Chapter 23

Union-Find

CS 573: Algorithms, Fall 2013
November 14, 2013

23.1 Union Find

23.2 Union-Find

23.2.1 Requirements from the data-structure

23.2.1.1 Requirements from the data-structure

(A) Maintain a collection of sets.
(B) **makeSet**(x) - creates a set that contains the single element x.
(C) **find**(x) - returns the set that contains x.
(D) **union**(A, B) - returns set = union of A and B. That is $A \cup B$.
... merges the two sets A and B and return the merged set.

23.2.2 Amortized analysis

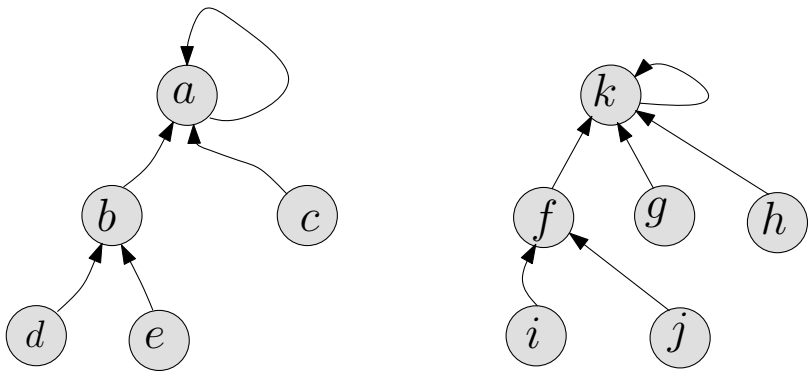
23.2.2.1 Amortized Analysis

(A) Use data-structure as a black-box inside algorithm.
... Union-Find in Kruskal algorithm for computing MST.
(B) Bounded worst case time per operation.
(C) Care: *overall* running time spend in data-structure.
(D) ***amortized running-time*** of operation is the average time it takes to perform an operation on the data-structure.
(E) Amortized time of an operation is $\dfrac{\text{overall running time}}{\text{number of operations}}$.

23.2.3 The data-structure

23.2.4 Reversed Trees

23.2.4.1 Representing sets in the Union-Find DS



The Union-Find representation of the sets $A = \{a, b, c, d, e\}$ and $B = \{f, g, h, i, j, k\}$. The set A is uniquely identified by a pointer to the root of A, which is the node containing a.

23.2.5 Reversed Trees

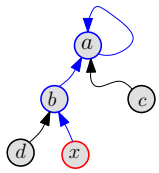
23.2.5.1 !esrever ni retteb si gnihtyreve esuaceB

(A) Reversed Trees:
  (A) Every element is stored in its own node.
  (B) A node has one pointer to its parent.
  (C) A set is uniquely identified with the element stored in the root.

(B) **makeSet**: Create a singleton pointing to itself: 

(C) **find**(x): Start from node containing x, traverse up the tree (using parent pointers), till arriving to root.

Thus, doing a **find**(x) operation in the reversed tree shown on the right, involve going up the tree from $x \to b \to a$, and returning a as the set.

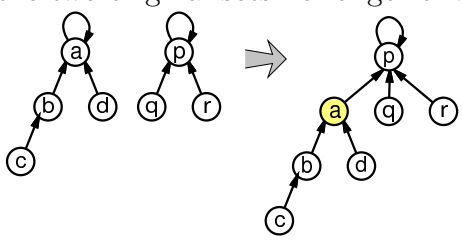


23.2.6 Union operation in reversed trees

23.2.6.1 Just hang them on each other.

union(a, p): Merge two sets.

(A) Hanging the root of one tree, on the root of the other.
(B) A destructive operation, and the two original sets no longer exist.



23.2.6.2 Pseudo-code of naive version...

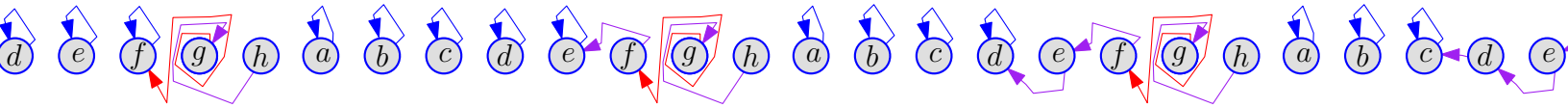
```
makeSet(x)
  p̄(x) ← x
```

```
find(x)
  if x = p̄(x) then
    return x
          return
find(p̄(x))
```

```
union( x, y )
  A ← find(x)
  B ← find(y)
  p̄(B) ← A
```

23.2.7 Example...

23.2.7.1 The long chain



After: **makeSet**(a), **makeSet**(b), **makeSet**(c), **makeSet**(d), **makeSet**(e), **makeSet**(f), **makeSet**(g), **makeSet**(h)

union(g, h)

union(f, g)

union(e, f)

union(d, e)

union(c, d)

union(b, c)

union(a, b)

23.2.7.2 Find is slow, hack it!

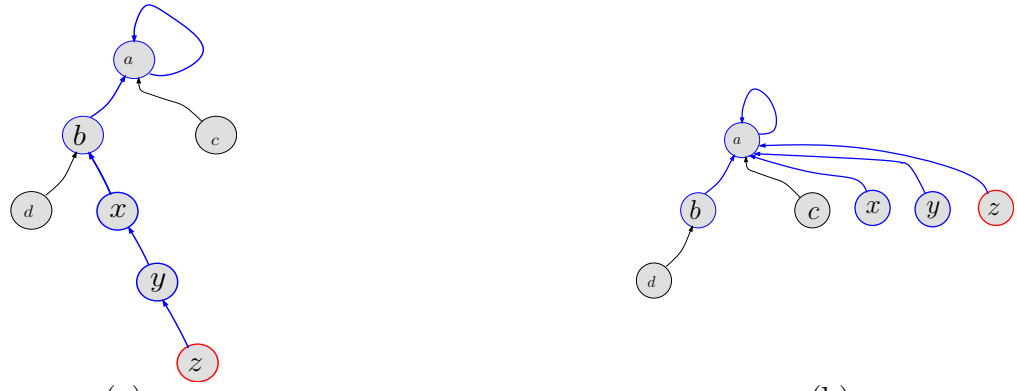
find might require $\Omega(n)$ time.

So, the question is how to further improve the performance of this data-structure. We are going to do this, by using two "hacks":

(i) ***Union by rank***: Maintain for every tree, in the root, a bound on its depth (called ***rank***). Always hang the smaller tree on the larger tree.

(ii) ***Path compression***: Since, anyway, we travel the path to the root during a **find** operation, we might as well hang all the nodes on the path directly on the root.

23.2.7.3 Path compression in action...



(a) (b)

(a) The tree before performing **find**(z), and (b) The reversed tree after performing **find**(z) that uses path compression.

23.2.7.4 Pseudo-code of improved version...

```
makeSet(x)
  p̄(x) ← x
  rank(x) ← 0
```

```
find(x)
  if x ≠ p̄(x) then
    p̄(x) ← find(p̄(x))
  return p̄(x)
```

```
union(x, y )
  A ← find(x)
  B ← find(y)
  if rank(A) > rank(B) then
    p̄(B) ← A
  else
    p̄(A) ← B
    if rank(A) = rank(B) then
      rank(B) ← rank(B) + 1
```

23.3 Analyzing the Union-Find Data-Structure

23.3.0.5 Definition

Definition 23.3.1. *A node in the union-find data-structure is a* **leader** *if it is the root of a (reversed) tree.*

23.3.0.6 Lemma

Lemma 23.3.2. *Once a node stop being a leader (i.e., the node in top of a tree), it can never become a leader again.*

Proof: Note, that an element x can stop being a leader only because of a **union** operation that hanged x on an element y. From this point on, the only operation that might change x parent pointer, is a **find** operation that traverses through x. Since path-compression can only change the parent pointer of x to point to some other element y, it follows that x parent pointer will never become equal to x again. Namely, once x stop being a leader, it can never be a leader again. ■

23.3.0.7 Another Lemma

Lemma 23.3.3. *Once a node stop being a leader then its rank is fixed.*

Proof: The rank of an element changes only by the **union** operation. However, the **union** operation changes the rank, only for elements that are leader after the operation is done. As such, if an element is no longer a leader, than its rank is fixed. ■

23.3.0.8 Ranks are strictly monotonically increasing

Lemma 23.3.4. *Ranks are monotonically increasing in the reversed trees, as we travel from a node to the root of the tree.*

23.3.0.9 Proof...

(A) Claim: $\forall u \to v$ in DS: $\text{rank}(u) < \text{rank}(v)$.
(B) Proof by induction. Base: all singletons. Holds.
(C) Assume claim holds at time t, before an operation.
(D) If operation is **union** (A, B), and assume that we hanged $\text{root}(A)$ on $\text{root}(B)$.
Must be that $\text{rank}(\text{root}(B))$ is now larger than $\text{rank}(\text{root}(A))$ (verify!).
Claim true after operation!
(E) If operation **find**: traverse path π, then all the nodes of π are made to point to the last node v of π.
By induction, $\text{rank}(v) >$ rank of all other nodes of π.
All the nodes that get compressed, the rank of their new parent, is larger than their own rank. ■

23.3.0.10 Trees grow exponentially in size with rank

Lemma 23.3.5. *When a node gets rank k than there are at least $\geq 2^k$ elements in its subtree.*

Proof: The proof is by induction. For $k = 0$ it is obvious since a singleton has a rank zero, and a single element in the set. Next observe that a node gets rank k only if the merged two roots has rank $k-1$. By induction, they have 2^{k-1} nodes (each one of them), and thus the merged tree has $\geq 2^{k-1} + 2^{k-1} = 2^k$ nodes. ■

23.3.0.11 Having higher rank is rare

Lemma 23.3.6. *# nodes that get assigned rank k throughout execution of Union-Find DS is at most $n/2^k$.*

Proof: Again, by induction. For $k = 0$ it is obvious.

Charge a node v of rank k to two elements u and v of rank $k-1$ that were leaders used to create new larger set.

After the merge v is of rank k and u is of rank $k-1$ and it is no longer a leader. Thus, we can charge this event to the two (no longer active) nodes of degree $k-1$. Namely, u and v.

By induction: algorithm created at most $n/2^{k-1}$ nodes of rank $k-1 \implies$ # nodes of rank k created by algorithm is $\leq \left(n/2^{k-1}\right)/2 = n/2^k$. ■

23.3.0.12 Find takes logarithmic time

Lemma 23.3.7. *The time to perform a single* **find** *operation when we perform union by rank and path compression is* $O(\log n)$ *time.*

Proof: The rank of the leader v of a reversed tree T, bounds the depth of a tree T in the Union-Find data-structure. By the above lemma, if we have n elements, the maximum rank is $\lg n$ and thus the depth of a tree is at most $O(\log n)$. ■

23.3.0.13 $\log^*$ in detail

$\log^*(n)$: number of times one has to take lg of a number to get a number smaller than two.

Thus, $\log^* 2 = 1$ and $\log^* 2^2 = 2$. Similarly, $\log^* 2^{2^2} = 1 + \log^*(2^2) = 2 + \log^* 2 = 3$. Similarly, $\log^* 2^{2^{2^2}} = \log^*(65536) = 4$.

Things get really exciting, when one considers

$$\log^* 2^{2^{2^{2^2}}} = log^* 2^{65536} = 5.$$

However, $\log^*$ is a monotone increasing function. And $\beta = 2^{2^{2^{2^2}}} = 2^{65536}$ is a huge number (considerably larger than the number of atoms in the universe). Thus, for all practical purposes, $\log^*$ returns a value which is smaller than 5.

23.3.0.14 Can do much better!

Theorem 23.3.8. *If we perform a sequence of* m *operations over* n *elements, the overall running time of the Union-Find data-structure is* $O((n+m)\log^* n)$.

(A) Intuitively: (in the amortized sense) Union-Find data-structure takes constant time per operation (unless n is larger than β which is unlikely).
(B) Not quite correct if n sufficiently large...

23.3.0.15 The tower function...

Definition 23.3.9. $\mathrm{Tower}(b) = 2^{\mathrm{Tower}(b-1)}$ *and* $\mathrm{Tower}(0) = 1$.

$\mathrm{Tower}(i)$: a tower of $2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}$ of height i.

Observe that $\log^*(\mathrm{Tower}(i)) = i$.

Definition 23.3.10. *For* $i \geq 0$, *let* $\mathrm{Block}(i) = [\mathrm{Tower}(i-1)+1, \mathrm{Tower}(i)]$*; that is*

$$\mathrm{Block}(i) = \left[z, 2^{z-1}\right] \qquad \textit{for} \qquad z = \mathrm{Tower}(i-1)+1.$$

Also $\mathrm{Block}(0) = [0,1]$. *As such,*

$\mathrm{Block}(0) = \left[0,1\right]$, $\mathrm{Block}(1) = \left[2,2\right]$, $\mathrm{Block}(2) = \left[3,4\right]$, $\mathrm{Block}(3) = \left[5,16\right]$, $\mathrm{Block}(4) = \left[17,65536\right]$, $\mathrm{Block}(5) = \left[65537, 2^{65536}\right]$ ...

23.3.0.16 Running time of find...

(A) RT of **find**(x) proportional to length of the path from x to the root of its tree.
(B) ...since start from x and we visit the sequence:
$x_1 = x, x_2 = \overline{\mathrm{p}}(x) = \overline{\mathrm{p}}(x_1), \ldots, x_i = \overline{\mathrm{p}}(x_{i-1}), \ldots, x_m =$ root of tree.
(C) $\mathrm{rank}(x_1) < \mathrm{rank}(x_2) < \mathrm{rank}(x_3) < \ldots < \mathrm{rank}(x_m)$.
(D) RT of **find**(x) is $O(m)$.

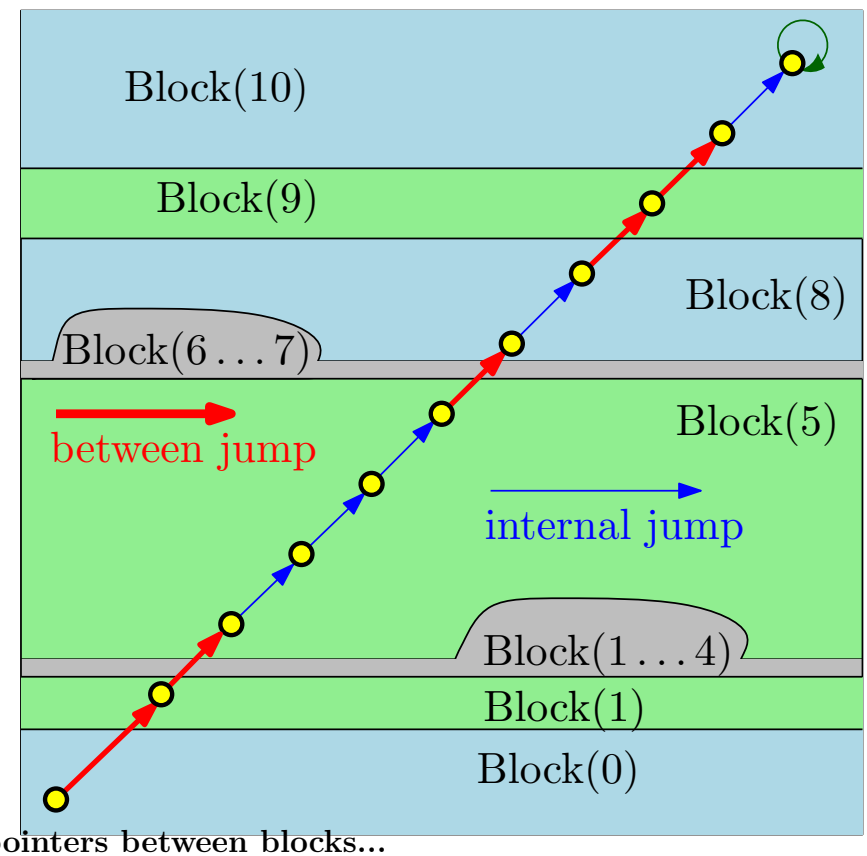
Definition 23.3.11. *A node x is in the ith block if* $\mathrm{rank}(x) \in \mathrm{Block}(i)$.

(E) Looking for ways to pay for the **find** operation.
(other two operations take constant time).

23.3.0.17 Blocks and jumping pointers

(A) maximum rank of node v is $O(\log n)$.
(B) # of blocks is $O(\log^* n)$, as $O(\log n) \in \mathrm{Block}(c \log^* n)$, ($c$: constant).
(C) **find** (x): π path used.
(D) partition π into each by rank.
(E) Price of **find** length π.
(F) For node x: $\nu = \mathrm{index}_{\mathrm{B}}(x)$ index of block containing $\mathrm{rank}(x)$. $\mathrm{rank}(x) \in \mathrm{Block}\Big(\mathrm{index}_{\mathrm{B}}(x)\Big)$.
(G) $\mathrm{index}_{\mathrm{B}}(x)$: ***block of*** x

23.3.0.18 The path of find operation, and its pointers



23.3.0.19 The pointers between blocks...

(A) During a **find** operation...
(B) π: path traversed.

(C) Ranks of the nodes visited in π monotone increasing.
(D) Once leave block ith, never go back!
(E) charge visit to nodes in π next to element in a different block...
(F) to total number of blocks $\leq O(\log^* n)$.

23.3.0.20 Jumping pointers

Definition 23.3.12. *π: path traversed by* **find**.
$x \in \pi$, $\overline{\mathrm{p}}(x)$ is in a different block, is a **jump between blocks**.
jump inside a block is an **internal jump** *(i.e., x and $\overline{\mathrm{p}}(x)$ are in same block).*

23.3.0.21 Not too many jumps between blocks

Lemma 23.3.13. *During a single* **find**(x) *operation, the number of jumps between blocks along the search path is $O(\log^* n)$.*

Proof: Consider the search path $\pi = x_1, \ldots, x_m$, and consider the list of numbers $0 \leq \mathrm{index}_{\mathrm{B}}(x_1) \leq \mathrm{index}_{\mathrm{B}}(x_2) \leq \ldots \leq \mathrm{index}_{\mathrm{B}}(x_m)$. We have that $\mathrm{index}_{\mathrm{B}}(x_m) = O(\log^* n)$. As such, the number of elements x in π such that $\mathrm{index}_{\mathrm{B}}(x) \neq \mathrm{index}_{\mathrm{B}}(\overline{\mathrm{p}}(x))$ is at most $O(\log^* n)$. ■

23.3.0.22 Benefits of an internal jump

(A) x and $\overline{\mathrm{p}}(x)$ are in same block.
(B) $\mathrm{index}_{\mathrm{B}}(x) = \mathrm{index}_{\mathrm{B}}(\overline{\mathrm{p}}(x))$.
(C) **find** passes through x.
(D) $r_{\mathrm{bef}} = \mathrm{rank}(\overline{\mathrm{p}}(x))$ before **find** operation.
(E) $r_{\mathrm{aft}} = \mathrm{rank}(\overline{\mathrm{p}}(x))$ after **find** operation.
(F) By path compression: $r_{\mathrm{aft}} > r_{\mathrm{bef}}$.
(G) $\implies$ parent pointer of x jump forward and the new parent has higher rank.
(H) Charge internal block jumps to this "progress".

23.3.0.23 Your parent can be promoted only a few times before leaving block

Lemma 23.3.14. *At most $|\mathrm{Block}(i)| \leq \mathrm{Tower}(i)$* **find** *operations can pass through an element x, which is in the ith block (i.e., $\mathrm{index}_{\mathrm{B}}(x) = i$) before $\overline{\mathrm{p}}(x)$ is* no longer *in the ith block. That is $\mathrm{index}_{\mathrm{B}}(\overline{\mathrm{p}}(x)) > i$.*

Proof: By above discussion, the parent of x increases its rank every-time an internal jump goes through x. Since there at most $|\mathrm{Block}(i)|$ different values in the ith block, the claim follows. The inequality $|\mathrm{Block}(i)| \leq \mathrm{Tower}(i)$ holds by definition:

$$\mathrm{Block}(i) = [\mathrm{Tower}(i-1) + 1, \mathrm{Tower}(i)]$$

■

23.3.0.24 Few elements are in the bigger blocks

Lemma 23.3.15. *There are at most $n/\mathrm{Tower}(i)$ nodes that have ranks in the ith block throughout the algorithm execution.*

Proof: By lemma, the number of elements with rank in the ith block

$$\leq \sum_{k \in \mathrm{Block}(i)} \frac{n}{2^k} = \sum_{k=\mathrm{Tower}(i-1)+1}^{\mathrm{Tower}(i)} \frac{n}{2^k}$$
$$= n \cdot \sum_{k=\mathrm{Tower}(i-1)+1}^{\mathrm{Tower}(i)} \frac{1}{2^k} \leq \frac{n}{2^{\mathrm{Tower}(i-1)}} = \frac{n}{\mathrm{Tower}(i)}.$$

■

23.3.0.25 Total number of internal jumps is $O(n)$

Lemma 23.3.16. *The number of internal jumps performed, inside the ith block, during the lifetime of the union-find data-structure is $O(n)$.*

Proof: An element x in the ith block, can have at most $|\mathrm{Block}(i)|$ internal jumps, before all jumps through x are jumps between blocks, by lemma... There are at most $n/\mathrm{Tower}(i)$ elements with ranks in the ith block, throughout the algorithm execution, by other Lemma. Thus, the total number of internal jumps is

$$|\mathrm{Block}(i)| \cdot \frac{n}{\mathrm{Tower}(i)} \leq \mathrm{Tower}(i) \cdot \frac{n}{\mathrm{Tower}(i)} = n.$$

■

23.3.0.26 Total number of internal jumps

Lemma 23.3.17. *The number of internal jumps performed by the Union-Find data-structure overall is $O(n \log^* n)$.*

Proof: Every internal jump can be associated with the block it is being performed in. Every block contributes $O(n)$ internal jumps throughout the execution of the union-find data-structures, by Lemma 16. There are $O(\log^* n)$ blocks. As such there are at most $O(n \log^* n)$ internal jumps. ■

23.3.0.27 Result...

Lemma 23.3.18. *The overall time spent on m* **find** *operations, throughout the lifetime of a union-find data-structure defined over n elements, is $O((m+n)\log^* n)$.*

Theorem 23.3.19. *If we perform a sequence of m operations over n elements, the overall running time of the Union-Find data-structure is $O((n+m)\log^* n)$.*