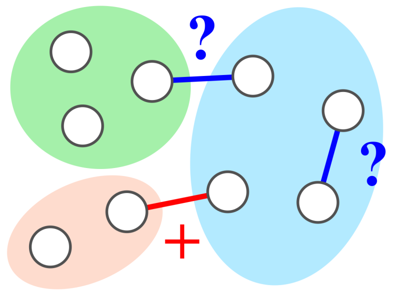
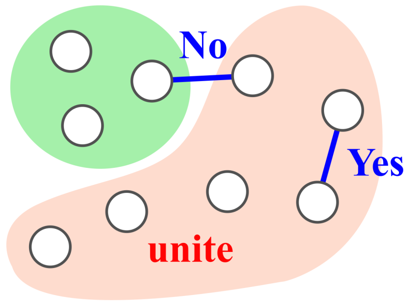
CSC11F: Advanced Data Structures and Algorithms

Disjoint Sets

Yutaka Watanobe (142-A, yutaka@u-aizu.ac.jp)

1 Introduction





Some applications involve grouping n distinct elements into a collection of disjoint sets. Two important operations are then finding which set a given element belongs to and uniting two sets.

One of the many applications of disjoint-set data structures arises in determining the connected components of an undirected graph. Another application of disjoint sets is Kruskal's algorithm to find minimum spanning trees.

The operations for disjoint-set can be performed by a basic graph traversal algorithm (DFS or BFS), but such a traversal for each query is too slow because DFS or BFS takes $O(V + E)$ which results in $O(Q(V + E))$ where Q is the number of queries.

2 Disjoint Set

A disjoint-set data structure maintains a collection $C = \{S_1, S_2, ..., S_n\}$ of disjoint dynamic sets. Each set is identified by a representative, which is a member of the set. Letting x denotes an object, we wish to support the following operations:

- **makeSet**(x) creates a new set whose only member (and thus representative) is x. Since the sets are disjoint, we require that x not already be in some other set.
- **union**(x, y) unites the dynamic sets that contain x and y, say S_x and S_y, into a new set that is the union of these two sets. The two sets are assumed to be disjoint prior to the operation. The representative of the resulting set is any member of $S_x \cup S_y$, although many implementations of union specifically choose the representative of either S_x or S_y as the new representative. Since we require the sets in the collection to be disjoint, we destroy sets S_x and S_y, removing them from the collection C.
- **findSet**(x) returns a pointer to the representative of the (unique) set containing x.

3 Disjoint-set Forests

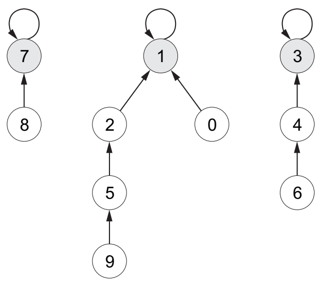


Figure 1: A disjoint-set forest

In a faster implementation of disjoint sets, we represent sets by rooted trees, with each node containing one member and each tree representing one set. In a disjoint-set forest, or a union find tree, illustrated in Figure 1, each member points only to its parent.

The root of each tree contains the representative and is its own parent. As we shall see, although the straightforward algorithms that use this representation are no faster than ones that use the linked-list representation, by introducing two heuristics - "union by rank" and "path compression" - we can achieve the asymptotically fastest disjoint-set data structure.

We perform the three disjoint-set operations as follows.

A makeSet operation simply creates a tree with just a given node x.

We perform a findSet operation from the given node x by following parent pointers until we find the root of the tree. The nodes visited on this path toward the root constitute the find path.

A union operation, shown in Figure 2, causes the root of one tree to point to the root of the other.

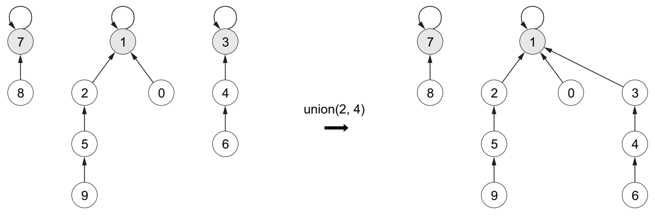


Figure 2: A union operation

3.1 Heuristics to improve the running time

A sequence of $n-1$ union operations may create a tree that is just a linear chain of n nodes. By using two heuristics, however, we can achieve a running time that is almost linear in the total number of operations m.

The first heuristic is "union by rank". The idea is to make the root of the tree with fewer nodes point to the root of the tree with more nodes. Rather than explicitly keeping track of the size of the subtree rooted at each node, we shall use an approach that eases the analysis. For each node, we maintain a rank that is an upper bound on the height of the node. In union by rank, the root with smaller rank is made to point to the root with larger rank during a union operation.

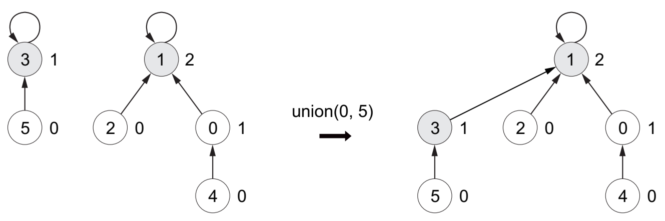


Figure 3: Union of trees with different ranks

For example, Figure 3 shows how we should merge two trees with different ranks. In this case, we should update the parent of node 3 so that the height of the obtained tree remains (if you update the parent of node 1, the height of the obtained tree increases). On the other hand, as shown in Figure 4, if the rank of both representatives are the same, the rank of the new representative increases.

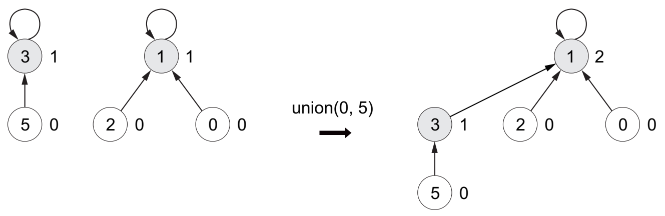


Figure 4: Union of trees with the same ranks

The second heuristic, path compression, is also quite simple and very effective. As sown in Figure 5, we use it during findSet operations to make each node on the find path point directly to the root. We can observe that path compression does not change any ranks.

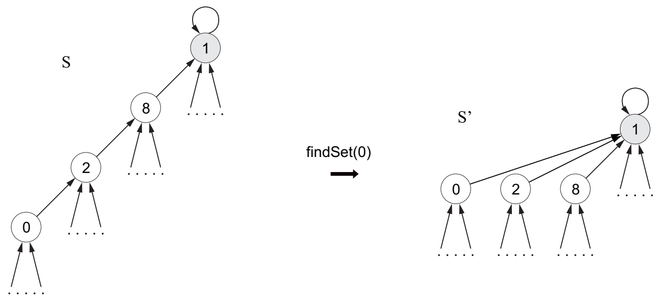


Figure 5: Path compression

3.2 Pseudocode for disjoint-set forests

To implement a disjoint-set forest with the union-by-rank heuristic, we must keep track of ranks. With each node x, we maintain the integer value rank[x], which is an upper bound on the height of x (the number of edges in the longest path between x and a descendant leaf).

When a singleton set is created by makeSet, the initial rank of the single node in the corresponding tree is 0. Each findSet operation leaves all ranks unchanged.

When applying union to two trees, there are two cases, depending on whether the roots have equal rank. If the roots have unequal rank, we make the root of higher rank the parent of the root of lower rank, but the ranks themselves remain unchanged. If, instead, the roots have equal ranks, we arbitrarily choose one of the roots as the parent and increment its rank.

Let us put this method into pseudocode. We designate the parent of node x by p[x]. The link procedure, a subroutine called by union, takes pointers to two roots as inputs.

```
makeSet(x)
1  p[x] = x
2  rank[x] = 0

union(x, y)
1  link(findSet(x), findSet(y))

link(x, y)
1  if rank[x] > rank[y]
2    p[y] = x
3  else
4    p[x] = y
4    if rank[x] == rank[y]
5      rank[y] = rank[y] + 1

findSet(x)
1  if x != p[x]
2    p[x] = findSet(p[x])
3  return p[x]
```

The findSet procedure is a two-pass method: it makes one pass up the find path to find the root, and it makes a second pass back down the find path to update each node so that it points directly to the root. Each call of findSet(x) returns p[x] in line 3. If x is the root, then line 2 is not executed and p[x] = x is returned. This is the case in which the recursion bottoms out. Otherwise, line 2 is executed, and the recursive call with parameter p[x] returns a pointer to the root. Line 2 updates node x to point directly to the root, and this pointer is returned in line 3.

3.3 Effect of the heuristics on the running time

Separately, either union by rank or path compression improves the running time of the operations on disjoint-set forests, and the improvement is even greater when the two heuristics are used together. Alone, union by rank yields a running time of $O(m \log n)$, and this bound is tight.

When we use both union by rank and path compression, the worst-case running time is $O(m\alpha(n))$, where $\alpha(n)$ is a very slowly growing function. We can view the running time as linear in m in all practical situations.

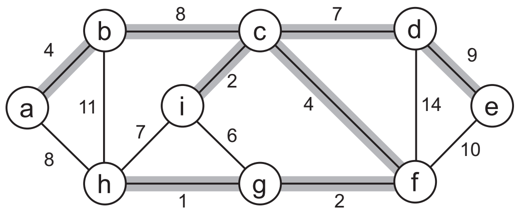
4 Application of Disjoint Sets

One of application of disjoint sets is Kruskal's algorithm to find minimum spanning trees of given graph $G = (V, E)$.

We wish to find an acyclic subset $T \subseteq E$ that connects all of the vertices and whose total weight

$$w(T) = \sum_{(u,v)\in T} w(u, v)$$

is minimized. Since T is acyclic and connects all of the vertices, it must form a tree, which we call a spanning tree. We call the problem of determining the tree T the minimum spanning-tree (MST) problem.



The figure shows a minimum spanning tree for a connected graph. In this example, the weights on edges are shown, and the edges in a minimum spanning tree are shaded. The total weight of the tree is 37.

Assume that we have a connected, undirected graph $G = (V, E)$ with a weight function $w : E \to R$, and we wish to find a MST for G. We use a greedy approach. It is captured by the "generic" algorithm, which grows the minimum spanning tree one edge at a time. The algorithm manages a set of edges T, maintaining the following loop invariant:

- Prior to each iteration, T is a subset of some minimum spanning tree.

At each step, we determine an edge (u, v) that can be added to T without violating this invariant, in the sense that $T \cup \{(u, v)\}$ is also a subset of a MST. We call such an edge a safe edge for T, since it can be safely added to T while maintaining the invariant.

Here is the Generic Algorithm for MST.

```
genericMST(G, w)
    T = empty
    while T does not form a spanning tree
        find an edge (u, v) that is safe for T
        T = T ∪ {(u, v)}
    return T
```

This is a "generic" algorithm which can be a basis to implement different algorithms. Kruskal's algorithm is based on the generic algorithm employing the disjoint sets data structure as follows.

```
kruskal(g)
    MST = empty
    edges = E

    sort edges in ascending order of their weights

    # create disjoint sets of |V| elements
    DisjointSet ds(|V|)

    for e in edges
        u = the first end-point of e
        v = the second end-point of e

        if ds.findSet(u) != ds.findSet(v)
            ds.unite(u, v)
            add e to MST
```

The computational complexity depends on the sorting of the edges, so it will be O($|E|$ log $|E|$).

5 Assignments

Place

https://onlinejudge.u-aizu.ac.jp/beta/room.html#CSC11F_2023_Week_02

Duration

1 week.

Problem A: Disjoint Set: Union Find Tree

Write a program which manipulates a disjoint set $S = \{S_1, S_2, ..., S_k\}$. First of all, the program should read an integer n, then make a disjoint set where each element consists of $0, 1, ...n-1$ respectively.

Next, the program should read an integer q and manipulate the set for q queries. There are two kinds of queries for different operations:

- unite(x, y): unites sets that contain x and y, say S_x and S_y, into a new set.
- same(x, y): determine whether x and y are in the same set.

Input

$n\ q$
$com_1\ x_1\ y_1$
$com_2\ x_2\ y_2$
...
$com_q\ x_q\ y_q$

In the first line, n and q are given. Then, q queries are given where com represents the type of queries. '`0`' denotes *unite* and '`1`' denotes *same* operation.

Output

For each *same* operation, print `1` if x and y are in the same set, otherwise `0`, in a line.

Constraints

- $1 \leq n \leq 10,000$
- $1 \leq q \leq 100,000$
- $0 \leq x, y < n$
- $x \neq y$

Sample Input

```
5 12
0 1 4
0 2 3
1 1 2
1 3 4
1 1 4
1 3 2
0 1 3
1 2 4
1 3 0
0 0 4
1 0 2
1 3 0
```

Sample Output

```
0
0
1
1
1
0
1
1
```

Problem B: Minimum Spanning Tree

Find the sum of weights of edges of the Minimum Spanning Tree for a given weighted undirected graph $G = (V, E)$.

Input

$|V|\ |E|$
$s_0\ t_0\ w_0$
$s_1\ t_1\ w_1$
:
$s_{|E|-1}\ t_{|E|-1}\ w_{|E|-1}$

, where $|V|$ is the number of vertices and $|E|$ is the number of edges in the graph. The graph vertices are named with the numbers $0, 1, ..., |V|-1$ respectively.

s_i and t_i represent source and target verticess of i-th edge (undirected) and w_i represents the weight of the i-th edge.

Output

Print the sum of the weights of the Minimum Spanning Tree.

Constraints

- $1 \leq |V| \leq 10,000$
- $0 \leq |E| \leq 100,000$
- $0 \leq w_i \leq 10,000$
- The graph is connected
- There are no parallel edges
- There are no self-loops

Sample Input 1

```
4 6
0 1 2
1 2 1
2 3 1
3 0 1
0 2 3
1 3 5
```

Sample Output 1

```
3
```

Sample Input 2

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

Sample Output 2

```
5
```

Problem C: Weighted Union Find Trees

There is a sequence $A = a_0, a_1, ..., a_{n-1}$. You are given the following information and questions.

- relate(x, y, z): a_y is greater than a_x by z
- diff(x, y): report the difference between a_x and a_y $(a_y - a_x)$

Input

n q
$query_1$
$query_2$
:
$query_q$

In the first line, n and q are given. Then, q information/questions are given in the following format:

`0` x y z

or

`1` x y

where '`0`' of the first digit denotes the *relate* information and '`1`' denotes the *diff* question.

Output

For each *diff* question, print the difference between a_x and a_y $(a_y - a_x)$.

Constraints

- $2 \leq n \leq 100,000$
- $1 \leq q \leq 200,000$
- $0 \leq x, y < n$
- $x \neq y$
- $0 \leq z \leq 10000$
- There are no inconsistency in the given information

Sample Input

```
5 6
0 0 2 5
0 1 2 3
1 0 1
1 1 3
0 1 4 8
1 0 4
```

Sample Output

```
2
?
10
```