Chapter 5

Data Structures

Algorithm Theory
WS 2017/18

Fabian Kuhn

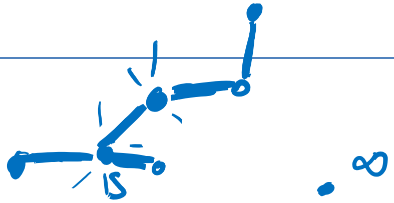
Summary: Binary and Fibonacci Heaps

	Binary Heap	**Fibonacci Heap**
initialize	$O(1)$	$O(1)$
insert	$O(\log n)$	$O(1)$
get-min	$O(1)$	$O(1)$
delete-min	$O(\log n)$	$O(\log n)$ *
decrease-key	$O(\log n)$	$O(1)$ *
merge	$O(m \cdot \log n)$	$O(1)$
is-empty	$O(1)$	$O(1)$

Dijkstra: $O(|E| + |V| \log |V|)$

* **amortized time**

Minimum Spanning Trees



Prim Algorithm:

(S)

1. Start with any node v (v is the initial component)
2. In each step:
   Grow the current component by adding the minimum weight edge e connecting the current component with any other node

Kruskal Algorithm:

1. Start with an empty edge set
2. In each step:
   Add minimum weight edge e such that e does not close a cycle

Implementation of Prim Algorithm

Start at node s, very similar to Dijkstra's algorithm:

1. Initialize $\underline{d(s) = 0}$ and $\underline{d(v) = \infty}$ for all $v \neq s$
2. All nodes $s \geq v$ are unmarked

add all nodes to an empty priority queue Q (d(v): key)

3. Get unmarked node u which minimizes $d(u)$:

get-min $\rightarrow$ u

Dijkstra: $d(u) + w(e)$

4. For all $e = \{u, v\} \in E$, $d(v) = \underbrace{\min\{d(v), w(e)\}}$

potentially update d(v) for all neighbors of u

5. mark node u

delete-min

6. Until all nodes are marked

Implementation of Prim Algorithm

Implementation with Fibonacci heap:

n nodes
m edges

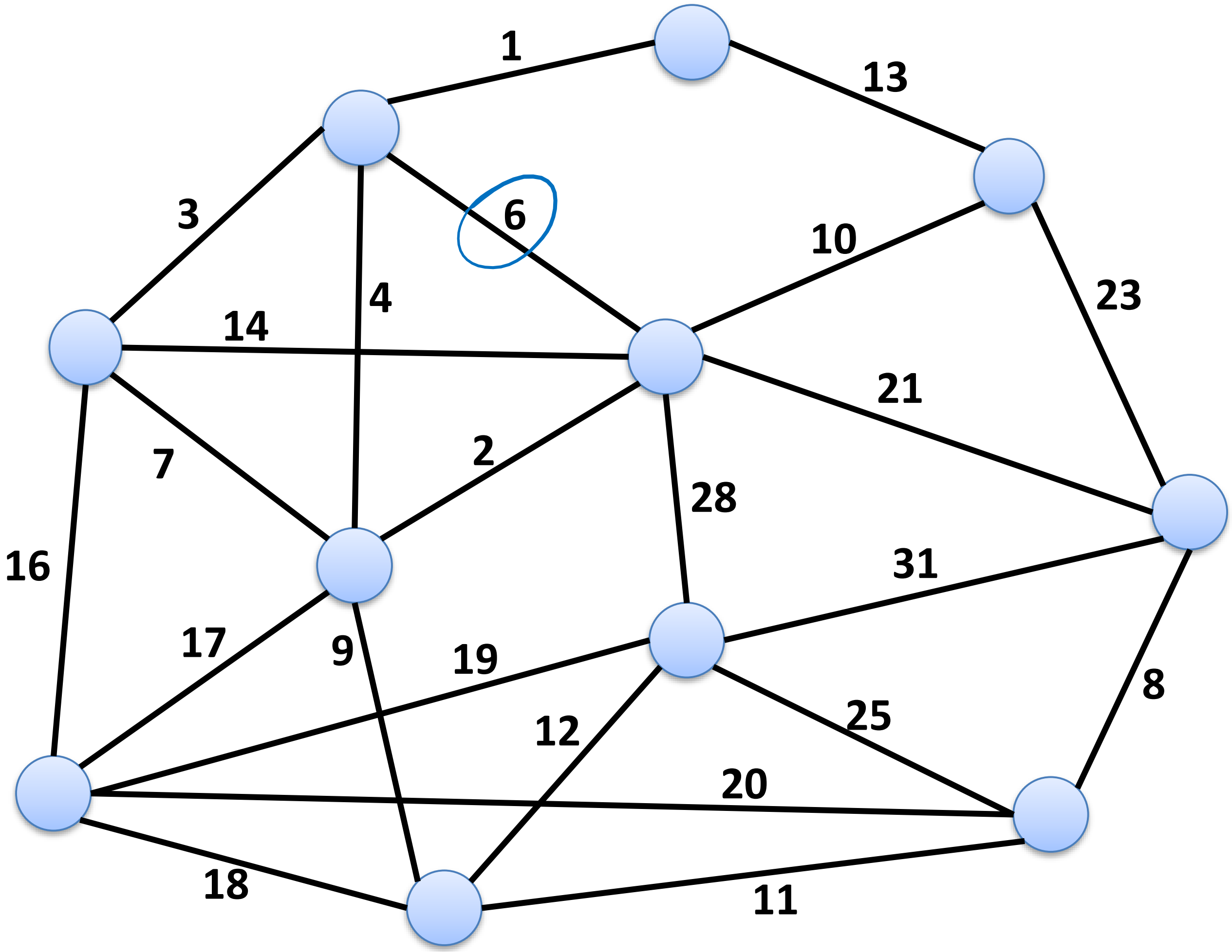
- Analysis identical to the analysis of Dijkstra's algorithm:

  $O(n)$ insert and delete-min operations

  $O(m)$ decrease-key operations

- Running time: $\boldsymbol{O(m + n\log n)}$

Kruskal Algorithm



1. Start with an empty edge set
2. In each step: Add minimum weight edge *e* such that *e* does *not* close a cycle

Implementation of Kruskal Algorithm

1. Go through edges in order of increasing weights

   sort edges by weight $O(m \log n)$ (if weights are nice, this might be faster)

2. For each edge e: $(e = \{u, v\})$

   if *e* does not close a cycle then

   need to be able to check whether e closes a cycle

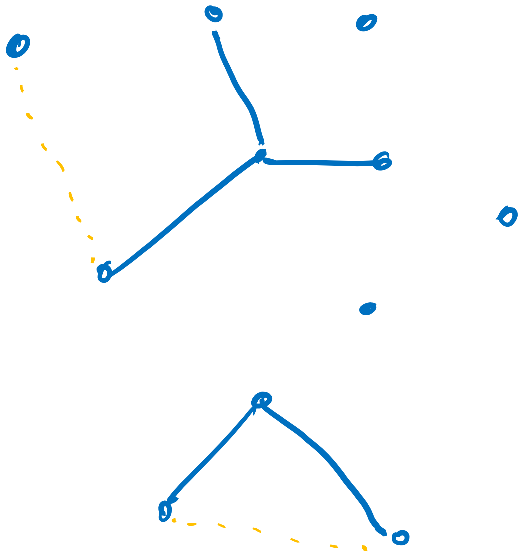
   $\Longleftrightarrow$

   check whether u & v are in the same component

   add *e* to the current solution

   add $\{u, v\}$

   need to merge components of u & v

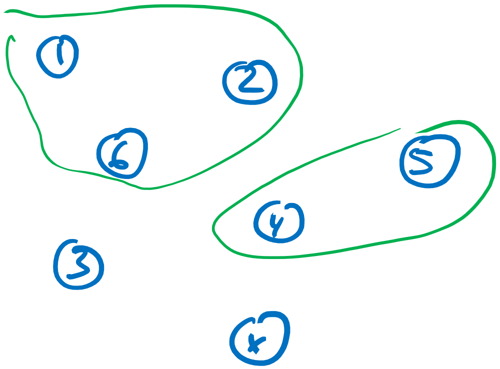


Union-Find Data Structure

Also known as **Disjoint-Set Data Structure**…

Manages partition of a set of elements

- set of disjoint sets



Operations:

- **make_set(x):** create a new set that only contains element x
- **find(x):** return the set containing x
- **union(x, y):** merge the two sets containing x and y

Implementation of Kruskal Algorithm

1. Initialization:
   For each node v: $\mathrm{make\_set}(v)$

2. Go through edges in order of increasing weights:
   Sort edges by edge weight

3. For each edge $e = \{u, v\}$:

```
if $\mathbf{find}(\boldsymbol{u}) \neq \mathbf{find}(\boldsymbol{v})$ then
    add $e$ to the current solution
    $\underline{\mathbf{union}(\boldsymbol{u}, \boldsymbol{v})}$
```

Managing Connected Components

- Union-find data structure can be used more generally to manage the connected components of a graph

  … if edges are added incrementally

- $\mathrm{make\_set}(v)$ for every node v
- $\mathrm{find}(v)$ returns component containing v
- $\mathrm{union}(u, v)$ merges the components of u and v
  (when an edge is added between the components)

- Can also be used to manage biconnected components

Basic Implementation Properties

Representation of sets:

- Every set S of the partition is identified with a representative, by one of its members $x \in S$

Operations:

- make_set(x): x is the representative of the new set $\{x\}$
- find(x): return representative of set S_x containing x
- union(x, y): unites the sets S_x and S_y containing x and y and returns the new representative of $S_x \cup S_y$

Observations

Throughout the discussion of union-find: f: # find ops

- $\underline{n}$: total number of make_set operations
- $\underline{m}$: total number of operations (make_set, find, and union)

Clearly:

- $m \geq n$ (exactly n make-set ops)
- There are at most $n - 1$ union operations

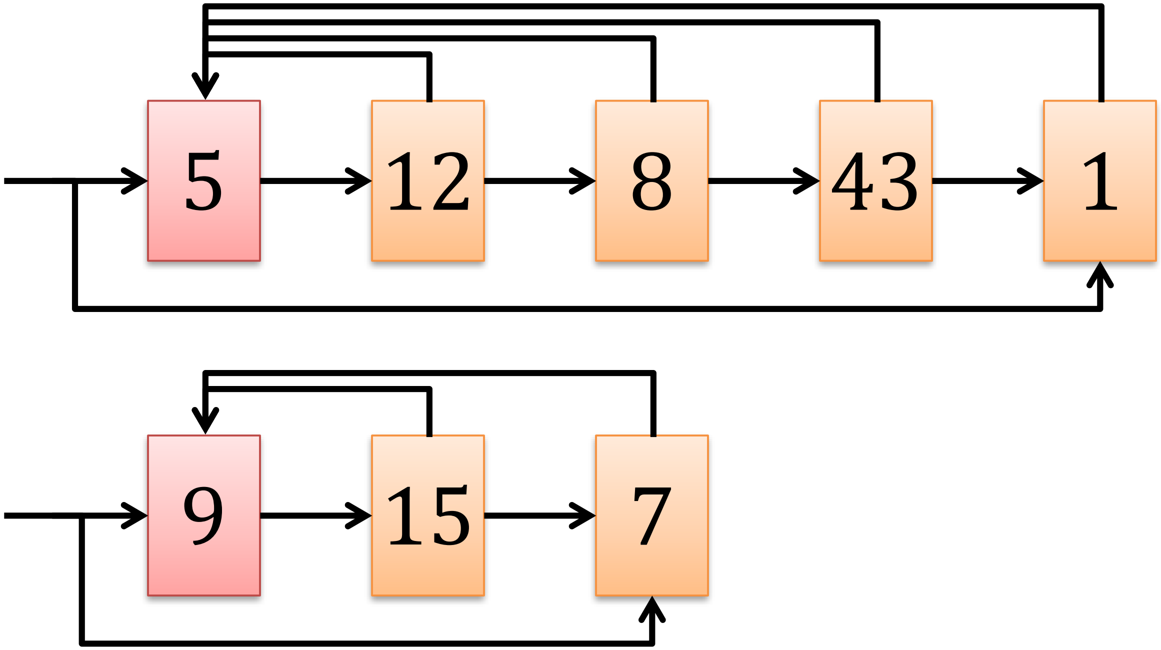
Remark:

- We assume that the n make_set operations are the first n operations
  - Does not really matter…

Linked List Implementation

Each set is implemented as a linked list:

- representative: first list element (all nodes point to first elem.)
  in addition: pointer to first and last element ↙ makes find cheap



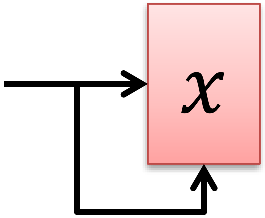
- sets: $\{1,5,8,12,43\}, \{7,9,15\}$; representatives: $5, 9$

Linked List Implementation

make_set(x):

- Create list with one element:

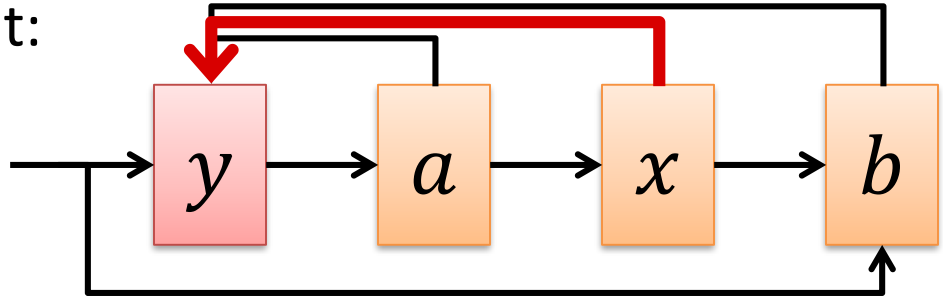
time: $O(1)$



find(x):

- Return first list element:

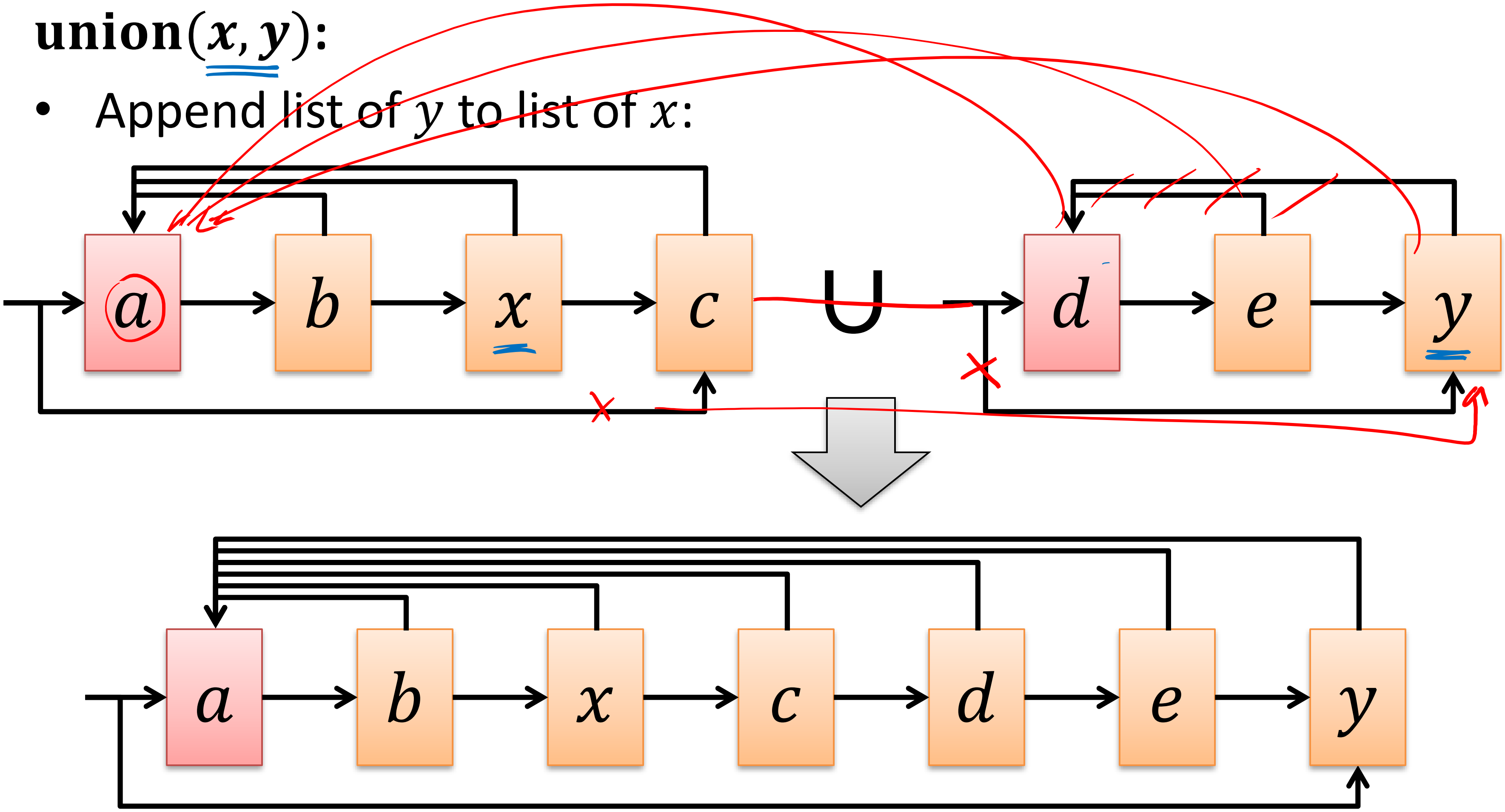
time: $O(1)$



Linked List Implementation

union(x, y):

- Append list of y to list of x:



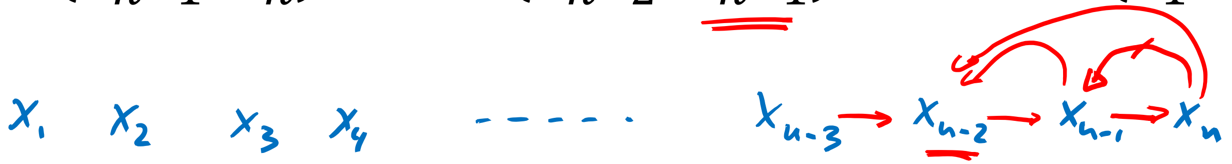
cost: # pointer redirections = O(length of 2^{nd} list)

Time: O(length of list of y)

Cost of Union (Linked List Implementation)

Total cost for $n-1$ union operations can be $\Theta(n^2)$:

- $\text{make\_set}(x_1), \text{make\_set}(x_2), \dots, \text{make\_set}(x_n),$
  $\text{union}(x_{n-1}, x_n), \text{union}(x_{n-2}, x_{n-1}), \dots, \text{union}(x_1, x_2)$



pointer redir. : $1 + 2 + 3 + \dots = \Theta(n^2)$

$\Longrightarrow$ avg. cost per union: $\Theta(n)$

Weighted-Union Heuristic

- In a bad execution, average cost per union can be $\Theta(n)$
- Problem: The longer list is always appended to the shorter one

Idea:

- In each union operation, append shorter list to longer one!

Cost for union of sets S_x and S_y: $O\big(\min\{|S_x|, |S_y|\}\big)$

Theorem: The overall cost of $\underline{m}$ operations of which at most $\underline{n}$ are $\mathrm{make\_set}$ operations is $\underline{\boldsymbol{O(m + n\log n)}}$.

Weighted-Union Heuristic

Theorem: The overall cost of m operations of which at most n are make_set operations is $\boldsymbol{O(\underline{m} + \underline{n \log n})}$.

Proof:

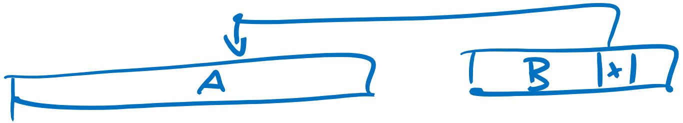
total cost of make-set & find operations : $O(m)$

need to bound total cost of the union operations

= # pointer redirections

consider a fixed element x

How often do we need to redirect the repr. pointer of x



Size of the set containing x at least doubles

$\Longrightarrow \leq \log_2 n$ redir. of repr. pointer of x

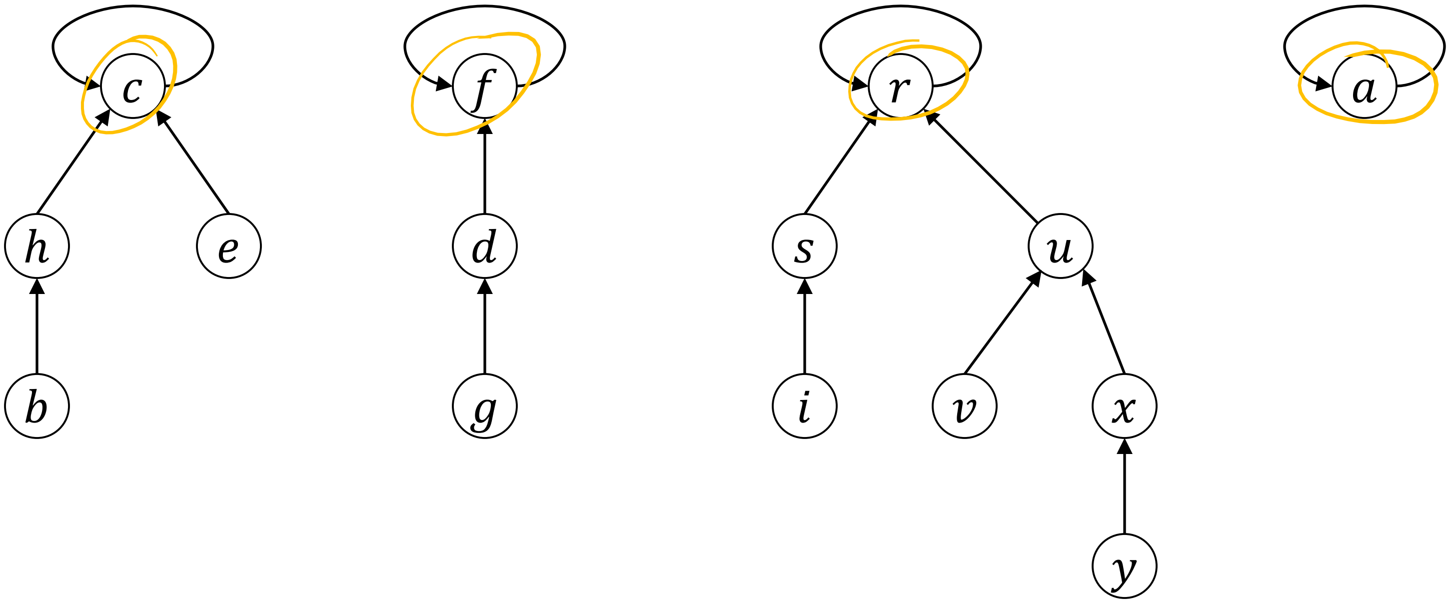
Kruskal's MST alg.

Sorting: $O(m \log n)$

union-find part

$O(m + n \log n)$

Disjoint-Set Forests

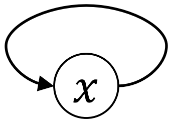


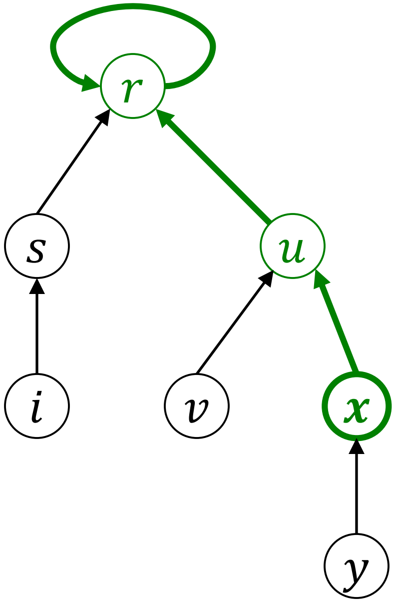
- Represent each set by a tree

- Representative of a set is the root of the tree

Disjoint-Set Forests

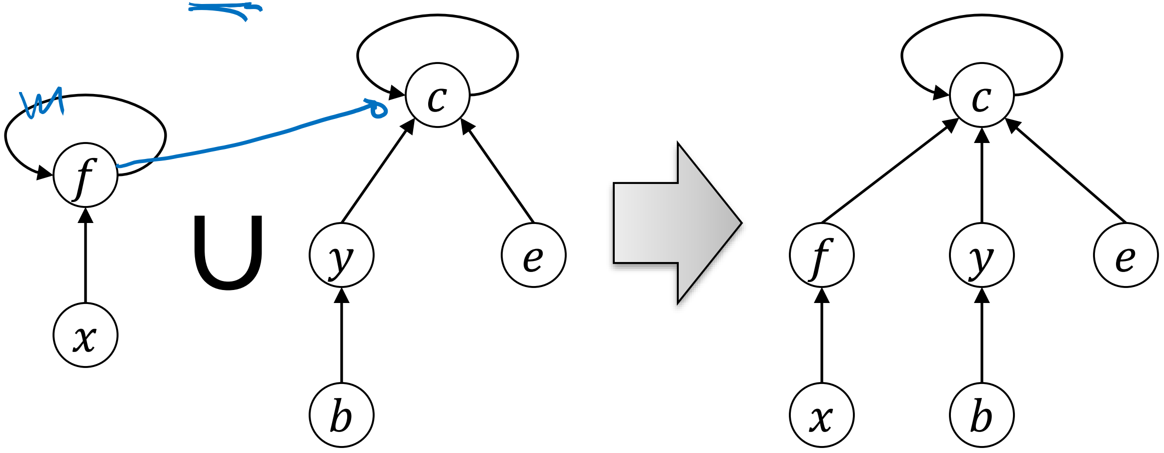
make_set(x): create new one-node tree





find(x): follow parent point to root
(parent pointer to itself)

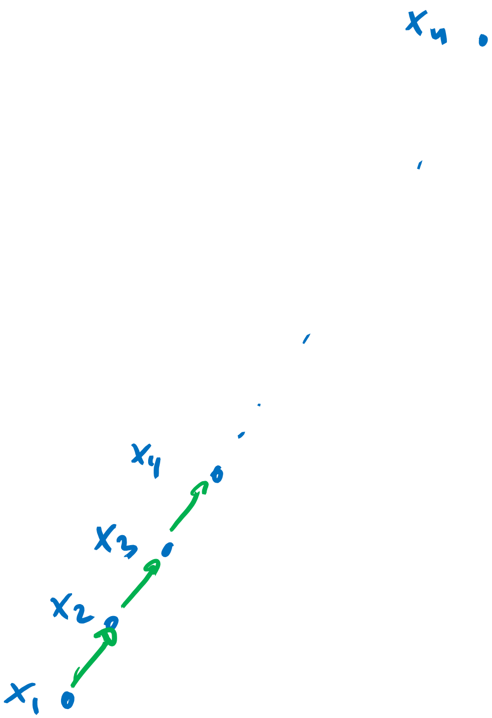
union(x, y): attach tree of x to tree of y



Bad Sequence

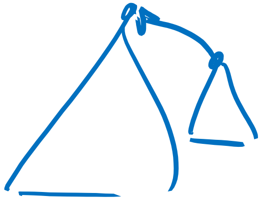
Bad sequence leads to tree(s) of depth $\Theta(n)$

- $\text{make\_set}(x_1), \text{make\_set}(x_2), \dots, \text{make\_set}(x_n),$ $\text{union}(x_1, x_2), \text{union}(\underline{x_1}, x_3), \dots, \text{union}(x_1, x_n)$



Union-By-Size Heuristic

Union of sets S_1 and S_2:



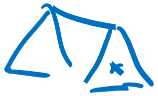
- Root of trees representing S_1 and S_2: r_1 and r_2
- W.l.o.g., assume that $|S_1| \geq |S_2|$
- Root of $S_1 \cup S_2$: r_1 (r_2 is attached to r_1 as a new child)

Theorem: If the union-by-size heuristic is used, the **worst-case cost of a find-operation is $O(\log n)$**

Proof: depth of a tree of size k is at most $\log_2 k$

depth of element x : $d_x \Rightarrow$ size of tree containing x $\geq 2^{d_x}$

$d_x = 0$ ✓ how can d_x grow?



→ size of tree at least doubles

Similar Strategy: union-by-rank

- rank: essentially the depth of a tree

Union-Find Algorithms

Recall: m operations, n of the operations are make_set-operations

Linked List with Weighted Union Heuristic:

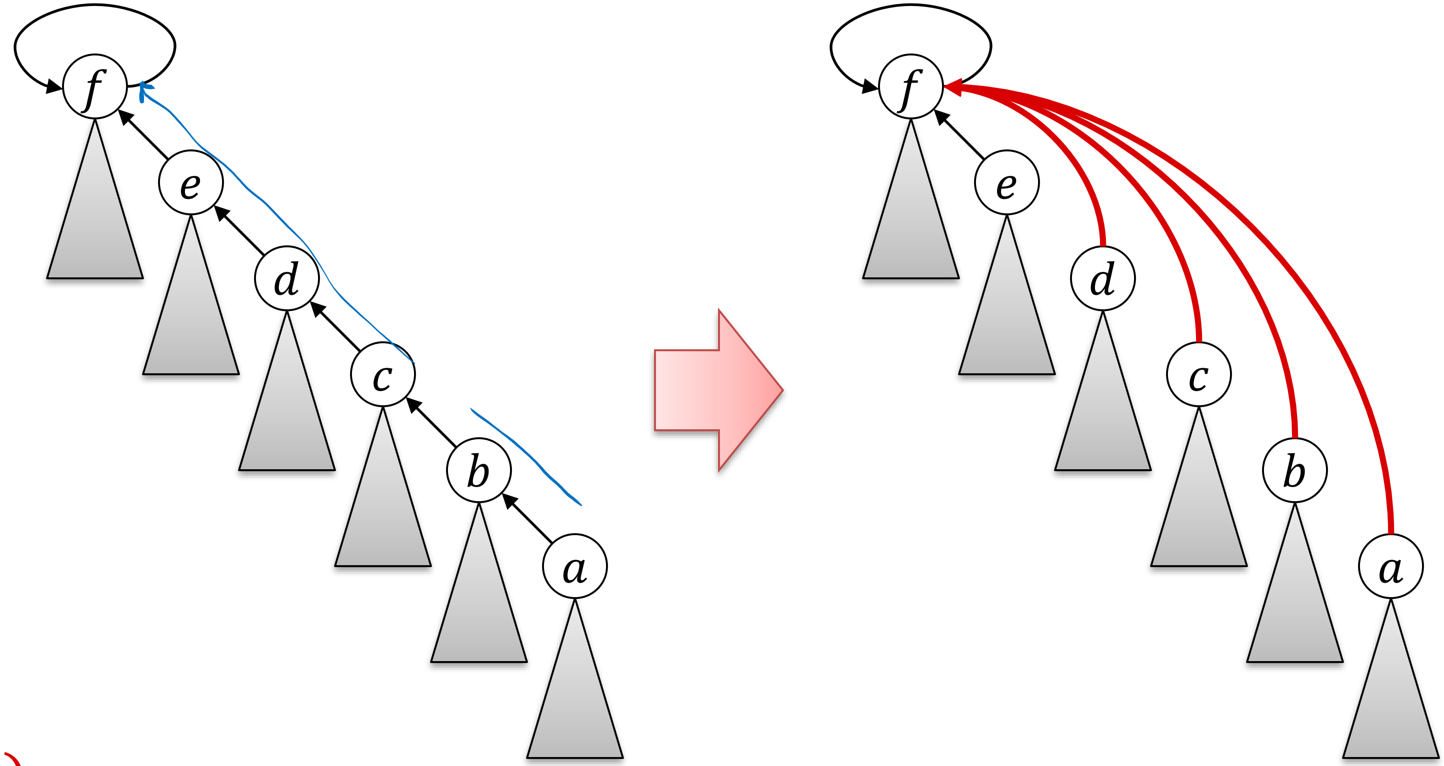
- make_set: worst-case cost $O(1)$
- find : worst-case cost $O(1)$
- union : amortized worst-case cost $O(\log n)$

Disjoint-Set Forest with Union-By-Size Heuristic:

- make_set: worst-case cost $O(1)$
- find : worst-case cost $O(\log n)$
- union : worst-case cost $O(\log n)$

Can we make this faster?

Path Compression During Find Operation



find(a):

```
if $a \neq a.parent$ then
    $a.parent := \text{find}(a.parent)$
return $a.parent$
```

Complexity With Path Compression

When using only path compression (without union-by-rank):

m: total number of operations

- f of which are find-operations
- n of which are make_set-operations

  → at most $n - 1$ are union-operations

$m \gg n$

f large $\rightarrow$ $f \cong m$

Total cost: $$\mathbf{O}\left(m + f \cdot \left\lceil \log_{2+f/n} n \right\rceil\right) = \underbrace{O\left(m + f \cdot \log_{2+m/n} n\right)}_{\text{if } m \gg n}$$

$$O\left(m \cdot \log_{2+m/n} n\right)$$

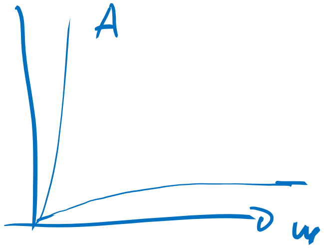
$m = n^{1.1}$

Union-By-Size and Path Compression

Theorem:

Using the combined union-by-rank and path compression heuristic, the running time of m disjoint-set (union-find) operations on n elements (at most n make_set-operations) is

$$\boldsymbol{\Theta(m \cdot \alpha(m,n))},$$



Where $\alpha(m,n)$ is the inverse of the Ackermann function.

grows extremely slowly

grows extremely fast

in practice: $\alpha(m,n) \leq 4$

Kruskal:

sorting: $O(m \log n)$

union-find: $O(m \, \alpha(m,n))$

Ackermann Function and its Inverse

Ackermann Function:

For $k, \ell \geq 1$,

$$A(k,\ell) := \begin{cases} 2^{\ell}, & \text{if } k = 1, \ell \geq 1 \\ A(k-1,2), & \text{if } k > 1, \ell = 1 \\ A\big(k-1, A(k,\ell-1)\big), & \text{if } k > 1, \ell > 1 \end{cases}$$

Inverse of Ackermann Function:

$$\alpha(m,n) := \min\{k \geq 1 \mid \underline{A(k, \lfloor m/n \rfloor}) > \log_2 n\}$$

Inverse of Ackermann Function

- $\alpha(m,n) \coloneqq \min\{k \geq 1 \mid A(k, \lfloor m/n \rfloor) > \log_2 n\}$

$$m \geq n \Longrightarrow A(k, \lfloor m/n \rfloor) \geq A(k,1) \Longrightarrow \alpha(m,n) \leq \min\{k \geq 1 | A(k,1) > \log n\}$$

- $A(1,\ell) = 2^{\ell}, \quad A(k,1) = A(k-1,2),$
  $A(k,\ell) = A\big(k-1, A(k,\ell-1)\big)$

$A(k,1)$

$A(2,1) = A(1,2) = 4$

$A(3,1) = A(2,2) = A(1, A(2,1)) = A(1,4) = 2^4 = 16$

$A(4,1) = A(3,2) = A(2, A(3,1)) = A(2,16) = A(1, A(2,15)) = 2^{A(2,15)}$

$A(2,15) = A(1, A(2,14))$

$\downarrow$

$= 2^{2^{A(2,14)}}$

$= \left.2^{\cdot^{\cdot^{\cdot^{2}}}}\right\} 16$

$10^{80} \approx 2^{250}$