Capítulo 3

Union-Find Retroativo

Neste capítulo falaremos do union-find retroativo, introduzida por Demaine *et al.* (2007). Ela 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`, com isso, conseguimos 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*.

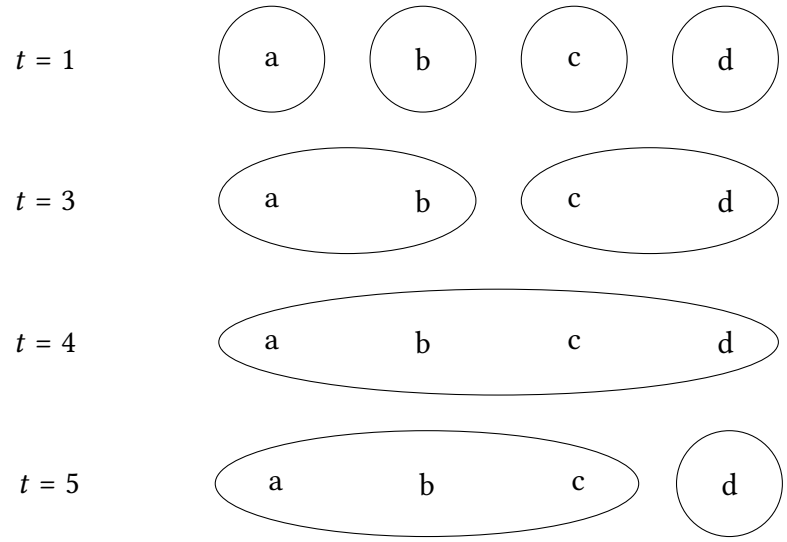

Figura 3.1: *Representação dos conjuntos com os elementos* $\{a, b, c, d\}$ *após a seguinte sequência de operações:* `create_union(a, b, 2)`, `create_union(c, d, 3)`, `create_union(b, c, 4)` *e* `delete_union(3)`.

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.

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 o conjunto que contém estes vértices. Portanto, caso alguma dessas uniões tenha acontecido em um instante maior que t, 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 suportamentos 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
```