AED2 - Algoritmos e Estr. de Dados II

Aula 19: Union-Find (Disjoint Set Union)

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br

CEFET-MG - Campus Timoteo
Dep. Engenharia de Computação

Fevereiro de 2026

Sumário

1. O Problema da Conectividade Dinâmica

Imagine uma rede de computadores em expansão ou um mapa de cidades.

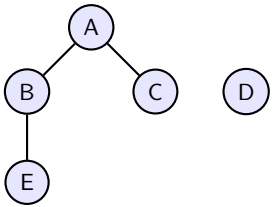
Precisamos responder a duas perguntas **rapidamente**:

1. O Problema da Conectividade Dinâmica

Imagine uma rede de computadores em expansão ou um mapa de cidades.

Precisamos responder a duas perguntas **rapidamente**:

1. **Estão conectados?** Existe um caminho (mesmo indireto) entre as entidades A e B? (*Query*)

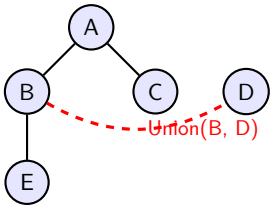


1. O Problema da Conectividade Dinâmica

Imagine uma rede de computadores em expansão ou um mapa de cidades.

Precisamos responder a duas perguntas **rapidamente**:

1. **Estão conectados?** Existe um caminho (mesmo indireto) entre as entidades A e B? (*Query*)
2. **Conectar agora!** Adicione uma nova conexão direta entre A e B. (*Command*)



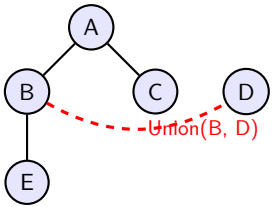
1. O Problema da Conectividade Dinâmica

Imagine uma rede de computadores em expansão ou um mapa de cidades.

Precisamos responder a duas perguntas **rapidamente**:

1. **Estão conectados?** Existe um caminho (mesmo indireto) entre as entidades A e B? (*Query*)
2. **Conectar agora!** Adicione uma nova conexão direta entre A e B. (*Command*)

Diferente de BFS/DFS (projetados para grafos estáticos), precisamos de uma estrutura focada em **arestas adicionadas dinamicamente** ao longo do tempo.



2. A Abstração: Disjoint Set Union (DSU)

O **Union-Find** (ou DSU) é a estrutura desenhada exatamente para este problema. Ele gerencia uma coleção de **conjuntos disjuntos** (onde nenhum elemento pertence a mais de um conjunto simultaneamente).

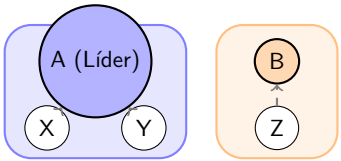
Operações Fundamentais

2. A Abstração: Disjoint Set Union (DSU)

O **Union-Find** (ou DSU) é a estrutura desenhada exatamente para este problema. Ele gerencia uma coleção de **conjuntos disjuntos** (onde nenhum elemento pertence a mais de um conjunto simultaneamente).

Operações Fundamentais

- `Find(i)`: Encontra e retorna o **representante** (ou líder) do conjunto. Serve para verificar se dois elementos estão no mesmo conjunto (`Find(A) == Find(B)`).

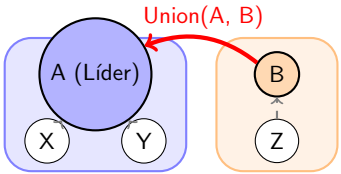


2. A Abstração: Disjoint Set Union (DSU)

O **Union-Find** (ou DSU) é a estrutura desenhada exatamente para este problema. Ele gerencia uma coleção de **conjuntos disjuntos** (onde nenhum elemento pertence a mais de um conjunto simultaneamente).

Operações Fundamentais

- `Find(i)`: Encontra e retorna o **representante** (ou líder) do conjunto. Serve para verificar se dois elementos estão no mesmo conjunto (`Find(A) == Find(B)`).
- `Union(i, j)`: Une os conjuntos que contêm `i` e `j` em um único conjunto.

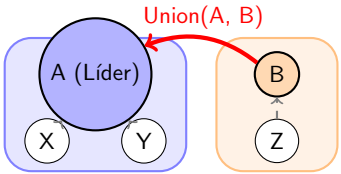


2. A Abstração: Disjoint Set Union (DSU)

O **Union-Find** (ou DSU) é a estrutura desenhada exatamente para este problema. Ele gerencia uma coleção de **conjuntos disjuntos** (onde nenhum elemento pertence a mais de um conjunto simultaneamente).

Operações Fundamentais

- `Find(i)`: Encontra e retorna o **representante** (ou líder) do conjunto. Serve para verificar se dois elementos estão no mesmo conjunto (`Find(A) == Find(B)`).
- `Union(i, j)`: Une os conjuntos que contêm `i` e `j` em um único conjunto.



O Truque da Representação

Cada conjunto é representado internamente como uma **árvore** em uma floresta. A **raiz** de cada árvore é o

3. A Estrutura Básica (Floresta de Arrays)

A magia do DSU é que precisamos de apenas um **array simples** para modelar toda a floresta:

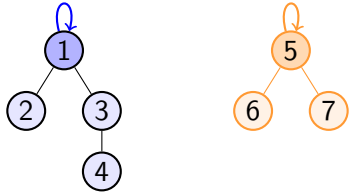


Figure: Floresta Union-Find. Raízes apontam para si mesmas.

3. A Estrutura Básica (Floresta de Arrays)

A magia do DSU é que precisamos de apenas um **array simples** para modelar toda a floresta:

- `parent[i]`: Armazena o índice do "pai" imediato do nó `i`.

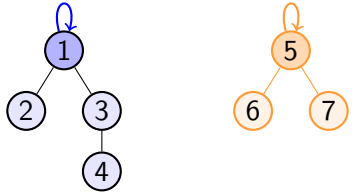


Figure: Floresta Union-Find. Raízes apontam para si mesmas.

3. A Estrutura Básica (Floresta de Arrays)

A magia do DSU é que precisamos de apenas um **array simples** para modelar toda a floresta:

- `parent[i]`: Armazena o índice do "pai" imediato do nó `i`.
- Se `parent[i] == i`, significa que `i` não tem pai: ele é a raiz (o representante).

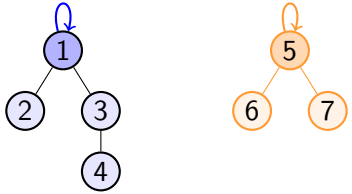


Figure: Floresta Union-Find. Raízes apontam para si mesmas.

3. A Estrutura Básica (Floresta de Arrays)

A magia do DSU é que precisamos de apenas um **array simples** para modelar toda a floresta:

- `parent[i]`: Armazena o índice do "pai" imediato do nó `i`.
- Se `parent[i] == i`, significa que `i` não tem pai: ele é a raiz (o representante).
- **Inicialização:** Inicialmente, como não há arestas, cada elemento é um conjunto isolado. Logo, `parent[i]` $= i$ para todo i.

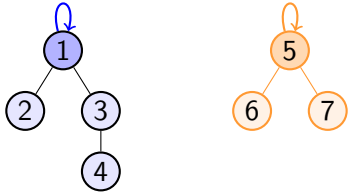


Figure: Floresta Union-Find. Raízes apontam para si mesmas.

4. O Problema da Abordagem Ingênua

Se fizermos o `Union(i, j)` simplesmente mandando a raiz de i apontar cegamente para a raiz de j, podemos criar um gargalo fatal.

4. O Problema da Abordagem Ingênua

Se fizermos o `Union(i, j)` simplesmente mandando a raiz de i apontar cegamente para a raiz de j, podemos criar um gargalo fatal.

4. O Problema da Abordagem Ingênua

Se fizermos o `Union(i, j)` simplesmente mandando a raiz de i apontar cegamente para a raiz de j, podemos criar um gargalo fatal.

A árvore degenera em uma **Lista Encadeada**. Percorrer do nó 5 até a raiz 1 custará $\mathcal{O}(N)$.

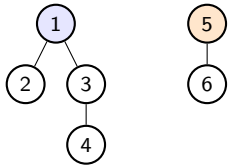
5. Otimização 1: Union by Size (União Inteligente)

A regra tática de fusão: **sempre anexe a árvore menor (menos elementos) à raiz da árvore maior**. Isso preserva o balanceamento e garante uma altura máxima de $\mathcal{O}(\log N)$.

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

    if (rootI != rootJ) {
        if (size[rootI] < size[rootJ]) {
            parent[rootI] = rootJ;
            size[rootJ] += size[rootI];
        } else {
            parent[rootJ] = rootI;
            size[rootI] += size[rootJ];
        }
    }
}
```

Ex: `Union(5, 1)`



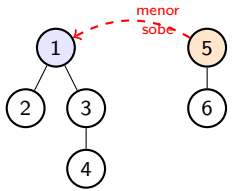
5. Otimização 1: Union by Size (União Inteligente)

A regra tática de fusão: **sempre anexe a árvore menor (menos elementos) à raiz da árvore maior**. Isso preserva o balanceamento e garante uma altura máxima de $\mathcal{O}(\log N)$.

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

    if (rootI != rootJ) {
        if (size[rootI] < size[rootJ]) {
            parent[rootI] = rootJ;
            size[rootJ] += size[rootI];
        } else {
            parent[rootJ] = rootI;
            size[rootI] += size[rootJ];
        }
    }
}
```

Ex: `Union(5, 1)`



A árvore laranja virou sub-árvore da azul!

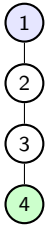
6. Otimização 2: Path Compression (Achatamento)

A sacada genial: durante o `Find`, ao subir até a raiz, aproveitamos a volta na recursão para **reapontar todos os nós visitados diretamente para a raiz suprema**.

```
public int find(int i) {
    if (parent[i] == i)
        return i; // Caso base: chegou na raiz

    // Mágica Recursiva: Atualiza o pai para
    // a raiz suprema encontrada, esmagando
    // a árvore e deixando-a plana!
    return parent[i] = find(parent[i]);
}
```

Ex: `Find(4)`



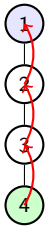
6. Otimização 2: Path Compression (Achatamento)

A sacada genial: durante o `Find`, ao subir até a raiz, aproveitamos a volta na recursão para **reapontar todos os nós visitados diretamente para a raiz suprema**.

```
public int find(int i) {
    if (parent[i] == i)
        return i; // Caso base: chegou na raiz

    // Mágica Recursiva: Atualiza o pai para
    // a raiz suprema encontrada, esmagando
    // a árvore e deixando-a plana!
    return parent[i] = find(parent[i]);
}
```

Ex: `Find(4)`



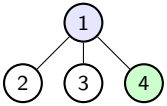
6. Otimização 2: Path Compression (Achatamento)

A sacada genial: durante o `Find`, ao subir até a raiz, aproveitamos a volta na recursão para **reapontar todos os nós visitados diretamente para a raiz suprema**.

```
public int find(int i) {
    if (parent[i] == i)
        return i; // Caso base: chegou na raiz

    // Mágica Recursiva: Atualiza o pai para
    // a raiz suprema encontrada, esmagando
    // a árvore e deixando-a plana!
    return parent[i] = find(parent[i]);
}
```

Ex: `Find(4)`



A árvore foi esmagada na volta da recursão!

As próximas chamadas de `Find` nestes nós custarão exato **1 passo**.

7. Implementação Completa (A Classe DSU)

```java
public class DSU {
    private int[] parent;
    private int[] size;
    private int numSets; // Rastreia quantidade de componentes isolados restantes

    public DSU(int n) {
        parent = new int[n]; size = new int[n]; numSets = n;
        for (int i = 0; i < n; i++) {
            parent[i] = i; size[i] = 1; // Árvore unitária
        }
    }

    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), rootJ = find(j);
        if (rootI != rootJ) {
            if (size[rootI] < size[rootJ]) {
                parent[rootI] = rootJ; size[rootJ] += size[rootI];
```

8. 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`) é $\mathcal{O}(\alpha(N))$.

O que é $\alpha(N)$?

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

8. 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`) é $\mathcal{O}(\alpha(N))$.

O que é $\alpha(N)$?

$\alpha(N)$ 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.

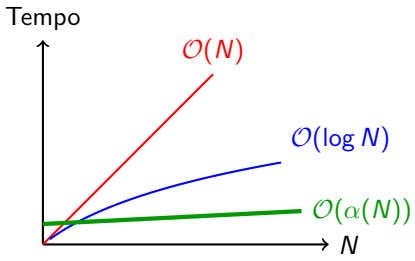
8. Análise de Complexidade e a Inversa de Ackermann

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

O que é $\alpha(N)$?

$\alpha(N)$ 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 quase instantaneamente, sua inversa $\alpha(N)$ cresce de forma **insuportavelmente lenta**.



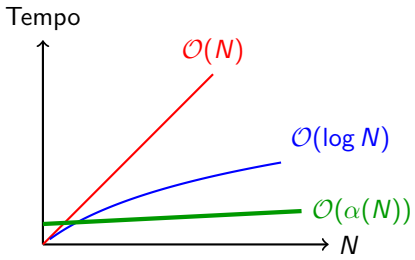
8. Análise de Complexidade e a Inversa de Ackermann

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

O que é $\alpha(N)$?

$\alpha(N)$ 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 quase instantaneamente, sua inversa $\alpha(N)$ cresce de forma **insuportavelmente lenta**.



Conclusão Prática para Engenheiros

Para qualquer N cabível na memória de um cluster de supercomputadores na Terra, temos $\alpha(N) \leq 4$. Na prática, o

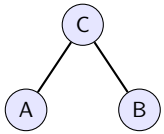
9. Aplicações no Mundo Real

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

9. Aplicações no Mundo Real

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

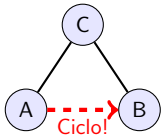
- **Algoritmo de Kruskal:** É o coração tático do Kruskal, permitindo rejeitar ciclos em fração de segundo para montar uma MST.



9. Aplicações no Mundo Real

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

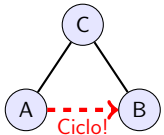
- **Algoritmo de Kruskal:** É o coração tático do Kruskal, permitindo rejeitar ciclos em fração de segundo para montar uma MST.
- **Detecção Rápida de Ciclos:** Testar proativamente se conectar A a B vai "fechar um anel" na sua topologia de rede.

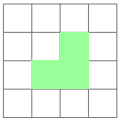


9. Aplicações no Mundo Real

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

- **Algoritmo de Kruskal:** É o coração tático do Kruskal, permitindo rejeitar ciclos em fração de segundo para montar uma MST.
- **Detecção Rápida de Ciclos:** Testar proativamente se conectar A a B vai "fechar um anel" na sua topologia de rede.
- **Processamento de Imagem (Flood Fill):** Localizar rapidamente *Componentes Conexos* (pixels vizinhos com a mesma cor) em visão computacional.



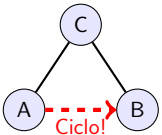


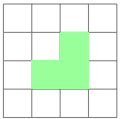
Pixels Conexos

9. Aplicações no Mundo Real

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

- **Algoritmo de Kruskal:** É o coração tático do Kruskal, permitindo rejeitar ciclos em fração de segundo para montar uma MST.
- **Detecção Rápida de Ciclos:** Testar proativamente se conectar A a B vai "fechar um anel" na sua topologia de rede.
- **Processamento de Imagem (Flood Fill):** Localizar rapidamente *Componentes Conexos* (pixels vizinhos com a mesma cor) em visão computacional.
- **Jogos e Tabuleiros:** Processar cadeias de conquista em jogos como Go, Hex ou cálculo de ilhas procedurais.





Pixels Conexos

10. Exemplo Passo-a-Passo: Inicialização

Considere 6 elementos: $\{0, 1, 2, 3, 4, 5\}$. No início, cada um é seu próprio conjunto.

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

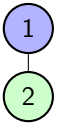
6 conjuntos isolados $\Rightarrow \{0\}, \{1\}, \{2\}, \{3\}, \{4\}, \{5\}$. Cada no e raiz (aponta para si mesmo).

11. Exemplo: Union(1, 2) e Union(3, 4)

Passo 1: Union(1, 2)

11. Exemplo: Union(1, 2) e Union(3, 4)

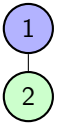
Passo 1: Union(1, 2)



$parent[2] = 1$, $size[1] = 2$

11. Exemplo: Union(1, 2) e Union(3, 4)

Passo 1: Union(1, 2)

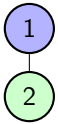


$parent[2] = 1$, $size[1] = 2$

Passo 2: Union(3, 4)

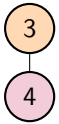
11. Exemplo: Union(1, 2) e Union(3, 4)

Passo 1: Union(1, 2)



$parent[2] = 1$, $size[1] = 2$

Passo 2: Union(3, 4)



$parent[4] = 3$, $size[3] = 2$

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

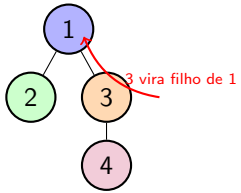
4 conjuntos: $\{0\}$, $\{1, 2\}$, $\{3, 4\}$, $\{5\}$

12. Exemplo: Union(1, 3) — Fusao de Arvores Iguais

Ambas as arvores tem $size = 2$. A regra: quando empata, a **segunda raiz** se torna filha da primeira.

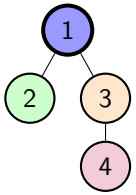
12. Exemplo: Union(1, 3) — Fusao de Arvores Iguais

Ambas as arvores tem *size* = 2. A regra: quando empata, a **segunda raiz** se torna filha da primeira.



12. Exemplo: Union(1, 3) — Fusao de Arvores Iguais

Ambas as arvores tem *size* = 2. A regra: quando empata, a **segunda raiz** se torna filha da primeira.

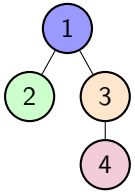


indice	0	1	2	3	4	5
parent[]	0	1	1	**1**	3	5
size[]	1	**4**	1	2	1	1

3 conjuntos: $\{0\},\ \{1,2,3,4\},\ \{5\}$ $\Rightarrow$ Find(4) = Find(2) = 1 (**mesmo conjunto!**)

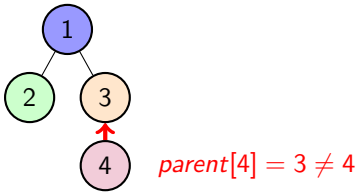
13. Animacao do Find: Caminhando ate a Raiz

Find(4): Precisamos subir a cadeia de pais ate encontrar $parent[i] = i$.



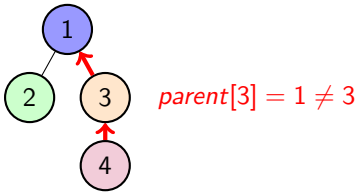
13. Animacao do Find: Caminhando ate a Raiz

Find(4): Precisamos subir a cadeia de pais ate encontrar $parent[i] = i$.



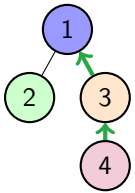
13. Animacao do Find: Caminhando ate a Raiz

Find(4): Precisamos subir a cadeia de pais ate encontrar $parent[i] = i$.



13. Animacao do Find: Caminhando ate a Raiz

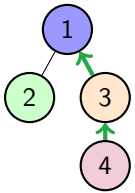
Find(4): Precisamos subir a cadeia de pais ate encontrar $parent[i] = i$.



$parent[1] = 1$ ✓ **Raiz!**

13. Animacao do Find: Caminhando ate a Raiz

Find(4): Precisamos subir a cadeia de pais ate encontrar $parent[i] = i$.



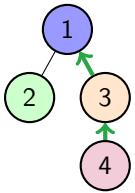
$parent[1] = 1$ ✓ **Raiz!**

Sem Path Compression: 2 saltos

Custo proporcional a **profundidade** do no. Imagine uma cadeia de 1000 nos...

13. Animacao do Find: Caminhando ate a Raiz

Find(4): Precisamos subir a cadeia de pais ate encontrar $parent[i] = i$.



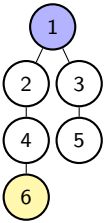
$parent[1] = 1$ ✓ **Raiz!**

Sem Path Compression: 2 saltos

Custo proporcional a **profundidade** do no. Imagine uma cadeia de 1000 nos...

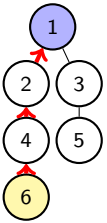
14. Path Compression: Antes vs Depois (Animacao)

ANTES do Find(6)



14. Path Compression: Antes vs Depois (Animacao)

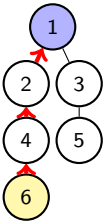
ANTES do Find(6)



3 saltos!

14. Path Compression: Antes vs Depois (Animacao)

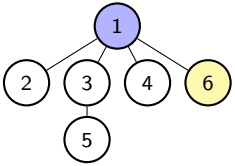
ANTES do Find(6)



3 saltos!

$\Rightarrow$

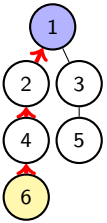
DEPOIS do Find(6)



1 salto direto!

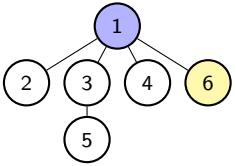
14. Path Compression: Antes vs Depois (Animacao)

ANTES do Find(6)



3 saltos!

DEPOIS do Find(6)



1 salto direto!

Efeito Colateral Positivo

Path Compression nao "conserta" apenas o no buscado — **todos os nos no caminho** sao reapontados! Os nos 6, 4 **e** 2 agora sao filhos diretos da raiz 1.

15. Union by Rank vs Union by Size

Existem duas variantes da "uniao inteligente". Ambas garantem $\mathcal{O}(\log N)$ de altura.

Union by Size (nosso padrao)

- Mantem $size[r]$ = total de elementos
- A raiz da arvore **menor** vira filha da **maior**
- *size* e atualizado a cada Union
- **Mais intuitivo** e util quando precisamos saber o tamanho dos conjuntos

Union by Rank

- Mantem $rank[r] \approx$ altura estimada
- A raiz de **menor rank** vira filha da de **maior rank**
- Rank so incrementa se ambos forem iguais
- Path Compression **nao altera** o rank (e um upper bound)

Qual escolher?

Na pratica, ambas sao equivalentes em desempenho. Use **Union by Size** quando precisar saber $|S_i|$, e **Rank** quando so importa a conectividade.

16. Exercicio Guiado: Trace Completo

Execute as operacoes sobre $\{0, 1, 2, 3, 4, 5, 6, 7\}$ com Union by Size + Path Compression.

16. Exercicio Guiado: Trace Completo

Execute as operacoes sobre $\{0, 1, 2, 3, 4, 5, 6, 7\}$ com Union by Size + Path Compression.

1. Union(0, 1) $\Rightarrow p[1] = 0, s[0] = 2$

16. Exercicio Guiado: Trace Completo

Execute as operacoes sobre $\{0, 1, 2, 3, 4, 5, 6, 7\}$ com Union by Size + Path Compression.

1. Union(0, 1) $\Rightarrow p[1] = 0, s[0] = 2$
2. Union(2, 3) $\Rightarrow p[3] = 2, s[2] = 2$

16. Exercicio Guiado: Trace Completo

Execute as operacoes sobre $\{0, 1, 2, 3, 4, 5, 6, 7\}$ com Union by Size + Path Compression.

1. Union(0, 1) $\Rightarrow p[1] = 0, s[0] = 2$
2. Union(2, 3) $\Rightarrow p[3] = 2, s[2] = 2$
3. Union(0, 2) $\Rightarrow p[2] = 0, s[0] = 4$

16. Exercicio Guiado: Trace Completo

Execute as operacoes sobre $\{0, 1, 2, 3, 4, 5, 6, 7\}$ com Union by Size + Path Compression.

1. Union(0, 1) $\Rightarrow p[1] = 0, s[0] = 2$
2. Union(2, 3) $\Rightarrow p[3] = 2, s[2] = 2$
3. Union(0, 2) $\Rightarrow p[2] = 0, s[0] = 4$
4. Union(5, 6) $\Rightarrow p[6] = 5, s[5] = 2$

16. Exercicio Guiado: Trace Completo

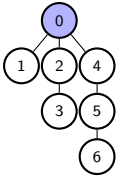
Execute as operacoes sobre $\{0, 1, 2, 3, 4, 5, 6, 7\}$ com Union by Size + Path Compression.

1. Union(0, 1) $\Rightarrow p[1] = 0, s[0] = 2$
2. Union(2, 3) $\Rightarrow p[3] = 2, s[2] = 2$
3. Union(0, 2) $\Rightarrow p[2] = 0, s[0] = 4$
4. Union(5, 6) $\Rightarrow p[6] = 5, s[5] = 2$
5. Union(4, 5) $\Rightarrow p[5] = 4, s[4] = 3$

16. Exercicio Guiado: Trace Completo

Execute as operacoes sobre $\{0, 1, 2, 3, 4, 5, 6, 7\}$ com Union by Size + Path Compression.

1. Union(0, 1) $\Rightarrow p[1] = 0, s[0] = 2$
2. Union(2, 3) $\Rightarrow p[3] = 2, s[2] = 2$
3. Union(0, 2) $\Rightarrow p[2] = 0, s[0] = 4$
4. Union(5, 6) $\Rightarrow p[6] = 5, s[5] = 2$
5. Union(4, 5) $\Rightarrow p[5] = 4, s[4] = 3$
6. Union(0, 4) $\Rightarrow p[4] = 0, s[0] = 7$

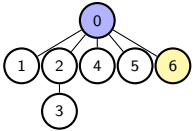


16. Exercicio Guiado: Trace Completo

Execute as operacoes sobre $\{0, 1, 2, 3, 4, 5, 6, 7\}$ com Union by Size + Path Compression.

1. Union(0, 1) $\Rightarrow p[1] = 0, s[0] = 2$
2. Union(2, 3) $\Rightarrow p[3] = 2, s[2] = 2$
3. Union(0, 2) $\Rightarrow p[2] = 0, s[0] = 4$
4. Union(5, 6) $\Rightarrow p[6] = 5, s[5] = 2$
5. Union(4, 5) $\Rightarrow p[5] = 4, s[4] = 3$
6. Union(0, 4) $\Rightarrow p[4] = 0, s[0] = 7$
7. Find(6)? 6$\rightarrow$5$\rightarrow$4$\rightarrow$0 $\mathbf{= 0}$

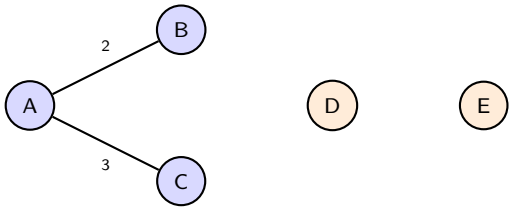
Apos Find(6): nos 6, 5, 4 apontam direto para 0. Proximo Find(6) = **1 salto**!



Path Compression!

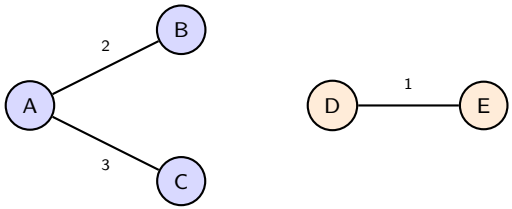
17. DSU + Kruskal: A Parceria Perfeita

No algoritmo de Kruskal, o DSU decide em $\mathcal{O}(\alpha(N))$ se uma aresta **cria ciclo**.



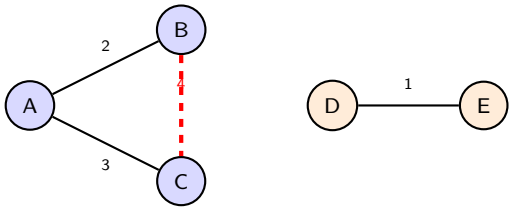
17. DSU + Kruskal: A Parceria Perfeita

No algoritmo de Kruskal, o DSU decide em $\mathcal{O}(\alpha(N))$ se uma aresta **cria ciclo**.



17. DSU + Kruskal: A Parceria Perfeita

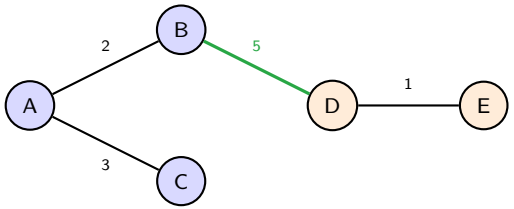
No algoritmo de Kruskal, o DSU decide em $\mathcal{O}(\alpha(N))$ se uma aresta **cria ciclo**.



Find(B)=A, Find(C)=A $\Rightarrow$ **MESMO CONJUNTO! Rejeitar!**

17. DSU + Kruskal: A Parceria Perfeita

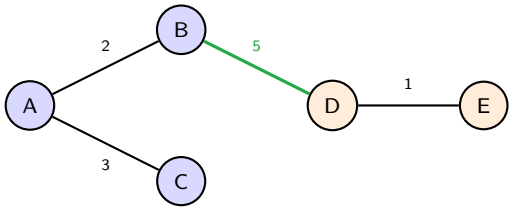
No algoritmo de Kruskal, o DSU decide em $\mathcal{O}(\alpha(N))$ se uma aresta **cria ciclo**.



Find(B)=A, Find(D)=D $\Rightarrow$ **Conjuntos diferentes! Aceitar!**

17. DSU + Kruskal: A Parceria Perfeita

No algoritmo de Kruskal, o DSU decide em $\mathcal{O}(\alpha(N))$ se uma aresta **cria ciclo**.



Find(B)=A, Find(D)=D $\Rightarrow$ **Conjuntos diferentes! Aceitar!**

18. Resumo e Takeaways

Abordagem	**Find**	**Union**
Ingenua (lista/array)	$\mathcal{O}(N)$	$\mathcal{O}(N)$
Union by Size apenas	$\mathcal{O}(\log N)$	$\mathcal{O}(\log N)$
Path Compression apenas	$\mathcal{O}(\log N)$ amort.	$\mathcal{O}(\log N)$ amort.
Size + Compression	$\mathcal{O}(\alpha(N))$	$\mathcal{O}(\alpha(N))$

Licoes Chave

18. Resumo e Takeaways

Abordagem	**Find**	**Union**
Ingenua (lista/array)	$\mathcal{O}(N)$	$\mathcal{O}(N)$
Union by Size apenas	$\mathcal{O}(\log N)$	$\mathcal{O}(\log N)$
Path Compression apenas	$\mathcal{O}(\log N)$ amort.	$\mathcal{O}(\log N)$ amort.
Size + Compression	$\mathcal{O}(\alpha(N))$	$\mathcal{O}(\alpha(N))$

Licoes Chave

1. DSU = floresta de arrays com duas otimizacoes

18. Resumo e Takeaways

Abordagem	**Find**	**Union**
Ingenua (lista/array)	$\mathcal{O}(N)$	$\mathcal{O}(N)$
Union by Size apenas	$\mathcal{O}(\log N)$	$\mathcal{O}(\log N)$
Path Compression apenas	$\mathcal{O}(\log N)$ amort.	$\mathcal{O}(\log N)$ amort.
Size + Compression	$\mathcal{O}(\alpha(N))$	$\mathcal{O}(\alpha(N))$

Licoes Chave

1. DSU = floresta de arrays com duas otimizacoes
2. Path Compression: "esmaga" a arvore no Find

18. Resumo e Takeaways

Abordagem	**Find**	**Union**
Ingenua (lista/array)	$\mathcal{O}(N)$	$\mathcal{O}(N)$
Union by Size apenas	$\mathcal{O}(\log N)$	$\mathcal{O}(\log N)$
Path Compression apenas	$\mathcal{O}(\log N)$ amort.	$\mathcal{O}(\log N)$ amort.
Size + Compression	$\mathcal{O}(\alpha(N))$	$\mathcal{O}(\alpha(N))$

Licoes Chave

1. DSU = floresta de arrays com duas otimizacoes
2. Path Compression: "esmaga" a arvore no Find
3. Union by Size: mantem arvores balanceadas

18. Resumo e Takeaways

Abordagem	**Find**	**Union**
Ingenua (lista/array)	$\mathcal{O}(N)$	$\mathcal{O}(N)$
Union by Size apenas	$\mathcal{O}(\log N)$	$\mathcal{O}(\log N)$
Path Compression apenas	$\mathcal{O}(\log N)$ amort.	$\mathcal{O}(\log N)$ amort.
Size + Compression	$\mathcal{O}(\alpha(N))$	$\mathcal{O}(\alpha(N))$

Licoes Chave

1. DSU = floresta de arrays com duas otimizacoes
2. Path Compression: "esmaga" a arvore no Find
3. Union by Size: mantem arvores balanceadas
4. Juntas: $\alpha(N) \leq 4$ na pratica $\approx \mathcal{O}(1)$

18. Resumo e Takeaways

Abordagem	**Find**	**Union**
Ingenua (lista/array)	$\mathcal{O}(N)$	$\mathcal{O}(N)$
Union by Size apenas	$\mathcal{O}(\log N)$	$\mathcal{O}(\log N)$
Path Compression apenas	$\mathcal{O}(\log N)$ amort.	$\mathcal{O}(\log N)$ amort.
Size + Compression	$\mathcal{O}(\alpha(N))$	$\mathcal{O}(\alpha(N))$

Licoes Chave

1. DSU = floresta de arrays com duas otimizacoes
2. Path Compression: "esmaga" a arvore no Find
3. Union by Size: mantem arvores balanceadas
4. Juntas: $\alpha(N) \leq 4$ na pratica $\approx \mathcal{O}(1)$

Quando NAO usar

- Quando precisa **desfazer** unioes (DSU nao suporta split nativamente)
- Quando o grafo e **direcionado** (use Tarjan/Kosaraju)
- Quando precisa listar **todos os elementos** de um conjunto eficientemente