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.

Sumário

Capítulo 1

Introdução

Estruturas de dados retroativas bla bla bla

1.1 Retroatividade Parcial

1.2 Retroatividade Total

Capítulo 2

Link-Cut Trees

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, 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 criando 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 árvores genéricas que nossa estrutura sintetiza. 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 arvore 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 como seu 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 contem.

TODO: colocar imagem de uma árvore representada e seus caminhos preferidos.

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 contem 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 com v mais profundo que u na árvore representada — e concatena os respectivos caminhos preferidos, transformando $\{u, v\}$ 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 sera 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.

- `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.
- `get_parent_path_node(u)`: retorna 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`.
- `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 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 da raiz da árvore representada e vai até u. 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

```
FUNCAO access(u)
   last ← NULL
   current_root ← u
     ▻ concatena todos os caminhos preferidos de u até a raiz da árvore representada
   enquanto current_root ≠ NULL
      make_identifier(current_root)   ▻ faz virar o identificador de seu caminho preferido
      join(current_root, last)         ▻ concatena um novo pedaço de caminho preferido
      ao caminho em que last é identificador
      last ← current_root              ▻ current_root agora é identificador
      current_root ← get_parent_path_node(current_root)
```

cont ⟶

⟶ *cont*

```
    make_identifier(u)
fim
```

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

2.3.2 Make Root

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

Programa 2.2

```
FUNCAO make_root(u)
    access(u)
    reverse_path(u)
fim
```

2.3.3 Link e Cut

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

```
FUNCAO link(u, v, w)
    uv_edge ← new Node(w)   ▹ cria nó com peso w
      ▹ ligando (v) - (uv_edge)
    make_root(v)
    access(v)
```

cont ⟶

⟶ *cont*

```
    join(v, uv_edge)
      ▹ ligando (uv_edge)-(u)
    make_root(u);
    access(u);
    access(uv_edge);
    join(uv_edge, u)
fim
```

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

```
FUNCAO 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)
fim
```

2.3.4 Is Connected

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

```
FUNCAO is_connected(u, v)
    access(u)
    u_tree_root ← get_path_end_node(u)
```

cont ⟶

```
⟶ cont
    access(v)
    v_tree_root ⟵ get_path_end_node(v)
    devolva (u_tree_root = v_tree_root)
fim
```

2.3.5 Maximum Edge

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

```
FUNCAO maximum_edge(u, v)
    make_root(v);
    access(u);   ▹ cria um caminho preferido entre u e v
    devolva get_maximum_path_value(u)
fim
```

E com isso, encerramos a explicação da implementação dos métodos da Link-Cut Tree.

2.4 Splay Trees

No artigo original, os autores utilizam uma árvore binária enviesada como estrutura para os caminhos preferidos. Porém, quatro anos depois, Sleator e Tarjan (1985) apresentaram a Splay Tree, 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 Tree como uma arvore auxiliar que cuida de manter os caminhos preferidos.

A Splay Tree é uma árvore binária de busca auto-ajustável, capaz de realizar as operações de inserção, deleção e busca. Em particular, para seu uso como árvore auxiliar, estamos interessados na sua operação `splay`, que 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 podermos lidar com os pesos nas arestas da Link-Cut Tree, fazemos com que cada aresta da árvore representada vire um nó na árvore auxiliar. Isso nos permite calcular eficientemente o peso máximo de uma aresta em um caminho preferido, dado que podemos facilmente manter o peso máximo dos vértices em cada sub-árvore de uma Splay Tree.

TODO: colocar imagem de um preferred path e sua respectiva splay tree.

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 á 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.

TODO: colocar imagem de uma Splay antes e depois da inversão, assim como sua árvore representada.

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`: guarda o peso de uma aresta da árvore representada transformado em vértice na árvore auxiliar.
- `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.

TODO: Colocar figura de uma Splay antes e depois do Splay em uma folha

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, recebemos um nó u da Splay Tree, e enquanto este nó não é raiz de nossa árvore, conduzimos a seguinte rotina:

- Verifico se `parent` é a raiz da árvore, caso positivo, vou para o último item.
- Caso contrario, propagamos o 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 a comparação a seguir entre os filhos corretos.
- Em seguida, checamos se `grandparent`, `parent` e u estão em uma orientação de *zig-zig*, *zag-zag* ou *zig-zag*, como exemplificadas na figura abaixo. Dependendo da orientação, fazemos uma rotação em u ou em `parent`, sempre com a ideia de diminuirmos em 1 a profundidade de u.
- Por último, fazemos uma rotação em u, o que o coloca na profundidade que inicialmente estava o nó `grandparent`.

TODO: Colocar figura mostrando configurações de zig-zig, zag-zag e zig-zags.

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.

Programa 2.7

```
FUNCAO splay(u)
  enquanto !u.is_root()   ▷ u não ser raiz da LCT e nem da Splay
    parent ← u.parent
    grandparent ← parent.parent
    se !parent.is_root()
        ▷ propagamos o is_reversed bit do grandparent para o parent para garantir
       que a condicional a seguir usa os filhos corretos para a comparação
      grandparent.push_reversed_bit()
      parent.push_reversed_bit()
      se (grandparent.r_child = parent) = (parent.r_child = u))
          ▷ zig-zig ou zag-zag
        rotate(parent)
      senao
          ▷ zig-zag
        rotate(u)
    rotate(u)
  u.push_reversed_bit()
fim
```

Agora, olharemos a função responsável por realizar as rotações. Basicamente ela pode ser fatorada em quatro partes:

- Primeiramente propagamos as reversões de `grandparent`, `parent` e u, garantindo que estaremos acessando e manipulando os filhos corretos destes respectivos nós.
- Em seguida, caso o `parent` não seja a raiz da Splay Tree, o trocamos de lugar com u, efetivamente colocando u como algum dos filhos de `grandparent`.
- Agora, basta colocarmos `parent` como algum dos filhos u, espelhando a orientação inicial em que u estava como filho de `parent`.
- Por último, recalculamos os valores máximos nas sub-árvores de `parent` e de u.

2.4.2 Split e Join

Temos também dois métodos importantes para a manutenção dos caminhos preferidos, `split` e `join`, responsáveis por separar e concatenar caminhos preferidos, respectivamente.

Programa 2.8

```
FUNCAO split(u)
  se u.l_child ≠ NULL
    u.l_child.parent ⟵ NULL
  u.l_child ⟵ NULLl
fim
```

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.

Programa 2.9

```
FUNCAO join(u, v)
  se v ≠ NULL
    v.parent ⟵ u
  u.r_child ⟵ v
  ▹ atualiza max_subtree_value com o máximo entre o value dos dois filhos e de u
  u.recalculate_max_subtree_value()
fim
```

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, 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

```
FUNCAO reverse_path(u)
   u.is_reversed ⟵ !u.is_reversed   ▹ inverte o valor do bit
   u.push_reversed_bit()   ▹ inverte os filhos de u e propaga a inversão do bit
fim
```

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.

Programa 2.11 Join

```
FUNCAO get_path_end_node(u)
   splay(u)
   smallest_value ⟵ u
   enquanto smallest_value.l_child ≠ NULL
      smallest_value ⟵ smallest_value.l_child
   splay(smallest_value)   ▹ garantido que sera mais rápido na próxima vez
   devolva smallest_value
fim
```

Programa 2.12

```
FUNCAO get_parent_path_node(u)
   splay(u)
   devolva u.parent
fim
```

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 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

```
FUNCAO get_maximum_path_value(u)
   devolva u.max_subtree_value
fim
```

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`.

TODO: Incluir figura de uma Splay antes e depois da reversão do caminho, colocando na legenda o que cada operação retornaria.

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

Referências

[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 (jul. de 1985), pgs. 652–686. issn: 0004-5411. doi: 10.1145/3828.3835. url: https://doi.org/10.1145/3828.3835 (citado na pg. 8).