Data Structures – LECTURE 17

Union-Find on disjoint sets

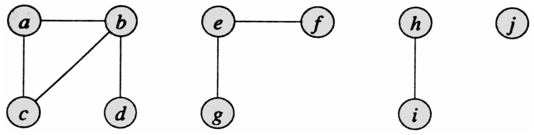
- Motivation
- Linked list representation
- Tree representation
- Union by rank and path compression heuristics

Chapter 21 in the textbook (pp 498—509).

Motivation

- Perform repeated union and find operations on disjoint data sets.
- Examples:
  - Kruskal's MST algorithm
  - Strongly connected components
- Goal: define an ADT that supports Union-Find queries on disjoint data sets efficiently.
- Target: average $O(n)$ time, where n is the total number of elements in all sets.

Example: connected components



Initial set **S** = {{*a*}, {*b*}, {*c*}, {*d*}, {*e*}, {*f*}, {*g*}, {*h*}, {*i*}, {*j*}}
(*b*,*d*) **S** = {{*a*}, {*b*,*d*}, {*c*}, {*e*}, {*f*}, {*g*}, {*h*}, {*i*}, {*j*}}
(*e*,*g*) **S** = {{*a*}, {*b*,*d*}, {*c*}, {*e*,*g*}, {*f*}, {*h*}, {*i*}, {*j*}}
(*a*,*c*) **S** = {{*a*,*c*}, {*b*,*d*}, {*e*,*g*}, {*f*}, {*h*}, {*i*}, {*j*}}
(*h*,*i*) **S** = {{*a*,*c*}, {*b*,*d*}, {*e*,*g*}, {*f*}, {*h*,*i*}, {*j*}}
(*a*,*b*) **S** = {{*a*,*c*,*b*,*d*}, {*e*,*g*}, {*f*}, {*h*,*i*}, {*j*}}
(*e*,*f*) **S** = {{*a*,*c*,*b*,*d*}, {*e*,*f*,*g*}, {*h*,*i*}, {*j*}}
(*b*,*c*) **S** = {{*a*,*c*,*b*,*d*}, {*e*,*f*,*g*}, {*h*,i}, {*j*}}

Union-Find Abstract Data Type

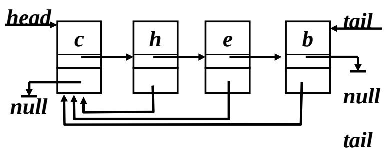
- Let $S = \{S_1, S_2, \ldots, S_k\}$ be a dynamic collection of disjoint sets.
- Each set S_i is identified by a representative member.
- Operations:
  - Make-Set(x): create a new set S_x, whose only member is x (assuming x is not already in one of the sets).
  - Union(x, y): replace two disjoint sets S_x and S_y represented by x and y by their union.
  - Find-Set(x): find and return the representative of the set S_x that contains x.

Disjoint sets: linked list representation

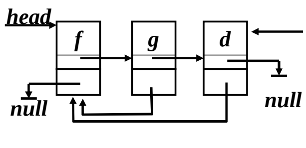
- Each set is a linked list, and the representative is the head of the list. Elements point to the successor and to the head of the list.
- Make-Set: create a new list: $O(1)$.
- Find-Set: search for an element down the list: $O(n)$.
- Union: link the tail of L_1 to the head of L_2, and make each element of L_2 point to the head of L_1: $O(n)$.
- A sequence of n Make-Set operations + n–1 Union operations will take $n+\Sigma i = \Theta(n^2)$ operations.
- The amortized time for one operation is thus $\Theta(n)$.

Example: linked list representation

$S_1=\{c,h,e,b\}$

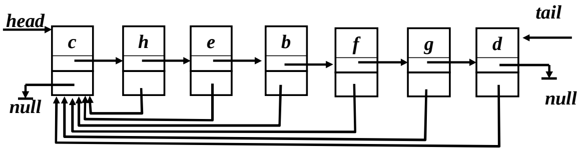


$S_2=\{f,g,d\}$



Example: linked list representation

S_1 U S_2= ={c,h,e,b}U{f,g,d}



Weighted-union heuristic

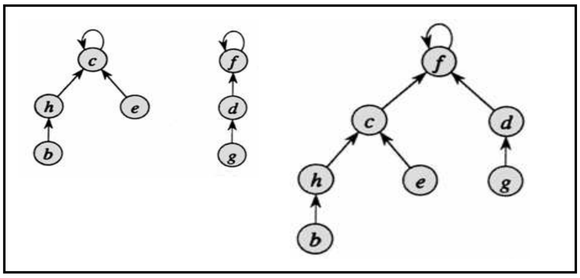
- When doing the union of two lists, append the shorter one to the longer one.
- A single operation will still take $O(n)$ time.
- However, for a sequence of $m>n$ Make-Set, Union, and Find-Set operations, of which n are Make-Set, the total time is $O(m + n\lg n)$ instead of $O(mn)$!
- Proof outline: an object x has its pointer updated at most *ceiling*(lg n) times, since it cannot always be on a long list.

Disjoint sets: tree representation

- Each set is a tree, and the representative is the root.
- Each element points to its parent in the tree. The root points to itself.
- Make-Set: takes $O(1)$.
- Find-Set: takes $O(h)$ where h is the height of the tree.
- Union is performed by finding the two roots, and choosing one of the roots, to point to the other. This takes $O(h)$.
- The complexity therefore depends on how the trees are maintained! In the worst case, no improvement.

Example: disjoint sets tree representation

$S=\{S_1, S_2\}$, $S_1=\{c,h,e,b\}$, $S_2=\{f,g,d\}$



Union by rank

- We want to make the trees as shallow as possible → trees must be balanced.
- When taking the union of two trees, make the root of the shorter tree become the child of the root of the longer tree.
- Keep track of the estimated size of each sub-tree: → keep the *rank* of each node.
- Every time Union is performed, update the rank of the root.

Union by rank: pseudocode

Make-Set(x)
 pointer[x] ← x
 rank[x] ← 0

Union(x,y)
 Link(Find-Set(x), Find-Set(y))

Find-Set(x)
 if $x \neq$ *pointer*[x] **then** *pointer*[x] ← Find-Set(*pointer*[x])
return *pointer*[x]

Link(x,y)
 if *rank*[x] > *rank*[y] **then** *pointer*[x] ← y
 else *pointer*[y] ← x
  if *rank*[x] = *rank*[y] **then** *rank*[x] ← *rank*[y]+1

Complexity of Find-Set (1)

- Claim: the maximum height of a tree when using height balancing is $O(\lg n)$.
- Proof: By induction on the number of Union operations used to create the tree.

  When the tree height is h, the number of nodes is at least 2^h.

  Basis: Clearly true for the first union operation, where h=1 and the resulting tree has two nodes.

  Induction step: True for tree of height h.

Complexity of Find-Set (2)

For the next union operation, there are two cases:

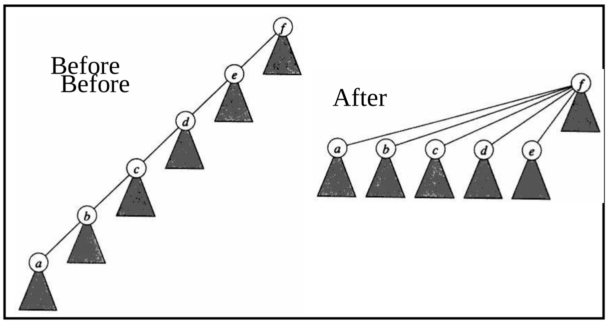
- The tree height does not grow: one tree was shorter than the other, in which case it is clearly true, because h didn't grow and the number of nodes did.
- The height does grow, because both tree heights were the same. By the induction hypothesis, each sub-tree has at least 2^h nodes, so the new tree has at least $2.2^h = 2^{h+1}$ nodes. Thus, the height of the tree grew by 1 to h+1, which proves the induction step.

Overall complexity: $O(m \lg n)$.

Path compression

- Speed up Union-Find operations by shortening the sub-tree paths to the root.
- During a Find-Set operation, make each node on the find path point directly to the root.
- Complexity: The worst-case time complexity of n Make-Set operations and m Find-Set operations is $\Theta(n + m(1+ \log_{2+n/m} m))$. (Analysis omitted).

Example: path compression



Union by rank and path compression

- When both heuristics are used, the worst-case time complexity is $O(m\ \alpha(n))$ where $\alpha(n)$ is the *inverse Ackerman function*.
- The inverse Ackerman function grows so slowly that for all practical purposes $\alpha(n) \leq 4$ for very large n.

Summary

- Union-Find has many applications.
- For a sequence of $m>n$ Make-Set, Union, and Find-Set operations, of which n are Make-Set:
  - List implementation: $O(m + n\lg n)$ with weighted union heuristic.
  - Tree implementation: union by rank + path compression yields $O(m\ \alpha(n))$ complexity.