CSC 317: Data Structures and Algorithm Analysis

Dilip Sarkar

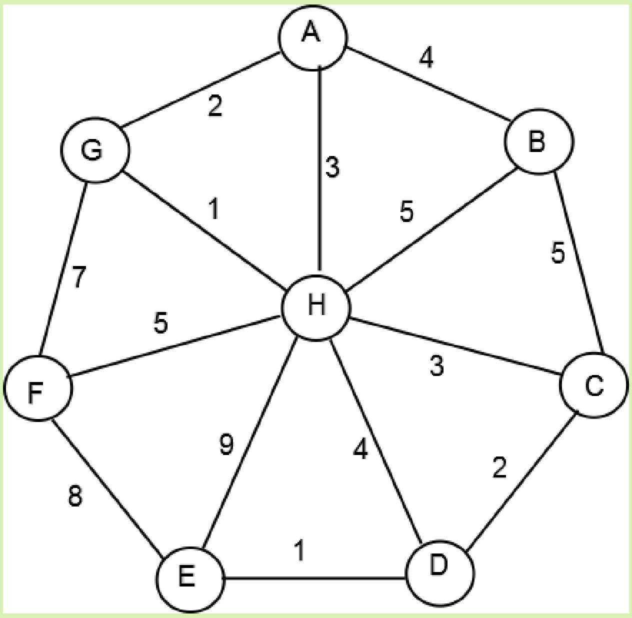
Department of Computer Science
University of Miami

Outline I

- Finding the Least-Expensive Optical Network Layout
- Data Structure for Disjoint Sets
- Linked-list Representation
- Rooted-tree Representation
- An Application of Disjoint-sets Data Structure
- Disjoint-set forests

Finding the Least-Expensive Optical Network Layout

- A Wheel-City



- The wheel-city mayor wants to connect all homes with an optical network.
- He wants to spend the least amount of money for the project.
- The city has been represented as a graph.
- The cost for laying optical cable between two houses are shown on the links
- How to find the least-expensive layout?
- All we need to do is find a **least-expensive tree** layout.
- Can you give an algorithm?

Disjoint Sets and Operations on Them

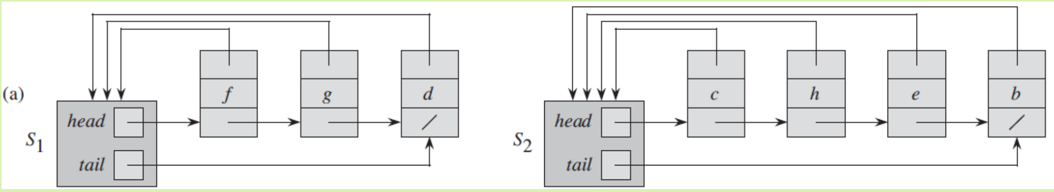
- Let us consider four sets:

  $S_1 = \{a, b, c, d\}, S_2 = \{e, f, g\}, S_3 = \{h, i\}, S_4 = \{j\}$
  These sets are disjoint, that is, for $1 \le i \ne j \le 4$, $S_i \bigcap S_j = \emptyset$
  Notation: Instead of using symbol to name a set, we will be using a new notation.
  A set is named with one of the elements of the set.
  - Thus, the set S_1 can be named as a, or b, or c or d.
    This naming may appear to be strange, but it will help us for representation and operations on the sets.
- Operations on the disjoint sets
  - Make-Set(x): create a new set whose only member is x.
    Since the sets are disjoint, we require that x not already be in some other set.
  - Find-Set(x): returns a pointer to the representation of the (unique) set containing x.
    Find operation is over the collection of sets.
    Suppose the set S_1 is named with its element b. Find-Set(d) will return b.
  - Union(x, y): unites the dynamic sets that contains x and y;
    We can then name new set as x or y.
    A **good** choice **can** improve performance.

Linked-list Representation of Disjoint Sets I

- Linked-list Representation

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



- MAKE-SET(x): create a new set whose only member is x.

  Since the sets are disjoint, we require that x not already be in some other set.

  MAKE-SET(x)

  $x.p = x$

  $x.rank = 0$ // Will be used latter

- Cost of MAKE-SET(x) operation:

  $O(1)$

Linked-list Representation of Disjoint Sets II

- Operations on the disjoint sets
  - FIND-SET(x): returns a pointer to the representation of the (unique) set containing x.

    Find operation is over the collection of sets.

    FIND-SET(x)

    if $x \neq x.p$

    $x.p =$ FIND-SET($x.p$)

    return $x.p$

  - Cost of FIND-SET(x) operation:

    $O(1)$

Linked-list Representation of Disjoint Sets II

- Operations on the disjoint sets
  - UNION(x, y): unites the dynamic sets that contains x and y;
    We can then name new set as x or y.
    A **good** choice **can** improve performance.

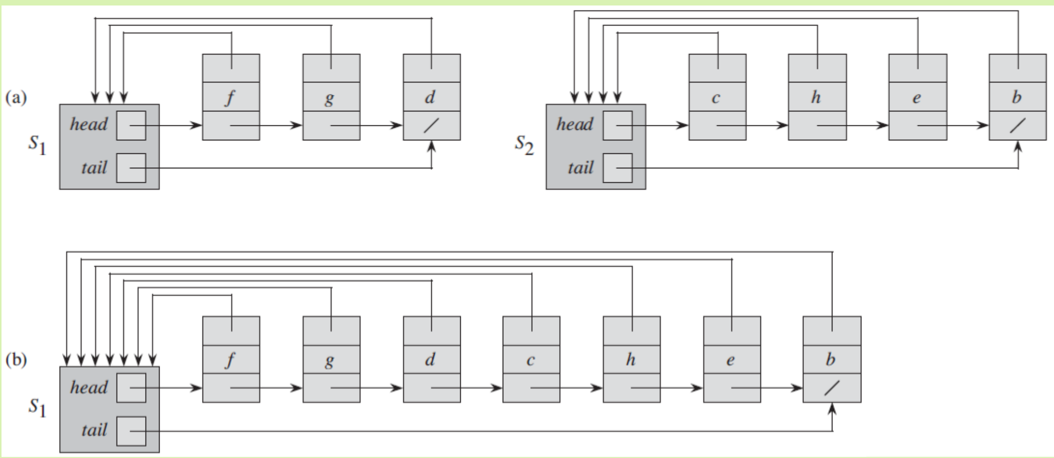
```
UNION(x, y)
    z1 = FIND-SET(x)
    z2 = FIND-SET(y)
    z1.p = z2
    z2.p = 0
```

  - Cost of FIND-SET(x) operation:
    $O(n)$

Linked-list Representation of Disjoint Sets III

- Cost of UNION(x, y) operation:

  $O(n)$; because we need to change pointer for all the elements of one of the two sets.



- Pointers for elements of set S_2, that is, c, h, e, and b have been changes to the head of set S_1.

Linked-list Representation of Disjoint Sets IV

- Complexity Analysis

Operation	Number of objects updated
$\textsc{Make-Set}(x_1)$	1
$\textsc{Make-Set}(x_2)$	1
...	
$\textsc{Make-Set}(x_n)$	1
$\textsc{Union}(x_2, x_1)$	1
$\textsc{Union}(x_3, x_2)$	2
...	
$\textsc{Union}(x_n, x_{n-1})$	$n-1$
	$\sum_{i=1}^{n} 1 + \sum_{i=1}^{n-1} i = O(n^2)$

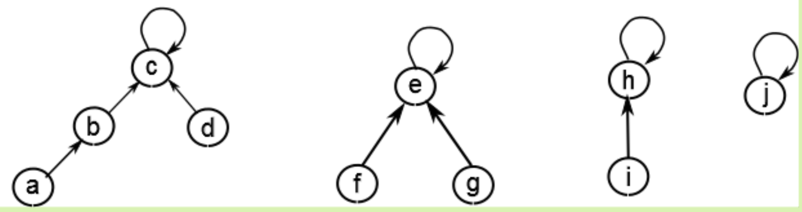
Rooted-tree Representation of Disjoint Sets I

- Rooted-tree Representation
- The sets are $S_1 = \{a, b, c, d\}$, $S_2 = \{e, f, g\}$, $S_3 = \{h, i\}$, $S_4 = \{j\}$

Parent	b	c	c	c	e	e	e	h	h	j
Count	0	0	4	0	3	0	0	2	0	1
Name	a	b	c	d	e	f	g	h	i	j

- **Recall notation**: S_1 is named as c — one of the elements in S_1. Similarly, S_2 is named as e, $S_3$is named as h, and S_4 is named as j

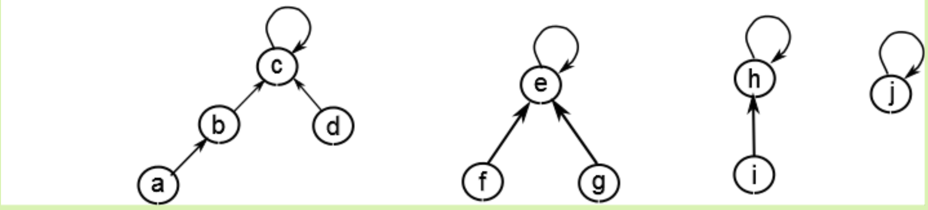
Parent	b	c	c	c	e	e	e	h	h	j
Count	0	0	4	0	3	0	0	2	0	1
Name	a	b	c	d	e	f	g	h	i	j



Rooted-tree Representation of Disjoint Sets II

- Rooted-tree Representation
- **Recall notation**: S_1 is named as c — one of the elements in S_1. Similarly, S_2 is named as e, $S_3$is named as h, and S_4 is named as j

Parent	b	c	c	c	e	e	e	h	h	j
Count	0	0	4	0	3	0	0	2	0	1
Name	a	b	c	d	e	f	g	h	i	j



- Note the direction of pointers from children to root

  Advantage: tree is not degree-limited.

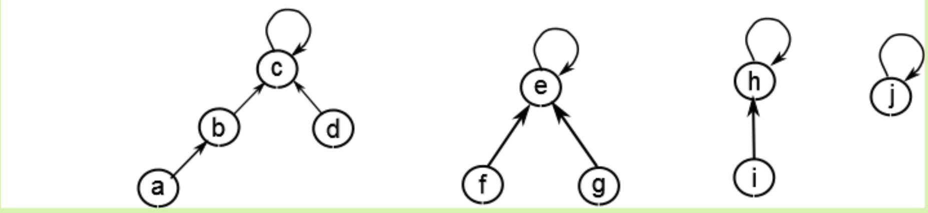
  Every node has a counter – value is size of the set for the root, '0' for others

  Value in the counter will be used for $\textsc{Union}(x, y)$ operation.

Rooted-tree Representation of Disjoint Sets III

- Rooted-tree Representation

Parent	b	c	c	c	e	e	e	h	h	j
Count	0	0	4	0	3	0	0	2	0	1
Name	a	b	c	d	e	f	g	h	i	j



- Cost of $\textsc{Make-Set}(x)$ operation:

  $O(1)$
- Cost of $\textsc{Find-Set}(x)$ operation:

  Distance of the element from the 'root'; for any good implementation, height is $\log_2 n$ or less.

Linked-list Representation of Disjoint Sets II

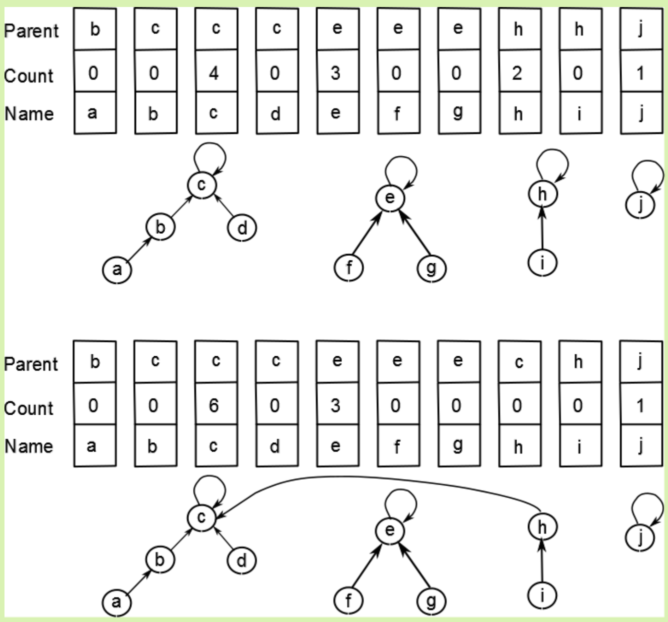
- Operations on the disjoint sets
  - UNION(x, y): unites the dynamic sets that contains x and y;
    We can then name new set as x or y.
    A **good** choice **can** improve performance.

WEIGHTED-UNION(x, y)
$z1$ = FIND-SET(x) // $z1$ is the root of the tree where x is
$z2$ = FIND-SET(y) // $z2$ is the root of the tree where y is
if $z1.count > z2.count$ // in book *rank* is similar to *count*
    $z1.p = z2$
    $z1.count = z1.count + z2.count$
    $z2.count = 0$;
else
    $z2.p = z1$
    $z1.count = z1.count + z2.count$
    $z1.count = 0$;

  - Cost of UNION(x, y) operation:
    $O(\log_2 n)$; because cost of find operation is $O(\log_2 n)$

Rooted-tree Representation of Disjoint Sets IV

- Rooted-tree Representation



- After UNION(d, h) operation, the new root is c.
  After UNION(d, h) operation, the **new** count at c is 6.
- If we use the root of the bigger set as the new root, the height is no more than $\log_2 n$.

Theorem

An Application of Disjoint Sets Data Structures I

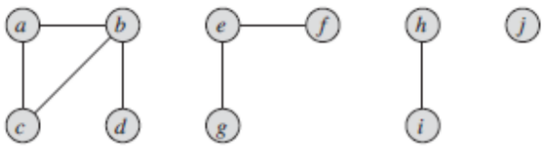
- Find the number of connected components in a graph.

```
CONNECTED-COMPONENTS(G)
for each vertex v ∈ G.V
    MAKE-SET(v)
for each edge (u, v) ∈ G.E
    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
```


An Application of Disjoint Sets Data Structures II

- Find the number of connected components in a graph. An example.



(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)

Disjoint-set forests