

# Algorithms and Data Structures: Minimum Spanning Trees

28th Oct, 1st & 4th Nov, 2011

*ADS: lects 10, 11, 12 – slide 1 – 28th Oct, 1st & 4th Nov, 2011*

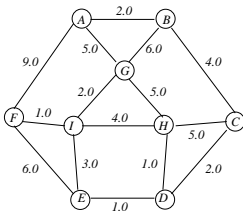
# Weighted Graphs

## Definition 1

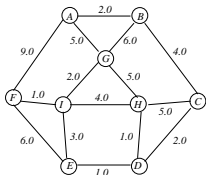
A *weighted* (directed or undirected graph) is a pair  $(\mathcal{G}, W)$  consisting of a graph  $\mathcal{G} = (V, E)$  and a *weight function*  $W : E \rightarrow \mathbb{R}$ .

In this lecture, we always assume that **weights are non-negative**, i.e., that  $W(e) \geq 0$  for all  $e \in E$ .

## Example



## Representations of Weighted Graphs (as Matrices)

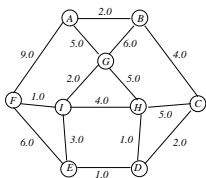


### Adjacency Matrix

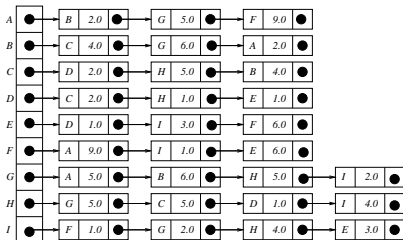
$$\begin{pmatrix} 0 & 2.0 & 0 & 0 & 0 & 9.0 & 5.0 & 0 & 0 \\ 2.0 & 0 & 4.0 & 0 & 0 & 0 & 6.0 & 0 & 0 \\ 0 & 4.0 & 0 & 2.0 & 0 & 0 & 0 & 5.0 & 0 \\ 0 & 0 & 2.0 & 0 & 1.0 & 0 & 0 & 1.0 & 0 \\ 0 & 0 & 0 & 1.0 & 0 & 6.0 & 0 & 0 & 3.0 \\ 9.0 & 0 & 0 & 0 & 6.0 & 0 & 0 & 0 & 1.0 \\ 5.0 & 6.0 & 0 & 0 & 0 & 0 & 0 & 5.0 & 2.0 \\ 0 & 0 & 5.0 & 1.0 & 0 & 0 & 5.0 & 0 & 4.0 \\ 0 & 0 & 0 & 0 & 3.0 & 1.0 & 2.0 & 4.0 & 0 \end{pmatrix}$$

ADS: lects 10, 11, 12 – slide 3 – 28th Oct, 1st & 4th Nov, 2011

# Representations of Weighted Graphs (as Adj. lists)



## Adjacency Lists



# Connecting Sites

## Problem

Given a collection of *sites* and *costs* of connecting them, find a minimum cost way of connecting all sites.

## Our formal model

- ▶ Sites are vertices of a *weighted graph*, and the weights of the edges represent the cost of connecting their endpoints.
- ▶ It is reasonable to assume that the graph is *undirected* and *connected*.
- ▶ The *cost* of a *subgraph* is the sum of the costs of its edges.
- ▶ The problem is to find a *subgraph of minimum cost* that *connects all vertices*.

Compare this to the shortest path problem, where we are only interested in minimum cost connections between pairs of vertices.

## Spanning Trees

$\mathcal{G} = (V, E)$  undirected connected graph and  $W$  weight function.

$\mathcal{H} = (V^H, E^H)$  with  $V^H \subseteq V$  and  $E^H \subseteq E$  subgraph of  $\mathcal{G}$ .

- ▶ The *weight* of  $\mathcal{H}$  is the number

$$W(\mathcal{H}) = \sum_{e \in E^H} W(e).$$

- ▶  $\mathcal{H}$  is a *spanning subgraph* of  $\mathcal{G}$  if  $V^H = V$ .

### Observation 2

*A connected spanning subgraph of minimum weight is a tree.*

# Minimum Spanning Trees

$(\mathcal{G}, W)$  undirected connected weighted graph

## Definition 3

A *minimum spanning tree (MST)* of  $\mathcal{G}$  is a connected spanning subgraph  $\mathcal{T}$  of  $\mathcal{G}$  of minimum weight.

The **minimum spanning tree problem**:

**Input:** Undirected connected weighted graph  $(\mathcal{G}, W)$

**Output:** An MST of  $\mathcal{G}$

# Prim's Algorithm

## Idea

Grow an MST out of a single vertex by always adding edges of minimum weight.

A *fringe edge* for a subtree  $\mathcal{T}$  of a graph is an edge with exactly one endpoint in  $\mathcal{T}$  (so  $e = (u, v)$  with  $u \in \mathcal{T}$  and  $v \notin \mathcal{T}$ ).

**Algorithm** PRIM( $\mathcal{G}, W$ )

1.  $\mathcal{T} \leftarrow$  one vertex tree with arbitrary vertex of  $\mathcal{G}$
2. **while** there is a fringe edge **do**
3.     add fringe edge of minimum weight to  $\mathcal{T}$
4. **return**  $\mathcal{T}$

This algorithm uses a **Greedy strategy**.

First time we have used the Greedy approach (seriously) in this course.

### Correctness of Prim's algorithm

1. Throughout the execution of PRIM,  $\mathcal{T}$  remains a tree.

*Proof:* To show this we need to show that throughout the algorithm,  $\mathcal{T}$  is (i) **always connected** and (ii) **never contains a cycle**.

(i) Only edges with an endpoint in  $\mathcal{T}$  are added to  $\mathcal{T}$ , so  $\mathcal{T}$  remains connected.

(ii) We never add any edge which has both endpoints in  $\mathcal{T}$  (we only allow a single endpoint), so the algorithm will never construct a cycle.

## Correctness of Prim's algorithm (cont'd)

2. All vertices will eventually be added to  $\mathcal{T}$ .

*Proof:* by contradiction ... (depends on our assumption that the graph  $\mathcal{G}$  was connected.)

- ▶ Suppose  $w$  is a vertex that never gets added to  $\mathcal{T}$  (as usual, in proof by contradiction, we suppose the *opposite* of what we want).
- ▶ Let  $v = v_0 e_1 v_1 e_2 \dots v_n = w$  be a path from some vertex  $v$  inside  $\mathcal{T}$  to  $w$  (we know such a path *must* exist, because  $\mathcal{G}$  is connected). Let  $v_i$  be the **first** vertex on this path that never got added to  $\mathcal{T}$ .
- ▶ After  $v_{i-1}$  was added to  $\mathcal{T}$ ,  $e_i = (v_{i-1}, v_i)$  would have become a fringe edge. Also, it would have remained as a fringe edge unless  $v_i$  was added to  $\mathcal{T}$ .
- ▶ So eventually  $v_i$  would have been added, because Prim's algorithm *only stops* if there are no fringe edges. So our assumption was wrong. So we *must* have  $w$  in  $\mathcal{T}$  for every vertex  $w$ .

### Correctness of Prim's algorithm (cont'd)

3. Throughout the execution of PRIM,  $\mathcal{T}$  is contained in some MST of  $\mathcal{G}$ .  
(This will be enough to prove Correctness, because at the end, when all vertices are added, we will have an MST)

*Proof: (by induction)*

- ▶ Suppose that  $\mathcal{T}$  is contained in an MST  $\mathcal{T}'$  and that fringe edge  $e = (x, y)$  is then added to  $\mathcal{T}$ . We shall prove that  $\mathcal{T} + e$  is also contained in *some* MST  $\mathcal{T}''$  (not necessarily the MST  $\mathcal{T}'$ ).
- ▶ If  $e$  is contained in  $\mathcal{T}'$ , then our proof is easy, we can simply let  $\mathcal{T}'' = \mathcal{T}'$ .
- ▶ Otherwise, consider some path  $\mathcal{P}$  from  $x$  to  $y$  in  $\mathcal{T}'$ . The path  $\mathcal{P}$  contains *exactly one* edge  $e' = (x', y')$  which is currently (at the time we are adding  $e = (x, y)$ ) a competing fringe edge for Prim's algorithm.
- ▶ Then  $W(e) \leq W(e')$  (otherwise  $e'$  would be added instead of  $e$ ).
- ▶ Let  $\mathcal{T}'' = \mathcal{T}' + e - e'$ .
- ▶  $\mathcal{T}''$  is a tree (we add  $e$  to make a cycle, then delete  $e'$  to break it).
- ▶  $\mathcal{T}''$  has the same vertices as  $\mathcal{T}'$ , thus it is a spanning tree.
- ▶ Moreover,  $W(\mathcal{T}'') \leq W(\mathcal{T}')$ , thus  $\mathcal{T}''$  is also a Minimum Spanning Tree (MST).

# Toward an Implementation

## Improvement

- ▶ Instead of fringe edges, we think about adding *fringe vertices* to the tree.
- ▶ A *fringe vertex* is a vertex  $y$  not in  $\mathcal{T}$  that is an endpoint of a fringe edge.
- ▶ The *weight* of a fringe vertex  $y$  is

$$\min\{W(e) \mid e = (x, y) \text{ fringe edge}\}$$

- ▶ To be able to recover the tree, with each vertex  $y$  we store its *parent* in the tree.

We store the fringe vertices in a *priority queue*.

## Priority Queues with Decreasing Key

A *priority queue* is an ADT for storing a collection of elements with an associated *key* - remember *Inf 2B*.

The following methods are supported:

- ▶ `INSERT( $e, k$ )`: Insert element  $e$  with key  $k$ .
- ▶ `GET-MIN()`: Return an element with minimum key; an error occurs if the priority queue is empty.
- ▶ `EXTRACT-MIN()`: Return and remove an element with minimum key; an error occurs if the priority queue is empty.
- ▶ `IS-EMPTY()`: Return `TRUE` if the priority queue is empty and `FALSE` otherwise.

To update the keys during the execution of `PRIM`, we need priority queues supporting the following additional method:

- ▶ `DECREASE-KEY( $e, k$ )`: Set the key of  $e$  to  $k$  and update the priority queue. It is assumed that  $k$  is smaller than or equal to the old key of  $e$ .

## Implementation of Prim's Algorithm

### Algorithm PRIM( $\mathcal{G}$ , $W$ )

1. Initialise parent array  $\pi$ :  
 $\pi[v] \leftarrow \text{NIL}$  for all vertices  $v$
2. Initialise weight array:  
 $\text{weight}[v] \leftarrow \infty$  for all vertices  $v$
3. Initialise priority queue  $Q$
4.  $v \leftarrow$  arbitrary vertex of  $\mathcal{G}$
5.  $Q.\text{INSERT}(v, 0)$
6.  $\text{weight}[v] = 0$
7. **while not**( $Q.\text{IS-EMPTY}()$ ) **do**
8.      $y \leftarrow Q.\text{EXTRACT-MIN}()$
9.     **for all**  $z$  adjacent to  $y$  **do**
10.         RELAX( $y, z$ )
11. **return**  $\pi$

### Algorithm RELAX( $y, z$ )

1.  $w \leftarrow W((y, z))$
2. **if**  $\text{weight}[z] = \infty$  **then**
3.      $\text{weight}[z] \leftarrow w$
4.      $\pi[z] \leftarrow y$
5.      $Q.\text{INSERT}(z, w)$
6. **else if**  $w < \text{weight}[z]$  **then**
7.      $\text{weight}[z] \leftarrow w$
8.      $\pi[z] \leftarrow y$
9.      $Q.\text{DECREASE KEY}(z, w)$

## Analysis of Prim's Algorithm

Let  $n$  be the number of vertices and  $m$  the number of edges of the input graph.

- ▶ Lines 1–6,11 of Prim require time  $\Theta(n)$
- ▶  $Q$  contains each of the  $n$  vertices of  $\mathcal{G}$  at most once. Thus the loop in lines 7–10 is iterated at most  $n$  times.  
Thus, disregarding for now the time required to execute the inner loop (9-10), the execution of the loop requires time

$$\Theta((n \cdot T_{\text{EXTRACT-MIN}}(n)))$$

- ▶ The inner loop is executed at most *once for every edge*. Thus its execution requires time

$$\Theta(m \cdot T_{\text{RELAX}}(n, m)).$$

## Analysis of Prim's Algorithm (cont'd)

- ▶ Disregarding the time needed to execute INSERT and DECREASE-KEY, the execution of RELAX requires time  $\Theta(1)$ .
- ▶ INSERT is executed at most once for every vertex, which requires time

$$\Theta(n \cdot T_{\text{INSERT}}(n))$$

- ▶ DECREASE-KEY is executed at most once for every edge, which requires time

$$\Theta(m \cdot T_{\text{DECREASE KEY}}(n))$$

Overall, we get

$$T_{\text{PRIM}}(n, m) = \Theta\left(n(T_{\text{EXTRACT-MIN}}(n) + T_{\text{INSERT}}(n)) + mT_{\text{DECREASE KEY}}(n)\right)$$

## Priority Queue Implementations

- ▶ **Array:** Elements are simply stored in an array.
- ▶ **Heap:** Elements are stored in a binary heap (see Inf2B (ADS note 7)), [CLRS] Section 6.5)
- ▶ **Fibonacci Heap:** Sophisticated variant of the simple binary heap (see [CLRS] Chapters 19 and 20)

method	running time		
	Array	Heap	Fibonacci Heap
INSERT	$\Theta(1)$	$\Theta(\lg n)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(n)$	$\Theta(\lg n)$	$\Theta(\lg n)$
DECREASE KEY	$\Theta(1)$	$\Theta(\lg n)$	$\Theta(1)$ (amortised)

## Running Time of Prim's Algorithm

### Recall:

$$T_{\text{PRIM}}(n, m) = \Theta\left(n(T_{\text{EXTRACT-MIN}}(n) + T_{\text{INSERT}}(n)) + mT_{\text{DECREASE KEY}}(n)\right)$$

This yields

- ▶ With **array** implementation of priority queue:

$$T_{\text{PRIM}}(n, m) = \Theta(n^2).$$

- ▶ With **heap** implementation of priority queue:

$$T_{\text{PRIM}}(n, m) = \Theta((n + m) \lg n).$$

- ▶ With **Fibonacci heap** implementation of priority queue:

$$T_{\text{PRIM}}(n, m) = \Theta(n \lg n + m).$$

## Remarks

- ▶ The Fibonacci heap implementation is mainly of theoretical interest. It is not much used in practice because it is very complicated and the constants hidden in the  $\Theta$ -notation are large.
- ▶ For *dense graphs* with  $m = \Theta(n^2)$ , the array implementation is probably the best, because it is so simple.
- ▶ For *sparser graphs* with  $m \in O(\frac{n^2}{\lg n})$ , the heap implementation is a good alternative, since it is still quite simple, but more efficient for smaller  $m$ .

Instead of using binary heaps, the use of  $d$ -ary heaps for some  $d \geq 1$  can speed up the algorithm (see [Sedgewick] for a discussion of practical implementations of Prim's algorithm).

## Reading Assignment (Prim's Alg)

[CLRS] Chapter 23 (pages 561–579). *This is Chapter 24 (pages 498–513) of [CLR].*

### Online Resources

Wikipedia on:

Minimum Spanning Trees:

[http://en.wikipedia.org/wiki/Minimum\\_spanning\\_tree](http://en.wikipedia.org/wiki/Minimum_spanning_tree)

Prim's Algorithm

[http://en.wikipedia.org/wiki/Prim's\\_algorithm](http://en.wikipedia.org/wiki/Prim's_algorithm)

## Problems (Prim's Alg)

1. Exercise 23.2-1, page 573 of [CLRS] Ex. 24.2-1, pg. 510 of [CLR].
2. Exercise 23.2-5, page 574 of [CLRS] Ex. 24.2-4, pg. 510 of [CLR].
3. In line 3 of Prim's algorithm, there may be more than one fringe edge of minimum weight. Suppose we add all these minimum edges in one step. Does the algorithm still compute a MST?
4. Prove that our *implementation* of Prim's algorithm on slide 10 is correct - ie, that it computes an MST. What is the difference between this and the suggested algorithm above (qn 3)?

## Kruskal's Algorithm

A different approach to computing MSTs.

A *forest* is a graph whose connected components are trees.

### Idea

Starting from the spanning forest without any edges, repeatedly add edges of minimum weight until the forest becomes a tree.

**Algorithm** KRUSKAL( $\mathcal{G}, W$ )

1.  $F \leftarrow \emptyset$
2. **for all**  $e \in E$  in the order of increasing weight **do**
3.     **if** the endpoints of  $e$  belong to different connected components of  $(V, F)$
4.         **then**  
           $F \leftarrow F \cup \{e\}$
5. **return** tree with edge set  $F$

## Correctness of Kruskal's algorithm

1. Throughout the execution of `KRUSKAL`,  $(V, F)$  remains a spanning forest.

*Proof:*  $(V, F)$  is a spanning subgraph because the vertex set is  $V$ . It always remains a forest because edges with endpoints in different connected components never induce a cycle.

2. Eventually,  $(V, F)$  will be connected and thus a spanning tree.

*Proof:* Suppose that after the complete execution of the loop,  $(V, F)$  has a connected component  $(V_1, F_1)$  with  $V_1 \neq V$ . Since  $\mathcal{G}$  is connected, there is an edge  $e \in E$  with exactly one endpoint in  $V_1$ . This edge would have been added to  $F$  when being processed in the loop, so this can never happen.

3. Throughout the execution of `KRUSKAL`,  $(V, F)$  is contained in some MST of  $\mathcal{G}$ .

*Proof:* Similar to the proof of the corresponding statement for Prim's algorithm.

## Data Structures for Disjoint Sets

- ▶ A *disjoint set* data structure maintains a collection  $\mathcal{S} = \{S_1, \dots, S_k\}$  of *disjoint sets*.
- ▶ The sets are *dynamic*, i.e., they may change over time.
- ▶ Each set  $S_i$  is identified by some *representative*, which is some member of that set.

### Operations:

- ▶ MAKE-SET( $x$ ): Creates new set whose only member is  $x$ . The representative is  $x$ .
- ▶ UNION( $x, y$ ): Unites set  $S_x$  containing  $x$  and set  $S_y$  containing  $y$  into a new set  $S$  and removes  $S_x$  and  $S_y$  from the collection.
- ▶ FIND-SET( $x$ ): Returns representative of the set holding  $x$ .

## Implementation of Kruskal's Algorithm

**Algorithm** KRUSKAL( $\mathcal{G}, W$ )

1.  $F \leftarrow \emptyset$
2. **for all** vertices  $v$  of  $\mathcal{G}$  **do**
3.     MAKE-SET( $v$ )
4. sort edges of  $\mathcal{G}$  into non-decreasing order by weight
5. **for all** edges  $(u, v)$  of  $\mathcal{G}$  in non-decreasing order by weight **do**
6.     **if** FIND-SET( $u$ )  $\neq$  FIND-SET( $v$ ) **then**
7.          $F \leftarrow F \cup \{(u, v)\}$
8.         UNION( $u, v$ )
9. **return**  $F$

## Analysis of KRUSKAL

Let  $n$  be the number of vertices and  $m$  the number of edges of the input graph

- ▶ Line 1, line 9:  $\Theta(1)$
- ▶ Loop in Lines 2–3:  $\Theta(n \cdot T_{\text{MAKE-SET}}(n))$
- ▶ Line 4:  $\Theta(m \lg m)$
- ▶ Loop in Lines 5–8:  $\Theta\left(m \cdot T_{\text{FIND-SET}}(n) + n \cdot T_{\text{UNION}}(n)\right)$ .

Overall:

$$\Theta\left(n\left(T_{\text{MAKE-SET}}(n) + T_{\text{UNION}}(n)\right) + m\left(\lg m + T_{\text{FIND-SET}}(n)\right)\right)$$

With efficient implementation of disjoint set this amounts to

$$T(n, m) = \Theta(m \lg m).$$

## Amortized Analysis

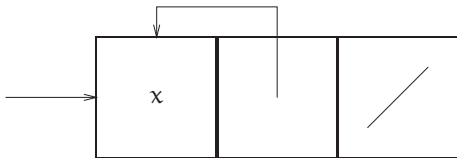
We want to analyse the following: ...

Given a sequence of  $m$  “Disjoint Set” operations (MAKE-SET, UNION, FIND-SET), bound the total running time to perform **a sequence of  $m$  operations**.

- ▶ Usually parametrize in terms of  $n$  (no. of MAKE-SET operations), not just  $m$ .
- ▶ (Of course) the bound we get will depend on the particular data structure/implementation we design for this problem.

## Linked List Implementation of Disjoint Sets

Each element represented by a pointer to a cell ( $x$ ):



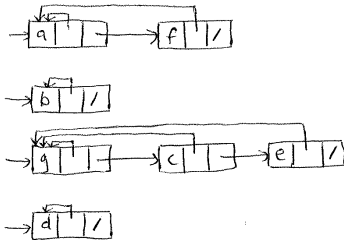
- ▶ Use a linked list for each set.
- ▶ Representative of the set is at the head of the list.
- ▶ Each cell has a pointer direct to the representative (head of the list).

Example ( $\{a, f\}$ ,  $\{b\}$ ,  $\{g, c, e\}$ ,  $\{d\}$  :)

Linked list representation of

---

$\{a, f\}$ ,  $\{b\}$ ,  $\{g, c, e\}$ ,  $\{d\}$  :



The representatives are  $a, b, g$  and  $d$ .

## Analysis of Linked List Implementation

MAKE-SET: constant time.

FIND-SET: constant time.

UNION: Naive implementation of

UNION( $x, y$ )

appends  $x$ 's list onto end of  $y$ 's list.

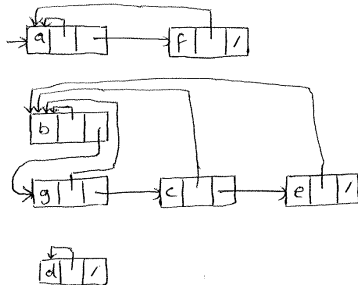
**Assumption:** Representative  $y$  of each set has attribute  $last[y]$ : a pointer to last cell of  $y$ 's list. (Not shown in diagrams.)

**Snag:** have to update “representative pointer” in each cell of  $x$ 's list to point to the representative (head) of  $y$ 's list.

Cost is:

$\Theta(\text{length of } x\text{'s list}).$

## Example (cont'd)



after doing Union (g, b).

## Conventions for Further Analysis

Express running time in terms of:

$n$ : the number of MAKE-SET operations,

$m$ : the number of MAKE-SET, UNION and FIND-SET operations.

### Note

1. After  $n - 1$  UNION operations only one set remains.
2.  $m \geq n$ .

## A nasty example

**Take:**  $n = \lceil m/2 \rceil + 1$ ,  $q = m - n = \lfloor m/2 \rfloor - 1$ .

**Elements:**  $x_1, x_2, \dots, x_n$ .

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( $x_{q-1}, x_q$ )	$q - 1$
Total	$\Theta(m^2)$

# The Weighted-Union Heuristics

## Idea

Record length of each list. To execute

$\text{UNION}(x, y)$

append **shorter** list to **longer** one (breaking ties arbitrarily).

## Theorem 4

*Using the linked-list representation of disjoint sets and the weighted-union heuristic, a sequence of  $m$  MAKE-SET, UNION & FIND-SET operations,  $n$  of which are MAKE-SET operations, takes*

$$O(m + n \lg n)$$

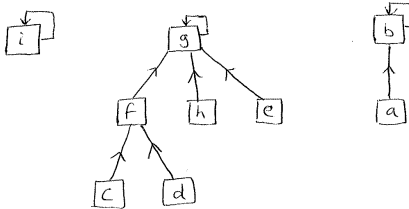
*time.*

**Crucial Proof Idea:** Each element can appear at most  $\lg n$  times in the shorter list of a UNION. (because if  $x$  is in the shortest list, the UNION operation **doubles** the size of  $x$ 's list).

*ADS: lects 10, 11, 12 – slide 34 – 28th Oct, 1st & 4th Nov, 2011*

# The Forest Implementation of Disjoint-Sets

Each set represented by a rooted tree:



## Basic Operations

**MAKE-SET:** Constant time.

**FIND-SET:** Follow pointers to root. (Path followed is called the *find path*.) Cost proportional to height of tree.

**UNION:** Naive strategy: root of tree of  $x$  made to point to that of  $y$ . Cost proportional to height of  $x$ 's tree plus the height of  $y$ 's tree.

Not faster than linked list implementation.

# Improving the Running Time

## General Strategy

Keep trees low.

## Two Heuristics

1. *Union-by-Rank*: Attach lower tree to the root of higher tree in UNION
2. *Path Compression*: Update tree during FIND-SET operations.

## The Union-by-Rank Heuristic

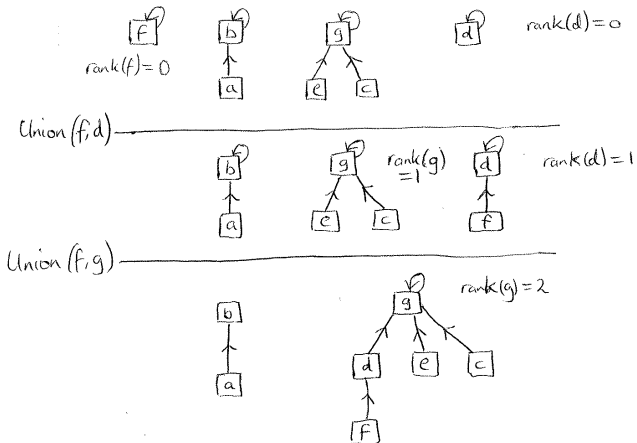
At each node  $x$  maintain a variable  $rank[x]$ , such that if  $x$  is the root of its tree,  $rank[x]$  is the height of this tree.

In executing

UNION( $x, y$ )

make the tree of smaller rank point to the one with larger rank. (Break ties arbitrarily.)

# Union-by-Rank (Union(f,d), then Union(f,g))



## The Height of the Trees

### Lemma 5

The height of a tree (in our datastructure) is at most

$$\lg(\text{size of the tree}).$$

**Proof** By induction on the number of UNION operations it is proved that a tree of height  $h$  has size at least  $2^h$ .

- ▶ As long as there are no UNIONS, all trees have height 0 and contain  $1 = 2^0$  node.
- ▶ Suppose UNION( $x, y$ ) is executed. Let  $r_x$  and  $r_y$  be the roots of the trees of  $x$  and  $y$ , resp.

Case 1:  $\text{rank}(r_x) < \text{rank}(r_y)$ .

Then  $r_x$  becomes child of  $r_y$ , and the height remains unchanged, but the number of nodes increases.

Case 2:  $\text{rank}(r_x) > \text{rank}(r_y)$ . Analogously.

Case 3:  $\text{rank}(r_x) = \text{rank}(r_y)$ .

Then height increase by one and the size of the new tree is

$$\text{size}(r_x) + \text{size}(r_y) \geq 2^{\text{rank}(r_x)} + 2^{\text{rank}(r_y)} = 2^{\text{rank}(r_x)+1},$$

which is the height of the new tree.

## Running Time with Union-by-Rank Only

MAKE-SET: constant time.

FIND-SET: Bounded by rank:  $O(\log n)$ .

UNION: Bounded by rank:  $O(\log n)$ .

Bottom line ( $m$  operations, of which  $n$  are MAKE-SET):

$$O(m \log n).$$

*NOT any better than Linked-Lists with Weighted union heuristic.*

# The Path-Compression Heuristics

## Idea

When performing

$\text{FIND-SET}(x)$

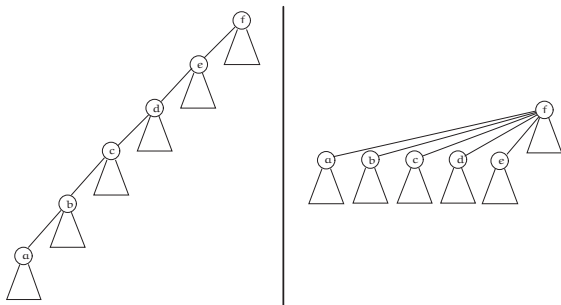
make each vertex on find path point to root.

## Implementation

**Algorithm**  $\text{FIND-SET}(x)$

1. **if**  $x \neq \pi(x)$  **then**
2.      $\pi(x) \leftarrow \text{FIND-SET}(\pi(x))$
3. **return**  $\pi(x)$

## Example



FIND-SET(*a*)

# The Ackermann Function

## Ackerman's Function

Function  $A : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$  defined by

$$\begin{aligned}A(1, j) &= 2^j && \text{for } j \geq 1 \\A(i, 1) &= A(i-1, 2) && \text{for } i \geq 2 \\A(i, j) &= A(i-1, A(i, j-1)) && \text{for } i, j \geq 2\end{aligned}$$

Other variants exist—last line of definition is crucial common point.

## Inverse

Not true mathematical inverse—grows as *slowly* as  $A(i, j)$  grows *fast*:

$$\alpha(m, n) = \min\{i \geq 1 \mid A(i, \lfloor m/n \rfloor) > \lg n\}.$$

## Small table of $A(i, j)$

	$j = 1$	$j = 2$	$j = 3$	$j = 4$
$i = 1$	$2^1$	$2^2$	$2^3$	$2^4$
$i = 2$	$2^2$	$2^{2^2}$	$2^{2^{2^2}}$	$2^{2^{2^{2^2}}}$
$i = 3$	$2^{2^2}$	$2^{2^{\dots^2}} \left. \vphantom{2^{2^{\dots^2}}} \right\} 16$	$2^{2^{\dots^2}} \left. \vphantom{2^{2^{\dots^2}}} \right\} 2^{\dots^2} \left. \vphantom{2^{2^{\dots^2}}} \right\} 16$	$2^{2^{\dots^2}} \left. \vphantom{2^{2^{\dots^2}}} \right\} 2^{\dots^2} \left. \vphantom{2^{2^{\dots^2}}} \right\} 2^{\dots^2} \left. \vphantom{2^{2^{\dots^2}}} \right\} 16$

**Historical importance of  $A(i, j)$ :** Showed that the class of *primitive recursive functions* does not include all *computable functions*.

$A(i, j)$  grows faster than any primitive recursive function.

## Analysis of Disjoint Forests with both Heuristics

### Theorem 6

*Using both the union-by-rank and path compression heuristic, the worst-case runtime for disjoint forests is*

$$O(m \alpha(m, n)).$$

### Remark

Ackermann's Function grows really really fast, so the inverse  $\alpha$  grows really really slowly.

For all practical purposes  $\alpha(m, n) \leq 4$ .

## Reading (Kruskal)

[CLRS] Chapter 21 (pp. 498–522) or [CLR] Chapter 22 (pp. 440–461)

[CLRS] Chapter 23 (pages 561–579). *This is Chapter 24 (pages 498–513) of [CLR].*

Wikipedia on Kruskal's Algorithm

[http://en.wikipedia.org/wiki/Kruskal's\\_algorithm](http://en.wikipedia.org/wiki/Kruskal's_algorithm)

## Problems (Kruskal)

1. Exercise 23.2-1, page 573 of [CLRS] *Ex. 24.2-1, pg. 510 of [CLR]*.
2. Suppose that all edge weights in a graph  $G$  are integers in the range 1 to  $|V|$ . How fast can you make Kruskal's algorithm run in this case?

What if the edge weights are integers in the range from 1 to  $C$ , for some constant  $C$ ?

*This is Exercise 23.2-4, page 574 of [CLRS] Ex. 24.2-4, pg. 510 of [CLR].*

3. Exercise 21.2-2, page 504 of [CLRS]. *22.2-2, page 446 of [CLR]*.
4. Exercise 21.3-1, page 509 of [CLRS]. *22.3-1, page 446 of [CLR]*.