Link-cut trees e aplicações em estruturas de dados retroativas

Felipe Castro de Noronha **Orientadora: Cristina Gomes Fernandes**

Departamento de Ciência da Computação, Instituto de Matemática e Estatística, Universidade de São Paulo

Resumo

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

Retroatividade

Introduzida por Demaine et al., as **estruturas de dados retroativas** fazem com que cada operação possua um instante de tempo associado, o que permite realizar operações em qualquer estado, passado ou presente, da estrutura. Além disso, é possível remover uma operação que aconteceu em um certo instante de tempo, fazendo com que seus efeitos desapareçam da estrutura.

Link-Cut tree

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

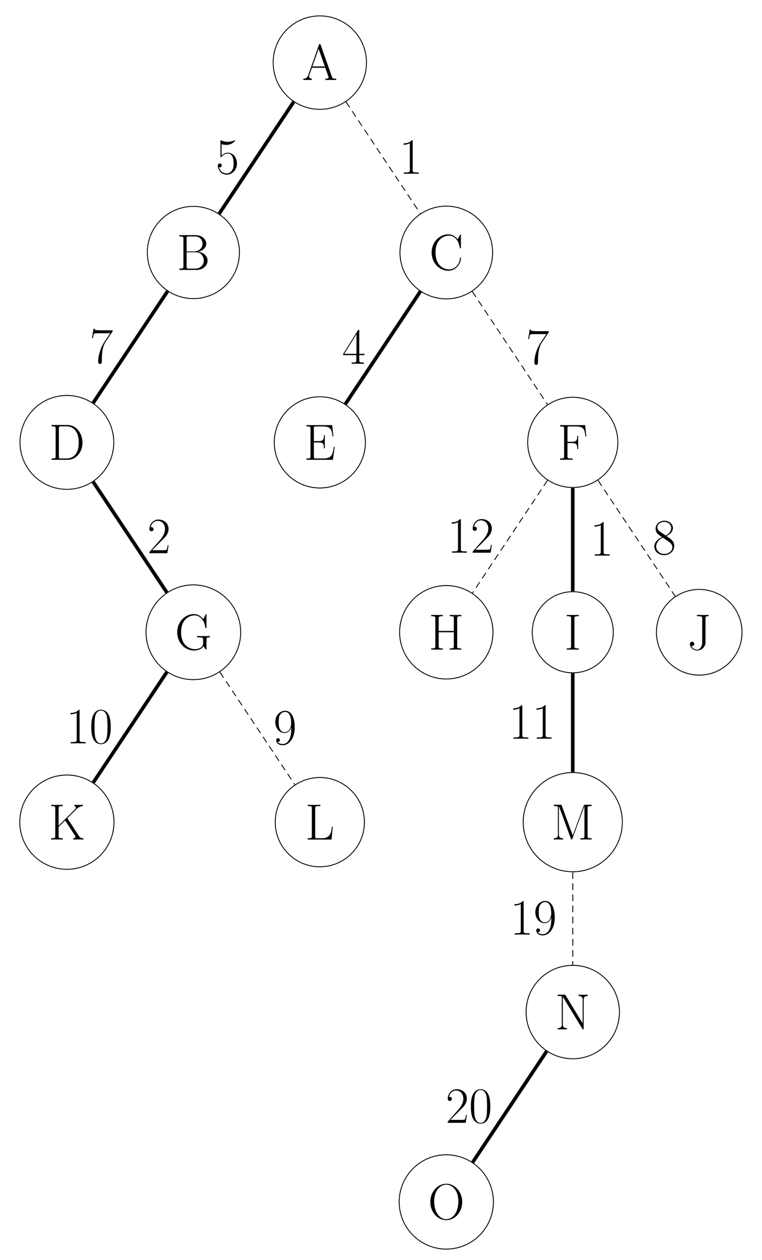

Figura 1: Árvore representada e seus caminhos preferidos, a estruturação interna de uma link-cut tree. Na figura acima, as arestas escuras representam caminhos preferidos, com isso, temos o seguinte conjunto de caminhos vértice-disjuntos $\{\langle K,G,D,B,A\rangle, \langle E,C\rangle, \langle M,I,F\rangle, \langle L\rangle, \langle H\rangle, \langle J\rangle, \langle O,N\rangle\}$.

Ideia: Dividir a floresta em caminhos vértice-disjuntos, chamados **caminhos preferidos**. Cada um desses caminhos é representado na forma de uma *splay tree*, uma estrutura que permite unir e quebrar estes caminhos de forma bastante eficiente.

As link-cut trees fornecem a seguinte interface:

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

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

Union-Find retroativo

O union-find é uma estrutura de dados utilizada para manter uma **coleção de conjuntos disjuntos**, isto é, conjuntos que não se intersectam.

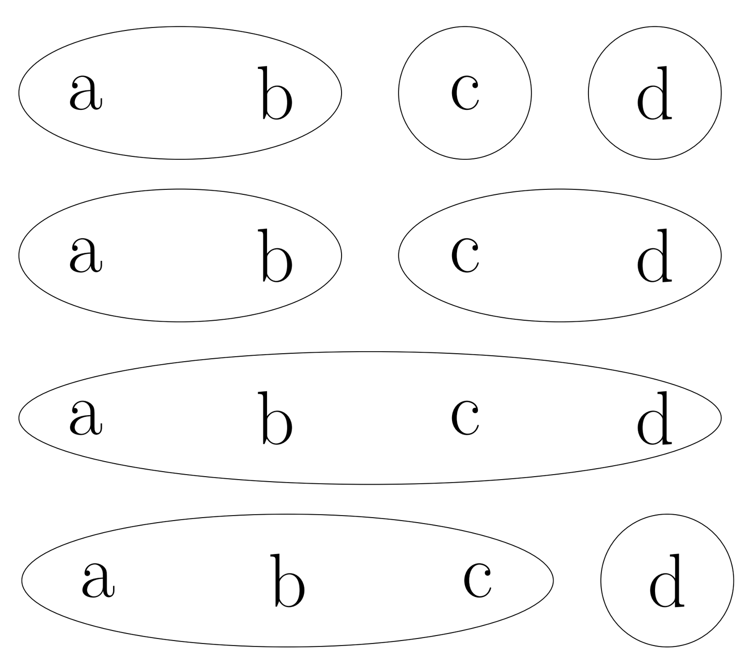

Figura 2: Representação dos conjuntos com os elementos $\{a, b, c, d\}$ após a seguinte sequência de operações: `create_union(a, b, 2)`, `create_union(c, d, 3)`, `create_union(b, c, 4)` e `delete_union(3)`. Cada linha mostra o estado atual da coleção imediatamente após uma operação.

Na sua versão retroativa, implementamos as seguintes operações:

- **`create_union(a, b, t)`**: adiciona a união dos conjuntos que contém a e b no instante de tempo t
- **`same_set(a, b, t)`**: consulta se dois elementos pertenciam ao mesmo conjunto no instante t
- **`delete_union(t)`**: desfaz a união realizada em t

Por exemplo, a Figura 2 mostra o estado de uma coleção de conjuntos disjuntos após quatro operações serem aplicadas. Antes da operação `delete_union(3)`, as consultas `same_set(a, b, 3)` e `same_set(c, d, 3)` retornam `verdadeiro`. Por outro lado `same_set(a, d, 3)` e `same_set(c, d, 3)` retornam `falso` após a chamada da função `delete_union(3)`.

Ideia: Fazer com que os elementos dos conjuntos sejam vértices na floresta mantida por uma link-cut tree, onde cada aresta representa uma operação de `union`. Assim, uma chamada **`create_union(a, b, 3)`** cria uma aresta de valor 3 entre os vértices a e b. Da mesma forma, uma chamada `delete_union(t)` simplesmente exclui a aresta criada no instante t. Para conferir se dois elementos a e b, no instante de tempo t, estão em um mesmo conjunto, basta conferir se eles estão em uma mesma árvore e se o valor da maior aresta no caminho entre eles é menor ou igual a t, o que significa que todas as uniões já foram realizadas no instante consultado.

Floresta geradora mínima retroativa

Como passo inicial temos que introduzir a **floresta geradora mínima incremental**, uma estrutura que utiliza as link-cut trees para fornecer uma maneira eficiente de consulta acerca da floresta geradora mínima de um grafo que está sempre crescendo, isto é, que está sofrendo a inserção de novas arestas.

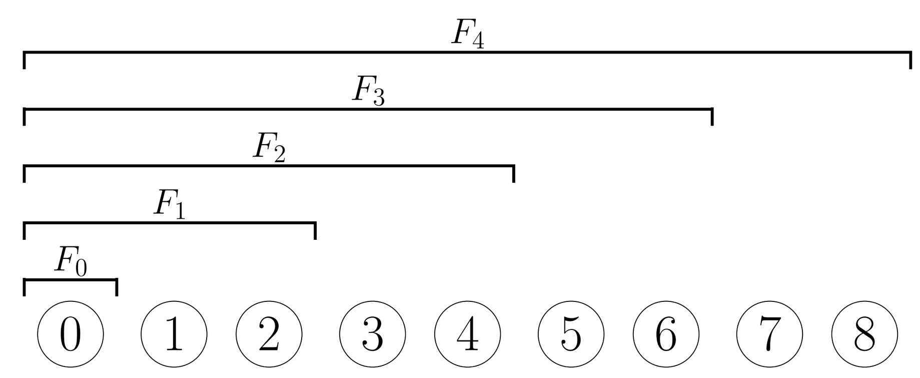

Figura 3: Representação da lista de 8 arestas inseridas. Neste caso, cada bloco tem tamanho 2. Assim, por exemplo, a estrutura F_3 contém todas as arestas adicionadas desde o instante 1 até o instante 6.

A **floresta geradora mínima retroativa** tem a seguinte interface:

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

Ideia: Organizar cada operação retroativa de inserção numa lista ordenada pelo instante de tempo em que a aresta foi inserida. Em seguida, utilizar a técnica de **square-root decomposition** para dividir essa lista em $\sqrt{m}$ blocos, onde m é o número total de operações na lista. Essa divisão — ou como chamamos, reconstrução — vai sendo refeita conforme novas operações de inserção vão sendo adicionadas, a fim de manter o tamanho dos blocos aproximadamente constante. Por último, é necessário distribuir as operações de cada bloco em diferentes florestas geradoras mínimas incrementais, fazendo com que uma consulta acerca do instante de tempo t possa ser realizada de maneira eficiente por uma estrutura que contenha um grafo com um estado próximo ao instante t.

Por último, além da ideia inicial de Andrade Júnior and Duarte Seabra [1] para a floresta geradora mínima retroativa, foi necessário adaptar a ideia apresentada por Demaine et al. [2] para transformar estruturas parcialmente retroativas em estruturas totalmente retroativas. Em particular, realizamos uma melhoria na etapa de reconstrução da estrutura, permitindo que ela seja realizada em tempo $\mathrm{O}(m \log n)$, onde n é o número de vértices na floresta. Adicionalmente, escrevemos um artigo descrevendo essa melhoria, visando a sua publicação em algum veículo da área teórica de ciência da computação.

Informações e contato

Para mais informações, acesse a página do trabalho: `https://linux.ime.usp.br/~felipen/mac0499`

Endereço para contato: `felipe.castro.noronha@usp.br`

Referências

[1] José Wagner de Andrade Júnior and Rodrigo Duarte Seabra. Fully Retroactive Minimum Spanning Tree Problem. *The Computer Journal*, 65(4):973–982, 12 2020. ISSN 0010-4620. doi: 10.1093/comjnl/bxaa135. URL `https://doi.org/10.1093/comjnl/bxaa135`.

[2] Erik D. Demaine, John Iacono, and Stefan Langerman. Retroactive data structures. *ACM Trans. Algorithms*, 2007. ISSN 1549-6325. doi: 10.1145/1240233.1240236. URL `https://doi.org/10.1145/1240233.1240236`.