Graph algorithms: Minimum Spanning Tree (and Disjoint sets data structures)

Rossano Venturini[1]

1 **Computer Science Department, University of Pisa, Italy**
rossano.venturini@unipi.it

Notes for the course *"Competitive Programming and Contests"* at Department of Computer Science, University of Pisa.
Web page: `https://github.com/rossanoventurini/CompetitiveProgramming`

These notes sketch the content of the 11th lecture. The topics of this lecture are treated in any introductory book on Algorithms. You may refer to Chapters 21 and 23 of *Introduction to Algorithms, 3rd Edition*, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, The MIT Press, 2009 for a more detailed explanation.

1 Disjoint sets data structures

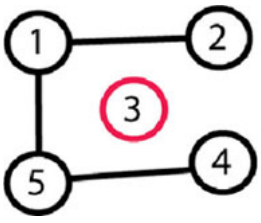
This section is based on the tutorial at `https://www.topcoder.com/community/data-science/data-science-tutorials/disjoint-set-data-structures/` and Chapter 21 of Cormen et al.

Consider the following problem.

▶ **Problem 1** (Groups of friends). *In a room are n persons, and we will define two persons are friends if they are directly or indirectly friends. If A is a friend with B, and B is a friend with C, then A is a friend of C too. A group of friends is a group of persons where any two persons in the group are friends. Given the list of persons that are directly friends find the number of groups of friends and the number of persons in each group.*

For example, $n = 5$ and the list of friends is: 1-2, 5-4, and 5-1. Here is the figure of the graph that represents the groups of friends. 1 and 2 are friends, then 5 and 4 are friends, and then 5 and 1 are friends, but 1 is friend with 2; therefore 5 and 2 are friends, etc.

The instance can be represented as the following graph.



Clearly, there are 2 groups.

The problem can be solved with BFS as we are looking for connected components.

We will see now an alternative solution.

A disjoint-set data structure is a structure that maintains a collection $S_1, S_2, S_3, \ldots, S_n$ of dynamic disjoint sets. Two sets are disjoint if their intersection is null. In a data structure of disjoint sets every set contains a *representative*, which is one member of the set.

A Disjoint set data structure has to support the following operations:

1. CreateSet(x), which creates a new set with one element $\{x\}$.
2. Union(x, y), which merges into one set the set that contains element x and the set that contains element y (x and y are in different sets). The original sets will be destroyed.
3. FindSet(x), which returns the representative or a pointer to the representative of the set that contains element x.

The problem above is soved by using a Disjoint set data struture as follows. At the beginning there are 5 groups (sets): $\{1\}, \{2\}, \{3\}, \{4\}, \{5\}$. Nobody is anybody's friend and everyone is the representative of his or her own group.

We then scan the frienships and use Union the sets containing the two persons. We also take count of how many distinc groups are there. How? By comparing the representatives of the two persons. If there were different, the merge will reduce the number of distinct sets by 1.

Linked-list representation of sets

One way to implement disjoint set data structures is to represent each set by a linked list.

Each element (object) will be in a linked list and will contain a pointer to the next element in the set and another pointer to the representative of the set.

CreateSet and FindSet takes constant time.

We perform Union(x, y) by appending y's list at the tail of x's list. This takes constant time. However, we also need to modify the representative of y's elements to equal the representitive of x. The latter operation is quite expensive.

We can easily perform a sequence of m operations on n elements that requires $\Theta(n^2)$ time.

A weighted-union heuristic

The "a weighted-union heuristic" makes the algorithm more efficient.

In this case, let's say that the representative of a set contains information about how many elements are in that set as well.

The optimization is to always append the smaller list onto the longer and, in case of ties, append arbitrarily.

This will bring the complexity of the algorithm to $\Theta(m + nlogn)$ where m is the number of operations (CreateSet, FindSet and Union) and n is the number of operations CreateSet.

Why do we have such time complexity?

Proof. Because Union merge two disjoint sets, we have at most $n-1$ such operations. The other operations cost constant time. Thus, we have to bound the cost of these $n-1$ Unions.

We start by determining for an element x an upper bound to the number of times its representative changes.

We know that everytime the representative of x changes, x was in the smaller set.

Thus, the new set of x has a size which is at least twice the size of the its previous set. Thus, the number of times cannot be larger that $\log n$.

Thus, the total cost of the $n-1$ Unions is $O(n \log n)$. ◀

Disjoint-set forests

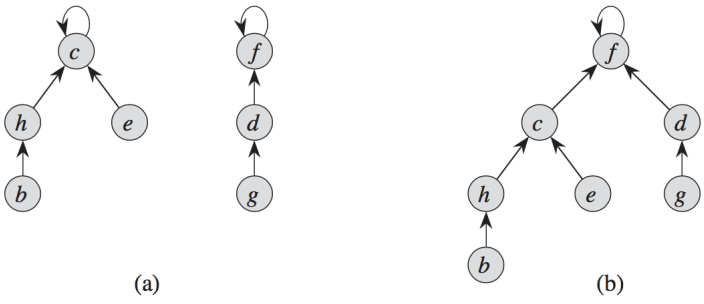
Instead of using linked-lists, we can use trees. Every element point to its parent, the root is the representative.

This representation can be made faster by using two heristics: *union by rank* and *path compression*.

This gives an asymptotically optimal disjoint-set data structure, which executes m operations on n elements in $\Theta(m\alpha(n))$ time, where $\alpha(n)$ is the inverse Ackermann.

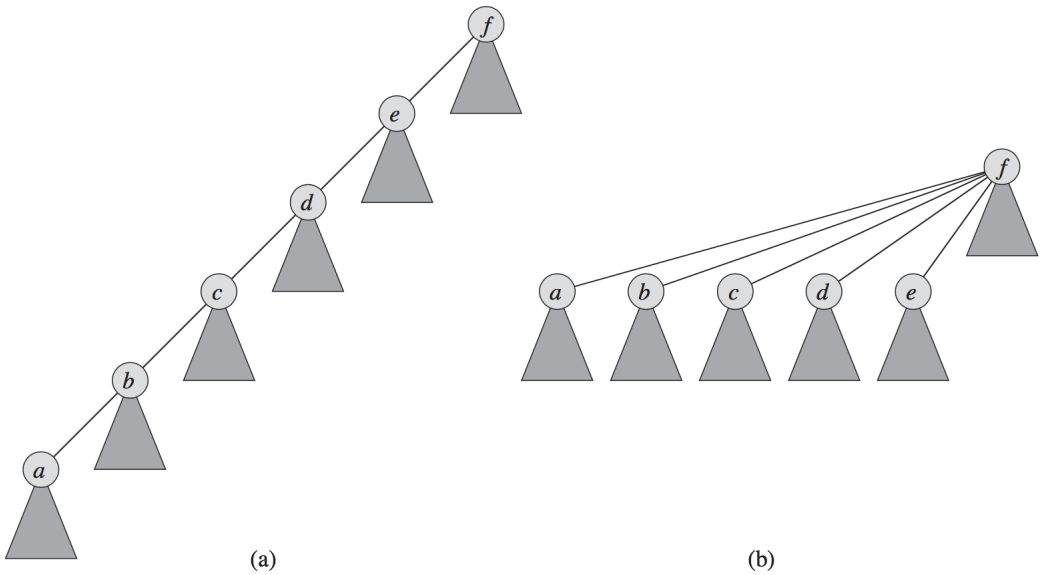
The idea in the first heuristic union by rank is to make the root of the tree with fewer nodes point to the root of the tree with more nodes.

Figure below shows an example of Union.



The idea in the second heuristic path compression, which is used for operation FindSet(x), is to make each node on the find path point directly to the root. This will not change any ranks.

Figure below shows an example of path compression.



For each node, we maintain a rank which is an upper bound on the height of the node. When Union(x, y) is called, the root with smaller rank is made to point to the root with larger rank.

If ranks of the two are equal, we increase the rank by one.

Below the pseudocode of the operations.

```
MAKE-SET(x)
1  x.p = x
2  x.rank = 0

UNION(x, y)
1  LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y)
1  if x.rank > y.rank
2       y.p = x
3  else x.p = y
4       if x.rank == y.rank
5            y.rank = y.rank + 1
```

The FIND-SET procedure with path compression is quite simple:

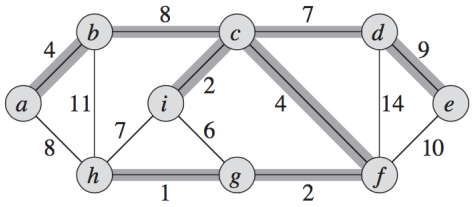
```
FIND-SET(x)
1  if x ≠ x.p
2       x.p = FIND-SET(x.p)
3  return x.p
```

The analysis of the data structure is skipped. Refer to Cormen et al for a complete (and intersting) proof.

2 Minimum spanning tree

▶ **Problem 2** (Minimum spanning tree). *Given an undirected graph $G = (V, E)$ where every edge (u, v) has a weight $w(u, v)$. The goal is to find a tree T such that T is formed with a subset of edges in E, it connects all the vertecis in V, and has the minimum possible total edge weight.*

Below an example of a graph and its minimum spanning tree.



There exists two efficient algorithms for this problem: Kruskal's algorithm and Prin's algorithm. The former runs in $\Theta(|E| \log |V|)$ time, while the latter runs in $\Theta(|E| + |V| \log |V|)$ time.

Below the pesudocode of the first one.

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3       MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6       if FIND-SET(u) ≠ FIND-SET(v)
7            A = A ∪ {(u, v)}
8            UNION(u, v)
9  return A
```

We omit the proof of correctness, which is left for self-study.