



# Conjuntos Disjuntos

Algoritmos e Estruturas de Dados

2019/2020



## Conjuntos disjuntos

### Objetivo

- resolver eficientemente o problema da equivalência
- estrutura de dados simples (vector)
- implementação rápida

### Desempenho

- análise complicada

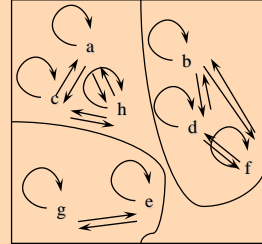
### Uso

- problemas de grafos
- equivalência de tipos em compiladores



## Relações de equivalência

- relação  $R$  definida num conjunto  $S$  se
  - $a R b = V$  ou  $a R b = F \quad \forall a, b \in S$
  - $a R b \Rightarrow a$  está relacionado com  $b$
- propriedades das relações de equivalência
  - reflexiva**  $a R a, \quad \forall a \in S$
  - simétrica**  $a R b \rightarrow b R a$
  - transitiva**  $a R b, b R c \rightarrow a R c$
- exemplos de relações
  - $\leq$ : reflexiva, transitiva; não é simétrica  $\Rightarrow$  não é de equivalência
  - “pertencer ao mesmo país” ( $S$  é o conjunto das cidades): reflexiva, simétrica, transitiva  $\Rightarrow$  relação de equivalência
- classe de equivalência de  $a \in S$ 
  - subconjunto de  $S$  que contém os elementos relacionados com  $a$
  - relação de equivalência induz uma partição de  $S$ : cada elemento pertence exactamente a uma classe



## Problema da equivalência dinâmica

**R: relação de equivalência**

Problema: dados  $a$  e  $b$ , determinar se  $a R b$

Solução: relação armazenada numa matriz bidimensional de booleanos

$\Rightarrow$  resposta em tempo constante

Dificuldade: relações definidas implicitamente

$\{a1, a2, a3, a4, a5\}$  (25 pares)

$a1 R a2, a3 R a4, a5 R a1, a4 R a2 \Rightarrow$  todos relacionados

° pretende-se obter esta conclusão rapidamente

Observação:  $a R b \leftarrow a$  e  $b$  pertencem à mesma classe de equivalência



## Problema da equivalência dinâmica

### Algoritmo abstrato

- Entrada: coleção de  $N$  conjuntos, cada um com um elemento
  - disjuntos ( $S_i \cap S_j = \emptyset$ )
  - só propriedade reflexiva
- Duas operações:
  - **Pesquisa**: devolve o nome do conjunto que contém um dado elemento
  - **União**: substitui dois conjuntos pela respectiva união (preserva a disjunção da coleção)
- Método: acrescentar o par  $a R b$  à relação
  - usa Pesquisa em  $a$  e em  $b$  para verificar se pertencem já à mesma classe de equivalência
    - se sim, o par é redundante
    - se não, aplica União às respetivas classes



## Problema da equivalência dinâmica

### Algoritmo abstrato

- algoritmo **dinâmico** (os conjuntos são alterados por União) e **online** (cada Pesquisa tem que ser respondida antes de o algoritmo continuar)
- valores dos elementos irrelevantes  $\Rightarrow$  basta numerá-los com uma função de dispersão, p.ex
- nomes concretos dos conjuntos irrelevantes  $\Rightarrow$  basta que a igualdade funcione



## Privilegiando a Pesquisa

- **Pesquisa com tempo constante para o pior caso:**
  - implementação: vetor indexado pelos elementos indica nome da classe respetiva
  - Pesquisa é  $O(1)$
  - União(a, b): se Pesquisa(a) = i e Pesquisa(b) = j, pode-se percorrer o vector mudando todos os i's para j  $\Rightarrow O(N)$
  - para N-1 Uniões (o máximo até ter tudo numa só classe)  $\Rightarrow O(N^2)$
- **Melhoramentos:**
  - colocar os elementos da mesma classe numa lista ligada para saltar diretamente de uns para os outros ao fazer a alteração do nome da classe (mas mantém o tempo do pior caso em  $O(N^2)$ )
  - registar o tamanho da classe de equivalência para alterar sempre a mais pequena; cada elemento é alterado no máximo  $\log N$  vezes (cada fusão duplica a classe)  $\Rightarrow$  com N-1 fusões e M Pesquisas  $O(N \log N + M)$

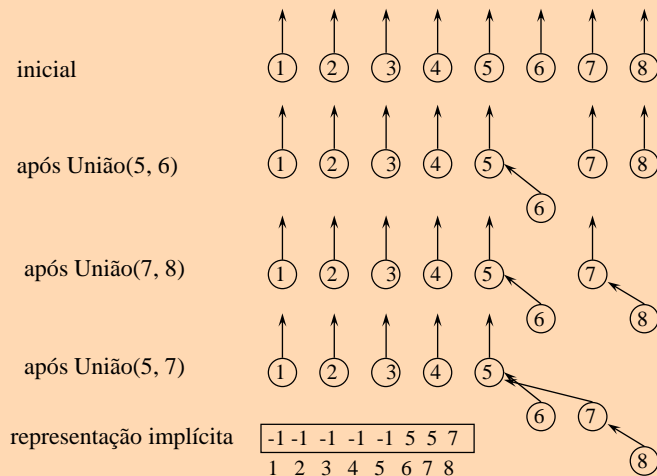


## Privilegiando a União

- **Representar cada conjunto como uma árvore**
  - a raiz serve como nome do conjunto
  - inicialmente, cada conjunto contém um elemento
  - árvores podem não ser binárias: cada nó só tem um apontador para o pai
  - as árvores são armazenadas implicitamente num vetor
    - p[i] contém o número do pai do elemento i
    - se i for raiz p[i] = 0
- **União:** fusão de duas árvores
  - colocar a raiz de uma a apontar para a outra ( $O(1)$ )
  - convenção: União(x, y) tem como raiz x
- **Pesquisa(x)** devolve a raiz da árvore que contém x
  - tempo proporcional à profundidade de x (N-1 no pior caso)
- Não é possível ter tempo constante simultaneamente para União e Pesquisa



## Exemplo



## Implementação

```
class DisjointSets
{
    vector<int> s;
public:
    explicit DisjointSets(int numElements);
    int find(int x) const;
    void unionSets(int root1, int root2);
};
```

### Construtor

```
DisjointSets::DisjointSets (int numElements) :
    s{numElements,-1}
{ }
```

### União (fraco)

```
void DisjointSets::unionSets (int root1,
                              int root2)
{
    s[root2] = s[root1];
}
```

### Pesquisa simples

```
int DisjointSets::find(int x) const
{
    if ( s[x] == -1 )
        return x;
    else
        return find(s[x]);
}
```



## Análise no caso médio \*

- Como definir "médio" relativamente à operação União?

Depende do modelo escolhido: (ver exemplo anterior, última situação)

- Considerando como equiprováveis as Uniões entre diferentes árvores
  - restam 5 árvores, há  $5 \cdot 4 = 20$  resultados equiprováveis da próxima União
  - $2/5$  de hipóteses de envolver a árvore maior
- Outro modelo: considerando como equiprováveis as Uniões entre dois quaisquer elementos de árvores diferentes
  - há 6 maneiras de fundir dois elementos de  $\{1, 2, 3, 4\}$  (não contando simetrias) e 16 maneiras de fundir um elemento de  $\{1, 2, 3, 4\}$  e um de  $\{5, 6, 7, 8\}$
  - probabilidade de a árvore maior estar envolvida:  $16/22$

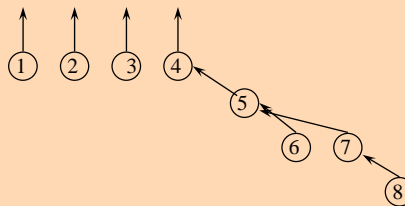
- O tempo médio depende do modelo:  $O(M)$ ,  $O(M \log N)$ ,  $O(MN)$  (o mais realista)

- tempo quadrático é mau, mas evitável



## União melhorada

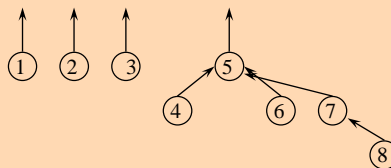
após União(4, 5)  
(altura 3)



### União-por-Tamanho

colocar a árvore menor como sub-árvore da maior (arbitrar em caso de empate)

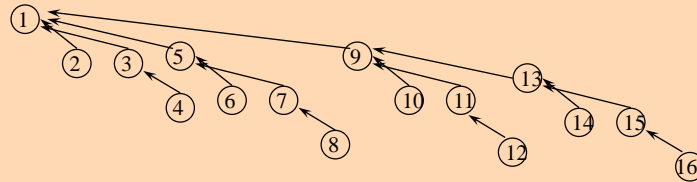
após União(4, 5)  
(altura 2)



## União-por-Tamanho

- Profundidade de cada nó - nunca superior a  $\log N$ 
  - um nó começa por ter profundidade 0
  - cada aumento de profundidade resulta de uma união que produz uma árvore pelo menos com o dobro do tamanho
  - logo, há no máximo  $\log N$  aumentos de profundidade
  - Pesquisa é  $O(\log N)$ ,  $M$  operações é  $O(M \log N)$

Pior caso para  $n=16$  (União entre árvores de igual tamanho)



- Registar a dimensão de cada árvore (na raiz respetiva e com sinal negativo)
  - o resultado de uma União tem dimensão igual à soma das duas anteriores



## União-por-Altura

- Em vez da dimensão, regista-se a altura
  - coloca-se a árvore mais baixa como subárvore da mais alta
  - altura só se altera quando as árvores a fundir têm a mesma altura
- Representação vectorial da situação após União(4, 5)

União-por-Tamanho

-1	-1	-1	5	-5	5	5	7
1	2	3	4	5	6	7	8

União-por-Altura

-1	-1	-1	5	-2	5	5	7
1	2	3	4	5	6	7	8

União (melhorado)

```

void DisjSets::unionSets(int root1, int root2) {
    if ( s[root2] < s[root1] )
        s[root1] = root2;
    else {
        if ( s[root1] == s[root2] )
            --s[root1];
        s[root2] = root1;
    }
}
  
```



## Compressão

- Algoritmo descrito é linear na maior parte das situações, mas no pior caso é  $O(M \log N)$  \*
  - já não é fácil melhorar União: atuar em Pesquisa

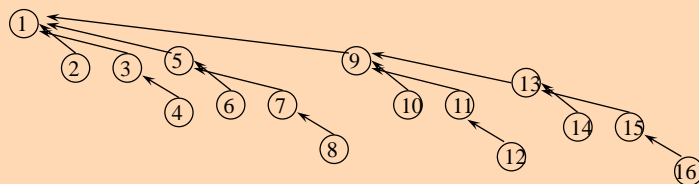
\* suponha: conjuntos estão numa fila, retiram-se 2 elementos da fila, faz-se a união que se coloca na fila

### Compressão do caminho

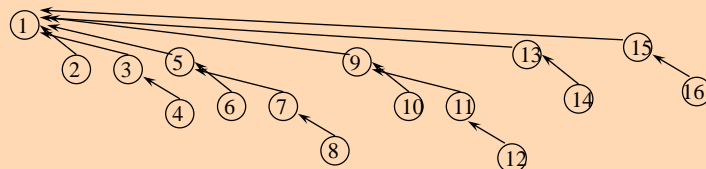
ao executar Pesquisa(x), todos os nós no caminho de x até à raiz ficam com a raiz como pai



## Compressão



Compressão após Pesquisa\_e\_Compressão(15)





## Pesquisa modificada

### Pesquisa com compressão

```
int DisjSets::find(int x) { // não const!
    if ( s[x] <= 0 )
        return x;
    else
        return s[x] = find(s[x]);
}
```

- profundidade de vários nós diminui
- com União arbitrária, a compressão garante M operações, no pior caso, em tempo  $O(M \log N)$
- a compressão é compatível com União-por-Tamanho
- não é completamente compatível com União-por-Altura: não é fácil computar eficientemente as alturas modificadas pela compressão
- não se modificam os valores: passam a ser entendidos como estimativas da altura, designados por nível
- ambos os métodos de União garantem M operações em tempo linear: não é evidente que a compressão traga vantagem em tempo médio: melhora o tempo no pior caso; a análise é complexa, apesar da simplicidade do algoritmo



## Aplicação \*

- Rede de computadores com uma lista de ligações bidirecionais; cada ligação permite a transferência de ficheiros de um computador para o outro
  - é possível enviar um ficheiro de um qualquer nó da rede para qualquer outro?
  - problema deve ser resolvido apresentando as ligações uma de cada vez
- O algoritmo começa por pôr cada computador em seu conjunto
  - o invariante é que dois computadores podem transferir ficheiros se estiverem no mesmo conjunto
  - esta capacidade determina uma relação de equivalência
  - à medida que se lêem as ligações vão-se fundindo os conjuntos
- O grafo da capacidade de transferência é conexo se no fim houver um único conjunto
  - com M ligações e N computadores o espaço requerido é  $O(N)$
  - com União-por-Tamanho e compressão de caminho obtém-se um tempo no pior caso praticamente linear

