CS711008Z Algorithm Design and Analysis

Lecture 7. UNION-FIND data structure [1]

Dongbo Bu

Institute of Computing Technology
Chinese Academy of Sciences, Beijing, China

[1] The slides were made based on Chapter 5 of Algorithms by S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani, Data Structure by Ellis Horowitz, Hopcroft and Ullman 1973, and Tarjan 1975, et al.

Outline

- Introduction to UNION-FIND data structure
- Various implementations of UNION-FIND data structure:
  - Array: store "set name" for each element separately. Easy to FIND set of any element, but hard to UNION two sets.
  - Tree: each set is organized as a tree with root as "set name". It is easy to UNION two sets, but hard to FIND set for an element.
  - Link-by-rank: maintain a balanced-tree to limit tree depth to $O(\log n)$, making FIND operations efficient.
  - Link-by-rank and path compression: compress path when performing FIND, making subsequent FIND operations much quicker.

UNION-FIND data structure

UNION-FIND: motivation

- Motivation: Suppose we have a collection of **disjoint sets**. The objective of UNION-FIND is to keep track of elements by using the following operations:
  - MAKESET(x): to create a new set $\{x\}$.
  - FIND(x): to find the set that contains the element x;
  - UNION(x, y): to union the two sets that contain elements x and y, respectively.
- Analysis: total running time of a sequence of m FIND and n UNION.

UNION-FIND is very useful

- UNION-FIND has extensive applications, such as:
  - Network connectivity
  - Kruskal's MST algorithm
  - Least common ancestor
  - Games (Go)
  - ......

An example: Kruskal's MST algorithm

- Basic idea: during the execution, F is always an **acyclic forest**, and the **safe edge** added to F is always a least-weight edge connecting two distinct components.

Figure 1: Joseph Kruskal

Kruskal's algorithm [1956]

MST-KRUSKAL(G, W)

for all vertex $v \in V$ **do**

    MAKESET(v);

end for

sort the edges of E into nondecreasing order by weight W;

for each edge $(u, v) \in E$ in the order **do**

    if FINDSET$(u) \neq$ FINDSET(v) **then**

        $F = F \cup \{(u, v)\}$;

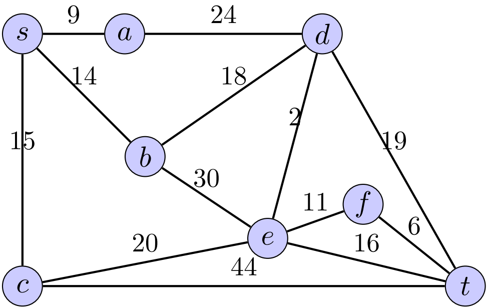
        UNION (u, v);

    end if

end for

- Here, UNION-FIND structure is used to detect whether a set of edges form a cycle.
- Specifically, each set represents a connected component; thus, an edge connecting two nodes in the same set is "unsafe", as adding this edge will form a cycle.

Kruskal's MST algorithm: an example

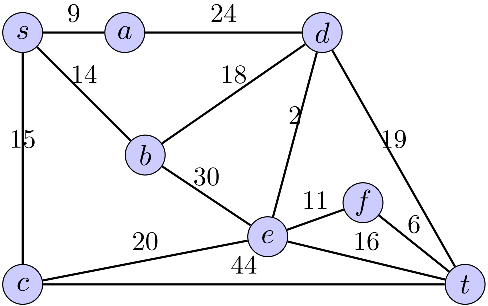


Kruskal's MST algorithm: an example

Step 1

Edge weight: $2, 6, 9, 11, 14, 15, 16, 18, 19, 20, 24, 30, 44$

Disjoint sets: $\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{s\}, \{t\}$

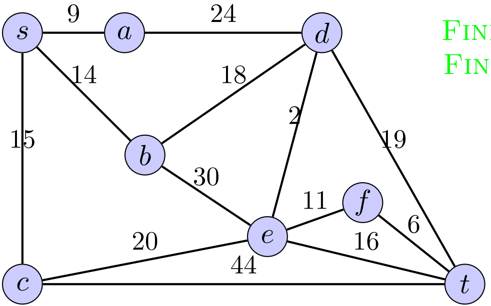


Kruskal's MST algorithm: an example

Step 1

Edge weight: $2, 6, 9, 11, 14, 15, 16, 18, 19, 20, 24, 30, 44$

Disjoint sets: $\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{s\}, \{t\}$



$\textsc{Find}(d)$ returns $\{d\}$

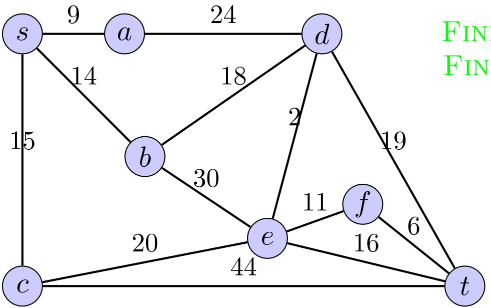
$\textsc{Find}(e)$ returns $\{e\}$

Kruskal's MST algorithm: an example

Step 1

Edge weight: $2, 6, 9, 11, 14, 15, 16, 18, 19, 20, 24, 30, 44$

Disjoint sets: $\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{s\}, \{t\}$



$\textsc{Find}(d)$ returns $\{d\}$

$\textsc{Find}(e)$ returns $\{e\}$

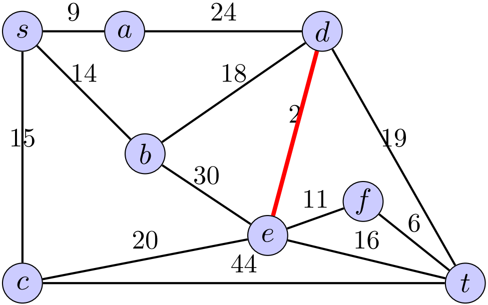
$\textsc{Union}(d, e)$

Kruskal's MST algorithm: an example

Step 1

Edge weight: 2, 6, 9, 11, 14, 15, 16, 18, 19, 20, 24, 30, 44

Disjoint sets: $\{a\}, \{b\}, \{c\}, \{d, e\}, \{f\}, \{s\}, \{t\}$

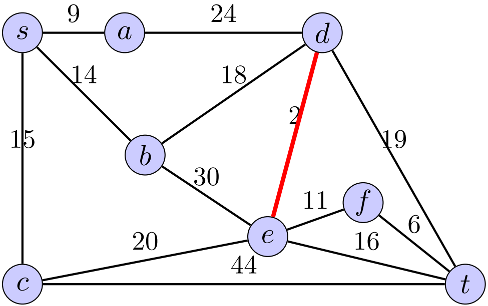


Kruskal's MST algorithm: an example

Step 2

Edge weight: $2, 6, 9, 11, 14, 15, 16, 18, 19, 20, 24, 30, 44$

Disjoint sets: $\{a\}, \{b\}, \{c\}, \{d, e\}, \{f\}, \{s\}, \{t\}$

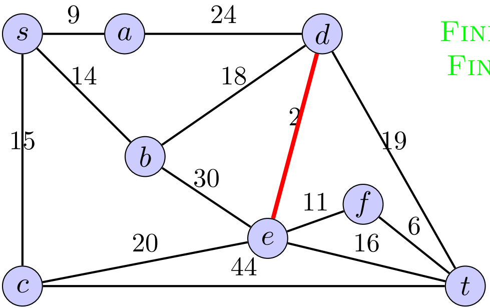


Kruskal's MST algorithm: an example

Step 2

Edge weight: $2, 6, 9, 11, 14, 15, 16, 18, 19, 20, 24, 30, 44$

Disjoint sets: $\{a\}, \{b\}, \{c\}, \{d, e\}, \{f\}, \{s\}, \{t\}$



$\textsc{Find}(f)$ returns $\{f\}$

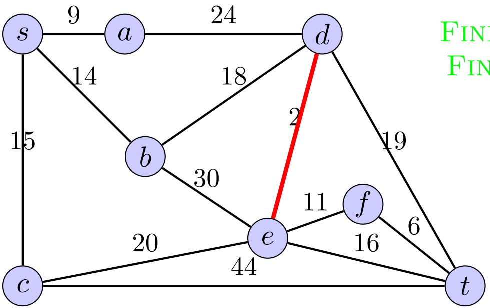
$\textsc{Find}(t)$ returns $\{t\}$

Kruskal's MST algorithm: an example

Step 2

Edge weight: $2, 6, 9, 11, 14, 15, 16, 18, 19, 20, 24, 30, 44$

Disjoint sets: $\{a\}, \{b\}, \{c\}, \{d, e\}, \{f\}, \{s\}, \{t\}$



$\textsc{Find}(f)$ returns $\{f\}$

$\textsc{Find}(t)$ returns $\{t\}$

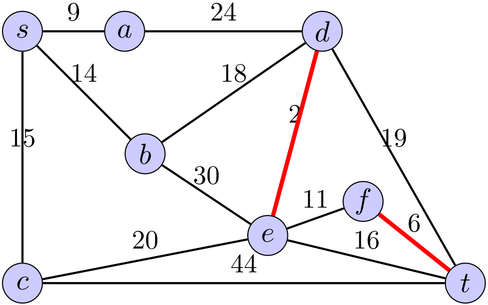
$\textsc{Union}(f, t)$

Kruskal's MST algorithm: an example

Step 2

Edge weight: $2, 6, 9, 11, 14, 15, 16, 18, 19, 20, 24, 30, 44$

Disjoint sets: $\{a\}, \{b\}, \{c\}, \{d, e\}, \{f, t\}, \{s\}$

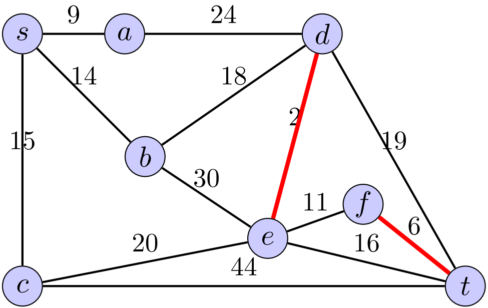


Kruskal's MST algorithm: an example

Step 3

Edge weight: $2, 6, 9, 11, 14, 15, 16, 18, 19, 20, 24, 30, 44$

Disjoint sets: $\{a\}, \{b\}, \{c\}, \{d, e\}, \{f, t\}, \{s\}$

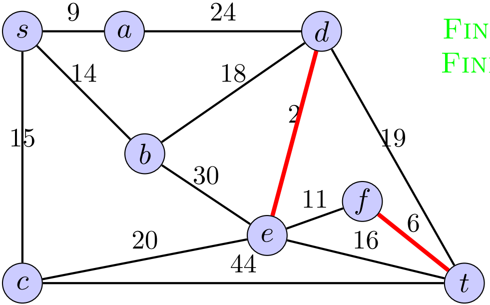


Kruskal's MST algorithm: an example

Step 3

Edge weight: $2, 6, 9, 11, 14, 15, 16, 18, 19, 20, 24, 30, 44$

Disjoint sets: $\{a\}, \{b\}, \{c\}, \{d, e\}, \{f, t\}, \{s\}$



$\textsc{Find}(s)$ returns $\{s\}$

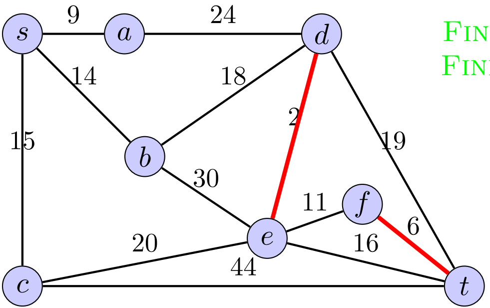
$\textsc{Find}(a)$ returns $\{a\}$

Kruskal's MST algorithm: an example

Step 3

Edge weight: $2, 6, 9, 11, 14, 15, 16, 18, 19, 20, 24, 30, 44$

Disjoint sets: $\{a\}, \{b\}, \{c\}, \{d, e\}, \{f, t\}, \{s\}$



$\textsc{Find}(s)$ returns $\{s\}$

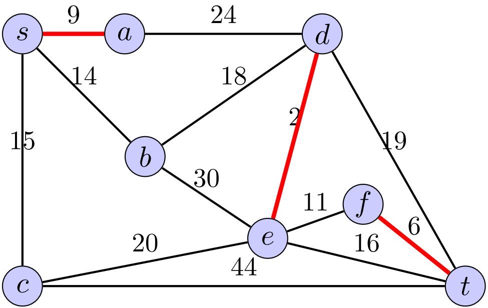
$\textsc{Find}(a)$ returns $\{a\}$

$\textsc{Union}(s, a)$

Step 3

Edge weight: $2, 6, 9, 11, 14, 15, 16, 18, 19, 20, 24, 30, 44$

Disjoint sets: $\{a, s\}, \{b\}, \{c\}, \{d, e\}, \{f, t\}$

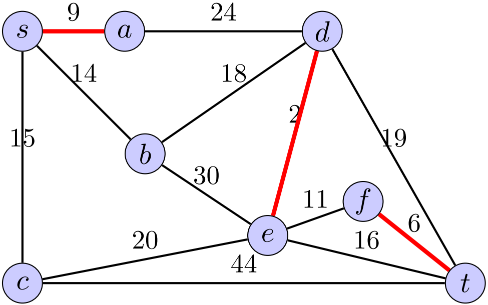


Kruskal's MST algorithm: an example

Step 4

Edge weight: $2, 6, 9, 11, 14, 15, 16, 18, 19, 20, 24, 30, 44$

Disjoint sets: $\{a, s\}, \{b\}, \{c\}, \{d, e\}, \{f, t\}$

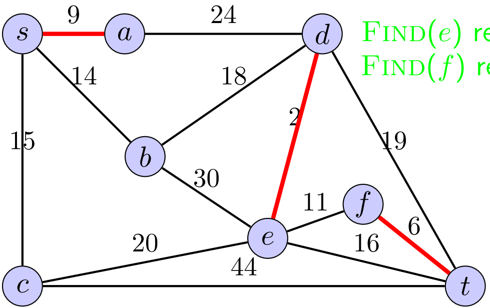


Kruskal's MST algorithm: an example

Step 4

Edge weight: $2, 6, 9, 11, 14, 15, 16, 18, 19, 20, 24, 30, 44$

Disjoint sets: $\{a, s\}, \{b\}, \{c\}, \{d, e\}, \{f, t\}$



$\textsc{Find}(e)$ returns $\{d, e\}$

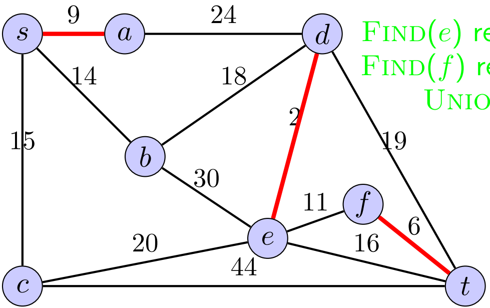
$\textsc{Find}(f)$ returns $\{f, t\}$

Kruskal's MST algorithm: an example

Step 4

Edge weight: $2, 6, 9, 11, 14, 15, 16, 18, 19, 20, 24, 30, 44$

Disjoint sets: $\{a, s\}, \{b\}, \{c\}, \{d, e\}, \{f, t\}$



$\text{FIND}(e)$ returns $\{d, e\}$

$\text{FIND}(f)$ returns $\{f, t\}$

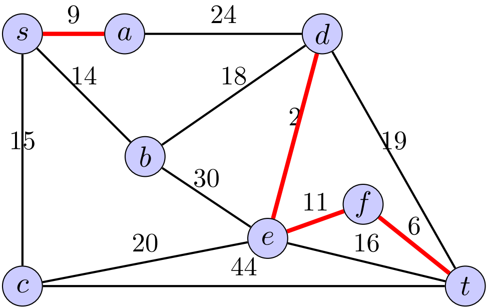
$\text{UNION}(e, f)$

Kruskal's MST algorithm: an example

Step 4

Edge weight: $2, 6, 9, 11, 14, 15, 16, 18, 19, 20, 24, 30, 44$

Disjoint sets: $\{a, s\}, \{b\}, \{c\}, \{d, e, f, t\}$

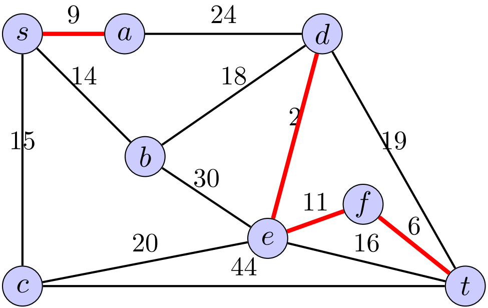


Kruskal's MST algorithm: an example

Step 5

Edge weight: $2, 6, 9, 11, 14, 15, 16, 18, 19, 20, 24, 30, 44$

Disjoint sets: $\{a, s\}, \{b\}, \{c\}, \{d, e, f, t\}$

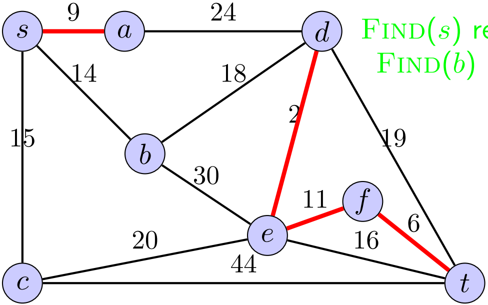


Kruskal's MST algorithm: an example

Step 5

Edge weight: $2, 6, 9, 11, 14, 15, 16, 18, 19, 20, 24, 30, 44$

Disjoint sets: $\{a, s\}, \{b\}, \{c\}, \{d, e, f, t\}$



$\textsc{Find}(s)$ returns $\{s, a\}$

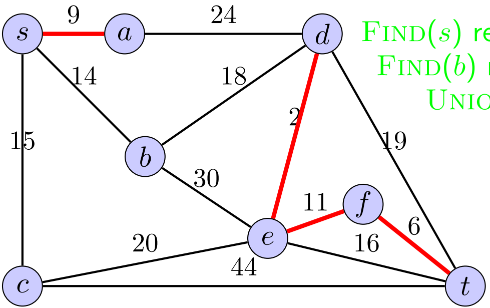
$\textsc{Find}(b)$ returns $\{b\}$

Kruskal's MST algorithm: an example

Step 5

Edge weight: $2, 6, 9, 11, 14, 15, 16, 18, 19, 20, 24, 30, 44$

Disjoint sets: $\{a, s\}, \{b\}, \{c\}, \{d, e, f, t\}$



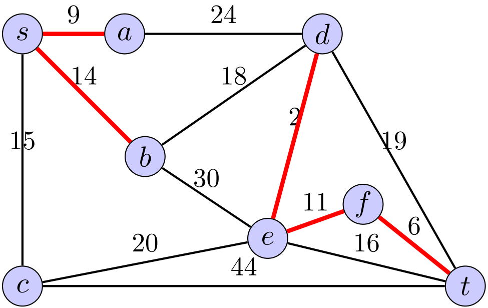
$\textsc{Find}(s)$ returns $\{s, a\}$
$\textsc{Find}(b)$ returns $\{b\}$
$\textsc{Union}(s, b)$

Kruskal's MST algorithm: an example

Step 5

Edge weight: $2, 6, 9, 11, 14, 15, 16, 18, 19, 20, 24, 30, 44$

Disjoint sets: $\{a, s, b\}, \{c\}, \{d, e, f, t\}$

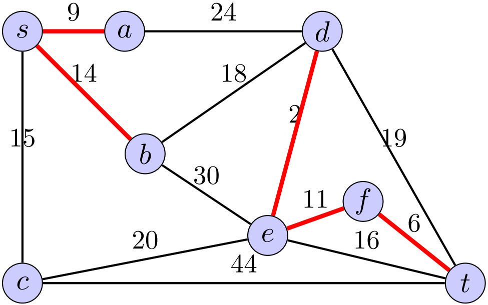


Kruskal's MST algorithm: an example

Step 6

Edge weight: $2, 6, 9, 11, 14, 15, 16, 18, 19, 20, 24, 30, 44$

Disjoint sets: $\{a, s, b\}, \{c\}, \{d, e, f, t\}$

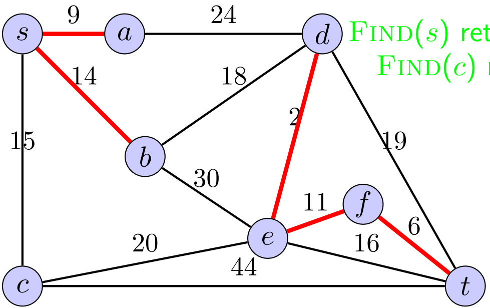


Kruskal's MST algorithm: an example

Step 6

Edge weight: $2, 6, 9, 11, 14, 15, 16, 18, 19, 20, 24, 30, 44$

Disjoint sets: $\{a, s, b\}, \{c\}, \{d, e, f, t\}$



$\text{FIND}(s)$ returns $\{s, a, b\}$

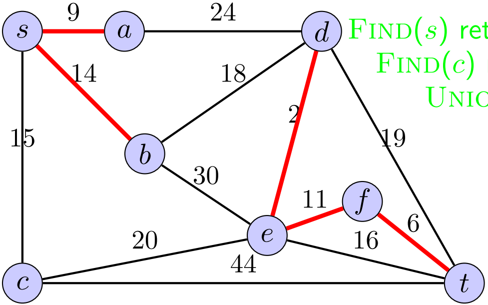
$\text{FIND}(c)$ returns $\{c\}$

Kruskal's MST algorithm: an example

Step 6

Edge weight: $2, 6, 9, 11, 14, 15, 16, 18, 19, 20, 24, 30, 44$

Disjoint sets: $\{a, s, b\}, \{c\}, \{d, e, f, t\}$



$\textsc{Find}(s)$ returns $\{s, a, b\}$

$\textsc{Find}(c)$ returns $\{c\}$

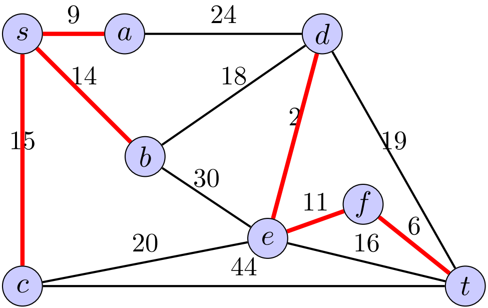
$\textsc{Union}(s, c)$

Kruskal's MST algorithm: an example

Step 6

Edge weight: $2, 6, 9, 11, 14, 15, 16, 18, 19, 20, 24, 30, 44$

Disjoint sets: $\{a, s, b, c\}, \{d, e, f, t\}$

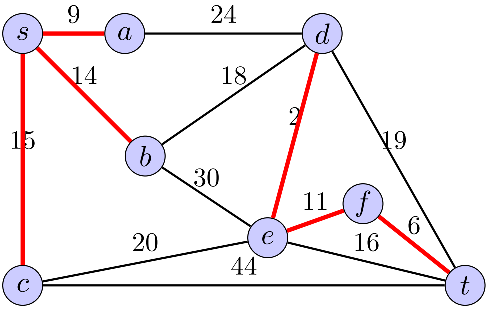


Kruskal's MST algorithm: an example

Step 7

Edge weight: $2, 6, 9, 11, 14, 15, 16, 18, 19, 20, 24, 30, 44$

Disjoint sets: $\{a, s, b, c\}, \{d, e, f, t\}$

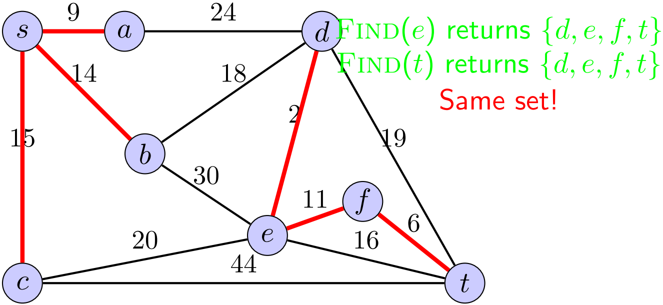


Kruskal's MST algorithm: an example

Step 7

Edge weight: $2, 6, 9, 11, 14, 15, \textcolor{green}{16}, 18, 19, 20, 24, 30, 44$

Disjoint sets: $\{a, s, b, c\}, \{d, e, f, t\}$

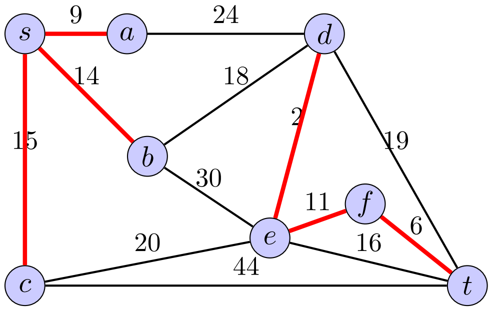


Kruskal's MST algorithm: an example

Step 8

Edge weight: $2, 6, 9, 11, 14, 15, 16, 18, 19, 20, 24, 30, 44$

Disjoint sets: $\{a, s, b, c\}, \{d, e, f, t\}$

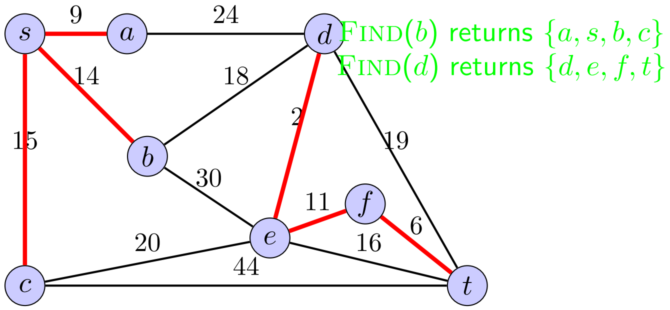


Kruskal's MST algorithm: an example

Step 8

Edge weight: $2, 6, 9, 11, 14, 15, 16, 18, 19, 20, 24, 30, 44$

Disjoint sets: $\{a, s, b, c\}, \{d, e, f, t\}$

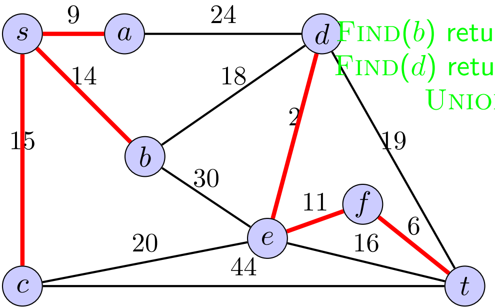


Kruskal's MST algorithm: an example

Step 8

Edge weight: $2, 6, 9, 11, 14, 15, 16, 18, 19, 20, 24, 30, 44$

Disjoint sets: $\{a, s, b, c\}, \{d, e, f, t\}$



$\textsc{Find}(b)$ returns $\{a, s, b, c\}$

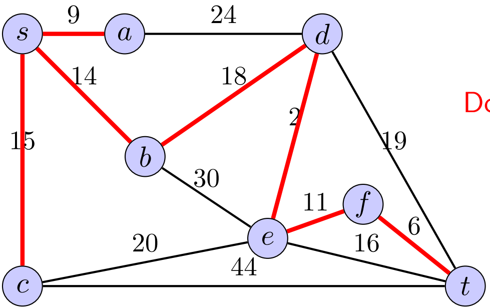
$\textsc{Find}(d)$ returns $\{d, e, f, t\}$

$\textsc{Union}(b, d)$

Step 8

Edge weight: $2, 6, 9, 11, 14, 15, 16, 18, 19, 20, 24, 30, 44$

Disjoint sets: $\{a, s, b, c, d, e, f, t\}$



Time complexity of KRUSKAL'S MST algorithm

Operation	Array	Tree	Link-by-rank	Link-by-rank + path compression
MAKESET	1	1	1	1
FIND	1	n	$\log n$	$\log^* n$
UNION	n	$\cancel{1}$ n	$\log n$	$\log^* n$
KRUSKAL'S MST	$O(n^2)$	$O(mn)$	$O(m \log n)$	$O(m \log^* n)$

KRUSKAL'S MST algorithm: n MAKESET, $n-1$ UNION, and m FIND operations.

Implementing UNION-FIND: array or linked list

Implementing UNION-FIND: array

- Basic idea: for each element, we record its "set name" individually.

	s	a	b	c	d	e	f	t
Set name:	0	1	2	3	4	5	6	7

- Operation:

  FIND(x)

  1: **return** SetName[x];

- Complexity: $O(1)$

Implementing UNION-FIND: array

- Operation:

  $\textsc{Union}(x, y)$

  $s_x = \textsc{Find}(x)$;
  $s_y = \textsc{Find}(y)$;
  for all element i **do**
      if SetName$[i]$==s_y **then**
          SetName$[i]$=s_x
      end if
  end for

	s	a	b	c	d	e	f	t
Set name:	0	1	2	3	4	5	6	7

$\textsc{Union}(d, e)$

Set name:	0	1	2	3	5	5	6	7

$\textsc{Union}(f, e)$

Set name:	0	1	2	3	6	6	6	7

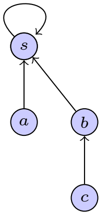
- Complexity: $O(n)$

Tree implementation: organizing a set into a tree with its root as representative of the set

Tree implementation: Find

- Basic idea: We use a tree to store elements of a set, and use root as "set name". Thus, only one representative should be maintained.

Set: $\{s, a, b, c\}$



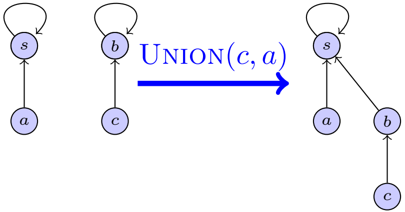
- Operation:

  $\textsc{Find}(x)$

  $r = x$;
  while $r != parent(r)$ **do**
  $\quad r = parent(r)$;
  end while
  return r;

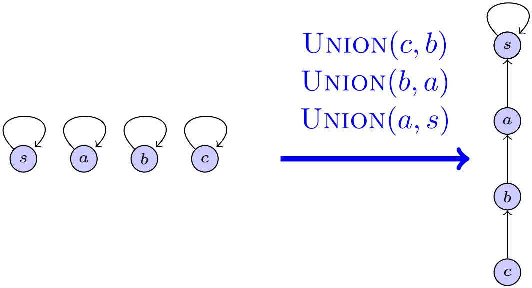
Tree implementation: $\textsc{Union}$

- Operation:
  $\textsc{Union}(x, y)$
  1: $r_x = \textsc{Find}(x)$;
  2: $r_y = \textsc{Find}(y)$;
  3: $parent(r_x) = r_y$;
- Example: $\textsc{Union}(c, a)$



Tree implementation: worst case

- Worst case: the tree degenerates into a linked list. For example, $\textsc{Union}(c, b)$, $\textsc{Union}(b, a)$, $\textsc{Union}(a, s)$.



- Complexity: $\textsc{Find}$ takes $O(n)$ time, and $\textsc{Union}$ takes $O(n)$ time.
- Question: how to keep a "good" tree shape to limit path length?

Link-by-rank: shorten the path by maintaining a balanced tree

Tree implementation with link-by-size

- Basic idea: We shorten the path by maintaining a balanced-tree. In fact, this will limit path length to $O(\log n)$.
- How to maintain a balanced tree? Each node is associated with a $rank$, denoting its height. The tree has a balanced shape via linking smaller tree to larger tree; if tie, increase the rank of new root by 1.

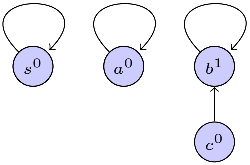
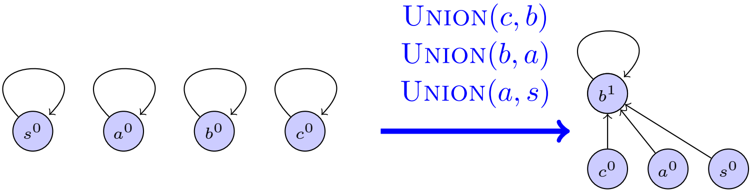


Figure 2: Three sets: $\{s\}$, $\{a\}$, $\{b, c\}$

Tree implementation with link-by-size: UNION operation

UNION(x, y)

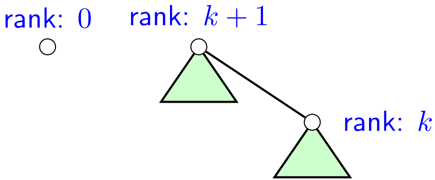
$r_x = $ FIND(x);
$r_y = $ FIND(y);
if $rank(r_x) > rank(r_y)$ **then**
    $parent(r_y) = r_x$;
else
    $parent(r_x) = r_y$;
    if $rank(r_x) == rank(r_y)$ **then**
        $rank(r_y) = rank(r_y) + 1$;
    end if
end if



Note: a node's rank will not change after it becomes an internal node

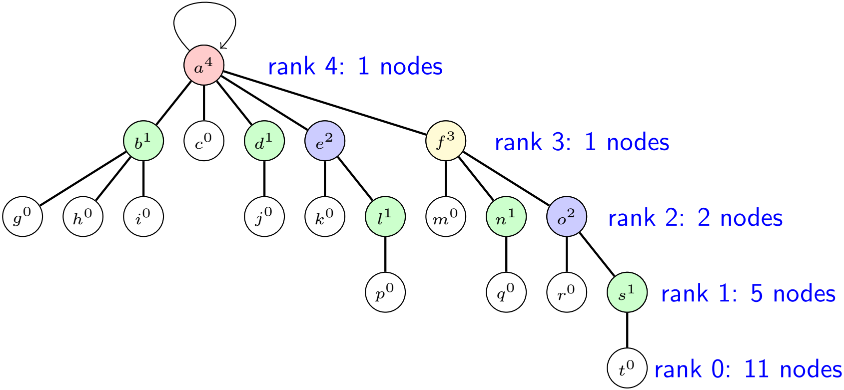
Properties of rank I

1. For any node x, $rank(x) < rank(parent(x))$.
2. Any tree with root rank of k contains at least 2^k nodes. (Hint: by induction on k.)
3. Once a root node was changed into internal node during a UNION operation, its rank will not change afterwards.



4. Suppose we have n elements. The number of rank k nodes is at most $\frac{n}{2^k}$. (Hint: Different nodes of rank k share no common descendants.)

Properties of rank II

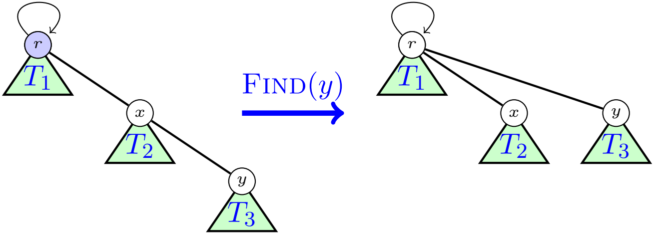


- Thus, all of the trees have height less than $\log n$, which means both FIND and UNION take $O(\log n)$ time.

Path compression: compress paths to make further $\textsc{Find}$ efficient

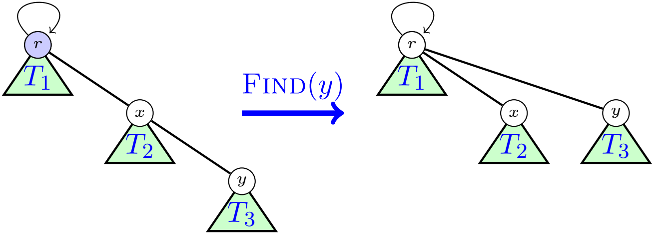
Path compression

- Basic idea: After finding the root r of the tree containing x, we change the parent of the nodes along the path to point directly to r. Thus, the subsequent $\textsc{Find}(x)$ operations will be efficient.



- Note: Path compression changes height of nodes but does not change rank of nodes. We always have $height(x) \leq rank(x)$; thus, the three properties still hold.

Path compression: FIND operation



FIND(x)

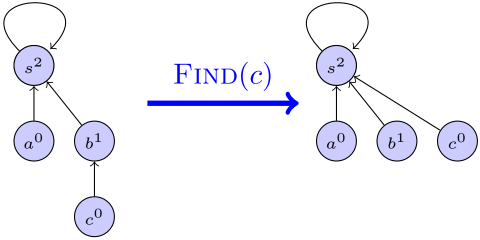
1: **if** $x != parent(x)$ **then**
2: $\quad parent(x) = \text{FIND}(parent(x))$;
3: **else**
4: $\quad$ **return** x;
5: **end if**

Some properties of FIND and UNION

- FIND operations change internal nodes only while UNION operations change root node only.
- Path compression changes parent node of certain internal nodes. However, it will not change the root nodes, rank of any node, and thus will not affect UNION operations.

Path compression: complexity

- Example: $\textsc{Find}(c)$



- A $\textsc{Find}(c)$ operation might takes long time; however, the path compression makes subsequent $\textsc{Find}(c)$ (and other middle nodes in the path) efficient.

Theorem

Starting from each item forming an individual set, any sequence of m operations (including $\textsc{Find}$ and $\textsc{Union}$) over n elements takes $O(m \log^* n)$ time.

Analysis of path compression: a brief history

- In 1972, Fischer proved a bound of $O(m \log \log n)$.
- In 1973, Hopcroft and Ullman proved a bound of $O(m \log^* n)$.
- In 1975, R. Tarjan et al. proved a bound using "inverse Ackerman function".
- Later, R. Tarjan, et. al. and Harfst and Reingold proved the bound using the potential function technique.

Here, we present the proof in *Algorithms* by S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani.

$\log^* n$: Iterated logarithm function

- Intuition: the number of logarithm operations to make n to be 1.
- $\log^* n = \begin{cases} 0 & \text{if } n = 1 \\ 1 + \log^*(\log n) & \text{otherwise} \end{cases}$

n	$\log^* n$
1	0
2	1
$[3, 2^2]$	2
$[5, 2^4]$	3
$[17, 2^{16}]$	4
$[65537, 2^{65536}]$	5

- Note: $\log^* n$ increases very slowly, and we have $\log^* n < 5$ unless n exceeds the number of atoms in the universe.

Analysis of rank

- Let's divide the nonzero ranks into groups as below.

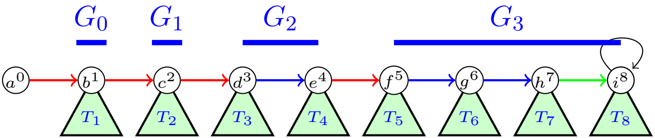
Group	Rank	Upper bound of #elements
0	1	$\frac{n}{2}$
1	2	$\frac{n}{2^2}$
2	$[3, 2^2]$	$\frac{n}{2^2}$
3	$[5, 2^4]$	$\frac{n}{2^4}$
4	$[17, 2^{16}]$	$\frac{n}{2^{16}}$
5	$[65537, 2^{65536}]$	$\frac{n}{2^{65536}}$

- Note:
  - Group number is $\log^* rank$ and the number of groups is at most $\log^* n$.
  - The number of elements in the rank group G_k $(k \geq 2)$ is at most $\frac{n}{\underbrace{2^{2^{\cdots^{2}}}}_{k}}$ as the number of nodes with rank r is at most $\frac{n}{2^r}$.

    We will see why the group was set to take the form $[\underbrace{2^{2^{\cdots^{2}}}}_{\text{k-1}}+1, \underbrace{2^{2^{\cdots^{2}}}}_{\text{k}}]$ soon.

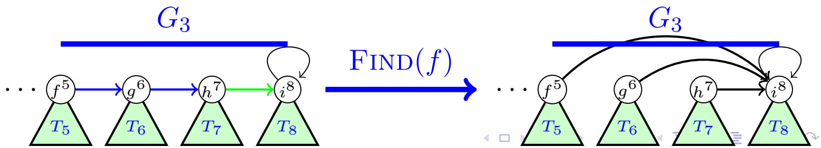
Amortized analysis: total time of m FIND operations

- Basic idea: a FIND operation might take long time; however, path compression makes subsequent FIND operations efficient.
- Let's consider a sequence of m FIND operations, and divide the traversed links into the following three types:
  - **Type 1:** links to **root**
  - **Type 2:** links traversed **between** different rank groups
  - **Type 3:** links traversed **within** the same rank groups
- For example, the links that $\text{FIND}(a)$ travels:

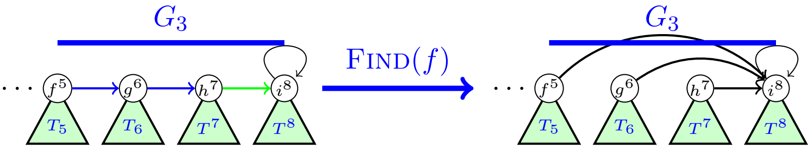


- The total time is $T = T_1 + T_2 + T_3$, where T_i denotes the number of links of type i. We have:
  - $T_1 = O(m)$.
  - $T_2 = O(m \log^* n)$. (Hint: there are at most $\log^* n$ groups.)
  - $T_3 = O(n \log^* n)$. (To be shown later.)
- Thus, $T = O(m \log^* n)$.

- Note that the $\textsc{Find}(f)$ operation of type 3 will change $parent(f)$: the rank of $parent(f)$ increases by at least 1. In the example shown below, $parent(f)$ changes from g^6 to i^8. Let's consider the next $\textsc{Find}(f)$ operation.
  1. If a $\textsc{Union}$ operation linked i^8 to another root node before the next $\textsc{Find}(f)$ operation, then this $\textsc{Find}(f)$ operation will again lead to the increase of the rank of $parent(f)$.
  2. Otherwise, $parent(f)$ is itself a root, and the next $\textsc{Find}(f)$ operation will be accounted into T_1.
- Hence, after at most 2^4 $\textsc{Find}(f)$ operations of type 3, $parent(f)$ is itself a root, or the rank of $parent(f)$ increase to make it lie in another group different from f, leading subsequent $\textsc{Find}(f)$ operations to be accounted into T_2 or T_1.



Why $T_3 = O(n \log^* n)$? continued



- Formally we have

$$\begin{aligned} T_3 &\leq \sum_{k=2}^{\log^* n} \sum_{f \in G_k} \underbrace{2^{2^{\cdots^2}}}_{\text{k}} && \text{(the largest rank in group } G_k \text{ is } \underbrace{2^{2^{\cdots^2}}}_{\text{k}}\text{)} \\ &\leq \sum_{k=2}^{\log^* n} \frac{n}{\underbrace{2^{2^{\cdots^2}}}_{\text{k}}} \underbrace{2^{2^{\cdots^2}}}_{\text{k}} && \text{(\#nodes in group } G_k \leq \frac{n}{\underbrace{2^{2^{\cdots^2}}}_{\text{k}}}\text{)} \\ &= O(n \log^* n) \end{aligned}$$

$T_3 = O(n \log^* n)$: another explanation using "credit"

- Let's give each node credits as soon as it ceases to be a root. If its rank is in the group $[k+1, 2^k]$, we give it 2^k credits.
- The total credits given to all nodes is $n \log^* n$. (Hint: each group of nodes receive n credits.)
- If $rank(f)$ and $rank(parent(f))$ are in the same group, we will charge f 1 credit.
- In this case, $rank(parent(f))$ increases by at least 1.
- Thus, after at most 2^k $\textsc{Find}$ operations, $rank(parent(f))$ will be in a higher group.
- Thus, f has enough credits until $rank(f)$ and $rank(parent(f))$ are in different group, which will be accounted into T_2.