Data Structures and Algorithms

Conf. dr. ing. Guillaume Ducoffe

guillaume.ducoffe@fmi.unibuc.ro

Algorithms on **Sets**

Reminder: Set = Unordered data collection with no repeated element.

Q1: How to represent a set?

Today's main objectives: Operations on Sets

- Disjoint Sets Data Structure ("Union Find")
- Partition Refinement.

Representation of a Set

<u>Reminder</u>: no repeated elements.

$\rightarrow$ Could be naively checked by maintaining a sorted collection.

Ex.: self-balanced Binary Search Trees.

A more efficient (and general) approach:

Element Uniqueness

Input: a dataset
Question: are all elements pairwise different?

- Can be solved using a **Hash Table**.

$\rightarrow$ A Set = a Hash Table + *Any* Data structure on the same elements.

Disjoint Sets

A Disjoint Sets Data Structure maintains a collection of pairwise disjoint sets. It supports the following three basic operations:

- `makeset(x)`: If x is not already present in the collection, then add a new singleton set whose unique elements equals x.
- `find(x)`: outputs the unique identifier of the set containing x.

  $\rightarrow$ In general, `find(x)` outputs an element of the set, also called its "representative".
  $\rightarrow$ We may force this representative to have special properties (*e.g.*, largest element in the set) with no computational overhead.
- `union(x,y)`: merge the respective sets of x,y into one.

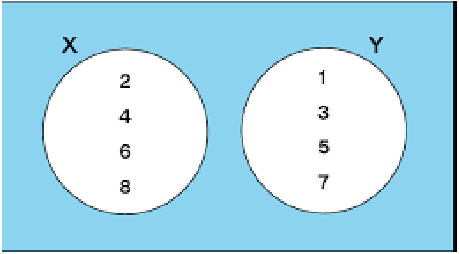
  $\rightarrow$ In some implementations, has $\mathcal{O}(n)$ **worst-case** complexity. But the amortized cost can be much lower than that.

Naive implementation

Assumption for what follows: The universe is $\{0, 1, 2, \ldots, n-1\}$.

- We simply store an n-vector associating to each element in a set its representative.

```
typedef vector<int> DisjointSets;
```



`[-1,1,2,1,2,1,2,1,2]`

Operations

```
//Complexity: O(1)
void makeset(DisjointSets& F, int x) {
   if(F[x] == -1) F[x] = x;
}
```

```
//Complexity: O(1)
int find(DisjointSets& F, int x) {
   return F[x];
}
```

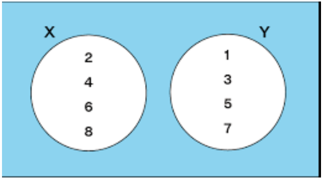
```
//Complexity: O(n)
void union(DisjointSets& F, int x, int y) {
   for(int i = 0; i < F.size(); i++)
      if(F[i]==F[x]) F[i] = F[y];
}
```

The Amortized cost of union also is $\mathcal{O}(n)$: consider union(0,1), union(0,2), ..., union(0,i), ... union(0,n-1).

A Better Approach

- We keep the vector of representatives. However, for each element x that is a representative, we now keep the list of all elements in its set in a separate array of lists (indexed by all elements).

```
typedef struct {
   vector<int> rep;
   vector< list<int> > set;
} DisjointSets;
```



rep: [-1,1,2,1,2,1,2,1,2]

set: [[],[1,3,5,7],[2,4,6,8],[],[],[],[],[],[]]

Operations

```
//Complexity: O(1)
void makeset(DisjointSets& F, int x) {
    if(F.rep[x] == -1) {
        F.rep[x] = x; F.set[x].push_back(x);
    }
}

//Complexity: O(1)
int find(DisjointSets& F, int x) {
    return F.rep[x];
}

//Complexity: O(n)
void union(DisjointSets& F, int x, int y) {
    int p = F.rep[x], q = F.rep[y];
    //Always merge the smallest set
    if(F.set[p].size() <= F.set[q].size()) {
        for(int i :  F.set[p]) { F.set[q].push_back(i); F.rep[i]=q; }
        F.set[p].erase(F.set[p].begin(),F.set[p].end());
    }else union(F,y,x);
}
```

Amortized complexity

Theorem

The cost of executing m operations is in $\mathcal{O}(m \log n)$.

$\rightarrow$ Equivalently: Amortized complexity is in $\mathcal{O}(\log n)$.

Proof:

1) We pay $\mathcal{O}(1)$ per makeset/find operation: $\mathcal{O}(m)$ in total

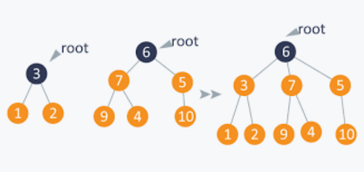
2) Each time an element changes her set, the size of her set doubles (at least).

3) Consequence: each element changes her set at most $\mathcal{O}(\log n)$ times.

4) The total number of elements changing their group at least once is no more than m.

A different perspective: Representing sets as **trees**

- The elements of each set are the nodes of a tree, whose root is the representative of this set.



Consequence: We can simulate Disjoint Sets with a **Dynamic Forest**

- `makeset(x)`: create a new tree whose unique node is x
- `find(x)`: reduces to `findRoot`
- `union(x,y)`: reduces to `link(findroot(x),findroot(y))`

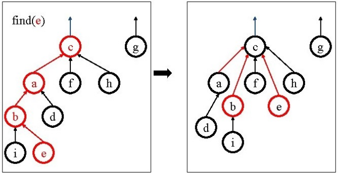
$\Longrightarrow$ $\mathcal{O}(\log n)$ worst-case per operation.

Improvements: Path Compression

Operation `find`

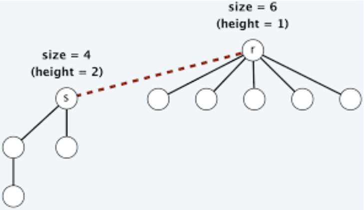
In order to access to the root (representative), we climb in the tree. *On our way, all visited nodes are reconnected as children of the root*.

$\rightarrow$ Speed-up of subsequent `find` operations.



Improvements: Union by rank/size

- Solution 1: Union by size.
    - Each node stores the size of its rooted subtree. If we merge two sets, then the root of the new set is the root of the biggest tree.



- Solution 2: Union by ranks.
    - Each node keeps a rank: **upper bound** on its depth. If we merge two sets, then the root of the new set is the root of larger rank.

Remark: both approaches ensure logarithmic depth.

Optimality

- Find/Union in worst-case $\mathcal{O}(\log n)$. – Trivial.

Define $\log^{(i)} n = \log\left(\log^{(i-1)} n\right)$.
Then, $\log^* n = \min\{i \mid \log^{(i)} n \leq 1\} \ll \log n$ is the iterated logarithm.

- (Hopcroft & Ullman, 1973): Find/Union in amortized $\mathcal{O}(\log^* n)$.

Optimality

- Find/Union in worst-case $\mathcal{O}(\log n)$. – Trivial.

Define $\log^{(i)} n = \log\left(\log^{(i-1)} n\right)$.
Then, $\log^* n = \min\{i \mid \log^{(i)} n \leq 1\} \ll \log n$ is the iterated logarithm.

- (Hopcroft & Ullman, 1973): Find/Union in amortized $\mathcal{O}(\log^* n)$.

Recall **Ackermann function**:
$A(0, n) = n+1$; $A(m+1, 0) = A(m, 1)$; $A(m+1, n+1) = A(m, A(m+1, n))$.
Its **inverse** is $\alpha(m, n) = \min\{i \geq 1 \mid A(i, \lfloor m/n \rfloor) \geq \log n\}$.

Theorem (Tarjan, 1979)

The amortized complexity of Find/Union is in $\mathcal{O}(\alpha(n, m))$.

This result cannot be improved.

Union/Find with Deletions

<u>Remark</u>: the standard Disjoint-Set data structure does not support deletions.

- To support deletion, each node is augmented with a Boolean field: indicating whether this element got deleted.
- Each root (representative) stores two pieces of information:
  - The size of its tree (Rk: this is $>$ than the size of the set)
  - The number of deleted elements in its tree.
- `delete(x)`: mark x as deleted.

The representative of x (operation find) increases the counter of deleted elements. If more than half of the nodes are deleted, then we completely rebuild this tree (using makeset/union operations).

$\rightarrow$ Amortized complexity remains in $\mathcal{O}(\alpha(m, n))$ for m operations.

Special Cases

Definition

A graph is a pair (V, E), where each element of E (called an edge) is a two-set of V (elements of V are called vertices).

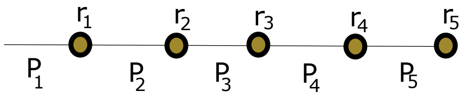
- Given a sequence $\mathcal{S}$ of m operations, define the graph $G(\mathcal{S})$ as follows:
  - For each `makeset(x)` operation, add a new vertex x to the graph.
  - For each `union(x,y)` operation, add an edge between x,y.

Theorem (Gabow & Tarjan, 1985)

If we are given the m operations in advance ("offline" setting) and $G(\mathcal{S})$ is a forest, then we can execute all m operations in $\mathcal{O}(m)$.

A simpler case: $G(\mathcal{S})$ is a path.

- We partition in $\mathcal{O}(n/b)$ sub-paths of length $\leq b$.



- If $b \ll logn$ then each sub-path is a binary word (Bitwise manipulation).

representative nodes $\Longleftrightarrow$ bits set to 0.

- We further maintain a classical Disjoint-Set data structure, *but* where in each set we only keep the roots r_i of the sub-paths.

Operations

- `makeset(x)`: corresponding bit set to 0
- `find(x)`: in the word of x's sub-path, find the next 0 after x.
  - Consider the word's complement (bitwise XOR). Discard all bits before x, Reverse the word, and then Use the logarithm function.

$\rightarrow$ Allows to find the representative if in the same sub-path.
$\rightarrow$ Otherwise, x is in the set of the sub-path root r. Call find operation on the Disjoint-Set data structure for roots. Let r' be the representative. Find the representative of r' in its sub-path.

- `union(x,y)`: x's bit set to 1. If x, y contain roots r_i in their respective sets, then also do a union in the Disjoint-Set data structure for the roots.
  - x's set contains a root $\iff$ all bits before/after x are set to 1.

Complexity: $\mathcal{O}(m) + \mathcal{O}(\frac{n}{b} \times \alpha(m, \frac{n}{b})) = \mathcal{O}(m)$

Partition Refinement

• Data Structure that maintains an ordered collection of pairwise disjoint sets, subject to the following basic operations:

- `init(V)`: initialize the structure with one set, equal to V.
- `refine(S)`: for each set X such that $X \cap S \neq \emptyset$ and $X \setminus S \neq \emptyset$, we replace X by the two consecutive new sets $X \cap S$ and $X \setminus S$.

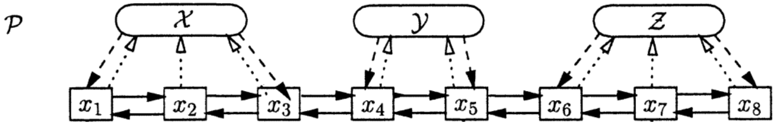
• Operation `init(V)` is in worst-case $\mathcal{O}(|V|)$. Each operation `refine(S)` is in worst-case $\mathcal{O}(|S|)$. Note that these are optimal runtimes!

Implementation

- Elements in V are maintained in a doubly-linked list $\mathcal{L}$, such that all elements in a same set X are consecutive.

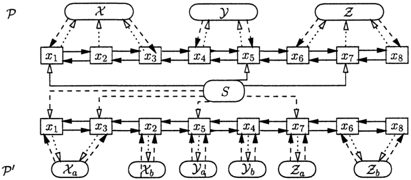
- Each set X of the partition is represented by a structure with two fields: pointers to its first and last elements in $\mathcal{L}$.

- Each node in the list $\mathcal{L}$ further stores a pointer to the set of its element. This mutual dependency between $\mathcal{L}$ and the set structures can be overcome by using an auxiliary Hash-table.



Refinement

- To each set X, associate an empty list $L[X]$.
- For each $s \in S$, access to its set X and add a pointer to the node containing s in $L[X]$. Put a pointer to X in an auxiliary Hash-table $\mathcal{H}$ (the keys of $\mathcal{H}$ are the sets intersecting S).
- For each set X in $\mathcal{H}$, if $L[X] \neq X$, then:
  - Update the first and last element of X as its first and last element snot in S (forward/backward search in $\mathcal{L}$).
  - Remove all elements in $L[X]$ from $\mathcal{L}$;
  - Reinsert $L[X]$ immediately before the first element of X (or immediately after the last element of X);
  - Create a new set from $L[X]$.



Questions