        rotate(u)
    end while
    u.push_reversed_bit()
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 manutenção 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.

Programa 2.8 Rotina Split

```
function SPLIT(u)
    if u.l_child ≠ NULL then
        u.l_child.parent ⟵ NULL
    end if
    u.l_child ⟵ NULL
end function
```

Primeiramente, falaremos do método `split(u)`, que recebe um nó u e separa o caminho preferido que o contém em dois: um com os vértices menos profundos que u em seu caminho, e outro com u e os vértices mais profundos que u. Para isso, o método simplesmente separa a sub-árvore esquerda de u, como mostrado acima. 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()` da link-cut tree.

Programa 2.9 Rotina Join

```
Require: u e v identificadores de seus caminhos preferidos
function JOIN(u, v)
    if v ≠ NULL then
        v.parent ⟵ u
    end if
    u.r_child ⟵ v
    ▷ atualiza max_subtree_value com o máximo entre o value dos dois filhos de u
    u.recalculate_max_subtree_value()
end function
```

De maneira complementar, temos a rotina `join(u, v)` que recebe dois nós e concatena os respectivos caminhos preferidos. Para isso, assume-se 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.

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.

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 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 na sua sub-árvore.

Programa 2.10 Rotina Revese Path

```
Require: u identificador de seu caminho preferido
  function REVESE_PATH(u)
      u.is_reversed ← ¬u.is_reversed
      u.push_reversed_bit() ▷ inverte os filhos de u e propaga a inversão do bit
  end function
```

Programa 2.11 Consulta Get Path End Node

```
  function GET_PATH_END_NODE(u)
      splay(u)
      smallest_value ← u
      while smallest_value.l_child ≠ NULL do
          smallest_value ← smallest_value.l_child
      end while
      splay(smallest_value)
      return smallest_value
  end function
```

Programa 2.12 Consulta Get Parent Path Node

```
  function GET_PARENT_PATH_NODE(u)
      splay(u)
      return u.parent
  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 vértice menos profundo do caminho preferido de u, fazendo isso ao acessar o vértice mais à esquerda na sua splay tree. Já o segundo método é responsável por retornar o vértice 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 `parent`.

Por último, temos a função `get_maximum_path_value(u)`, que recebe um vértice 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`.

Programa 2.13 Consulta Get Maximum Path Value

Require: u identificador de seu caminho preferido
  function GET_MAXIMUM_PATH_VALUE(u)
    return $u.max\_subtree\_value$
  end function

Com isso, temos todas as ferramentas necessárias para manipularmos a splay tree em seu uso como árvore auxiliar nas link-cut trees.

Capítulo 3

Union-Find Retroativo

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

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. Para isso, ela fornece duas operações principais:

- `same_set(a, b)`: retorna *verdadeiro* caso a e b estejam no mesmo conjunto, *falso* caso contrario.
- `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) utilizam a técnica de compressão de caminhos para mostrar uma implementação com complexidade $O(\alpha(n))$, 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 certo instante t.

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 criar 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 em um dado instante.

Por exemplo, a figura 3.1 mostra uma coleção de conjuntos disjuntos em quatro instantes de tempo. Neste caso, as consultas `same_set(a, b, 3)` e `same_set(c, d, 3)` retornariam *verdadeiro*, enquanto `same_set(a, d, 3)` e `same_set(c, d, 5)` retornariam *falso*.

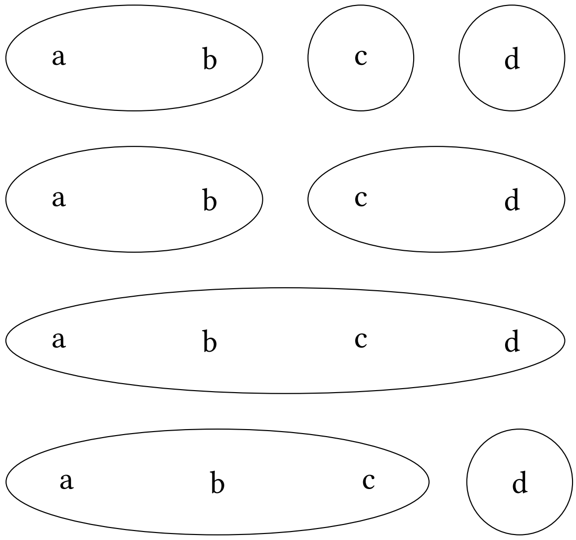


Figura 3.1: *Representação dos conjuntos com os elementos* $\{a, b, c, d\}$ *após a seguinte sequência de operações:* `create_union(a, b, 2)`, `create_union(c, d, 3)`, `create_union(b, c, 4)` *e* `delete_union(3)`. *Cada linha mostra o estado atual da coleção imediatamente após uma operação.*

Note 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.

3.2 Estrutura interna

Para implementarmos o union-find retroativo, vamos utilizar a link-cut tree como estrutura interna. Para isso, fazemos com que os elementos dos conjuntos sejam nós na floresta mantida pela link-cut tree. 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`.

Desta forma, para introduzirmos o caráter retroativo da estrutura, vamos utilizar o atributo `value` que mantemos nas arestas da link-cut tree. 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 pela link-cut tree, basicamente sem nenhuma computação adicional, podemos perceber que o union-find retroativo tem uma complexidade 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 link-cut tree. 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 da link-cut tree 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 consultado. 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

```
function SAME_SET(a, b, t)
    if ¬linkCutTree.is_connected(a, b) then
        return false
    end if
    return linkCutTree.maximum_edge(a, b) ≤ t
end function
```

3.4 Rotinas Create Union e Delete Union

Por último, temos as rotinas de criação e deleção de uniões. Aqui, as implementações são bem diretas, uma vez que essas operações se traduzem na criação e deleção de uma aresta na link-cut tree, 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. Ademais, para dar suporte a deleção de uma união criada em um instante t, precisamos criar um mapeamento que guarda o par de elementos unidos em cada instante. 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

function CREATE_UNION(*a, b, t*)
linkCutTree.create_node(a)
linkCutTree.create_node(b)
linkCutTree.link(a,b,t)
edges_by_time[t] ⟵ *(a, b)*
end function

Programa 3.3 Rotina Delete Union

function DELETE_UNION(*t*)
(u,v) ⟵ *edges_by_time[t]*
linkCutTree.cut(u,v)
edges_by_time.erase(t)
end function

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 a versão retroativa da floresta geradora mínima, que estudaremos no próximo capítulo.

4.1 Ideia

Primeiramente, a árvore geradora minima de uma grafo é um conjunto de arestas que conecta todos os vértices do grafo e tem peso mínimo. Em geral, caso o grafo não seja conexo, a floresta geradora mínima é o conjunto de árvores geradoras mínimas de cada uma das componentes do grafo.

Para resolver este problema, queremos uma estrutura que consiga manter um grafo não direcionado, com pesos nas arestas e que está sempre sofrendo a adição de novas arestas. Essa estrutura também deve ser capaz de calcular, de maneira eficiente, a floresta geradora mínima deste grafo. Desta forma, estamos interessados na seguinte interface:

- `add_edge(u, v, w)`: adiciona no grafo a aresta com pontas em u e v com peso w, possivelmente alterando a floresta geradora mínima.
- `get_msf()`: retorna uma lista com todas as arestas que compõem a floresta geradora mínima no momento atual.
- `get_msf_cost()`: retorna o custo da floresta geradora mínima no momento atual.

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. Tudo isso com um custo $O(\log n)$ para a adição de novas arestas, um custo linearmente proporcional ao tamanho da floresta geradora para a consulta das arestas que a compõem e um custo constante para a consulta do custo total da floresta.

4.2 Estrutura interna

Assim como no union-find retroativo, vamos utilizar a link-cut tree como a estrutura interna da solução deste problema. Para isso, queremos que a link-cut tree seja utilizada para manter a floresta geradora mínima 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 geradora mínima.

Um detalhe importante é que, para essa implementação, necessitamos de uma maneira de consultar qual a aresta com maior custo no caminho entre dois vértices na árvore, não apenas o seu respectivo valor. Para isso, modificamos a implementação da link-cut tree para incluir um novo parâmetro opcional `id` na rotina `link`, além de um novo método `maximum_edge_id`, que retorna o `id` da maior aresta 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 a floresta geradora minima, assim como um inteiro `current_msf_cost`, que armazena o respectivo custo. Estes atributos nos permitem responder de maneira eficiente as consultas de nossa estrutura, como mostraremos a seguir.

4.3 Consultas Get MSF e Get MST Cost

Primeiramente, para realizarmos a consulta acerca da composição da floresta geradora mínima, simplesmente percorremos a lista dos `id`'s das arestas que compõem a floresta e criamos uma nova lista com as arestas em si, utilizando o mapeamento fornecido pelo `edges_by_id`.

Programa 4.1 Consulta Get MSF

```
function GET_MSF
    msf ← []
    for each id in current_msf do
        msf.append(edges_by_id[id])
    end for
    return msf
end function
```

Já a consulta sobre o custo da floresta geradora mínima pode ser facilmente respondida retornando o inteiro `current_msf_cost` mantido pela rotina `add_edge`, explicada na seção a seguir.

Com isso, a primeira consulta tem um custo proporcional a O(m), onde m é o número de arestas inseridas no grafo, pois no pior caso o grafo pode ser a própria floresta geradora mínima, e a segunda consulta tem um custo O(1).

Programa 4.2 Consulta Get MSF Cost

```
function GET_MSF_COST
    return current_msf_cost
end function
```

4.4 Rotina Add Edge

Como a parte mais importante da nossa estrutura, a rotina `add_edge(u, v, w)` é responsável por adicionar uma nova aresta e ao grafo, possivelmente modificando a sua respectiva floresta geradora mínima. Para isso, vamos checar se a aresta deve ser adicionada à link-cut tree ou não, pois caso contrario, simplesmente a descartamos. Desta forma, este processo pode ser divido em duas partes.

A primeira delas consiste em verificar se u e v pertencem a componentes distintas do grafo. Neste caso, simplesmente adicionamos a aresta e na floresta geradora mínima, o que representa uma ligação entre as árvores geradoras mínimas em que u e v pertencem.

Caso u e v façam parte da mesma componente, devemos decidir se essa aresta e deve ou não substituir alguma aresta na árvore geradora mínima que acomoda estes vértices. Note que, ao adicionar essa nova aresta na árvore, estamos criando um ciclo, que consiste em todas as arestas no caminho simples de u até v mais e. Ademais, a adição da aresta e 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, preservando a estrutura de árvore e contribuindo para uma diminuição de seu custo total.

Programa 4.3 Rotina Add Edge

```
function ADD_EDGE(u,v,w)
    edge_id ← create_unique_edge_id()
    edges_by_id[edge_id] ← new edge(u, v, w, edge_id)
    if ¬linkCutTree.is_connected(u, v) then
        linkCutTree.link(u, v, w, edge_id)
        current_msf.append(edge_id)
        current_msf_cost += w
    else if linkCutTree.maximum_edge(u, v) > w then
        maximum_edge_id ← linkCutTree.maximum_edge_id(u, v)
        maximum_edge ← edges_by_id[maximum_edge_id]
        linkCutTree.cut(maximum_edge.u, maximum_edge.v)
        current_msf.erase(maximum_edge.id)
        current_msf_cost -= maximum_edge.w
        linkCutTree.link(u, v, w, edge_id)
        current_msf.append(edge_id)
        current_msf_cost += w
    end if
end function
```

Com isso, como esta rotina usa apenas os métodos fornecidos pela link-cut tree de uma maneira limitada, podemos concluir que ela tem uma complexidade proporcional $O(\log m)$.

4.5 Rotinas Extras

Por último, vamos falar de duas rotinas que serão uteis para a estrutura apresentada no próximo capítulo. Em particular, ambas possuem o mesmo objetivo, possibilitar consultas acerca da floresta geradora mínima 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_cost_after_operations(edges[])`, que recebem uma lista de arestas e retornam, respectivamente, as arestas que fazem parte da floresta geradora mínima e seu custo caso as arestas da lista fornecida fossem adicionas à estrutura. Desta forma, a execução destes métodos consiste em três etapas: adição das arestas na estrutura; a 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.

Finalmente, com estes métodos em mãos, podemos implementar as rotinas extras que estávamos interessados. Dado que a única diferença entre as duas implementações seria a chamada na segunda linha, vamos mostrar somente o pseudo-código da rotina `get_msf_after_operations`. Além disso, podemos perceber que a complexidade destes métodos é proporcional à $O(q \log m)$, onde q é o número de arestas que queremos adicionar.

Programa 4.4 Rotina Apply Rollback

```
function APPLY_ROLLBACK(operations_list[])
    revert(operations_list)
    for each (operation, edge) in operations_list do
        if operation = link then
            linkCutTree.cut(edge.u, edge.v)
            current_msf.erase(edge.id)
            current_msf_cost -= edge.w
        else
            linkCutTree.link(edge.u, edge.v, edge.w, edge.id)
            current_msf.append(edge.id)
            current_msf_cost += edge.w
        end if
    end for
end function
```

Programa 4.5 Rotina Get MSF After Operations

```
function GET_MSF_AFTER_OPERATIONS(edges[])
    rollback_operations ← apply_add_edge_operations(edges)
    msf ← get_msf()
    apply_rollback(rollback_operations)
    return msf
end function
```

Capítulo 5

Floresta Geradora Mínima retroativa

Neste capitulo, estudaremos a 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 anterior para solucionar o problema, oferecendo uma alternativa mais simples, porém mais limitada, a outras implementações na literatura, como a de HOLM *et al.* (2001).

5.1 Square-root decomposition

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

- `find_sum(l, r)`: encontra a soma de todos os valores no intervalo $[l, r]$;

- `update_value(i, x)`: atualiza o valor da posição i para x.

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 $\Sigma_{j=1}^{i} a_j$ — para respondermos as consultas `find_sum` em tempo constante, porém, acarretando a reconstrução de `prefix_sum` em toda chamada de `update_value`, o que consome $O(n)$.

$$\underbrace{a_1, a_2, a_3, a_4}_{block_1} \quad \underbrace{a_5, a_6, a_7, a_8}_{block_2} \quad \ldots \quad \underbrace{a_{n-3}, a_{n-2}, a_{n-1}, a_n}_{block_x}$$

5.2 Ideia

5.3 Consultas Get MSF e Get MST Cost

5.4 Rotina Add Edge

5.5 Complexidade

Referências

[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, 27).

[Demaine *et al.* 2007] Erik D. Demaine, John Iacono e Stefan Langerman. "Retroactive data structures". Em: *ACM Trans. Algorithms* 3.2 (2007), 13–es. issn: 1549-6325. doi: 10.1145/1240233.1240236. url: https://doi.org/10.1145/1240233.1240236 (citado na pg. 17).

[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).

[Holm *et al.* 2001] Jacob Holm, Kristian de Lichtenberg e Mikkel Thorup. "Polylogarithmic 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. 27).

[Sleator e Tarjan 1981] Daniel D. Sleator e Robert Endre Tarjan. "A data structure for dynamic trees". Em: *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*. STOC '81. Milwaukee, Wisconsin, USA: Association for Computing Machinery, 1981, pgs. 114–122. isbn: 9781450373920. doi: 10.1145/800076.802464. url: https://doi.org/10.1145/800076.802464 (citado na pg. 3).

[Sleator e Tarjan 1985] Daniel D. Sleator e Robert Endre 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 Endre 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).