Parma University, Math. Department

Algorithms and Data Structures
Disjoint sets

Marco Pellegrini

Spring 2002, Lecture 9

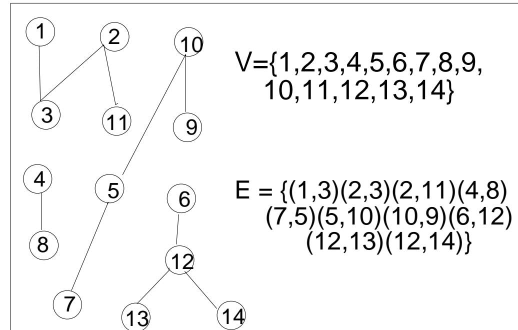
Summary of lecture 9

- Summary of this lecture: (CLR 22.1-3)
  Data Structures for Disjoint sets:
    Union operation
    Find operation

Disjoint set operations.

- We have a collection of elements, which we identify with numbers 1,2,3, etc.
- A Collection of sets S_1, S_2,..S_k.
- The property we assume is that an element is at most in one set. That is the sets are disjoint.
  We shall see the following operations:
- Make-set(x): take a element x not in any set and generates the set containing x, that is: {x}.
- Union(x,y): makes the union of the set containing x and the set containing y.
- Find-set(x): returns a pointer to the representative of the set containing x. Each set has a unique representative.

Example: connected components in graphs



V={1,2,3,4,5,6,7,8,9,
10,11,12,13,14}

E = {(1,3)(2,3)(2,11)(4,8)
(7,5)(5,10)(10,9)(6,12)
(12,13)(12,14)}

Connected-components.

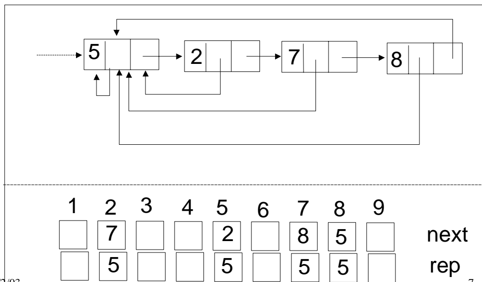
- We want to find the components of the graph.
  A component is a group of nodes that we can reach by following edges starting from a node.
-
```
Connected-component(V,E)
  FOR each vertex v in V DO
    Make-set(v)
  END FOR
  FOR each edge (u,v) in E DO
    IF Find-set(u) ≠ Find-set(v)
      THEN Union(u,v)
    END IF
  END FOR
```
- Same-component(u,v)
  RETURN [Find-set(u) = Find-set(v)]

Exercise

- Run the procedure Connected-component on the graph (that is V and E) given in the example.
- How many sets (that is components) do you have at the end.
- Hint: naturally I have not told you yet HOW make-set, union and find-set work but you know WHAT they are supposed to do.

Representing disjoint sets by lists.

- We can use circular singly linked lists and use the first element of the list as representative.



Operations

- Make-set and find-set can be done in O(1) on a circular list with back-pointer to the head.
- Union(x,y) is trickier. Appending two circular lists can be done in O(1), but to update the pointer to the representative we must scan one of the two lists.
- If n is the number of elements, m the number of operations, in the worst case we might use time $O(m^2)$ to perform the m operations.
- This happens when we grow a list by one element at a time and we always scan the longest list.

Size heuristic.

- Improvement: when we make the union scan the shortest of the two list.
- To know which is the shortest we carry a field SIZE linked to the representative which we update when we perform the union.
- This simple rule decreases the time complexity to O(m + n log n) for n elements, m operations.
- Proof. By a charging argument. Fix one element x, initially it is in a set of size 1. Then at every union the size of the set containing it at least doubles, that is 2,4,8...,n. This can happen only log n times. At every union we update repr(x). In total n log n updates. All other costs are O(m).

Homework

HW2.3 (CLR 22.2-1)
Write pseudocode for make-set,find-set, union using singly linked lists and the weighted union rule.
Each object x, has:

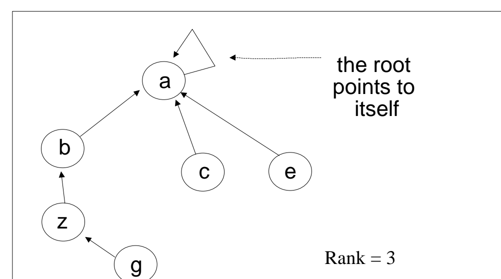
a) field repr[x] pointing to the representative of the set containing x,
b) field last[x] pointing to the last objectin the list containing x,
c) field size[x] giving the size of the listcontaining x.

Size[x] and Last[x] are correct only when x is a representative.

Forest-of-trees implementation.

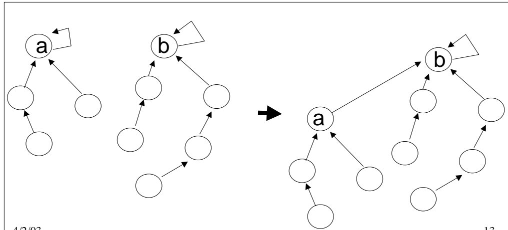
- Can we do better than the linked lists DS?
- Yes, buy it is hard to prove it!
- Idea: represent a set by a tree, we need only a pointer from a node to the parent.
- We could keep track of sizes, but for proof purposes we keep track of the *rank* of the trees.
- The rank is the length of the longest path.

Examples



Union by rank

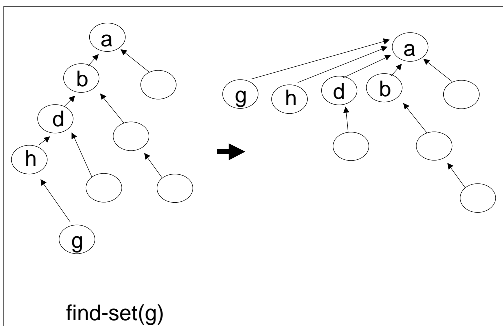
- The rank of the tree is the length of the longest path from root to leaf (without path compression)
- Union by rank is: attach the lowest rank tree under the root of the highest rank tree.



Union by rank

- If we merge two trees A,B of different rank the rank of the resulting tree is max{rank(A), rank(B)}.
- If they have the same rank, than the rank increases by 1.
-
```
Make-set(x)
  parent[x] := x
  rank[x] := 0
```
-
```
Union(x,y)
  a := find-set(x), b:= findset(y)
  IF rank[a] > rank[b]
    THEN parent[b] := a
    ELSE parent[a] := b
        IF rank[a] = rank[b]
          THEN rank[b] := rank[b]+1
        END IF
  END IF
```


Find-set with path compression.



find-set(g)

Find set + path compression.

- To implement find-set(x) we follow pointers from x to the root and then we go back making every node we visit on the path a child of the root.
-
```
Find-set(x)
  IF x ≠ p[x]
    THEN parent[x] := Find-set(parent[x])
  ENDIF
  RETURN parent[x]
```
- Exercise: simulate this code on the example of the previous slide find-set(g). Show for each call the node in input and the one in output and the change to the parent field.

Programming assignment

- Implement in C the Disjoins-sets DS using lists with weighted union rule. [DS1]
- Implement in C the Disjoint-set DS using trees with the rank rule and the path-compression. [DS2]
- Generate random Graph G with n nodes and n edges.
- Implement the connected components algorithm twice using DS1 the first time and DS2 the second time.[A1,A2]
- Run the A1 and A2 on the graph G (check that the output (that is the components) is the same.
- Plot the running times of A1 and A2 for n=100,200,500 1000, maximum number before the program crashes.

Performance of Union-Find DS

- n elements, m operations (union and find), f find operations.
- UF= Union find, PC= path compression, WH= weight heuristic, RH= rank heuristic.
- UF+WH is $O(m + n \log n)$.
- UF+RH is $O(m \log n)$
- UF+PC is $O(n+f \log n)$ if $f<n$, $O(f \log_{(1+f/n)} n)$ if $f>n$.
- UF+PC+RH is $O(m \log^* n)$

Where $\log^* n = \min\{i | \log^{(i)} n \leq 1\}$. And $\log^{(0)} n = \log n$, $\log^{(i)} n = \log(\log^{(i-1)} n)$.

$$F(i) = 2^{2^{2^{2^{2^{\cdot^{\cdot^{2}}}}}}} \Big\}\ \text{i times}$$

Log*(F(x))=x.

F(1) = 1

F(2) = 4

F(3) = 16

F(4) = 65,536

$F(5) = 2^{65536} > 10^{80}$

Analysis of UF+PC+RH

The function G(n)=log* n is the pseudo-inverse of F(n).

For all practical inputs log* n < 5.

The main idea is that to estimate the number of nodes *visited* by find operations, we can instead count the number of find operations *visiting* any node.

Lemma 1. For a node x that is not root , rank[x] < rank[parent[x]].

Proof: by construction.

Lemma 2. For a node x, rank[x] is an increasing function of time (non-decreasing). Constant when x ceases to be a root.

Proof: by construction.

For a root x, size(x) is the number of nodes in the tree rooted at x.

Lemma 3. Size(x) $\geq 2^{\text{rank(x)}}$.

Proof: by induction. Initially each node is a root and each node has rank 0, so the lemma is true.

Inductive step: suppose rank(x)<rank(y). After the union we have size at the root size(x)+size(y) > size(y) $\geq 2^{\text{rank(y)}}$. The rank of the root after the union is rank(y).

Suppose rank(x)=rank(y). After the union we have sixe at the root size(x)+size(y) $\geq 2^{\text{rank(y)}}+2^{\text{rank(y)}} \geq 2^{\text{rank(y)+1}}$. The rank of the root after the union is rank(y)+1.

Lemma 4. For each integer r, there are *at most* $n/2^r$ nodes of rank r at any given time.

Proof. Since the rank is an increasing function over each path from leaf to the root, no two nodes of same rank r are one the ancestor of the other. So for x and y both of rank r the subtrees rooted at x and y are *disjoint*.

Each subtree has at leat 2^r nodes, so there are at most $n/2^r$ nodes of rank r.

Lemma 5. No vertex has rank greater than log n.

Charging scheme.

We split the possible ranks r=[0,…, $\lceil \log n \rceil$] into groups.

Rank r goes into group G(r).

Equivalently group j contains ranks [F(j-1)+1,..,F(j)].

Case 1. If find visits x and group(x) ≠ group(parent(x)), or x is the root, charge 1 to the find operation.

Case 2. If find visits x and group(x) = group(parent(x)) charge 1 to the node x. (is not the root).

Costs

Case 1. A single find operation incur in case (1) when crossing the boundaries between two groups, so at most G(log n) = G(n) -1 times. So n find operations are charged O(nG(n)).

Case 2. While x and parent(x) are in the same group g, since the rank is increasing, we can charge x at most F(g)-F(g-1) times < F(g).

How many nodes do we have in group g?

$$\sum_{r=F(g-1)+1}^{F(g)} n/2^r \le (n/2^{F(g-1)+1}) \sum_{i=0} 2^{-i} \le (n/2^{F(g-1)}) = n/F(g)$$

So the total charge to group g is n. The number of groups is G(n). The total charge by case 2 is O(nG(n)).

Theorem 6. Union Find with Path Compression and Rank Heuristic, n elements and operations takes time O(n log* n).

Conclusions

- We have seen two DS for Disjoint-sets, sometimes called (Union-find ADS).
- The first based on lists, the second based on trees with union-by-rank and path-compression.
- The analysis tells us the second is asymptotically faster. Your programming assignment should test whether this happens also in experimental tests.
- Actually an even more complex analysis can prove that the right bound is O(m α(n,m)) where α(n,m) is a function that is even slower than log*(n).