
Data Structures for Disjoint Set

Union-Find Data Structure

Disjoint Set Data Structure

Disjoint Set Data Structure

- ▶ Storing a family of sets $\mathcal{S} = \{S_1, S_2, \dots, S_k\}$ with $i \neq j \rightarrow S_i \cap S_j = \emptyset$.
- ▶ Each set S_i is identified by a *representative* $s_i \in S_i$.
- ▶ Three operations: *Make-Set*, *Union*, *Find-Set*

Make-Set(x)

- ▶ Creates a new set $\{x\}$. (Clearly, x is representative of the set.)
- ▶ x cannot be in any other set already.

Union(x, y)

- ▶ Merges the sets containing x and y into one set.

Find-Set(x)

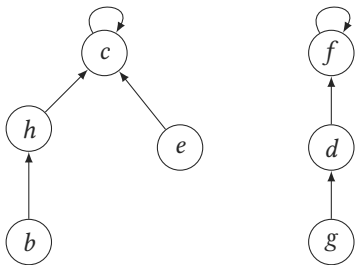
- ▶ Finds the representative of the set containing x .

Implementation

Idea

- ▶ Represent each set S_i as rooted tree (i. e., \mathcal{S} is a forest).
- ▶ Root of tree is representative

Example: $\mathcal{S} = \{\{b, c, e, h\}, \{d, f, g\}\}$

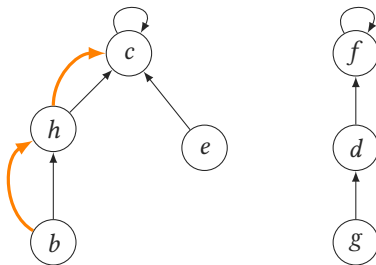


Implementation — Find-Set

Find-Set

- ▶ Follow pointers to root.

Example: $\text{Find}(b) = c$



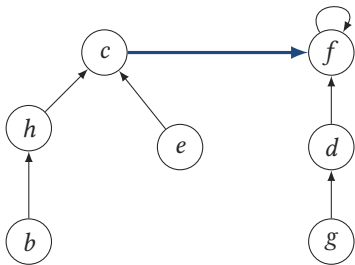
```
1 Procedure Find-Set( $x$ )  
2   While  $\text{par}(x) \neq x$   
3     Let  $x := \text{par}(x)$ .  
4   Return  $x$ 
```

Implementation — Union

Union(x, y)

- ▶ Find the representatives r_x and r_y of x and y (i. e., find roots of trees).
- ▶ Make r_x parent of r_y

Example: Union(b, g)



Implementation — Union

- 1 **Procedure** *Union*(x, y)
- 2 | Set par(Find-Set(x)) := Find-Set(y)

Implementation

Questions

- ▶ What is the worst-case runtime for these operations?
- ▶ Can we improve the runtime?

Questions

- ▶ What is the worst-case runtime for these operations?
- ▶ Can we improve the runtime?

Example

- ▶ Assume that we perform $\text{Union}(1, 2)$, $\text{Union}(1, 3)$, $\text{Union}(1, 4)$, \dots , $\text{Union}(1, n)$.
- ▶ Then, the runtime is in $O(n^2)$.

Improving Find-Set

Observation

- ▶ If we use Find-Set multiple times on the same element, we have to search for the root each time again.

Idea

- ▶ Update the parent pointer when calling Find-Set such that it points on the root.

```
1 Procedure Find-Set( $x$ )
2   If  $\text{par}(x) \neq x$  Then
3     Set  $\text{par}(x) := \text{Find-Set}(\text{par}(x))$ 
4   Return  $\text{par}(x)$ 
```

Improving Union

Idea: Union by Rank

- ▶ Keep track of height of a tree.
Number is denoted as *rank* of a vertex.
- ▶ Make root of smaller tree child of root of larger tree.

1 **Procedure** *Make-Set*(x)

2 $\text{par}(x) := x$

3 $\text{rank}(x) := 0$

Observation

- ▶ We only need to keep track of the rank of the root.

Improving Union

```
1 Procedure Union( $x, y$ )
2   Let  $x := \text{Find-Set}(x)$ .
3   Let  $y := \text{Find-Set}(y)$ .
4   If  $\text{rank}(x) > \text{rank}(y)$  Then
5     | Set  $\text{par}(y) := x$ .
6   Else
7     | Set  $\text{par}(x) := y$ .
8     | If  $\text{rank}(x) = \text{rank}(y)$  Then
9       | Set  $\text{rank}(y) := \text{rank}(y) + 1$ .
```

Assume our sets contain n elements in total.

Runtime

- ▶ Worst case for single operation: $O(\log n)$ (Why?)
- ▶ Worst case for m operations: $O(m \cdot \alpha(n))$
Thus, $O(\alpha(n))$ amortised runtime per operation.

α -Function

- ▶ Inverse Ackermann function
- ▶ $\alpha(\text{atoms in the universe}) \leq 4$
- ▶ Grows extremely slow. However, it is strictly speaking not constant.

Partition Refinement

Partition Refinement

Union-Find

- ▶ Start with a partition $\mathcal{P} = \{S_1, S_2, \dots, S_k\}$ of a set \mathcal{S}
- ▶ Step by step join two sets S_i and S_j together.
- ▶ *Union*(i, j): $\mathcal{P} := (\mathcal{P} \setminus \{S_i, S_j\}) \cup \{S_i \cup S_j\}$

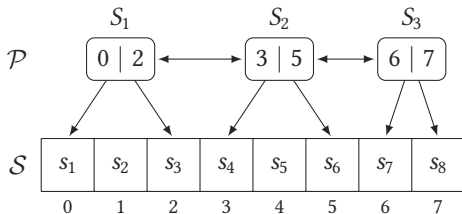
Partition Refinement

- ▶ Start with a partition $\mathcal{P} = \{S_1, S_2, \dots, S_k\}$ of a set \mathcal{S} (often $\mathcal{P} = \{\mathcal{S}\}$)
- ▶ Step by step, based on a set $X \subseteq \mathcal{S}$, split subsets S_i into $S_i \setminus X$ and $S_i \cap X$.
- ▶ *Refine*(X): $\mathcal{P} := \{S \setminus X, S \cap X \mid S \in \mathcal{P}\}$

Implementation – Data Structure

Data Structure

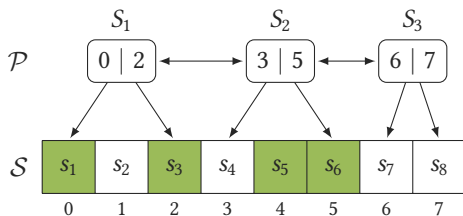
- ▶ Set \mathcal{S} is an array storing all its elements.
- ▶ Partition \mathcal{P} is a (doubly-linked) list of subsets S_i
- ▶ Subset S_i is represented by two integers which describe the interval (i. e., the first and last index) of S_i in the array \mathcal{S}



Implementation – Refinement

Refinement

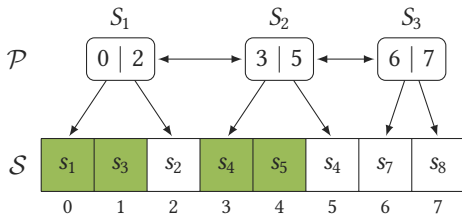
- ▶ Flag all elements $s_i \in X$.



Implementation – Refinement

Refinement

- ▶ Flag all elements $s_i \in X$.
- ▶ For each subset S_i
 - ▶ Reorder such that flagged elements are in front.



Implementation – Refinement

Refinement

- ▶ Flag all elements $s_i \in X$.
- ▶ For each subset S_i
 - ▶ Reorder such that flagged elements are in front.
 - ▶ Split S_i into two sets containing only flagged or non-flagged elements.

