CS302:Design and Analysis of Algorithms

Sets- Union and find operations on disjoint sets

Disjoint-set

- A ***disjoint-set data structure*** maintains a collection *S ={S1, S2,.... ,Sk}* of disjoint dynamic sets. We identify each set by a ***representative***, which is some member of the set.

Disjoint-set operations

- Let x be an object,

1. **MAKE-SET(x**) creates a new set whose only member (and thus representative) is x. Since the sets are disjoint, we require that x not already be in some other set.
2. **UNION(x, y)** unites the dynamic sets that contain x and y, say Sx and Sy, into a new set that is the union of these two sets.
   - We assume that the two sets are disjoint prior to the operation.
   - The representative of the resulting set is either any member of Sx U Sy, or chooses the representative of either Sx or Sy as the new representative. Remove sets Sx and Sy from the collection S.
3. **FIND-SET(x)** returns a pointer to the representative of the (unique) set containing x.

Analysis

- The running time of disjoint-set data structures is analyzed in terms of two parameters:
  - n, the number of MAKE-SET operations, and
  - m, the total number of MAKE-SET, UNION, and FIND-SET operations.
- Since the sets are disjoint, each UNION operation reduces the number of sets by one.
- After n -1 UNION operations, therefore, only one set remains. The number of UNION operations is thus at most n -1.
- since the MAKE-SET operations are included in the total number of operations m, we have m ≥ n. We assume that the *n* MAKE-SET operations are the first n operations performed.

Application: To compute the connected components of a graph

- The procedure CONNECTED-COMPONENTS that follows uses the disjoint-set operations to compute the connected components of a graph. Once CONNECTED-COMPONENTS has been run as a preprocessing step, the procedure SAME-COMPONENT answers queries about whether two vertices are in the same connected component. (The set of vertices of a graph *G* is denoted by *V*[*G*], and the set of edges is denoted by *E*[*G*].)

CONNECTED-COMPONENTS(*G*)

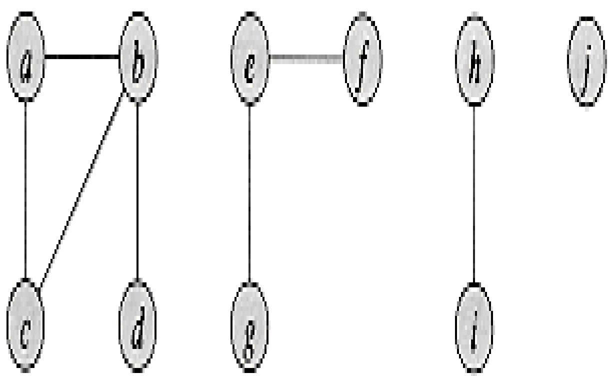
```
for each vertex v ∈ V[G]
    MAKE-SET(v)
for each edge (u,v) ∈ E[G]
    if FIND-SET(u) ≠ FIND-SET(v)
        UNION(u,v)
```

SAME-COMPONENT(*u*,*v*)

```
if FIND-SET(u) == FIND SET(v)
    return TRUE
else return FALSE
```

The procedure CONNECTED-COMPONENTS

- initially places each vertex *v* in its own set.
- Then, for each edge (*u, v*), it unites the sets containing *u* and *v*.
- After all the edges are processed, two vertices are in the same connected component if and only if the corresponding objects are in the same set.
- Thus, CONNECTED-COMPONENTS computes sets in such a way that the procedure SAME-COMPONENT can determine whether two vertices are in the same connected component



(a)

Edge processed	Collection of disjoint sets									
initial sets	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(b,d)	{a}	{b,d}	{c}		{e}	{f}	{g}	{h}	{i}	{j}
(e,g)	{a}	{b,d}	{c}		{e,g}	{f}		{h}	{i}	{j}
(a,c)	{a,c}	{b,d}			{e,g}	{f}		{h}	{i}	{j}
(h,i)	{a,c}	{b,d}			{e,g}	{f}		{h,i}		{j}
(a,b)	{a,b,c,d}				{e,g}	{f}		{h,i}		{j}
(e,f)	{a,b,c,d}				{e,f,g}			{h,i}		{j}
(b,c)	{a,b,c,d}				{e,f,g}			{h,i}		{j}

(b)

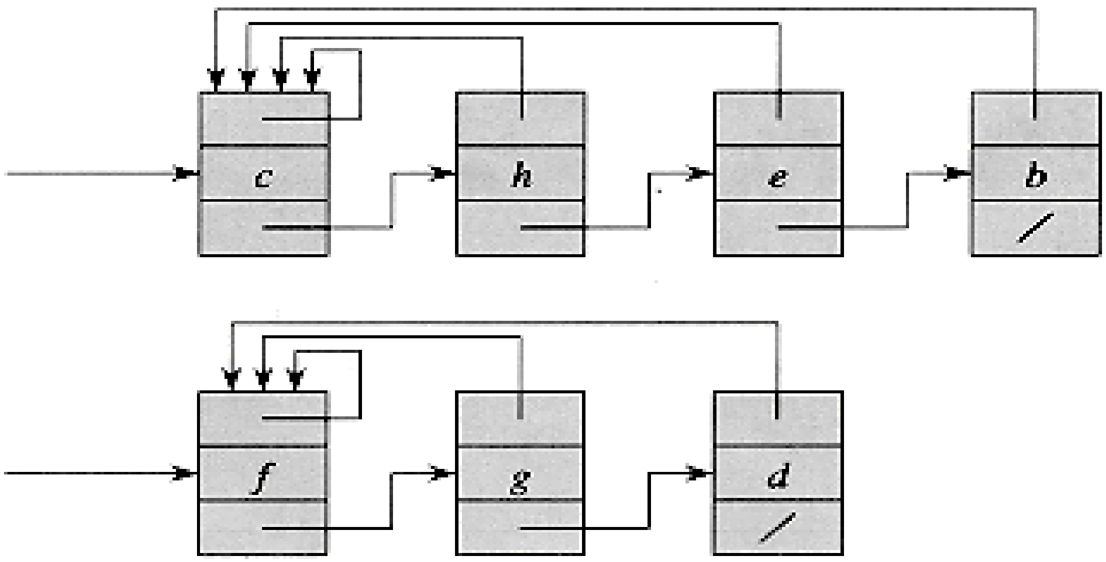
Figure (a) A graph with four connected components: {a, b, c, d}, {e, f, g}, {h, i}, and {j}. (b) The collection of disjoint sets after each edge is processed.

Linked-list representation of disjoint sets

- A simple way to implement a disjoint-set data structure is to represent each set **by a linked list**.
- The first object in each linked list serves as its set's representative.
- Each object in the linked list contains a set member, a pointer to the object containing the next set member, and a pointer back to the representative.
- Within each linked list, the objects may appear in any order.
- In linked-list representation,
  - MAKE-SET(x)- create a new linked list whose only object is x requiring $O(1)$ time.
  - For FIND-SET(x), we just return the pointer from x back to the representative requiring $O(1)$ time.

A simple implementation of union

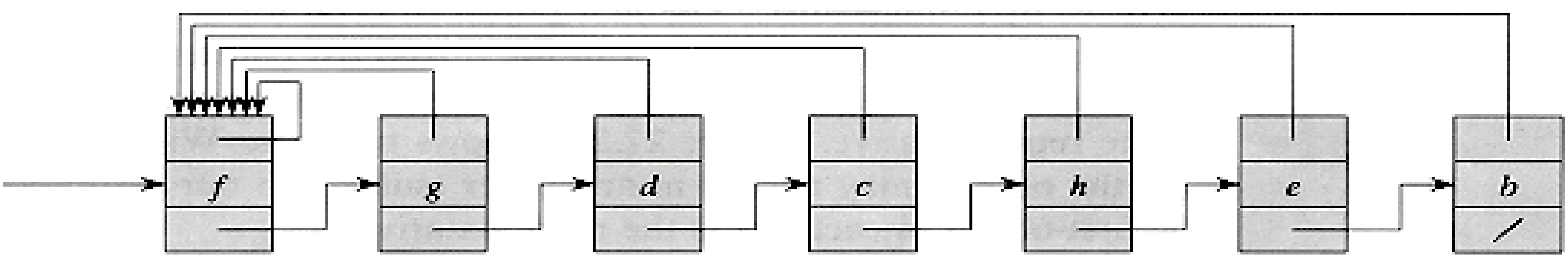
- we perform **UNION(*x*, *y*)** by appending *x*'s list onto the end of *y*'s list.
- The representative of the new set is the element that was originally the representative of the set containing *y*.
- Unfortunately, we must update the pointer to the representative for each object originally on *x*'s list, which takes time linear in the length of *x*'s list.
- A sequence of *m* operations that requires $\Theta(m^2)$ time.



(a)

Figure a) Linked-list representations of two sets. One contains objects b, c, e, and h, with c as the representative,

and the other contains objects d, f, and g, with f as the representative. Each object on the list contains a set member, a pointer to the next object on the list, and a pointer back to the first object on the list, which is the representative.



(b)

(b) The result of UNION(e, g). The representative of the resulting set is f.

Operation	Number of objects updated
MAKE-SET(x_1)	1
MAKE-SET(x_2)	1
$\vdots$	$\vdots$
MAKE-SET(x_n)	1
UNION(x_1, x_2)	1
UNION(x_2, x_3)	2
UNION(x_3, x_4)	3
$\vdots$	$\vdots$
UNION(x_{q-1}, x_q)	$q - 1$

Figure. A sequence of m operations that takes $O(m^2)$ time using the linked-list set representation and the simple implementation of UNION. For this example, $n = \lceil m/2 \rceil + 1$ and $q = m - n$.

A weighted-union heuristic

- The above implementation of the UNION procedure requires an average of (m) time per call because we may be appending a longer list onto a shorter list;
- we must update the pointer to the representative for each member of the longer list.
- Suppose instead that each representative also includes the length of the list and that we always append the smaller list onto the longer, with ties broken arbitrarily.
- With this simple ***weighted-union heuristic***, a single UNION operation can still take (m) time if both sets have (m) members.
- Using the linked-list representation of disjoint sets and the weighted-union heuristic, a sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, takes $O(m + n \text{ 1g } n)$ time.

Disjoint-set forests

- In a faster implementation of disjoint sets, we represent sets by rooted trees, with each node containing one member and each tree representing one set.
- In a ***disjoint-set forest***, each member points only to its parent.
- The root of each tree contains the representative and is its own parent.
- By using two heuristics--"union by rank" and "path compression"--we can achieve the asymptotically fastest disjoint-set data structure known

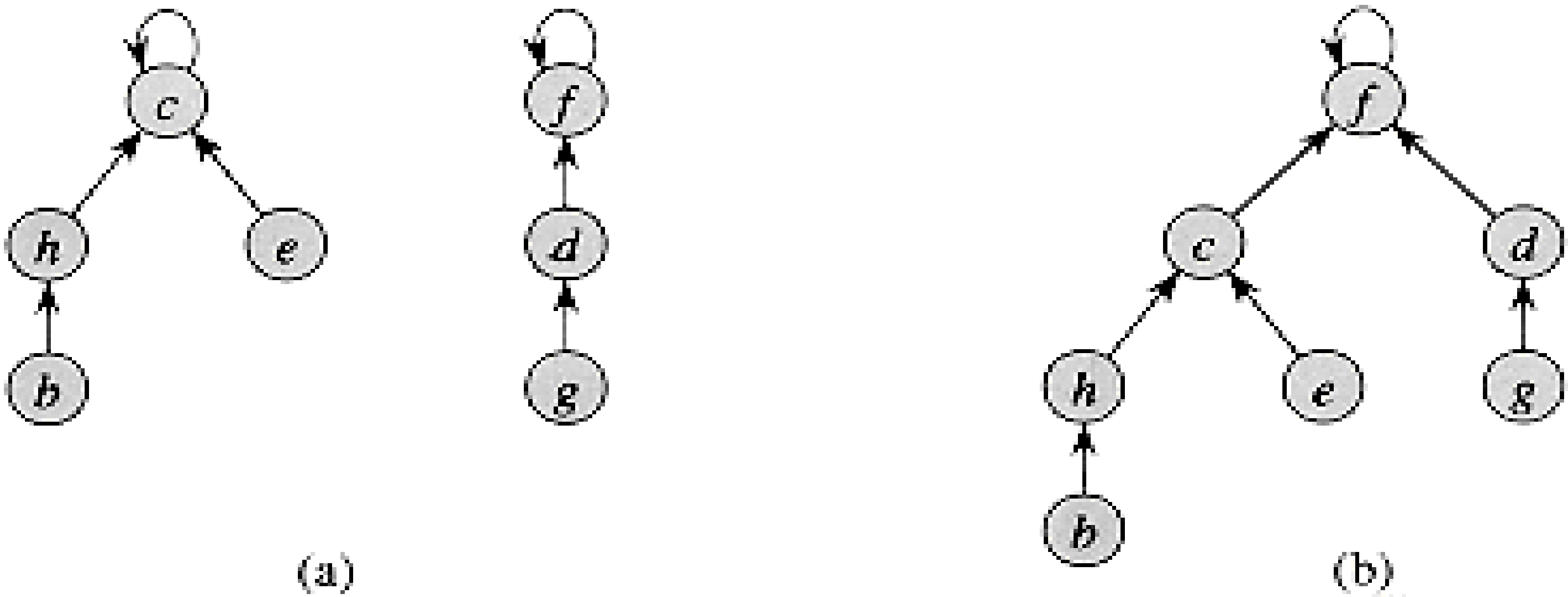


Figure shows A disjoint-set forest. (a) Two trees representing the two sets . The tree on the left represents the set {b, c, e, h}, with c as the representative, and the tree on the right represents the set {d, f, g}, with f as the representative. (b) The result of UNION(e, g).

- We perform the three disjoint-set operations as follows.
  1. A **MAKE-SET** operation simply creates a tree with just one node.
  2. We perform a **FIND-SET** operation by chasing parent pointers until we find the root of the tree. The nodes visited on this path toward the root constitute the ***find path***.
  3. A **UNION** operation, shown in Figure (b), causes the root of one tree to point to the root of the other.

Heuristics to improve the running time

- By using two heuristics, we can achieve a running time that is almost linear in the total number of operations *m*.

1)The first heuristic, ***union by rank***, is similar to the weighted-union heuristic we used with the linked-list representation.

  - The idea is to make the root of the tree with fewer nodes point to the root of the tree with more nodes.
  - Rather than explicitly keeping track of the size of the subtree rooted at each node, we shall use an approach that eases the analysis.
  - For each node, we maintain a ***rank*** that approximates the logarithm of the subtree size and is also an upper bound on the height of the node.
  - In union by rank, the root with smaller rank is made to point to the root with larger rank during a UNION operation.

- 2)The second heuristic, ***path compression***, is also quite simple and very effective.
  - we use it during FIND-SET operations to make each node on the find path point directly to the root.
  - Path compression does not change any ranks.

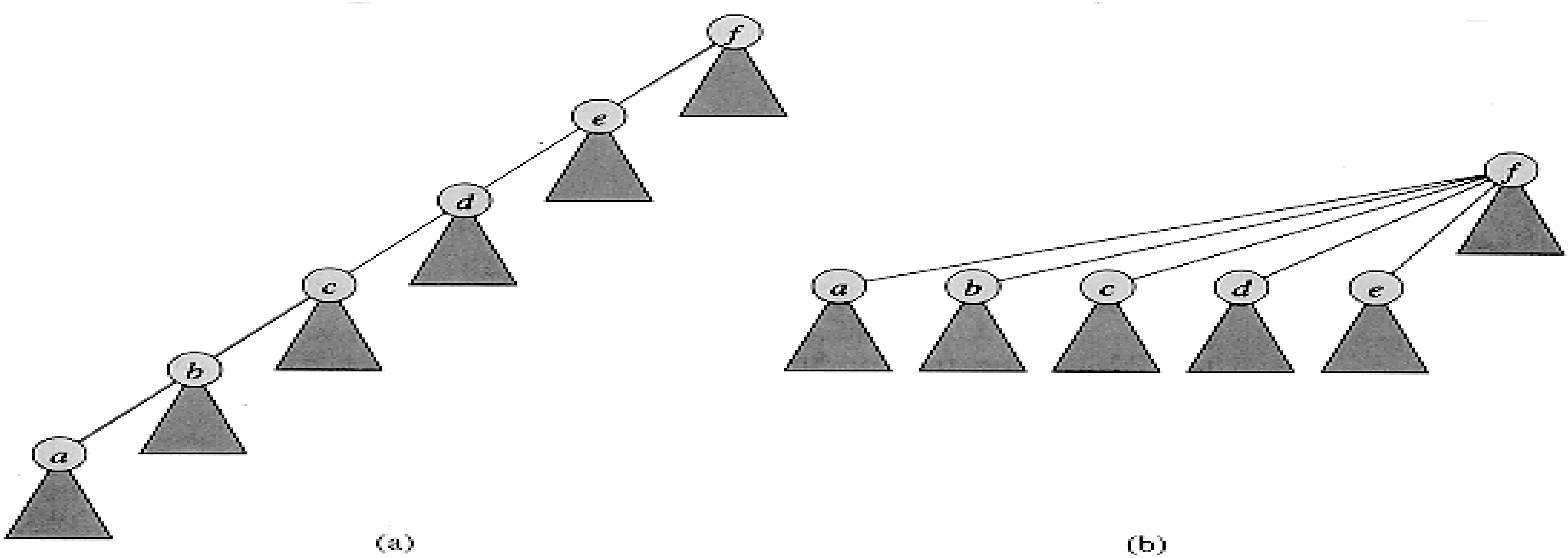


Figure shows the Path compression during the operation FIND-SET. Arrows and self-loops at roots are omitted.

(a) A tree representing a set prior to executing FIND-SET(a).

Triangles represent subtrees whose roots are the nodes shown. Each node has a pointer to its parent.

(b) The same set after executing FIND-SET(a). Each node on the find path now points directly to the root.

Pseudocode for disjoint-set forests

- To implement a disjoint-set forest with the union-by-rank heuristic, we must keep track of ranks.
- With each node x, we maintain the integer value $rank[x]$, which is an upper bound on the height of x (the number of edges in the longest path between x and a descendant leaf).
- When a singleton set is created by MAKE-SET, the initial rank of the single node in the corresponding tree is 0.
- Each FIND-SET operation leaves all ranks unchanged.
- When applying UNION to two trees, we make the root of higher rank the parent of the root of lower rank. In case of a tie, we arbitrarily choose one of the roots as the parent and increment its rank.

Pseudocode.

- *p*[*x*]- the parent of node *x*.
- The LINK procedure, a subroutine called by UNION, takes pointers to two roots as inputs.

```
MAKE-SET(x)
  p[x] ← x
  rank[x] ← 0

UNION(x,y)
  LINK(FIND-SET(x), FIND-SET(y))

LINK(x,y)
  if rank[x] > rank[y]
     then p[y] ← x
     else p[x] ← y
          if rank[x] = rank[y]
             then rank[y] ← rank[y] + 1
```

- The FIND-SET procedure with path compression is quite simple.

FIND-SET(x)

1 **if** $x \neq p[x]$

2 **then** $p[x] \leftarrow$ FIND-SET($p[x]$)

3 **return** $p[x]$

- The FIND-SET procedure is a ***two-pass method***: it makes one pass up the find path to find the root, and it makes a second pass back down the find path to update each node so that it points directly to the root.
- Each call of FIND-SET(x) returns $p[x]$ in line 3.
- If x is the root, then line 2 is not executed and $p[x] = x$ is returned. This is the case in which the recursion bottoms out.
- Otherwise, line 2 is executed, and the recursive call with parameter $p[x]$ returns a pointer to the root.
- Line 2 updates node x to point directly to the root, and this pointer is returned in line 3.

Effect of the heuristics on the running time

Running time of union by rank is O(m lg n) .

- if there are n MAKE-SET operations
  - (and hence at most n - 1 UNION operations) and
  - f FIND-SET operations,
  - **the path-compression heuristic** alone gives a worst-case running time of $\Theta(n+ f.(1+\log_{2+f/n} n))$.

Analysis of union by rank with path compression

- The worst-case running time of the combined union-by-rank and path-compression heuristic is $O(m\alpha(m, n))$ for m disjoint-set operations on n elements,
  - where $\alpha(m,n)$ is the *very* slowly growing function.
  - If $\alpha(m,n)<=4$, *then* the running time, $O(m \lg^* n)$.