Conjuntos disjuntos

Objectivo

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

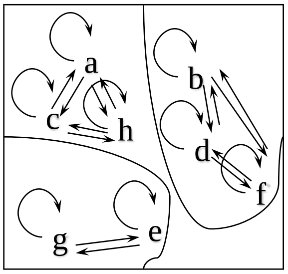
Desempenho

- 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, $\forall$ 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

⇒ 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 ⇒ todos relacionados

° pretende-se obter esta conclusão rapidamente

Observação: a R b ← a e b pertencem à mesma classe de equivalência

Algoritmo abstracto

- ☐ Entrada: colecção de n conjuntos, cada um com seu elemento
  - disjuntos
  - só propriedade reflexiva
- ☐ Duas operações
  - ° Busca - devolve o nome do conjunto que contém um dado elemento
  - ° União - dados dois conjuntos substitui-os pela respectiva união (preserva a disjunção da colecção)
- ☐ Método: acrescentar o par *a R b* à relação
  - ° usa Busca em *a* e em *b* para verificar se pertencem já à mesma classe
    - se sim, o par é redundante
    - se não, aplica União às respectivas classes

° algoritmo **dinâmico** (os conjuntos são alterados por União) e **em-linha** (cada Busca tem que ser respondida antes de o algoritmo continuar)

° valores dos elementos irrelevantes ⇒ basta numerá-los com uma função de dispersão

° nomes concretos dos conjuntos irrelevantes ⇒ basta que a igualdade funcione

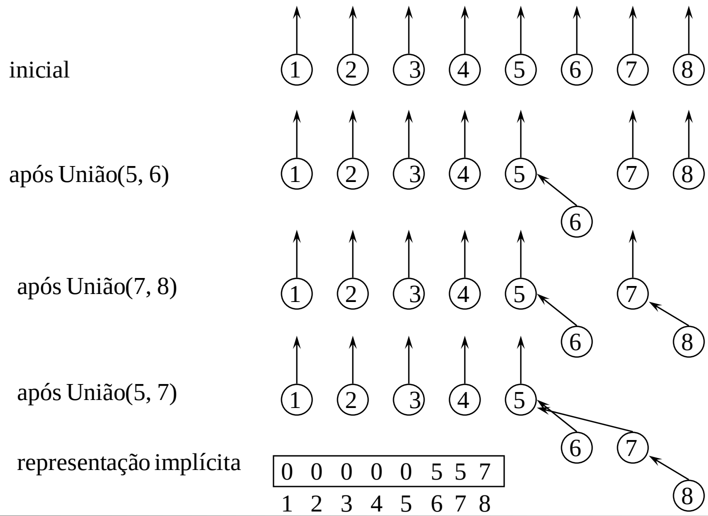
Privilegiando a Busca

- **Busca com tempo constante para o pior caso:**
  - implementação: vector indexado pelos elementos indica nome da classe respectiva
  - Busca fica uma consulta de O(1)
  - União(a, b): se Busca(a) = i e Busca(b) = j, pode-se percorrer o vector mudando todos os i's para j $\Rightarrow$ custo $\Theta(n)$
  - para n-1 Uniões (o máximo até ter tudo numa só classe) $\Rightarrow$ tempo $\Theta(n^2)$
  - se houver $\Omega(n^2)$ Busca, o tempo é O(1) para operação elementar; não é mau
- **Melhoramentos:**
  - colocar os elementos da mesma classe numa lista ligada para saltar directamente de uns para os outros ao fazer a alteração do nome da classe (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 Buscas $O(n \log n + m)$

Privilegiando a União

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

Exemplo



Implementação

construtor

```
Disj_Sets(int Num_Elements)
{
      S_Size = Num_Elements;
      S_Array = new int [S_Size + 1];
      for( int  i=0; i <= S_Size; i++)
         S_Array[i] = 0;
}
```

União (fraco)

```
Void
Unir( int Root1, int Root2 )
{
         S_Array [ Root2 ] = Root1;
}
```

Busca simples

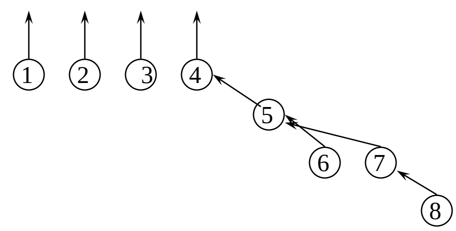
```
int
Busca( int X )
{
         if ( S_Array [ X ] <= 0 )
             return X;
         else
            return Busca( S_Array[ X ] );
}
```

Análise no caso médio

- **Como definir "médio" relativamente à operação União?**
  - depende do modelo escolhido (ver exemplo anterior; última situação)
  - 1 como no exemplo restaram 5 árvores, há 5*4 = 20 resultados equiprováveis da próxima União
    - 2/5 de hipóteses de envolver a árvore maior
  - 2 considerando como equiprováveis as Uniões entre dois quaisquer elementos de árvores diferentes
    - há 6 maneiras de fundir dois elementos de {1, 2, 3, 4} 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: Θ(m), Θ(m log n), Θ(m n)**
  - tempo quadrático é mau, mas evitável

União melhorada

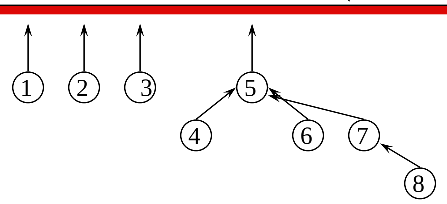
após União(4, 5)
(altura 3)



União-por-Tamanho

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

após União(4, 5)
(altura 2)

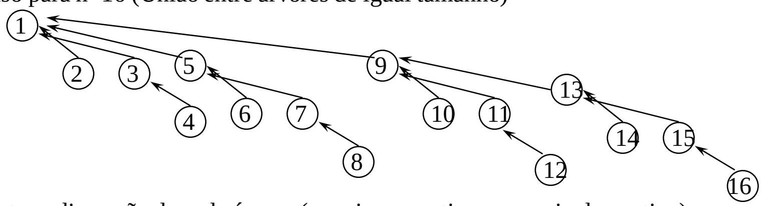


União-por-Tamanho

☐ **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
- Busca 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 respectiva e com sinal negativo)

- o resultado de uma União tem dimensão igual à soma das duas anteriores
- para m operações, dá O(m)

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

0	0	0	5	-2	5	5	7
1	2	3	4	5	6	7	8

União (melhorado)

```
void
União_por_Altura( int Root1, int Root2 )
{
        if (S_Array [ Root2 ] < S_Array[ Root1] )
            S_Array[ Root1 ] = Root2;
        else
        {
          if (S_Array [ Root1 ] == S_Array[ Root2] )
          S_Array [ Root1 ]--;
          S_Array [ Root2 ] = Root1;
        }
}
```

Compressão

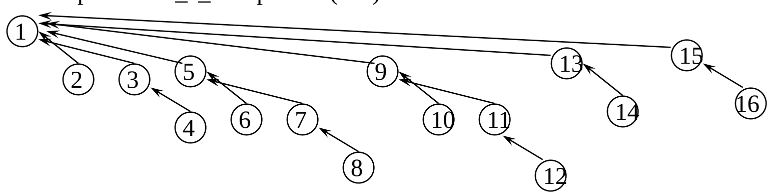
- algoritmo descrito é linear na maior parte das situações; mas no pior caso é O(m log n)

  - já não é fácil melhorar o União : actuar na Busca

Compressão do caminho

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

Compressão após Busca_e_Compressão(15)



Busca modificada

- profundidade de vários nós diminui
- com União arbitrária, a compressão garante m operações, no pior caso, em tempo O(m log n)
- desconhece-se qual é, em média, o comportamento só da compressão

Busca com compressão

```
int Busca ( int X )
{
        if ( S_Array [ X ] <= 0 )
           return X;
        else
          return S_Array[ X ] = Busca ( S_Array[ X ] );
}
```

- 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 bidireccionais; 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?
  - o problema deve ser resolvido em-linha, com as ligações apresentadas 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

Lema (para resultados de complexidade- Rever)

Ao executar uma sequência de Uniões, um nó de nível r tem que ter 2^r descendentes (incluindo o próprio)

Prova por indução

- a base, r = 0 é verdadeira: 2^0 = 1, e uma folha só tem um descendente (o próprio nó)
- seja T uma árvore de nível r com o mínimo número de descendentes e seja x a raiz de T
- suponha-se que a última União que envolveu x foi entre T1 e T2 e que x era a raiz de T1
- se T1 tivesse nível r, T1 seria uma árvore de altura r com menos descendentes do que T, o que contradiz o pressuposto de T ser uma árvore com o número mínimo de descendentes. Portanto, o nível de T1 ≤ r-1. O nível de T2 ≤ nível de T1. Uma vez que o nível de T é r e o nível só pode aumentar devido a T2, segue-se que nível de T2 = r-1. Então também o nível de T1 = r-1.
- pela hipótese de indução, cada uma das árvores tem 2^(r-1) descendentes, dando um total de 2^r e estabelecendo o lema.