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 text text text.

Keywords: Keyword1. Keyword2. Keyword3.

Sumário

Capítulo 1

Introdução

Estruturas de dados retroativas bla bla bla

1.1 Retroatividade Parcial

1.2 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 á 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.

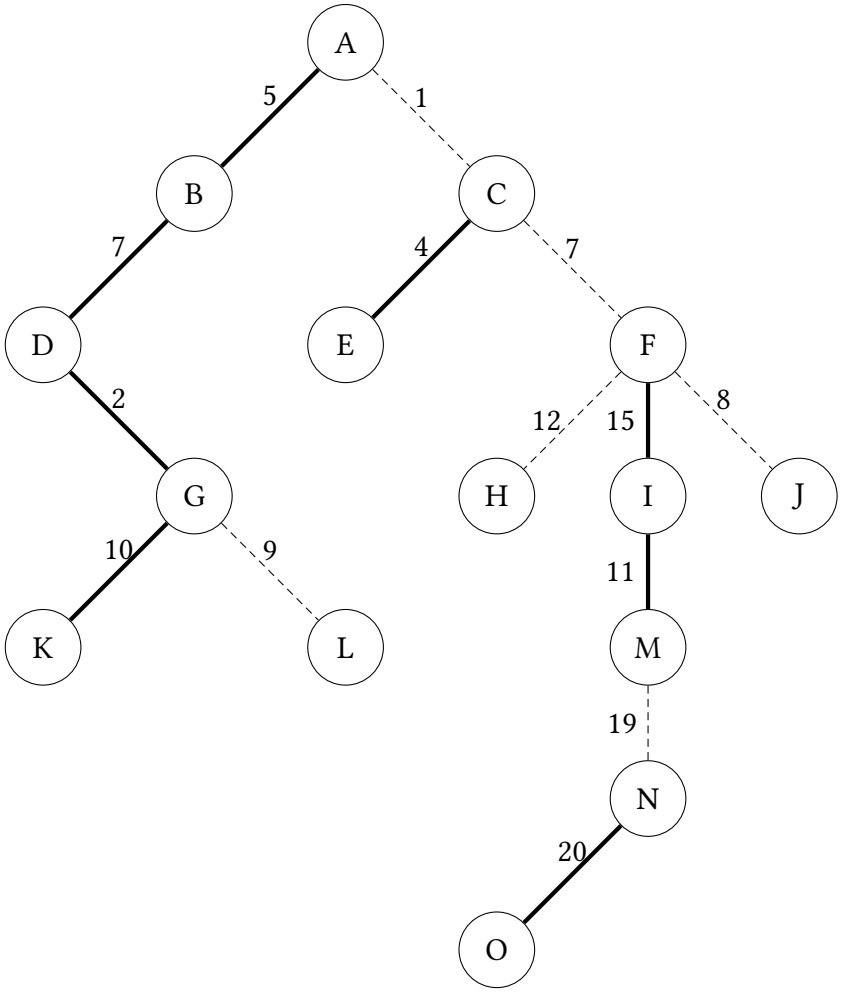


Figura 2.1: *Árvore representada e seus caminhos preferidos. Na figura acima, as arestas escuras representam caminhos preferidos, com isso, temos o seguinte conjunto de caminhos vértice-disjuntos* $\{\langle K, G, D, B, A\rangle, \langle E, C\rangle, \langle M, I, F\rangle, \langle L\rangle, \langle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

- `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 a parte superior do caminho mantido por `last`. Ao final dessa concatenação, temos que `current_root` é o identificador deste caminho que esta sendo construído, e após guardarmos seu valor em `last`, podemos prosseguir para o próximo passo, onde `current_root` agora corresponde ao no imediatamente em cima do caminho preferido que estamos construindo.

Programa 2.1 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 da árvore até u, utilizando `access(u)`. Em seguida, utilizamos a rotina `reverse_path(u)`, que inverte a

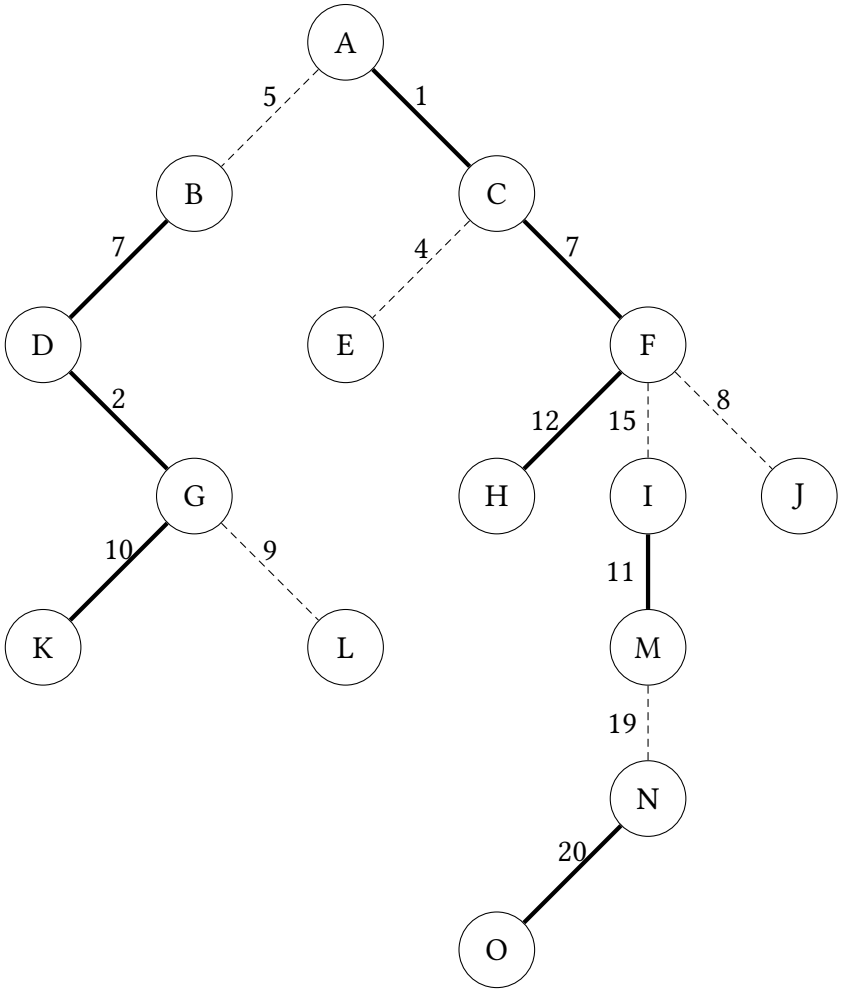


Figura 2.2: *Caminhos preferidos na árvore da figura 2.1 após uma operação de* `access` *no nó* H. *Com isso temos o novo conjunto de caminhos vértice-disjuntos* $\{\langle H, F, C, A\rangle, \langle K, G, D, B\rangle, \langle M, I\rangle, \langle E\rangle, \langle J\rangle, \langle L\rangle, \langle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

orientação deste caminho preferido recém-criado. Tal inversão coloca u como o vértice de menor profundidade da árvore representada, o que se traduz neste sendo a nova raiz.

Programa 2.2 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 primeiro 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 árvore representada viram vértices em nossa representação interna. Com isso, o primeiro passo é criar um vértice que tem seu valor definido como w, vamos chama-lo `uv_edge`. Dessa

forma, criaremos as seguintes conexões: $u \leftrightarrow uv\_edge \leftrightarrow v$.

Inicialmente, colocamos v como raiz de nossa 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 e o concatenando com um caminho que possui v e `uv_edge`.

Programa 2.3 Link

Require: u e v em árvores distintas
  function LINK(u, v, w)
    $uv\_edge := newNode(w)$ ▷ 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, 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 vai de u até u e `uv_edge`. 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 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 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 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
```

E com isso, 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, com uma implementação muito mais limpa do que a da versão original. Portanto, usaremos a splay trees para armazenar os caminhos preferidos.

Uma splay tree é uma árvore binária de busca auto-balanceável. Estas árvores utilizam rotações para 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 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.

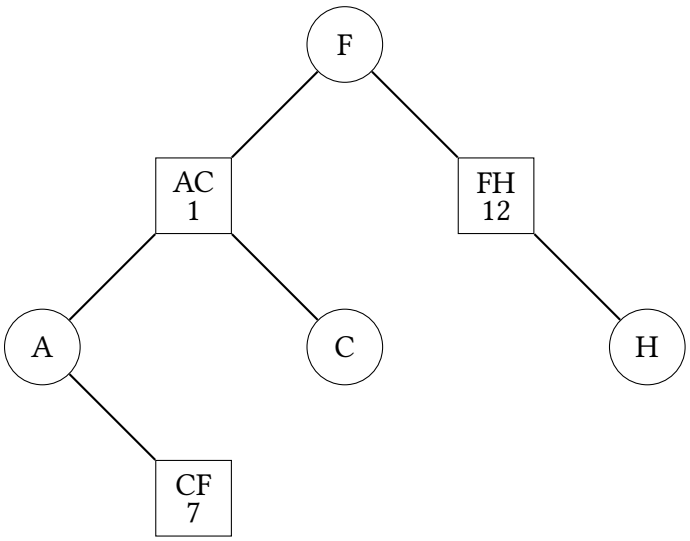


Figura 2.3: *Uma possível configuração da splay tree que armazena o caminho preferido* $\langle H, F, C, A \rangle$ *da figura 2.2, onde* F *é identificador do caminho. Os nós em formato retangular mostram as arestas da árvore representada, com o peso de tal aresta na parte inferior.*

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. Caso o nó em particular seja a raiz da árvore auxiliar, este campo armazena um ponteiro para o vértice que está logo acima do fim deste caminho preferido na árvore representada.
- `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.
- `is_reversed`: valor booleano para sinalizar se a sub-árvore do nó esta com sua ordem invertida ou não, isto é, se todas as posições de filhos esquerdos e direitos estão invertidas nessa sub-árvore.
- `max_subtree_value`: guarda o valor máximo armazenado na sub-árvore do nó.

2.4.1 Splay

Com a estrutura apresentada, podemos partir para a explicação de sua principal operação, a `splay`. Em poucas palavras, este método é responsável por receber um nó e fazer com que ele vire a raiz da splay tree, através de diversas rotações. Ademais, as operações de `splay` contribuem para diminuir a altura da árvore, melhorando o seu consumo de tempo.

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`.

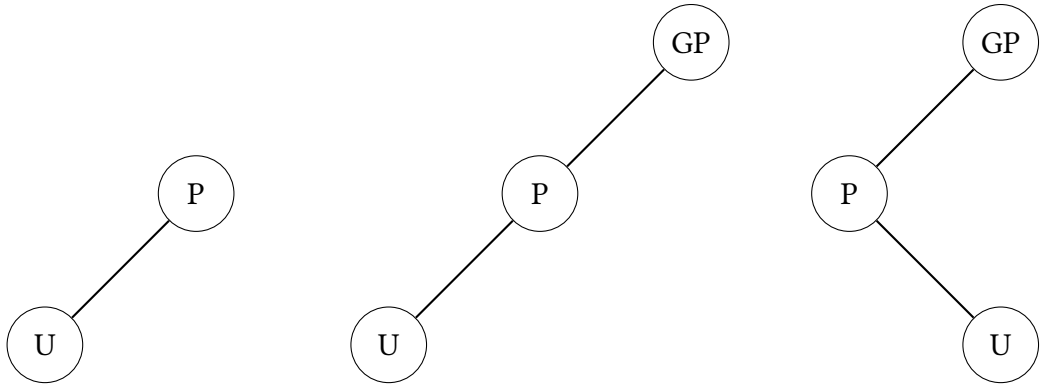
De modo a facilitarmos nossa explicação, chamamos `parent` o pai de um nó u e de `grandparent` o pai de `parent`.

Primeiramente, 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 rotação que diminui 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 necessárias. 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 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.

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. Assim, 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`.



(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 `grandparent`, respectivamente.*

Programa 2.7 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
```

2.4.2 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 e separa caminho preferido que o contem em dois. Para isso, ele 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.

De maneira complementar, temos a rotina `join(u, v)` que recebe dois nós e conca-

Programa 2.8 Split

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

Programa 2.9 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
```

tena 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, a sub-árvore direita inicial, que constitui a parte do caminho preferido de u que foi substituída, ficará com um apontador `parent` para u.

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`.

Programa 2.10 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
```

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 sera propagado na sub-árvore.

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 contem u. Em particular, a primeira rotina retorna o vértice menos profundo do caminho preferido de u, fazendo isso

Programa 2.11 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) ▷ garantido que sera mais rápido na próxima vez
    return smallest_value
end function
```

Programa 2.12 Get Parent Path Node

```
function GET_PARENT_PATH_NODE(u)
    splay(u)
    return u.parent
end function
```

ao acessar o vértice mais à esquerda na splay tree. Já o segundo método é responsável por retornar o vértice imediatamente acima do fim do caminho preferido que contêm 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`.

Programa 2.13 Get Maximum Path Value

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

Por último, temos a função `get_maximum_path_value(u)`, que recebe um vértice identificador de caminho u e retorna o maior valor de uma aresta no caminho preferencial de u, em termos práticos, retorna o valor de `max_subtree_value`.

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

Capítulo 3

Union-Find

Neste capítulo falaremos do union-find retroativo, introduzida por Demaine *et al.* (2007), ela será 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, que não se intersectam. Para isso, ela fornece duas operações principais, são elas:

- `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 de $O(\alpha(n))$, onde n é o número total de elementos nos conjuntos que estamos representando e $\alpha()$ é o inverso da função de Ackermann.

Na versão retroativa, estamos interessados em realizar as operações em uma linha de tempo, isto é, conseguirmos adicionar e remover operações `union` em certos instantes de tempo, assim como checar se dois elementos pertenciam a dado conjunto em um certo instante t.

Para isso, vamos trocar a operação `union(a, b)` 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`, com isso, conseguimos especificar o instante que estamos interessados em consultar.

Note que, em nenhum momento podemos fazer uma operação que seria invalida em

qualquer 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 Consultas Same Set

3.3 Rotinas Create Union e Delete Union

Referências

[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. 15).

[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. 15).

[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. 15).