Data Structures for Disjoint Set

Union-Find Data Structure

Disjoint Set Data Structure

- Storing a family of sets $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ with $i \neq j \to S_i \cap S_j = \emptyset$.
- 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)

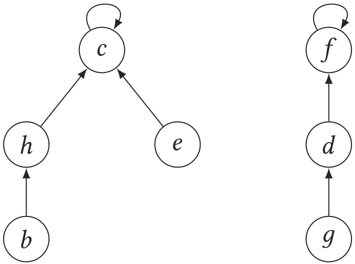
- Finds the representative of the set containing x.

Implementation

Idea

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

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

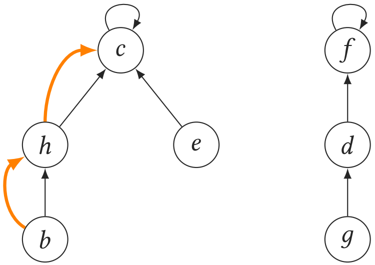


Implementation — Find-Set

Find-Set

- Follow pointers to root.

Example: Find$(b) = c$



Implementation — Find-Set

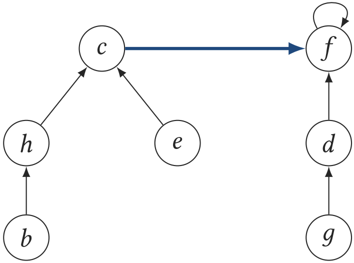
```
Procedure Find-Set(x)
    While par(x) ≠ x
        Let x := par(x).
    Return x
```

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

Procedure *Union*(x, y)
    Set $\mathrm{par}(\text{Find-Set}(x)) := \text{Find-Set}(y)$

Implementation

Questions

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

Implementation

Questions

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

Example

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

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.

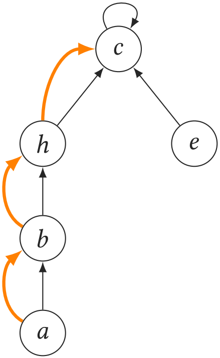
Idea

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

```
Procedure Find-Set(x)
    If par(x) ≠ x Then
        Set par(x) := Find-Set(par(x))
    Return par(x)
```

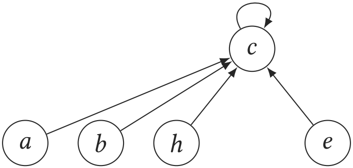
Improving Find-Set

Example: Find(a)



Improving Find-Set

Example: Find(a)



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.

Procedure *Make-Set(x)*
  $\mathrm{par}(x) := x$
  $\mathrm{rank}(x) := 0$

Observation

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

Improving Union

```
Procedure Union(x, y)
    Let x := Find-Set(x).
    Let y := Find-Set(y).
    If rank(x) > rank(y) Then
        Set par(y) := x.
    Else
        Set par(x) := y.
        If rank(x) = rank(y) Then
            Set rank(y) := rank(y) + 1.
```

Runtime

Assume our sets contain n elements in total.

Runtime

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

α-Function

- Inverse Ackermann function
- α(atoms in the universe) ≤ 4
- Grows extremely slow. However, it is strictly speaking not constant.

Partition Refinement

Partition Refinement

Union-Find

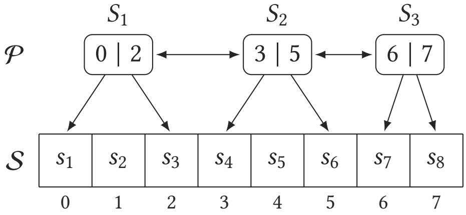
- Start with a partition $\mathcal{P} = \{S_1, S_2, \ldots, S_k\}$ of a set $\mathcal{S}$
- Step by step join two sets S_i and S_j together.
- *Union(i, j)*: $\mathcal{P} := \big(\mathcal{P} \setminus \{S_i, S_j\}\big) \cup \{S_i \cup S_j\}$

Partition Refinement

- Start with a partition $\mathcal{P} = \{S_1, S_2, \ldots, S_k\}$ 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)*: $\mathcal{P} := \{\, S \setminus X, S \cap X \mid S \in \mathcal{P} \,\}$

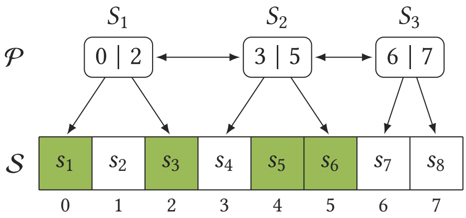
Data Structure

- 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 $\mathcal{S}$



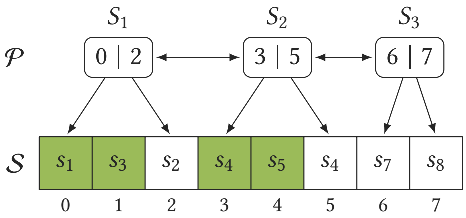
Refinement

- Flag all elements $s_i \in X$.



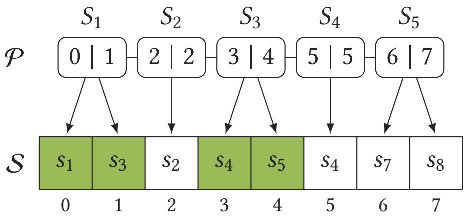
Refinement

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



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.



Computing Twins

Twins

In a graph G, two vertices u and v are called *twins* if they have the same neighbourhood. We distinguish between *true twins* where $N[u] = N[v]$ and *false twins* where $N(u) = N(v)$.

Algorithm Idea

- Let $\mathcal{S}$ be a copy of V and $\mathcal{P} = \{\mathcal{S}\}$.
- For each vertex v of G, Refine$\big(N[v]\big)$.

Observation

- Two vertices u and v are twins if and only if there is no vertex w such that Refine$\big(N[w]\big)$ separates u and v.
- Therefore, in the resulting partition $\mathcal{P} = \{S_1, S_2, \ldots, S_k\}$, each subset S_i is a set of twins.