Union-Find Disjoint Sets

Prof. **João Paulo** R. R. Leite
joaopaulo@**unifei**.edu.br
ECOE44 & ECOE45 – Maratona de Programação

Union-Find Disjoint Sets

A UFDS é a estrutura de dados que modela uma **coleção de conjuntos disjuntos** (que não se sobrepõem) e possui a habilidade de eficientemente (próximo de O(1)):

- **find**
  - determinar a qual conjunto um determinado elemento pertence;
  - verificar se dois elementos pertencem a um mesmo conjunto;
- **union**
  - e fazer uma operação de união com dois conjuntos diferentes, resultando em um conjunto maior.

Tal estrutura pode ser utilizada, por exemplo, para resolver o problema da busca por componentes conexas em um grafo não direcionado (mas não somente).

Union-Find Disjoint Sets

Essas operações, aparentemente simples, não são suportados eficientemente pela classe **set** da C++ STL, que vimos nas aulas passadas, apesar de seu nome. Manipular um vetor de sets para fazer operações de find e union pode ser custoso e não apresenta o mesmo desempenho da UFDS.

A grande inovação da UFDS é **escolher um elemento "pai"** de cada conjunto disjunto, que servirá como seu **representante** na estrutura. Se conseguirmos garantir que cada conjunto disjunto é representado por um "pai" diferente, as operações tornam-se muito mais simples.

- **find**: verifica se dois elementos têm o mesmo pai.
- **union**: faz com que dois conjuntos passem a ter o mesmo pai.

Union-Find Disjoint Sets

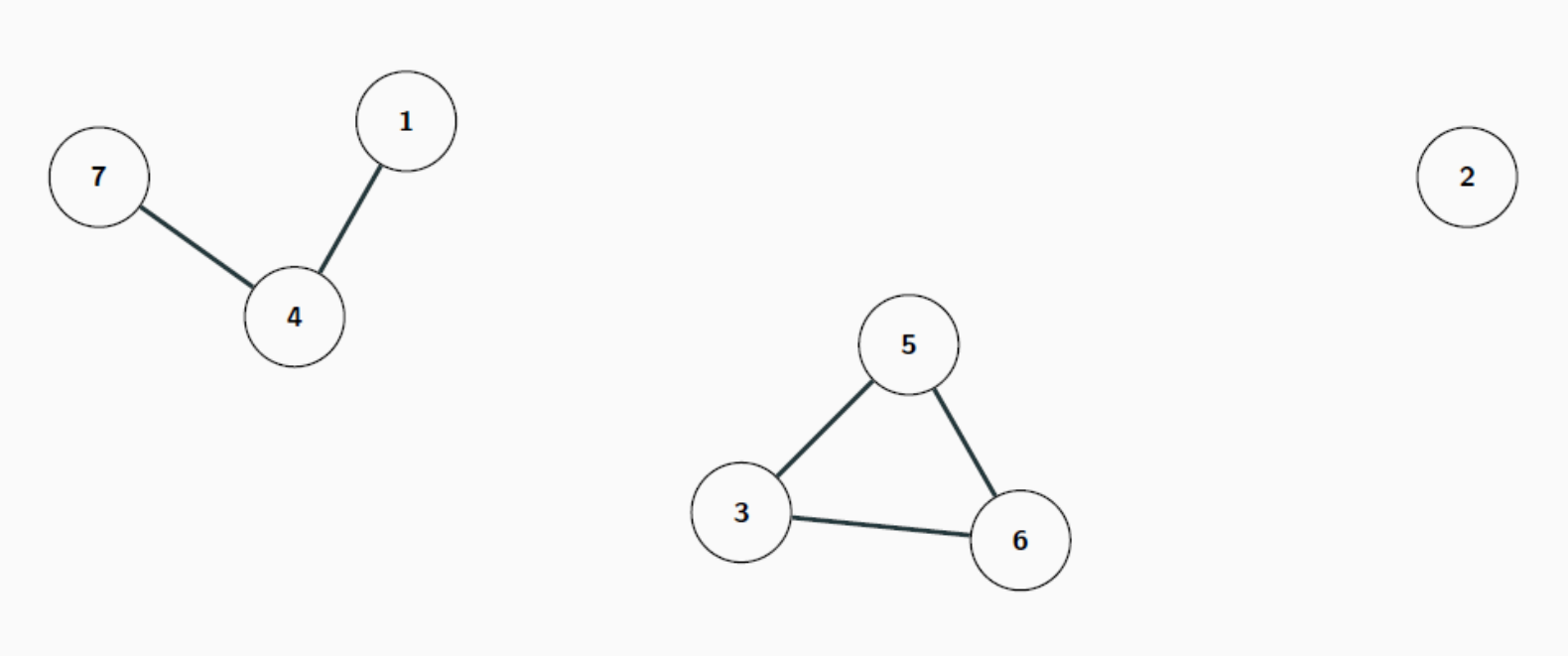
Em nosso código, será mantido um vetor chamado "**parent**", que contém os valores dos representantes ("pais") dos subconjuntos a que pertencem cada um dos *n* elementos (cada elemento está em exatamente um subconjunto disjunto).

Exemplo:

- **Itens**: {1, 2, 3, 4, 5, 6}
- **Subconjuntos disjuntos**: {1, 4}, {3, 5, 6}, {2}
- **Outro exemplo**: {1}, {2}, {3}, {4}, {5}, {6}.

Union-Find Disjoint Sets

O conceito pode ser visualizado muito claramente através da noção de grafos. Veja o exemplo, para itens de 1 a 7:

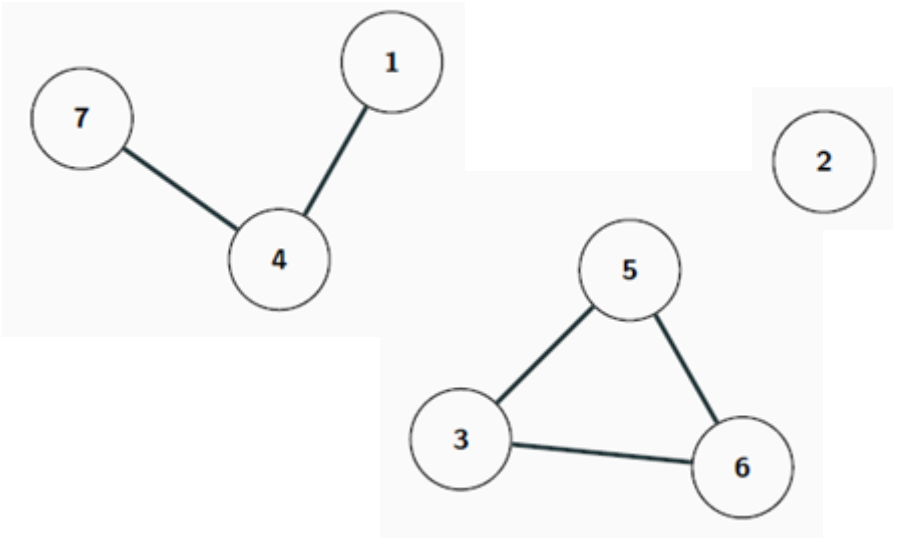


Para o exemplo anterior:

- **Itens**: {1, 2, 3, 4, 5, 6, 7}
- **Subconjuntos disjuntos**: {1, 4, 7}, {3, 5, 6}, {2}

Nossa função **find**(x) retornará um item representativo do conjunto:

- **find**(1) = 1
- **find**(4) = 1
- **find**(7) = 1
- **find**(3) = 5
- **find**(5) = 5
- **find**(6) = 5
- **find**(2) = 2



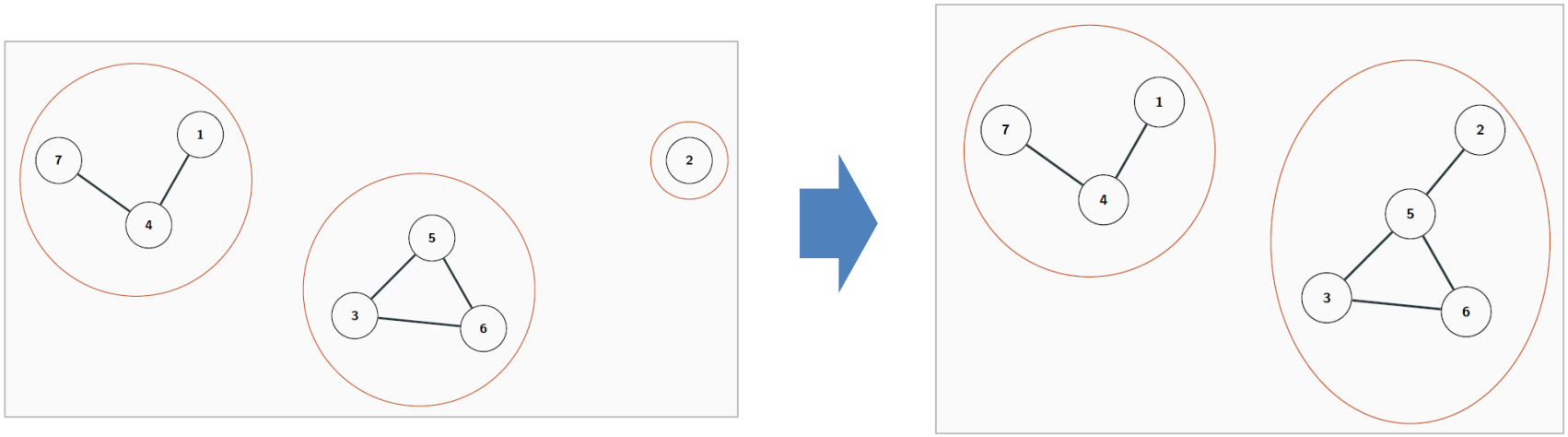
E como eu faço pra saber se dois elementos **a** e **b** estão em um mesmo subconjunto? Se e somente se **find**(a) == **find**(b). Podemos escrever uma função **isSameSet()** pra isso.

Para o mesmo exemplo:

- **Itens**: {1, 2, 3, 4, 5, 6, 7}
- **Subconjuntos disjuntos**: {1, 4, 7}, {3, 5, 6}, {2}

A função union(x, y) fará a união de dois subconjuntos: o que contém o elemento x com o que contém o elemento y.

- **union**(5, 2)
- **Subconjuntos resultantes:** {1, 4, 7}, {2, 3, 5 , 6}
- **union**(3, 6)
- **Subconjuntos resultantes:** {1, 4, 7}, {2, 3, 5 , 6}



E como ficaria o **código**?

```cpp
#include <iostream>
#include <vector>
#define MAXN 10
using namespace std;

int parents[MAXN]; // vetor global

void makeSet() { // constroi a ufds
  for(int i = 0; i < MAXN; i++) parents[i] = i;
}

int find(int elem) { // encontra representante
  if(elem == parents[elem]) return elem;
  return find(parents[elem]);
}

void unite(int x, int y) { // ounion é palavra reservada
  parents[find(y)] = find(x);
}

bool isSameSet(int x, int y) { // verifica conexao
  return find(x) == find(y);
}
```

```cpp
int main()
{
  int x, y;
  makeSet(); // inicializa estrutura

  while(true) { // fazendo conexões
    cin >> x >> y;
    if(x == -1 || y == -1) break;
    unite(x, y);
  }

  while(true) { // fazendo consultas
    cin >> x >> y;
    if(x == -1 || y == -1) break;
    if(isSameSet(x, y))
      cout << "Estao conectados" << endl;
    else
      cout << "NAO estao conectados" << endl;
  }
  return 0;
}
```

Union-Find Disjoint Sets

No entanto, o código apresentado no slide anterior ainda não é eficiente da maneira que gostaríamos. O problema está na função find, que pode ter que fazer n chamadas recursivas até encontrar o "pai" de um determinado elemento. Veja o caso abaixo:

vetor parent

0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8

Se quisesse verificar se elementos 0 e 9 estão em um mesmo conjunto, eu teria que verificar primeiramente o representante do conjunto do elemento 0, que é o próprio (essa foi fácil). A seguir, teria que verificar o representante do conjunto do 9. A chamada recursiva faria ele voltar para o 8, que voltaria para o 7, para o 6, para o 5... Até o 0. **São n passos**. **Isso não é tão bom**.

Union-Find Disjoint Sets

Uma solução bastante simples para este problema é uma técnica chamada de "***path compression***" ou compressão de caminho. Acrescentaremos um detalhe ao código que faça com que o vetor "parent" seja automaticamente atualizado sempre que for realizada uma busca com a função find, trocando um longo caminho de busca por um caminho bem mais curto, direto para o representante. Para isso, mudaremos um detalhe na função find:

```
int find(int elem) { // path compression
  if(elem == parents[elem]) return elem;
  return parents[elem] = find(parents[elem]);
}
```

Veja que, na primeira chamada de find, **o parent é atualizado** para o representante raiz do conjunto. Na próxima vez, será uma busca de um único passo (em vez de n).

E se estivermos com pressa, tem essa versão alternativa:

```
#define MAXN 1000
int p[MAXN]; // estrutura com os parents

int find(int x) {
  return (p[x] == x ? x : p[x] = find(p[x]));
}

void unite(int x, int y) {
  p[find(x)] = find(y);
}

// inicialização (main)
for(int i = 0; i < MAXN; i++) p[i] = i;
```

Union-Find Disjoint Sets

Exercício em Sala:

Uva 793 - **Network Connections**

Os três exercícios são para entrega até dia **06/11**.

Entrega através do e-mail joaopaulo@unifei.edu.br.

Devem ser realizados em dupla e somente valerão a nota se estiverem funcionando perfeitamente (***accepted***).

Arquivos estão no SIGAA, com nome problemas_ufds.zip. O arquivo contém dados de entrada (**in**), saída (**out**) e enunciados (**pdf**).