Chapter 21

Union-Find

CS 573: Algorithms, Fall 2014
November 6, 2014

21.1 Union Find

21.2 Union-Find

21.2.1 Requirements from the data-structure

21.2.1.1 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.

21.2.2 Amortized analysis

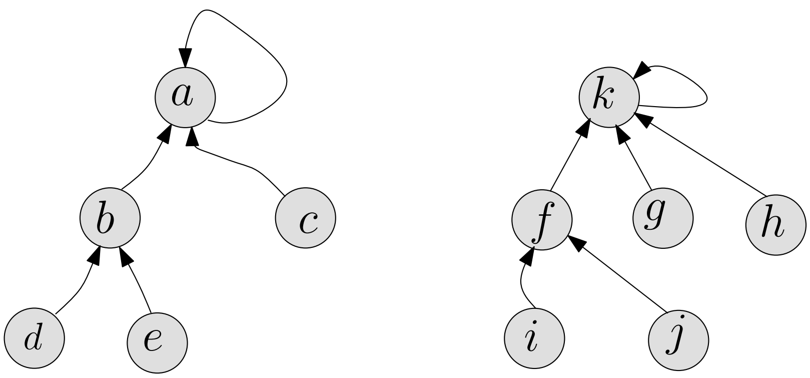
21.2.2.1 Amortized Analysis

(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
= average time to perform an operation on data-structure.
(E) Amortized time per operation $= \dfrac{\text{overall running time}}{\text{number of operations}}$.

21.2.3 The data-structure

21.2.4 Reversed Trees

21.2.4.1 Representing sets in the Union-Find DS

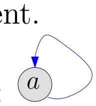


The Union-Find representation of the sets $A = \{a, b, c, d, e\}$ and $B = \{f, g, h, i, j, k\}$. The set A is uniquely identified by a pointer to the root of A, which is the node containing a.

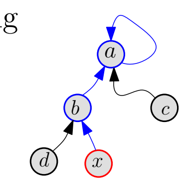
21.2.5 Reversed Trees

21.2.5.1 !esrever ni retteb si gnihtyreve esuaceB

(A) Reversed Trees:
  (A) Initially: Every element is its own node.
  (B) Node v: $\overline{\mathrm{p}}(v)$ pointer to its parent.
  (C) Set uniquely identified by root node/element.

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

(C) **find**(x):
  (A) Start from node containing x, traverse up tree, till arriving to root.
  (B) **find**(x):
  $x \to b \to a$
  (C) a: returned as set.

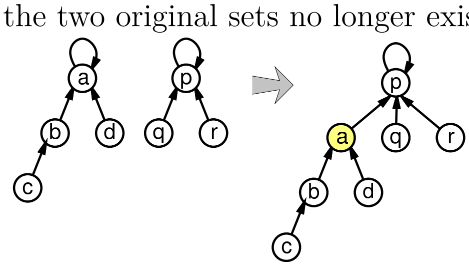


21.2.6 Union operation in reversed trees

21.2.6.1 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.



21.2.6.2 Pseudo-code of naive version...

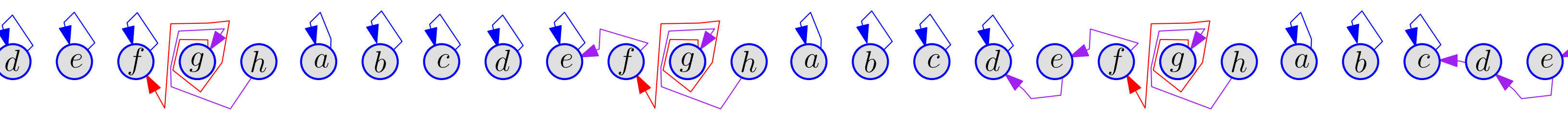
makeSet(x)
  $\overline{\mathrm{p}}(x) \leftarrow x$

find(x)
  if $x = \overline{\mathrm{p}}(x)$ **then**
    return x
  return **find**($\overline{\mathrm{p}}(x)$)

union(x, y)
  $A \leftarrow$ **find**(x)
  $B \leftarrow$ **find**(y)
  $\overline{\mathrm{p}}(B) \leftarrow A$

21.2.7 Example...

21.2.7.1 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)

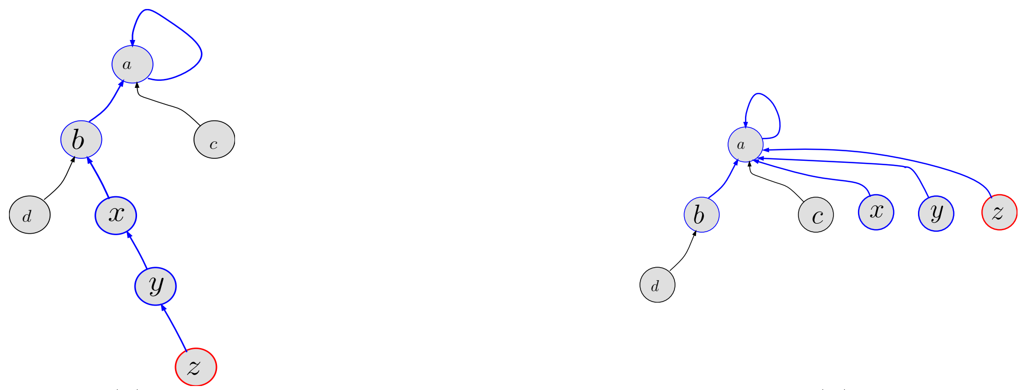
21.2.7.2 Find is slow, hack it!

(A) **find** might require $\Omega(n)$ time.
(B) **Q**: How improve performance?
(C) Two "hacks":

(i) ***Union by rank***:
Maintain in root of tree , a bound on its depth (***rank***).
Rule: Hang the smaller tree on the larger tree in **union**.

(ii) ***Path compression***:
During find, make all pointers on path point to root.

21.2.7.3 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.

21.2.7.4 Pseudo-code of improved version...

makeSet(x)
  $\overline{\mathrm{p}}(x) \leftarrow x$
  $\mathrm{rank}(x) \leftarrow 0$

find(x)
  if $x \neq \overline{\mathrm{p}}(x)$ **then**
    $\overline{\mathrm{p}}(x) \leftarrow$ **find**$(\overline{\mathrm{p}}(x))$
  return $\overline{\mathrm{p}}(x)$

union(x, y)
  $A \leftarrow$ **find**(x)
  $B \leftarrow$ **find**(y)
  if $\mathrm{rank}(A) > \mathrm{rank}(B)$ **then**
    $\overline{\mathrm{p}}(B) \leftarrow A$
  else
    $\overline{\mathrm{p}}(A) \leftarrow B$
    if $\mathrm{rank}(A) = \mathrm{rank}(B)$ **then**
      $\mathrm{rank}(B) \leftarrow \mathrm{rank}(B) + 1$

21.3 Analyzing the Union-Find Data-Structure

21.3.0.5 Definition

Definition 21.3.1. v: Node **UnionFind** data-structure $\mathcal{D}$
v is ***leader*** $\iff$ v root of a (reversed) tree in $\mathcal{D}$.

"When you're not a leader, you're little people."

21.3.0.6 Lemma

Lemma 21.3.2. *Once node v stop being a leader, can never become leader again.*

Proof: (A) x stopped being leader because **union** operation hanged x on y.
(B) From this point on...
(C) x might change only its parent pointer (**find**).
(D) x parent pointer will never become equal to x again.
(E) x never a leader again.

21.3.0.7 Another Lemma

Lemma 21.3.3. *Once a node stop being a leader then its rank is fixed.*

Proof: (A) rank of element changes only by **union** operation.
(B) **union** operation changes rank only for...
the "new" leader of the new set.
(C) if an element is no longer a leader, than its rank is fixed.

21.3.0.8 Ranks are strictly monotonically increasing

Lemma 21.3.4. *Ranks are monotonically increasing in the reversed trees...*
...along a path from node to root of the tree.

21.3.0.9 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. ∎

21.3.0.10 Trees grow exponentially in size with rank

Lemma 21.3.5. *When node gets rank $k \implies$ at least $\geq 2^k$ elements in its subtree.*

Proof: (A) Proof is by induction.
(B) For $k = 0$: obvious since a singleton has a rank zero, and a single element in the set.
(C) node u gets rank k only if the merged two roots u, v has rank $k - 1$.
(D) By induction, u and v have $\geq 2^{k-1}$ nodes before merge.
(E) merged tree has $\geq 2^{k-1} + 2^{k-1} = 2^k$ nodes.

21.3.0.11 Having higher rank is rare

Lemma 21.3.6. *# nodes that get assigned rank k throughout execution of Union-Find DS is at most $n/2^k$.*

Proof: (A) By induction. For $k = 0$ it is obvious.
(B) when v become of rank k. Charge to roots merged: u and v.
(C) Before union: u and v of rank $k - 1$
(D) After merge: $\text{rank}(v) = k$ and $\text{rank}(u) = k - 1$.
(E) u no longer leader. Its rank is now fixed.
(F) u, v leave rank $k - 1 \implies v$ enters rank k.
(G) By induction: at most $n/2^{k-1}$ nodes of rank $k - 1$ created.
$\implies$ # nodes rank k: $\leq \left(n/2^{k-1}\right)/2 = n/2^k$.

21.3.0.12 Find takes logarithmic time

Lemma 21.3.7. *The time to perform a single* **find** *operation when we perform union by rank and path compression is $O(\log n)$ time.*

Proof: (A) rank of leader v of reversed tree T, bounds depth of T.
(B) By previous lemma: max rank $\leq \lg n$.
(C) Depth of tree is $O(\log n)$.
(D) Time to perform **find** bounded by depth of tree.

21.3.0.13 $\log^*$ in detail

(A) $\log^*(n)$: number of times to take lg of number to get number smaller than two.
(B) $\log^* 2 = 1$
(C) $\log^* 2^2 = 2$.
(D) $\log^* 2^{2^2} = 1 + \log^*(2^2) = 2 + \log^* 2 = 3$.
(E) $\log^* 2^{2^{2^2}} = \log^*(65536) = 4$.
(F) $\log^* 2^{2^{2^{2^2}}} = log^* 2^{65536} = 5$.
(G) $\log^*$ is a monotone increasing function.
(H) $\beta = 2^{2^{2^{2^2}}} = 2^{65536}$: huge number
For practical purposes, $\log^*$ returns value ≤ 5.

21.3.0.14 Can do much better!

Theorem 21.3.8. *For a sequence of m operations over n elements, the overall running time of the* **UnionFind** *data-structure is $O((n + m)\log^* n)$.*

(A) Intuitively: **UnionFind** data-structure takes constant time per operation...
(unless n is larger than β which is unlikely).
(B) Not quite correct if n sufficiently large...

21.3.0.15 The tower function...

Definition 21.3.9. $\text{Tower}(b) = 2^{\text{Tower}(b-1)}$ and $\text{Tower}(0) = 1$.

$\text{Tower}(i)$: a tower of $2^{2^{2^{\cdot^{\cdot^{2}}}}}$ of height i.
Observe that $\log^*(\text{Tower}(i)) = i$.

Definition 21.3.10. For $i \geq 0$, let $\text{Block}(i) = [\text{Tower}(i-1) + 1, \text{Tower}(i)]$; that is

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

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

21.3.0.16 Running time of find...

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

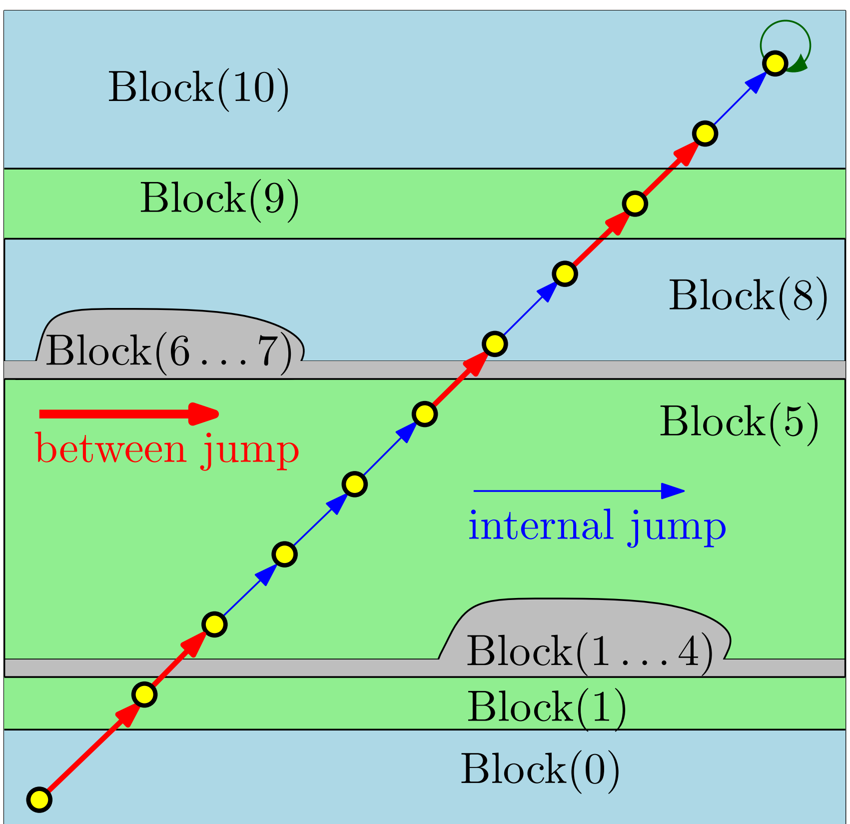
Definition 21.3.11. A node x is ***in the ith block*** if $\mathrm{rank}(x) \in \mathrm{Block}(i)$.

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

21.3.0.17 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, say 2).
(C) **find** (x): π path used.
(D) partition π into each by rank.
(E) Price of **find** length π.
(F) node x: $\nu = \mathrm{index_B}(x)$ index block containing $\mathrm{rank}(x)$.
(G) $\mathrm{rank}(x) \in \mathrm{Block}\big(\mathrm{index_B}(x)\big)$.
(H) $\mathrm{index_B}(x)$: ***block of x***

21.3.0.18 The path of find operation, and its pointers



21.3.0.19 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 $\leq O(\log^* n)$.

21.3.0.20 Jumping pointers

Definition 21.3.12. 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).

21.3.0.21 Not too many jumps between blocks

Lemma 21.3.13. *During a single* **find**(x) *operation, the number of jumps between blocks along the search path is* $O(\log^* n)$.

Proof: (A) $\pi = x_1, \ldots, x_m$: path followed by **find**.
(B) $x_i = \overline{\mathrm{p}}(x_{i-})$, for all i.
(C) $0 \leq \mathrm{index}_{\mathrm{B}}(x_1) \leq \mathrm{index}_{\mathrm{B}}(x_2) \leq \ldots \leq \mathrm{index}_{\mathrm{B}}(x_m)$.
(D) $\mathrm{index}_{\mathrm{B}}(x_m) = O(\log^* n)$.
(E) Number of elements in π such that $\mathrm{index}_{\mathrm{B}}(x) \neq \mathrm{index}_{\mathrm{B}}(\overline{\mathrm{p}}(x))$...
(F) ... at most $O(\log^* n)$.

21.3.0.22 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) $\Longrightarrow$ parent pointer x jumped forward...
(H) ...and new parent has higher rank.
(I) Charge internal block jumps to this "progress".

21.3.1 Changing parents...

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

Lemma 21.3.14. *At most* $|\mathrm{Block}(i)| \leq \mathrm{Tower}(i)$ **find** *operations can pass through an element* x*, which is in the* i*th block (i.e.,* $\mathrm{index}_{\mathrm{B}}(x) = i$*) before* $\overline{\mathrm{p}}(x)$ *is* no longer *in the* i*th block. That is* $\mathrm{index}_{\mathrm{B}}(\overline{\mathrm{p}}(x)) > i$.

Proof: (A) parent of x incr rank every-time internal jump goes through x.
(B) At most $|\mathrm{Block}(i)|$ different values in the ith block.
(C) $\mathrm{Block}(i) = [\mathrm{Tower}(i-1)+1, \mathrm{Tower}(i)]$
(D) Claim follows, as: $|\mathrm{Block}(i)| \leq \mathrm{Tower}(i)$.

21.3.1.2 Few elements are in the bigger blocks

Lemma 21.3.15. *At most $n/\text{Tower}(i)$ nodes are assigned ranks in the ith block throughout the algorithm execution.*

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

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

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

Lemma 21.3.16. *The number of internal jumps performed, inside the ith block, during the lifetime of the union-find data-structure is $O(n)$.*

Proof: (A) x in ith block, have at most $|\text{Block}(i)|$ internal jumps...
(B) ... all jumps through x are between blocks, by lemma...
(C) $\leq n/\text{Tower}(i)$ elements assigned ranks in the ith block, throughout algorithm execution.
(D) total number of internal jumps is $|\text{Block}(i)| \cdot \frac{n}{\text{Tower}(i)} \leq \text{Tower}(i) \cdot \frac{n}{\text{Tower}(i)} = n$.

21.3.1.4 Total number of internal jumps

Lemma 21.3.17. *The number of internal jumps performed by the Union-Find data-structure overall is $O(n \log^* n)$.*

Proof: (A) Every internal jump associated with block it is in.
(B) Every block contributes $O(n)$ internal jumps throughout time.
(By previous lemma.)
(C) There are $O(\log^* n)$ blocks.
(D) There are at most $O(n \log^* n)$ internal jumps.

21.3.1.5 Result...

Lemma 21.3.18. *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 21.3.19. *If we perform a sequence of m operations over n elements, the overall running time of the Union-Find data-structure is $O((n+m)\log^* n)$.*

21.3.1.6 More on strange functions...

Idea: Define a sequence of functions $f_i(x) = f_{i-1}^{(x)}(0)$

Function	Inverse function
$f_1(x) = x + 2$	$g_1(y) = y - 2$
$f_2(x) = 2x$	$g_2(y) = y/2$
$f_3(x) = 2^x$	$g_3(y) = \lg y$
$f_4(x) = \text{Tower}(x)$	$g_4(x) = \log^* x$
$f_5(x) = ...$	

$f_2(x) = f_1(f_2(x-1)) = 2x$ $f_3(x) = f_2(f_3(x-1)) = 2^x f_4(x) = f_3(f_4(x-1)) = \text{Tower}x$

$f_i(x) = f_{i-1}^{(x)}(1)$

$g_i(x) = \#$ of times one has to apply $g_{i-1}(\cdot)$ to x before we get number smaller than 2.

$A(n) = f_n(n)$: ***Ackerman function***.

Inverse Ackerman function:

$\alpha(n) = A^{-1}(n) = \min i$ s.t. $g_i(n) \le i$.

21.3.1.7 Union-Find: Tarjan result

Theorem 21.3.20 (?). *If we perform a sequence of m operations over n elements, the overall running time of the Union-Find data-structure is $O((n+m)\alpha(n))$.*

(The above is not quite correct, but close enough.)