Implementation

When implementing Kruskal's algorithm, it is convenient to use the edge list representation of the graph. The first phase of the algorithm sorts the edges in the list in $O(m \log m)$ time. After this, the second phase of the algorithm builds the minimum spanning tree as follows:

```
for (...) {
  if (!same(a,b)) unite(a,b);
}
```

The loop goes through the edges in the list and always processes an edge a–b where a and b are two nodes. Two functions are needed: the function `same` determines if a and b are in the same component, and the function `unite` joins the components that contain a and b.

The problem is how to efficiently implement the functions `same` and `unite`. One possibility is to implement the function `same` as a graph traversal and check if we can get from node a to node b. However, the time complexity of such a function would be $O(n+m)$ and the resulting algorithm would be slow, because the function `same` will be called for each edge in the graph.

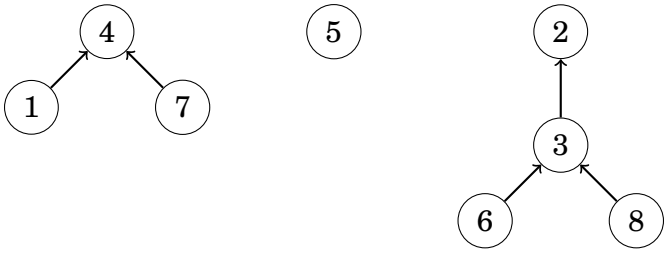
We will solve the problem using a union-find structure that implements both functions in $O(\log n)$ time. Thus, the time complexity of Kruskal's algorithm will be $O(m \log n)$ after sorting the edge list.

Union-find structure

A **union-find structure** maintains a collection of sets. The sets are disjoint, so no element belongs to more than one set. Two $O(\log n)$ time operations are supported: the `unite` operation joins two sets, and the `find` operation finds the representative of the set that contains a given element[2].

Structure

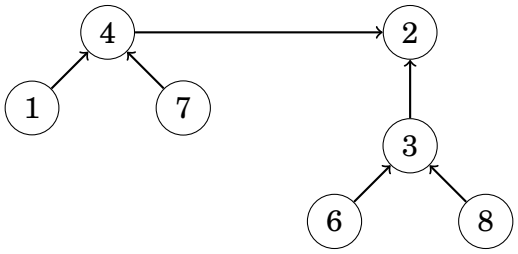
In a union-find structure, one element in each set is the representative of the set, and there is a chain from any other element of the set to the representative. For example, assume that the sets are $\{1,4,7\}$, $\{5\}$ and $\{2,3,6,8\}$:



[2]The structure presented here was introduced in 1971 by J. D. Hopcroft and J. D. Ullman [38]. Later, in 1975, R. E. Tarjan studied a more sophisticated variant of the structure [64] that is discussed in many algorithm textbooks nowadays.

In this case the representatives of the sets are 4, 5 and 2. We can find the representative of any element by following the chain that begins at the element. For example, the element 2 is the representative for the element 6, because we follow the chain $6 \to 3 \to 2$. Two elements belong to the same set exactly when their representatives are the same.

Two sets can be joined by connecting the representative of one set to the representative of the other set. For example, the sets $\{1,4,7\}$ and $\{2,3,6,8\}$ can be joined as follows:



The resulting set contains the elements $\{1,2,3,4,6,7,8\}$. From this on, the element 2 is the representative for the entire set and the old representative 4 points to the element 2.

The efficiency of the union-find structure depends on how the sets are joined. It turns out that we can follow a simple strategy: always connect the representative of the *smaller* set to the representative of the *larger* set (or if the sets are of equal size, we can make an arbitrary choice). Using this strategy, the length of any chain will be $O(\log n)$, so we can find the representative of any element efficiently by following the corresponding chain.

Implementation

The union-find structure can be implemented using arrays. In the following implementation, the array `link` contains for each element the next element in the chain or the element itself if it is a representative, and the array `size` indicates for each representative the size of the corresponding set.

Initially, each element belongs to a separate set:

```
for (int i = 1; i <= n; i++) link[i] = i;
for (int i = 1; i <= n; i++) size[i] = 1;
```

The function `find` returns the representative for an element x. The representative can be found by following the chain that begins at x.

```
int find(int x) {
    while (x != link[x]) x = link[x];
    return x;
}
```

The function `same` checks whether elements a and b belong to the same set. This can easily be done by using the function `find`:

```
bool same(int a, int b) {
    return find(a) == find(b);
}
```

The function `unite` joins the sets that contain elements a and b (the elements have to be in different sets). The function first finds the representatives of the sets and then connects the smaller set to the larger set.

```
void unite(int a, int b) {
    a = find(a);
    b = find(b);
    if (size[a] < size[b]) swap(a,b);
    size[a] += size[b];
    link[b] = a;
}
```

The time complexity of the function `find` is $O(\log n)$ assuming that the length of each chain is $O(\log n)$. In this case, the functions `same` and `unite` also work in $O(\log n)$ time. The function `unite` makes sure that the length of each chain is $O(\log n)$ by connecting the smaller set to the larger set.

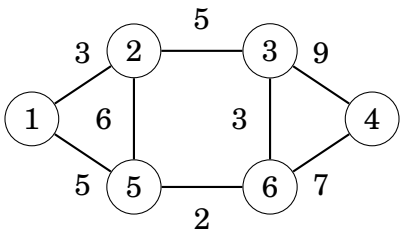
Prim's algorithm

Prim's algorithm[3] is an alternative method for finding a minimum spanning tree. The algorithm first adds an arbitrary node to the tree. After this, the algorithm always chooses a minimum-weight edge that adds a new node to the tree. Finally, all nodes have been added to the tree and a minimum spanning tree has been found.

Prim's algorithm resembles Dijkstra's algorithm. The difference is that Dijkstra's algorithm always selects an edge whose distance from the starting node is minimum, but Prim's algorithm simply selects the minimum weight edge that adds a new node to the tree.

Example

Let us consider how Prim's algorithm works in the following graph:



[3]The algorithm is named after R. C. Prim who published it in 1957 [54]. However, the same algorithm was discovered already in 1930 by V. Jarník.