Algoritmos e Estruturas de Dados II

Capítulo da Aula 19: Union-Find (Disjoint Set Union)

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br
CEFET-MG - Campus Timóteo

Fevereiro de 2026

1. O Canivete Suíço da Conectividade

O problema de **Conectividade Dinâmica** pergunta: 1. Os elementos A e B estão conectados? 2. Por favor, conecte A e B agora.

Diferente de BFS/DFS que analisam um grafo estático, o **Union-Find** é perfeito para quando as arestas são adicionadas dinamicamente.

- É a estrutura por trás do Kruskal.
- É usado para detectar ciclos em tempo real.
- É usado em processamento de imagens (rotulação de componentes).

2. A Estrutura Básica (Floresta)

Imagine cada elemento como um nó. Cada conjunto é uma **árvore**. A **Raiz** da árvore é o "Líder" (Representante) do conjunto.

- Se `Find(A) == Find(B)`, então eles têm o mesmo líder, logo estão conectados.
- Para fazer `Union(A, B)`, fazemos a raiz de A apontar para a raiz de B (ou vive-versa).

Arrays Necessários

- `parent[i]`: Armazena o pai do nó `i`. Se `parent[i] == i`, ele é a raiz.

3. As Duas Otimizações Críticas

Sem otimizações, a árvore pode virar uma linha reta (Linked List), e o `Find` fica $O(N)$. Com elas, fica praticamente $O(1)$.

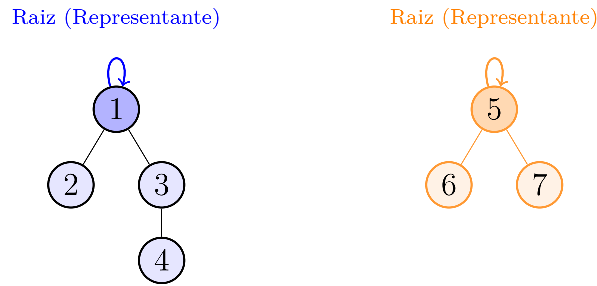

Figure 1: Union-Find: cada conjunto é uma árvore; a raiz é o representante.

3.1. Path Compression (Achatamento de Caminho)

Ao subir de um nó até a raiz durante o `Find`, aproveitamos para reconectar **todos** os nós visitados diretamente à raiz. Isso achata a árvore brutalmente.

```
public int find(int i) {
    if (parent[i] == i)
        return i;

    // Mágica Recursiva: Atualiza o pai para ser a raiz suprema
    return parent[i] = find(parent[i]);
}
```

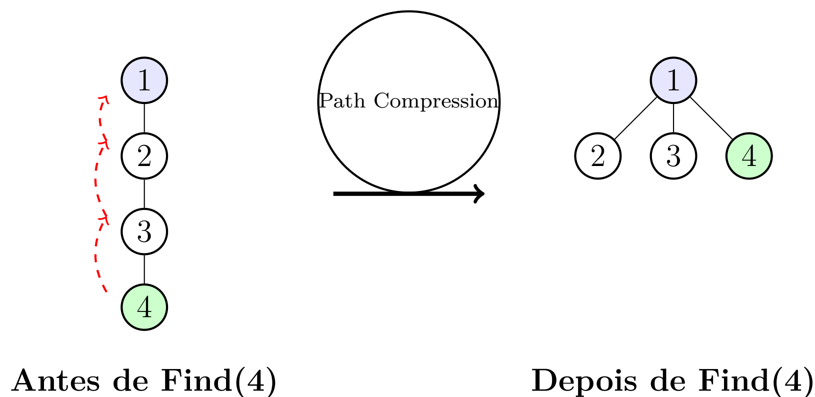

Figure 2: Path compression: ao fazer Find, reconectar todos os nós do caminho à raiz.

3.2. Union by Rank/Size (União Inteligente)

Ao unir duas árvores, sempre pendure a **menor** (menos profunda ou com menos nós) na **maior**. Isso evita que a árvore cresça desnecessariamente em altura.

```
// Variáveis globais
int[] parent;
int[] size; // ou 'rank'

public void union(int i, int j) {
    int rootI = find(i);
    int rootJ = find(j);

    if (rootI != rootJ) {
        // Otimização: A menor entra na maior
        if (size[rootI] < size[rootJ]) {
            int temp = rootI; rootI = rootJ; rootJ = temp;
        }

        parent[rootJ] = rootI; // rootJ agora é filho de rootI
        size[rootI] += size[rootJ];
    }
}
```

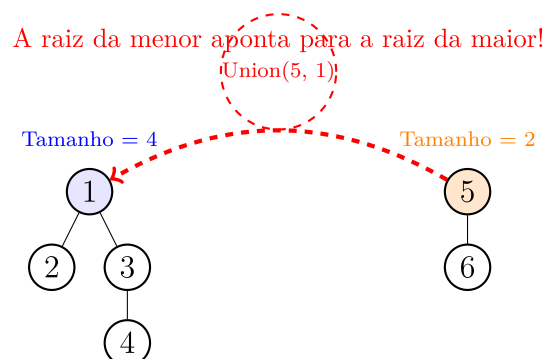

Figure 3: Union by rank: pendurar a árvore menor sob a maior para manter altura baixa.

3.3. Union by Rank vs Union by Size

Existem duas variantes principais para a união inteligente, ambas garantindo $\mathcal{O}(\log N)$ de altura máxima:

- **Union by Size (nosso padrão):** Mantém um array `size[r]` que armazena o número de elementos na árvore. A raiz da árvore **menor** vira filha da árvore **maior**. O `size` é atualizado em cada `Union`. É a variante mais intuitiva e muito útil quando o problema exige saber o tamanho atual do componente.

- **Union by Rank:** Mantém um array `rank[r]` que representa um limite superior (*upper bound*) para a altura da árvore. A raiz de **menor rank** vira filha da de **maior rank**. O rank só é incrementado se as duas raízes unidas possuírem o mesmo rank original. Diferente do *size*, o *rank* não é recalculado durante o Path Compression.

Na prática, ambas as variantes são equivalentes em desempenho algorítmico, operando perfeitamente bem para equilibrar as árvores.

4. Exemplo Passo-a-Passo das Operações

4.1. Inicialização

Considere 6 elementos: $\{0, 1, 2, 3, 4, 5\}$. No início, o construtor isola cada elemento no seu próprio conjunto. Cada nó é a sua própria raiz (aponta para si mesmo).

Índice	0	1	2	3	4	5
parent[]	0	1	2	3	4	5
size[]	1	1	1	1	1	1

Isso representa **6 conjuntos isolados** em uma floresta de raízes unitárias.

4.2. Executando Uniões Simples

Seja a sequência de operações: `Union(1, 2)` e `Union(3, 4)`. Utilizando a estratégia **Union by Size**, em caso de empate de tamanhos, adotamos a convenção em que a segunda árvore se torna sub-árvore da primeira.

- `Union(1, 2)`: o `size[1]` e `size[2]` valem 1. A raiz de 2 passa a apontar para 1 (`parent[2]` $= 1$). O `size[1]` vira 2.
- `Union(3, 4)`: a raiz de 4 passa a apontar para 3 (`parent[4]` $= 3$). O `size[3]` vira 2.

Índice	0	1	2	3	4	5
parent[]	0	**1**	**1**	**3**	**3**	5
size[]	1	**2**	1	**2**	1	1

Agora a floresta possui **4 conjuntos disjuntos**: $\{0\}, \{1, 2\}, \{3, 4\}, \{5\}$.

4.3. Fusão de Árvores Iguais

Considere agora `Union(1, 3)`. Ambas as árvores (com raízes 1 e 3) contam com 2 elementos. A raiz 3 se torna filha direta da raiz 1. O tamanho da árvore com raiz 1 passa a englobar todos os 4 elementos, ou seja, $2 + 2 = 4$.

Índice	0	1	2	3	4	5
parent[]	0	1	1	**1**	3	5
size[]	1	**4**	1	2	1	1

Restam **3 componentes conexos**: $\{0\}, \{1, 2, 3, 4\}, \{5\}$. Observe que `Find(4)` $=$ `Find(2)` $=$ 1, indicando rapidamente que 4 e 2 pertencem à mesma bolha de rede.

4.4. Animando o Find e o Path Compression

Suponha uma chamada para `Find(4)`. Precisamos varrer a cadeia subindo até onde `parent[i]` $== i$. O trajeto no array será: $4 \to 3 \to 1$.

Em uma implementação ingênua, essa busca demandaria 2 saltos. Contudo, na volta recursiva, a mágica do **Path Compression** reconecta **todos os nós visitados** diretamente à raiz suprema (o nó 1). Isso significa que as atribuições ocorrem: `parent[4]` $\leftarrow 1$ e em seguida `parent[3]` $\leftarrow 1$. Na próxima vez que testarmos a conectividade com `Find(4)`, o custo será **1 único salto**, pois a árvore foi comprimida. A otimização beneficia toda a linhagem consultada!

4.5. Trace Completo e Guiado

Para fixar de vez, siga as operações sobre os elementos $\{0, 1, 2, 3, 4, 5, 6\}$ usando **Union by Size + Path Compression**:

1. `Union(0, 1)` $\Rightarrow$ O nó 1 vira filho do 0. Temos `parent[1]` $= 0$, e `size[0]` $= 2$.
2. `Union(2, 3)` $\Rightarrow$ O nó 3 vira filho do 2. Temos `parent[3]` $= 2$, e `size[2]` $= 2$.
3. `Union(0, 2)` $\Rightarrow$ Empate de `size` 2 vs 2. A raiz 2 vira sub-árvore da 0. Agora `parent[2]` $= 0$ e `size[0]` $= 4$.
4. `Union(5, 6)` $\Rightarrow$ O nó 6 vira filho do 5. Temos `parent[6]` $= 5$, e `size[5]` $= 2$.
5. `Union(4, 5)` $\Rightarrow$ Nó 4 tem `size` 1, raiz 5 tem `size` 2. A menor (4) entra na maior. Logo, `parent[4]` $= 5$ e `size[5]` $= 3$.
6. `Union(0, 5)` $\Rightarrow$ Raiz 5 tem `size` 3 e raiz 0 tem `size` 4. O conjunto menor é incorporado: `parent[5]` $= 0$ e a nova raiz 0 atinge `size[0]` $= 7$.
7. `Find(6)` $\Rightarrow$ Rastreio original vai percorrer $6 \to 5 \to 0$. Retorna 0. Ao voltar, o *Path Compression* faz `parent[6]` $= 0$, esmagando esse braço da árvore e otimizando futuras consultas simultaneamente.

5. Implementação Completa e Segura

```java
public class DSU {
    private int[] parent;
    private int[] size;
    private int numSets; // Útil para saber quantos componentes restam

    public DSU(int n) {
        parent = new int[n];
        size = new int[n];
        numSets = n;

        for (int i = 0; i < n; i++) {
            parent[i] = i; // Cada um é seu próprio pai
```

```
            size[i] = 1;
        }
    }

    public int find(int i) {
        if (parent[i] == i) return i;
        return parent[i] = find(parent[i]); // Path Compression
    }

    public void union(int i, int j) {
        int rootI = find(i);
        int rootJ = find(j);

        if (rootI != rootJ) {
            // Union by Size
            if (size[rootI] < size[rootJ]) {
                // i é menor, j manda
                parent[rootI] = rootJ;
                size[rootJ] += size[rootI];
            } else {
                // j é menor, i manda
                parent[rootJ] = rootI;
                size[rootI] += size[rootJ];
            }
            numSets--; // Dois conjuntos viraram um
        }
    }

    public boolean isSameSet(int i, int j) {
        return find(i) == find(j);
    }

    public int getSize(int i) {
        return size[find(i)];
    }
}
```

6. Análise de Complexidade e a Inversa de Ackermann

Ao combinar o balanceamento da *Union by Size* com o esmagamento da *Path Compression*, a complexidade amortizada de cada operação (`Find` ou `Union`) fica em exatos $\mathcal{O}(\alpha(N))$.

O que é $\alpha(N)$? Ela é a **inversa da função de Ackermann**. A função de Ackermann original cresce tão assustadoramente rápido que:

- $Ackermann(4, 3)$ já é vastamente maior que o número estimado de átomos no universo observável.

- Como a função original dispara ao infinito, sua inversa, $\alpha(N)$, cresce de forma **insuportavelmente lenta**.

Conclusão prática para a Engenharia de Software: Para qualquer N cabível na memória dos atuais data centers terrestres, temos que $\alpha(N) \leq 4$. Na prática industrial, podemos afirmar que o DSU roda em **tempo constante estrito $\mathcal{O}(1)$**.

Abordagem	**Find**	**Union**
Ingênua (ligação cega)	$\mathcal{O}(N)$	$\mathcal{O}(N)$
Apenas Union by Size	$\mathcal{O}(\log N)$	$\mathcal{O}(\log N)$
Apenas Path Compression	$\mathcal{O}(\log N)$ amortizado	$\mathcal{O}(\log N)$ amortizado
Size + Compression	$\mathcal{O}(\alpha(N))$	$\mathcal{O}(\alpha(N))$

Table 1: Comparação de Desempenho e o Abismo entre as Abordagens.

7. Aplicações no Mundo Real

Quando você recorrerá ao Union-Find na sua carreira de engenheiro?

- **Algoritmo de Kruskal:** É o motor lógico principal do Kruskal, permitindo rejeitar arestas que fechariam ciclos em frações de microssegundos ao construir uma MST (Minimum Spanning Tree).
- **Detecção Rápida de Ciclos:** Pode testar proativamente se conectar um servidor A ao B vai "fechar um anel" indesejado em topologias de rede expansivas.
- **Processamento de Imagem (Flood Fill):** É muito empregado para delimitar *Componentes Conexos* (blocos contíguos de pixels da mesma cor) em análise e visão computacional.
- **Jogos e Tabuleiros:** Excelente para calcular áreas controladas dinamicamente e avaliar cadeias de conquista em jogos de estratégia em grelha, como o *Go* ou *Hex*.

7.1. DSU + Kruskal: A Parceria Perfeita

No algoritmo de Kruskal para obter a Árvore Geradora Mínima, o Union-Find avalia rapidamente o status das arestas. O protocolo de atuação obedece os passos:

1 Ordene todas as arestas pelo seu peso, de forma ascendente. Isso custa $\mathcal{O}(E \log E)$.

2 Para cada aresta analisada conectando (u, v):

   - Se `Find`$(u) \neq$ `Find`(v): Os pontos estão em aglomerados isolados. **Incorpore** a aresta à MST e efetue a fusão das regiões com `Union(u,v)`.
   - Se `Find`$(u) ==$ `Find`(v): Os pontos já são atingíveis um pelo outro. **Rejeite** a aresta sumariamente, evitando a formação do ciclo que ela traria.

3 Tempo total processual: Ordenação $\mathcal{O}(E \log E)$ somada aos acessos $\mathcal{O}(E \cdot \alpha(V))$. Assintoticamente, a ordenação domina, restando o Kruskal em $\approx \mathcal{O}(E \log E)$.

8. Exercícios

8.1. Exercícios Conceituais

1 Explique por que path compression e union by rank são necessárias para garantir complexidade quase constante. O que acontece sem essas otimizações?

2 Compare Union-Find com BFS/DFS para verificar conectividade. Quando cada abordagem é preferível?

3 Descreva a estrutura de dados subjacente ao Union-Find (floresta de árvores). Como ela evolui durante operações de union?

8.2. Exercícios Analíticos

1 Simule as operações Union-Find passo a passo:

- Union(0,1), Union(2,3), Union(1,2), Find(0), Find(3)

Mostre a estrutura da floresta após cada operação.

2 Para N elementos, qual é o número máximo de operações Union possíveis antes de todos estarem no mesmo conjunto? Qual é o número mínimo de operações Find necessárias para garantir que todos os elementos tenham path compression aplicado?

3 Analise o impacto de usar union by size vs union by rank na altura máxima das árvores resultantes.

8.3. Exercícios de Programação

1 Implemente Union-Find completo com path compression e union by rank. Adicione métodos para:

- Contar o número de conjuntos disjuntos.
- Obter o tamanho do conjunto de um elemento.
- Listar todos os elementos de um conjunto.

2 Use Union-Find para resolver problemas de conectividade dinâmica, como:

- Verificar se dois vértices estão no mesmo componente após adicionar arestas sequencialmente.
- Detectar quando um grafo se torna conexo.

3 Resolva problemas de juiz online que usam Union-Find, como problemas de Kruskal, detecção de ciclos e componentes conexos dinâmicos.