SCC-211
Lab. Algoritmos Avançados

Capítulo 10
Union Find

João Luís G. Rosa

Conjunto-disjunto

- Em computação, uma **estrutura de dados conjunto-disjunto** é uma estrutura de dados que considera um conjunto de elementos particionados em vários subconjuntos disjuntos.
- Um algoritmo **`union-find`** é um algoritmo que realiza duas operações úteis nesta estrutura de dados:
  - **`Find`**: Determina a qual conjunto um determinado elemento pertence. Também útil para determinar se dois elementos estão no mesmo conjunto.
  - **`Union`**: Combina ou agrupa dois conjuntos em um único conjunto.

Union-find

- Porque suporta estas duas operações, uma estrutura de dados conjunto-disjunto é conhecida como uma *estrutura de dados union-find* ou *conjunto merge-find*. Uma outra operação importante, *MakeSet*, que faz um conjunto conter apenas um dado elemento (um *singleton*), é geralmente trivial. Com estas três operações, muitos problemas práticos de particionamento podem ser resolvidos.

Union-find

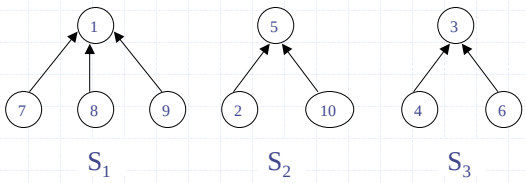
- Essas operações aparentemente simples não são suportadas *eficientemente* pelo **`set`** da STL do C++, que apenas lida com um único conjunto.
- Ter um **`vector`** de **`set`**s e iterar através de cada um para achar a que conjunto um item pertence é computacionalmente caro!
- O **`set_union`** do **`algorithm`** da STL do C++ também não é suficientemente eficiente uma vez que ele combina dois conjuntos em *tempo linear* e ainda teríamos de tratar com o "embaralhamento" do conteúdo dentro de um **`vector`** de **`set`**s.

Union-find

- Portanto, precisamos de nossa própria biblioteca para suportar esta estrutura de dados.
- As operações de **`find`** e **`union`** são úteis também para o algoritmo de Kruskal.

Exemplo 1

- Seja o seguinte exemplo:

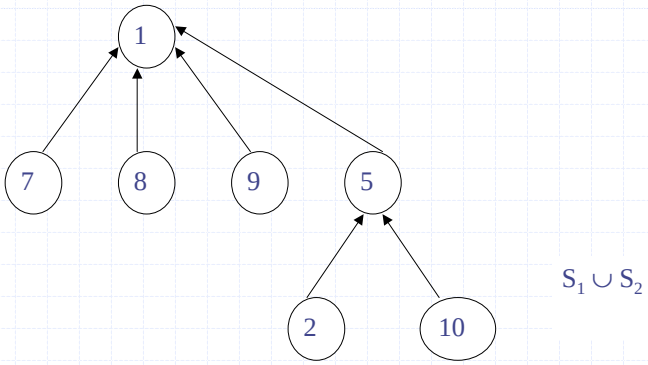


- $S_1 = \{1,7,8,9\}$, $S_2 = \{2,5,10\}$, $S_3 = \{3,4,6\}$.

Exemplo 1

- Operações:
  - **`Union`** de conjuntos disjuntos: $S_1 \cup S_2$ = {1,7,8,9,2,5,10}.
  - **`Find(i)`**. 4 está no conjunto S_3 e 9 está no conjunto S_1.
- Para obter a união de S_1 com S_2, como os vértices já foram ligados dos sucessores para os pais, basta tornar uma das árvores uma sub-árvore da outra.

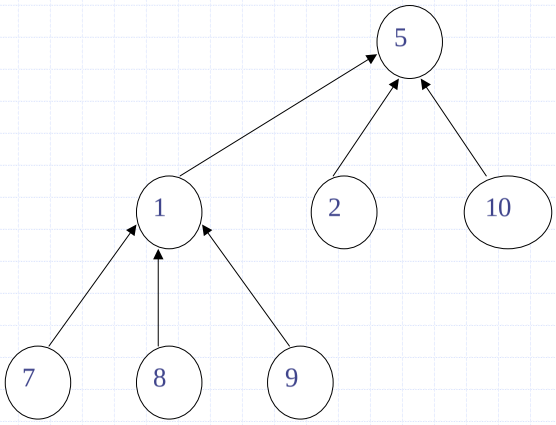
União de S_1 com S_2



$S_1 \cup S_2$

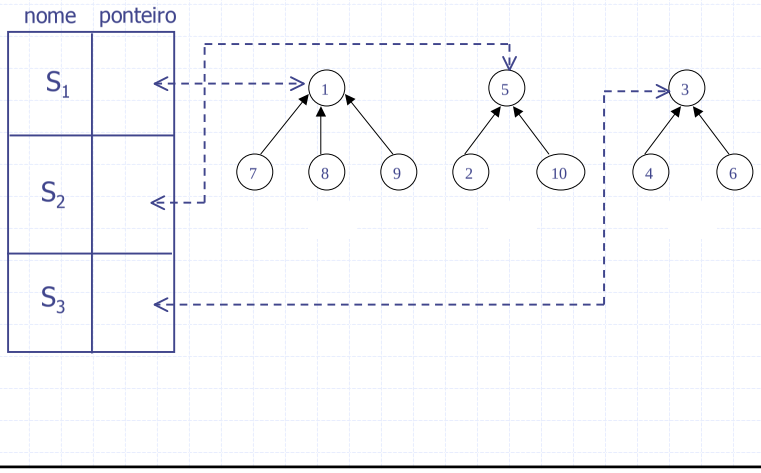
Laboratório de Algoritmos Avançados – Capítulo 10

União de S_1 com S_2



$S_1 \cup S_2$

Representação de Dados



União de S_1 com S_2

- Para os algoritmos **`union`** e **`find`**, ignora-se os nomes dos conjuntos que são identificados pelas raízes das árvores que os representam.
- Se o elemento *i* está em uma árvore de raiz *j* e *j* tem um ponteiro para a entrada *k* na tabela de nome do conjunto, então o nome do conjunto é *nome[k]*.
- Para $S_i \cup S_j$ une-se as árvores com raízes *FindPointer*(S_i) e *FindPointer*(S_j).

União de S_1 com S_2

- *FindPointer* é uma função que recebe o nome do conjunto e determina a raiz da árvore que o representa.
- Isso é feito examinando a tabela [nome do conjunto, ponteiro].
- Em muitas aplicações, o nome do conjunto é o elemento na raiz.

União de S_1 com S_2

- Operação **`Find(i)`** :
  - Determine a raiz da árvore que contém o elemento *i*.
- A função **`Union(i,j)`** requer que duas árvores com raízes *i* e *j* sejam unidas.
- Para simplificar assuma que os elementos do conjunto sejam os números de 1 a *n*.
- Utiliza-se então um vetor **`p[1:n]`**, onde *n* é o número máximo de elementos.

União de S_1 com S_2

- O *i*-ésimo elemento deste vetor representa o nó da árvore que contém o elemento *i*.
- Este elemento do vetor fornece o ponteiro para o pai do nó da árvore correspondente.
- Veja a representação dos conjuntos S_1, S_2 e S_3. Note que os nós raiz têm um pai –1:

i	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
p	-1	5	-1	3	-1	3	1	1	1	5

Algoritmos **Union** e **Find**

- Algoritmo **SimpleUnion(*i*,*j*)**

```
{
  p[i] = j;
}
```

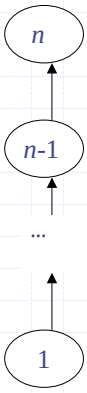
- Algoritmo **SimpleFind(*i*)**

```
{
  while (p[i] >= 0) do i = p[i];
  return i;
}
```


Algoritmos **Union** e **Find**

- A performance dos algoritmos não é muito boa.
- Por exemplo, q elementos onde cada um é seu próprio conjunto (ou seja, $S_i = \{i\}$, $1 \le i \le q$), então a configuração inicial consiste de uma floresta com q nós e $p[i] = -1$, $1 \le i \le q$.
  - Union(1,2), Union(2,3), Union(3,4), ... Union(n-1, n)
  - Find(1), Find(2), ..., Find(n)
- Esta sequência resulta na árvore degenerada do próximo slide.

Árvore degenerada



Algoritmos **`Union`** e **`Find`**

- Como o tempo de uma **`union`** é constante, as n -1 **`union`**s são processadas no tempo $O(n)$.
- Mas cada execução do **`find`** segue uma sequência de ponteiros para o pai, a partir do elemento a ser achado até a raiz.
- Como o tempo para cada **`find`** de um elemento no nível i de uma árvore é O(i), o tempo total necessário para processar os n **`find`**s é:

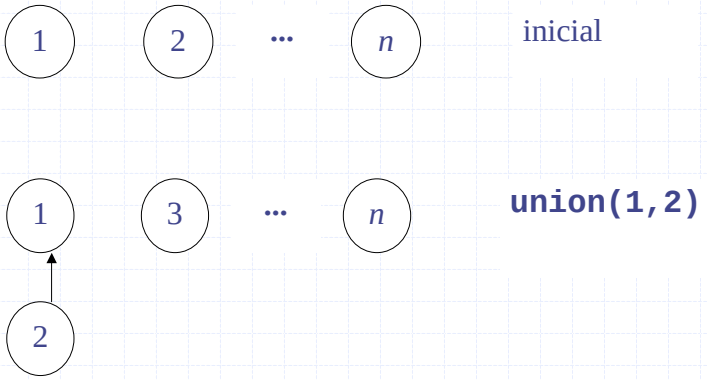
$$O(\sum_{i=1}^{n} i) = O(n^2)$$

Regra de Ponderação

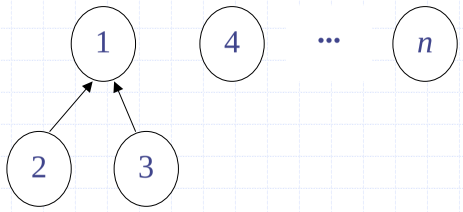
- Pode-se melhorar a performance dos algoritmos **`union`** e **`find`**, evitando a criação de árvores degeneradas:
  - *Regra de ponderação para* **`union(i,j)`**:
    - Se o número de nós na árvore com raiz *i* é menor que o número na árvore com raiz *j*, então faça *j* o pai de *i*; caso contrário, faça *i* o pai de *j*.
- Veja no próximo slide, onde se usou a regra de ponderação. As **`union`**s foram modificadas tal que os valores dos parâmetros de entrada correspondam às raízes das árvores a serem combinadas.

Laboratório de Algoritmos Avançados – Capítulo 10

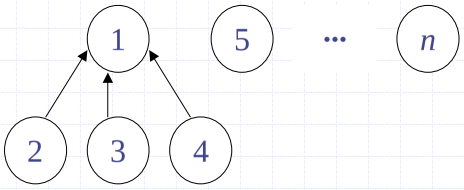
Inicial e **`union(1,2)`**



union(1,3) e union(1,4)

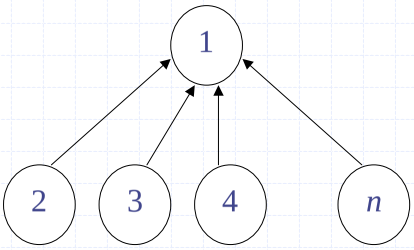


union(1,3)



union(1,4)

Finalmente union(1,n)



union(1,*n*)

Algoritmo **WeightedUnion(i,j)**


```
Algoritmo UniãoPonderada(i,j)
// conjuntos união com raízes i e j, i ≠j,
// usando a regra de ponderação.
// p[i] = -cont[i] e p[j] = -cont[j].
{
  temp = p[i] + p[j];
  if (p[i] > p[j])
  {// i tem menos nós
   p[i] = j; p[j] = temp;
  }
  else
  { // j tem números de nós menor ou igual
   p[j] = i; p[i] = temp;
  }
}
```

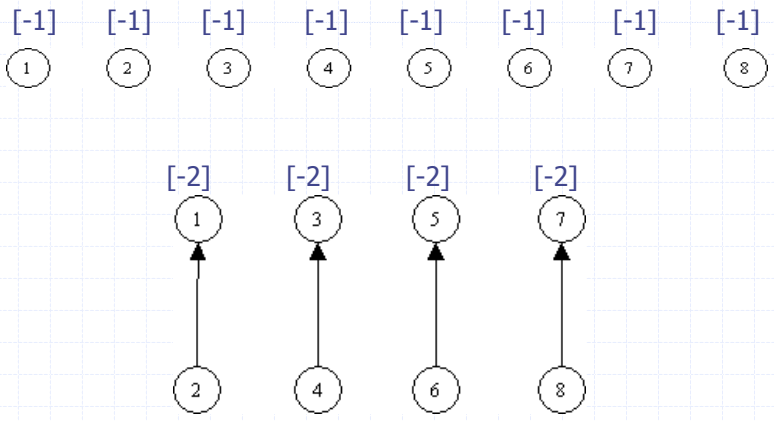
Algoritmo **WeightedUnion(i,j)**

- Neste algoritmo, o tempo necessário para realizar uma união cresceu mas ainda é limitado por uma constante, ou seja, $O(1)$.
- O algoritmo **`find`** não mudou.
- Lema 1: Assuma que se inicia com uma floresta de árvores, cada uma tendo um nó. Seja T a árvore com m nós criados como resultado de uma sequência de uniões realizadas com **`WeightedUnion`**. A altura de T não é maior que $\log_2 m+1$.

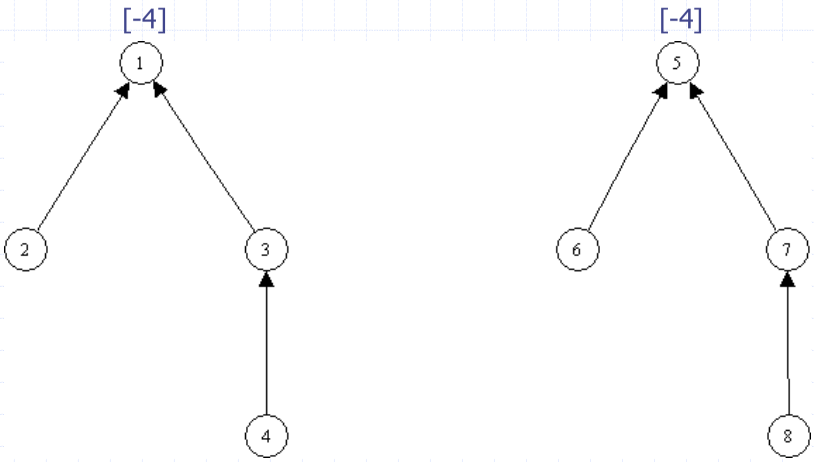
Exemplo 2

- Exemplo 2:
  - `p[i] = -cont[i] = -1`, $1 \leq i \leq 8 = n$
  - `union(1,2)`, `union(3,4)`, `union(5,6)`, `union(7,8)`, `union(1,3)`, `union(5,7)`, `union(1,5)`
- As árvores das figuras dos próximos slides são obtidas.
- Veja que a altura de cada árvore com m nós é $\log_2 m + 1$.

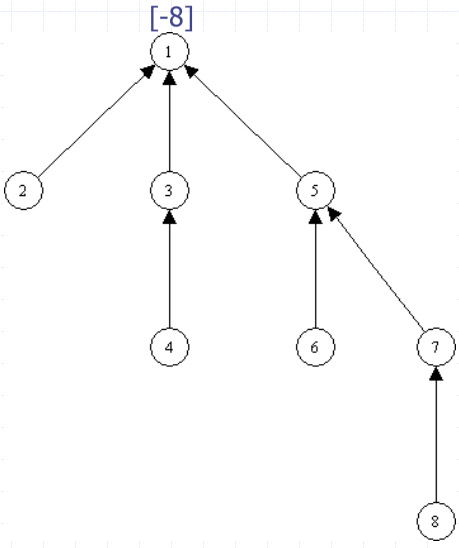
Exemplo 2



Exemplo 2



Exemplo 2



Exemplo 2

- Do Lema 1, segue que o tempo para processar um **`find`** é $O(\log m)$ se houver m elementos em uma árvore.
- Se uma sequência misturada de $u-1$ uniões e f operações **`find`** a serem processadas, o tempo torna-se $O(u + f \log u)$, já que nenhuma árvore tem mais de u nós.
- É claro que necessita-se de $O(n)$ de tempo adicional para iniciar a floresta de n árvores.
- Mas, é possível melhorar ainda mais: algoritmo **`find`** usando a *regra de desmoronamento.*

Regra do desmoronamento

- Regra de desmoronamento: Se **`j`** é um nó no caminho de **`i`** até sua raiz e **`p[i]`** ≠ **`raiz[i]`**, então faça **`raiz[i]`** = **`p[j]`**.
- O algoritmo **`CollapsingFind`** incorpora a regra de desmoronamento.
- Exemplo2: Considere a árvore criada pelo **`WeightedUnion`** na sequência de **`union`**s do exemplo 1. Agora processe os seguintes oito **`find`**s:

  `Find(8)`, **`Find(8)`**, …, **`Find(8)`**

Algoritmo **`CollapsingFind`**


```
// Ache a raiz da árvore que contém o elemento i.
// Use a regra de desmoronamento para desmoronar
// todos os nós de i a raiz
{
   r = i;
   while (p[r] > 0) do r = p[r]; // acha a raiz
   while (i ≠ r) do // desmorona nós de i a raiz r
   {
      s = p[i]; p[i] = r; i = s;
   }
   return r;
}
```

Algoritmo **`CollapsingFind`**

- Se **`SimpleFind`** é usado, cada **`Find(8)`** requer subir três campos de links dos pais, resultando num total de 24 movimentos para os oito **`find`**s.
- Quando **`CollapsingFind`** é usado, o primeiro **`Find(8)`** requer subir três links e então reseta dois links.
- Note que, mesmo que apenas dois links tenham de ser resetados, CollapseFind resetará três (o pai de 5 é resetado a 1).
- Cada um dos sete **`find`**s restantes requer subir apenas um campo de link.
- O custo total é de apenas 13 movimentos.

Algoritmo **`CollapsingFind`**

- Nos algoritmos **`WeightedUnion`** e **`CollapsingFind`**, o uso da regra de desmoronamento dobra o tempo para um item individual.
- Entretanto, reduz o tempo do pior caso para uma sequência de **`find`**s.
- Lema 2 [Tarjan e Van Leeuwen] Assuma que começa-se com uma floresta de árvores, cada uma tendo um nó. Seja $T(f, u)$ o tempo máximo necessário para processar qualquer sequência de f **`find`**s e u **`union`**s. Assuma que $u \geq \frac{1}{2}n$. Então:

  $k_1[n + f\,\alpha(f + n, n)] \leq T(f, u) \leq k_2[n + f\,\alpha(f + n, n)]$

  para algumas constantes positivas k_1 e k_2.

Algoritmo **`CollapsingFind`**

- A exigência de que $u \geq \frac{1}{2}n$ no Lema 2 não é significativa, já que quando $u \geq \frac{1}{2}n$ alguns elementos não estão envolvidos em nenhuma operação de **`union`**.
- Esses elementos mantêm-se em conjuntos singletons ao longo da sequência de operações **`union`** e **`find`** e podem ser eliminados.

Exemplo: Re-connecting Computer Sites (UVa 908)

- Consider the problem of selecting a set *T* of high-speed lines for connecting *N* computer sites, from a universe of *M* high-speed lines each connecting a pair of computer sites. Each high-speed line has a given monthly cost, and the objective is to minimize the total cost of connecting the *N* computer sites, where the total cost is the sum of the cost of each line included in set *T*. Consider further that this problem has been solved earlier for the set of *N* computer sites and *M* high-speed lines, but that a few *K* new high-speed lines have recently become available.

Exemplo: Re-connecting Computer Sites (UVa 908)

- Your objective is to compute the new set *T'* that may yield a cost lower than the original set *T*, due to the additional *K* new high-speed lines and when *M+K* high-speed lines are available.
- Input
  - The input will contain several test cases, each of them as described below. Consecutive test cases are separated by a single blank line.
  - The input is organized as follows:
  - A line containing the number *N* of computer sites, with *1 <= N <= 1000000*, and where each computer site is referred by a number *i*, *1 <= i <= N*.

Exemplo: Re-connecting Computer Sites (UVa 908)

- The set *T* of previously chosen high-speed lines, consisting of *N-1* lines, each describing a high-speed line, and containing the numbers of the two computer sites the line connects and the monthly cost of using this line. All costs are integers.
- A line containing the number *K* of new additional lines, *1 <= K <= 10*.
- *K* lines, each describing a new high-speed line, and containing the numbers of the two computer sites the line connects and the monthly cost of using this line. All costs are integers.
- A line containing the number *M* of originally available high-speed lines, with *N-1 <= M <= N (N-1) / 2*.

Exemplo: Re-connecting Computer Sites (UVa 908)

- *M* lines, each describing one of the originally available high-speed lines, and containing the numbers of the two computer sites the line connects and the monthly cost of using this line. All costs are integers.

◆ Output

- For each test case, the output must follow the description below. The outputs of two consecutive cases will be separated by a blank line.
- The output file must have one line containing the original cost of connecting the *N* computer sites with *M* high-speed lines and another line containing the new cost of connecting the *N* computer sites with *M+K* high-speed lines. If the new cost equals the original cost, the same value is written twice.

Exemplo: Re-connecting Computer Sites (UVa 908)

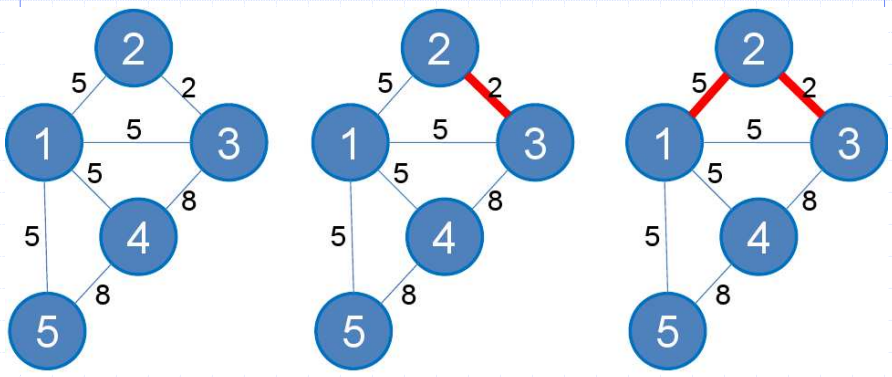
- Sample Input

```
5
1 2 5
1 3 5
1 4 5
1 5 5
1
2 3 2
6
1 2 5
1 3 5
1 4 5
1 5 5
3 4 8
4 5 8
```

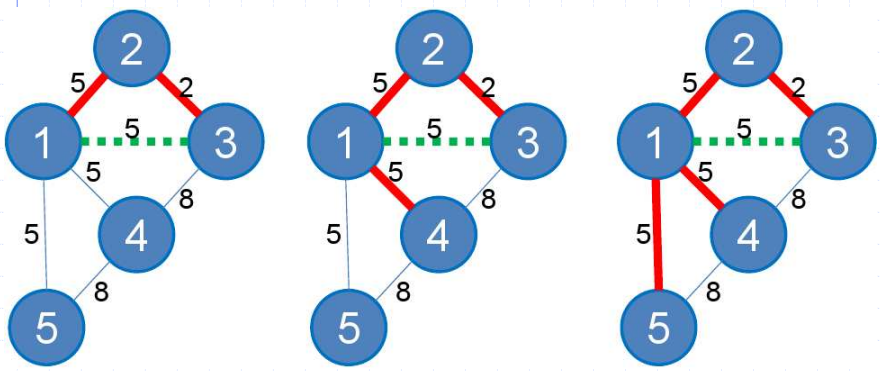
- Sample Output

```
20
17
```

Uma solução: Kruskal usando `union-find`



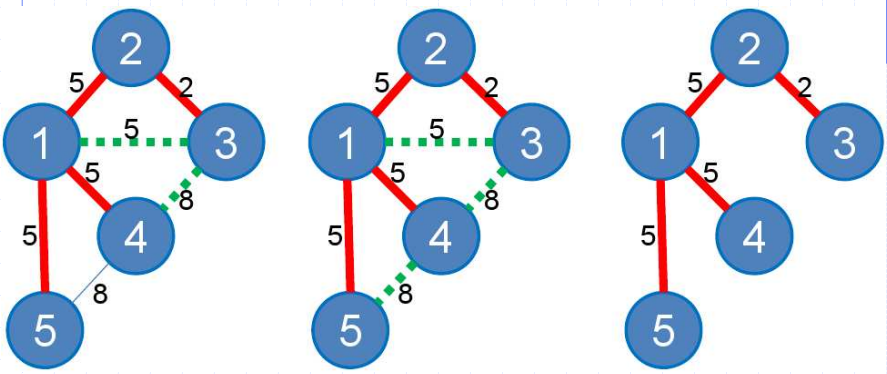
Kruskal usando `union-find`



Não pode conectar
1 e 3: ciclo!

Conecta 1 e 5:
MST formada!

Kruskal usando `union-find`



Mas o algoritmo de
Kruskal continua...

Como parar?

... sem modificação
nenhuma!

Algoritmo de Kruskal [2]

```
// sorted by edge cost, PQ default: sort descending :(
priority_queue< pair<int, ii> > EdgeList;
// trick: store (negative) weight(i, j) and pair(i, j)
// i.e. EdgeList.push(make_pair(-weight, make_pair(i,
   j)));
mst_cost = 0; initSet(V); // all V are disjoint initially
while (!EdgeList.empty())
{ // while ∃ more edges
 pair<int, ii> front = EdgeList.top(); EdgeList.pop();
 if (!isSameSet(front.second.first, front.second.second))
 {
   // if adding e to MST does not form a cycle
   mst_cost += (-front.first); // add –weight of e to MST
   unionSet(front.second.first, front.second.second);
 }
}
```


Exercícios para Nota

- Ubiquitous Religions (10583)
- Friends (10608)

Ubiquitous Religions (10583)

- There are so many different religions in the world today that it is difficult to keep track of them all. You are interested in finding out how many different religions students in your university believe in.
- You know that there are n students in your university ($0 < n \leq 50000$). It is infeasible for you to ask every student their religious beliefs. Furthermore, many students are not comfortable expressing their beliefs.

Ubiquitous Religions (10583)

- One way to avoid these problems is to ask m ($0 \leq m \leq n(n-1)/2$) pairs of students and ask them whether they believe in the same religion (e.g. they may know if they both attend the same church). From this data, you may not know what each person believes in, but you can get an idea of the upper bound of how many different religions can be possibly represented on campus. You may assume that each student subscribes to at most one religion.

Ubiquitous Religions (10583)

- The **input** consists of a number of cases. Each case starts with a line specifying the integers *n* and *m*. The next *m* lines each consists of two integers i and j, specifying that students i and j believe in the same religion. The students are numbered 1 to *n*. The end of input is specified by a line in which *n* = *m* = 0.
- For each test case, print on a single line the case number (starting with 1) followed by the maximum number of different religions that the students in the university believe in.

Ubiquitous Religions (10583)

- Sample Input

```
10 9
1 2
1 3
1 4
1 5
1 6
1 7
1 8
1 9
1 10
10 4
2 3
4 5
4 8
5 8
0 0
```

- Sample Output

```
Case 1: 1
Case 2: 7
```

Friends (10608)

- There is a town with N citizens. It is known that some pairs of people are friends. According to the famous saying that "The friends of my friends are my friends, too" it follows that if A and B are friends and B and C are friends then A and C are friends, too.
- Your task is to count how many people there are in the largest group of friends.

Friends (10608)

- Input
  - Input consists of several datasets. The first line of the input consists of a line with the number of test cases to follow. The first line of each dataset contains tho numbers N and M, where N is the number of town's citizens ($1 \le N \le 30000$) and M is the number of pairs of people ($0 \le M \le 500000$), which are known to be friends. Each of the following M lines consists of two integers A and B ($1 \le A \le N$, $1 \le B \le N$, $A \ne B$) which describe that A and B are friends. There could be repetitions among the given pairs.

Friends (10608)

- Output
  - The output for each test case should contain one number denoting how many people there are in the largest group of friends.
- Sample input

```
2
3 2
1 2
2 3
10 12
1 2
3 1
3 4
5 4
3 5
4 6
5 2
2 1
7 10
1 2
9 10
8 9
```

- Sample Output

```
3
6
```

Laboratório de Algoritmos Avançados – Capítulo 10

Referências

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. **Introduction to Algorithms**. Second edition. The MIT Press, 2001.
2. Halim, S, Halim, F. **Competitive Programming**. Lulu, 2010
3. Horowitz, E., Sahni, S., Rajasekaran, S. **Computer Algorithms**. Computer Science Press, 1997.
4. Wikipedia: http://en.wikipedia.org/wiki/Disjoint-set_data_structure