Chapter 5
Data Structures

Algorithm Theory
WS 2019/20

Fabian Kuhn

Minimum Spanning Trees

- **Minimum spanning tree (MST)** problem
  - Classic graph-theoretic optimization problem

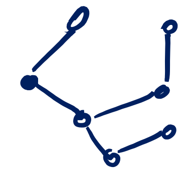
- **Given**: weighted graph
- **Goal**: spanning tree with min. total weight

- Several greedy algorithms work

- Kruskal's algorithm:
  - Start with empty edge set
  - As long as we do not have a spanning tree:
    add minimum weight edge that doesn't close a cycle

Minimum Spanning Trees

Prim Algorithm:



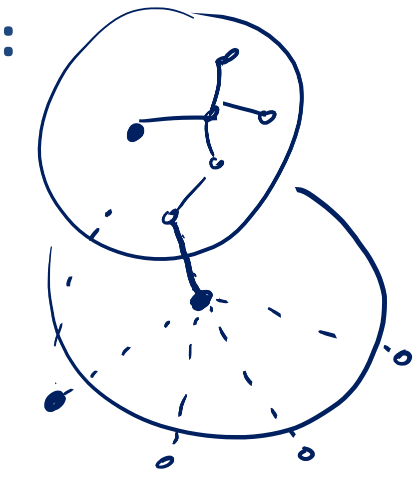
1. Start with any node v (v is the initial component)
2. In each step:
   Grow the current component by adding the minimum weight edge e connecting the current component with any other node

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

   add all nodes to an empty pr. queue Q (with key d(v))

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

   get-min / delete-min

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

   potentially update d(v) of neighbors of u → decrease-key

5. mark node u

   ↳ time (with Fib. heaps) $O(m + n \log n)$

6. Until all nodes are marked

Implementation of Prim Algorithm

Implementation with Fibonacci heap:

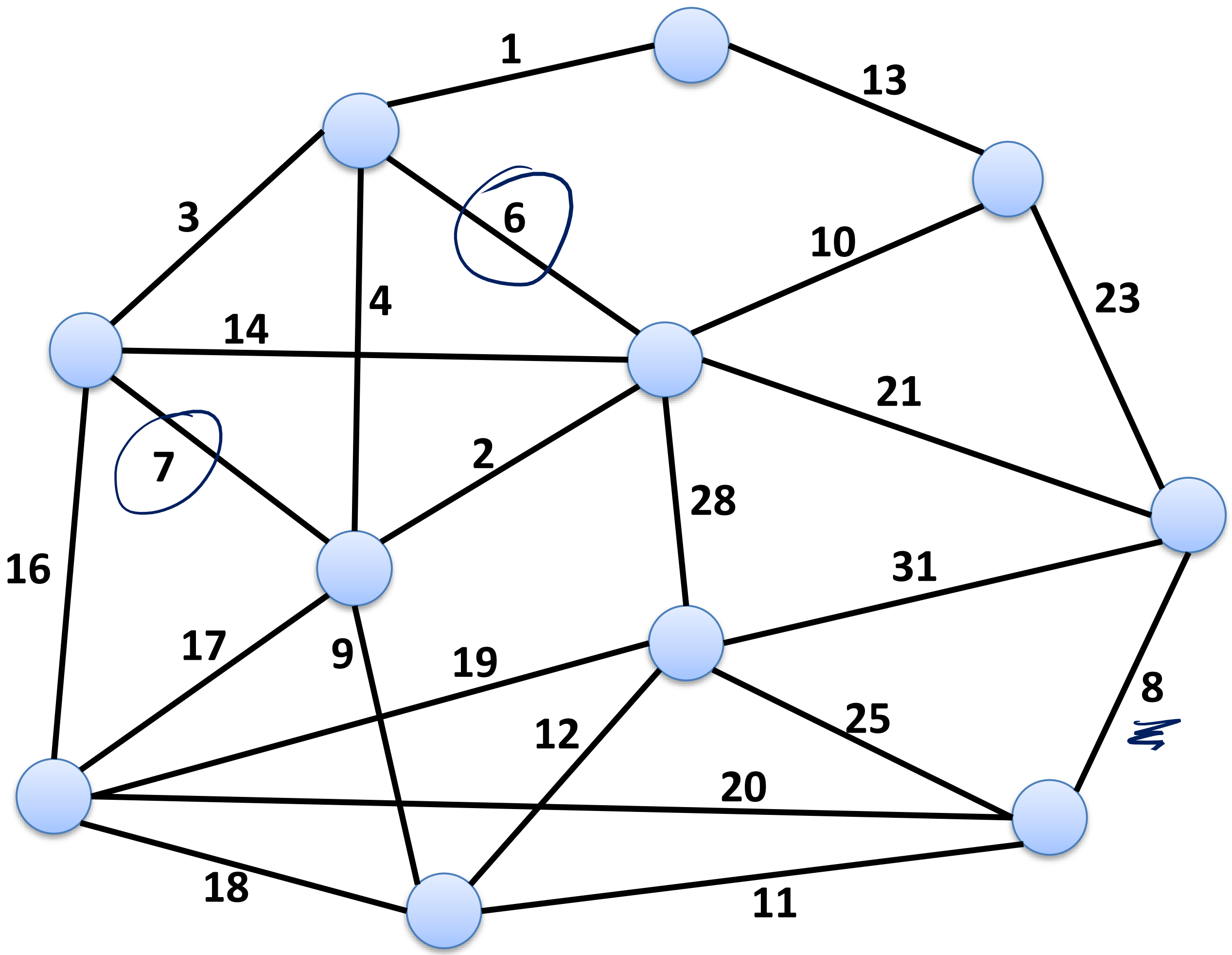
- Analysis identical to the analysis of Dijkstra's algorithm:

  $O(n)$ 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 cheaper)

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

   if ***e*** **does not close a cycle then**

   need to check if e closes a cycle

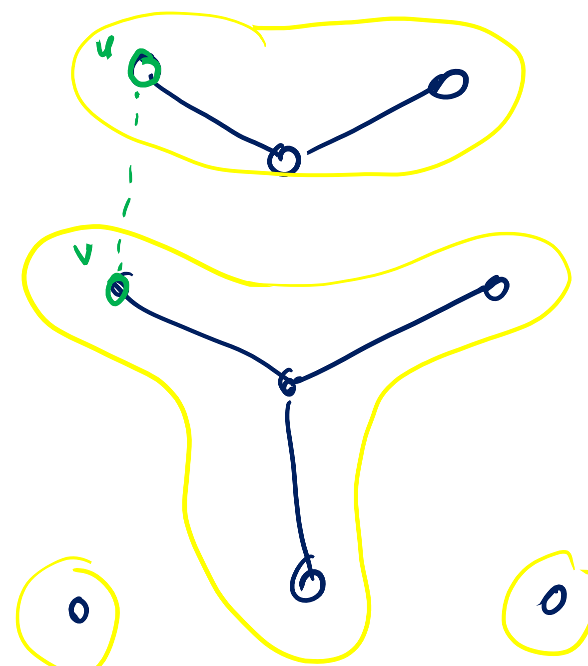
   ⇕

   check if u & v are in the same connected component

   add ***e*** **to the current solution**

   add $\{u,v\}$

   need to merge the 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: $\text{make\_set}(v)$

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

3. For each edge $e = \{u, v\}$:
   if $\mathbf{find}(u) \neq \mathbf{find}(v)$ then

   add e to the current solution

   $\mathbf{union}(u, v)$

\# operations

make-set : $|V|$

find : $2|E|$ ←

union : $|V| - 1$

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

- $\text{make\_set}(v)$ for every node v
- $\text{find}(v)$ returns component containing v
- $\text{union}(u, v)$ 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:

- $\text{make\_set}(x)$: x is the representative of the new set $\{x\}$
- $\text{find}(x)$: return representative of set S_x containing x
- $\text{union}(x, y)$: 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:

- n: total number of make_set operations
- m: total number of operations (make_set, find, and union)

Clearly:

- $m \geq n$
- There are at most $n - 1$ union operations

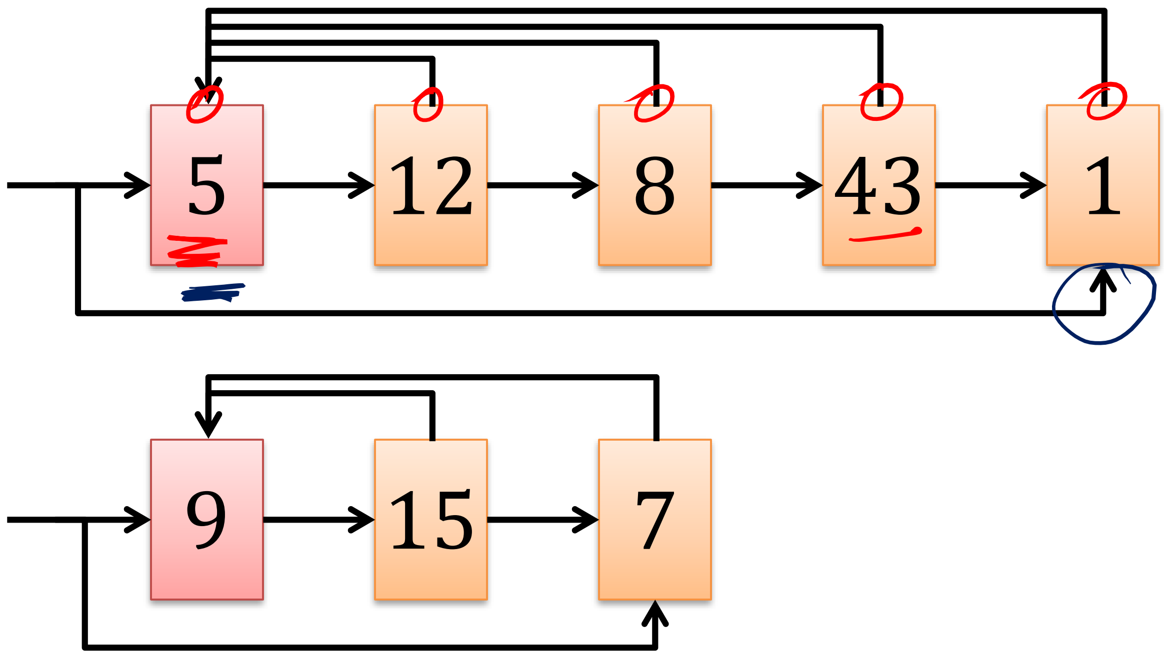
Remark:

- 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



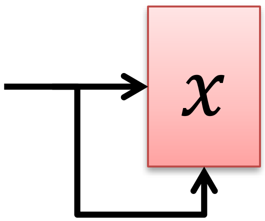
- sets: $\{1,5,8,12,43\}, \{7,9,15\}$; representatives: $5, 9$

Linked List Implementation

make_set(x):

- Create list with one element:

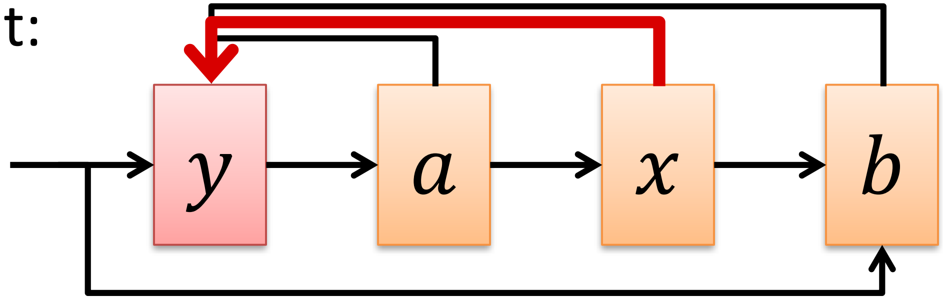
  time: $O(1)$



find(x):

- Return first list element:

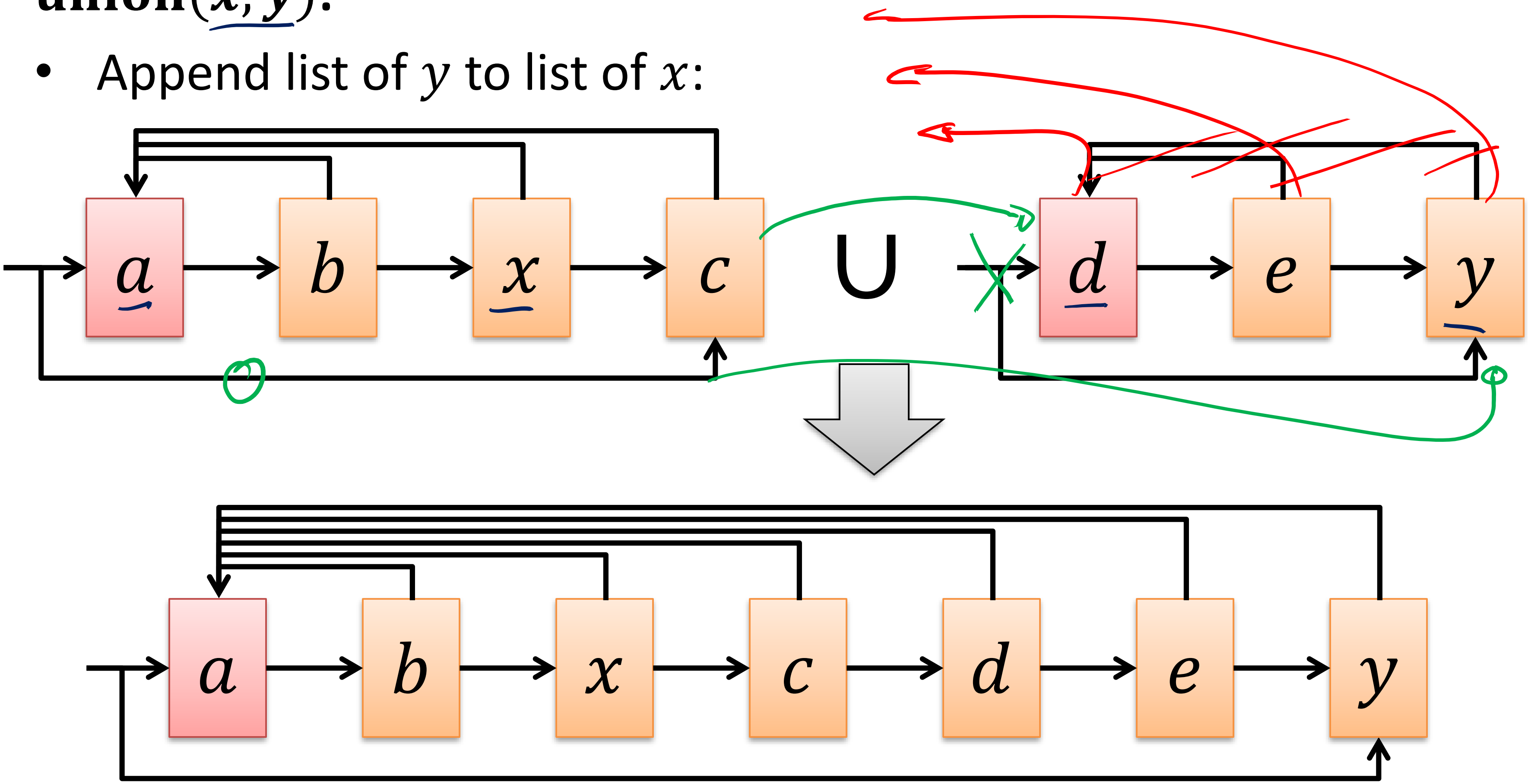
  time: $O(1)$



Linked List Implementation

union(x, y):

- Append list of y to list of x:

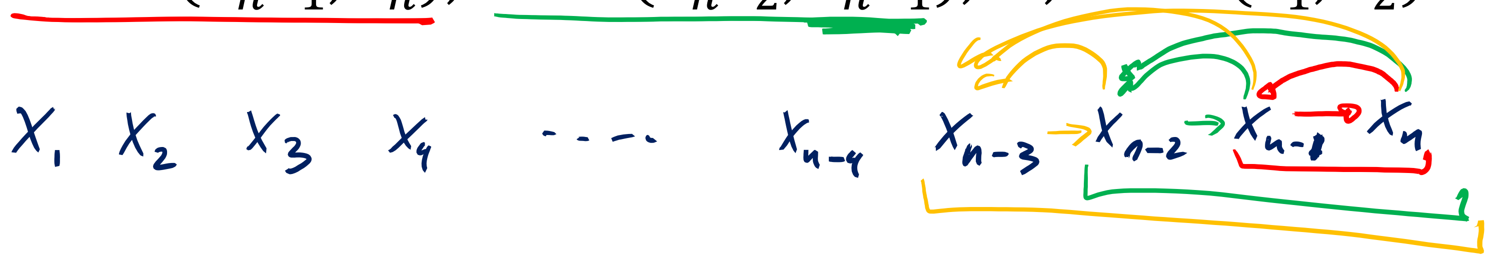


Time: O(length of list of y)

Cost of Union (Linked List Implementation)

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

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



\# operations $1 + 2 + 3 + \dots + n-1 = \Theta(n^2)$

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

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

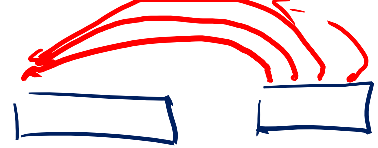
Idea:

- 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 m operations of which at most n are make_set operations is $\boldsymbol{O(m + n\log n)}$.

Weighted-Union Heuristic



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

Proof:

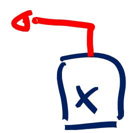
total cost of make-set & find operations: $O(m)$

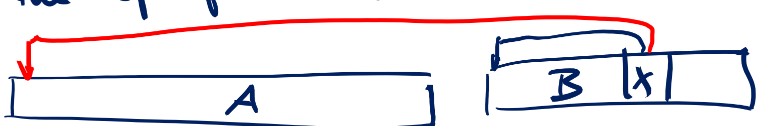
Need to bound the total cost of the union operations

Count # "repr. pointer" redirections

Consider a fixed element x

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





Size of set containing x at least doubles

$\Rightarrow \leq \log_2 n$ redirections for x

$\Rightarrow$ total # repr. ptr. redir $= O(n \log n)$ □

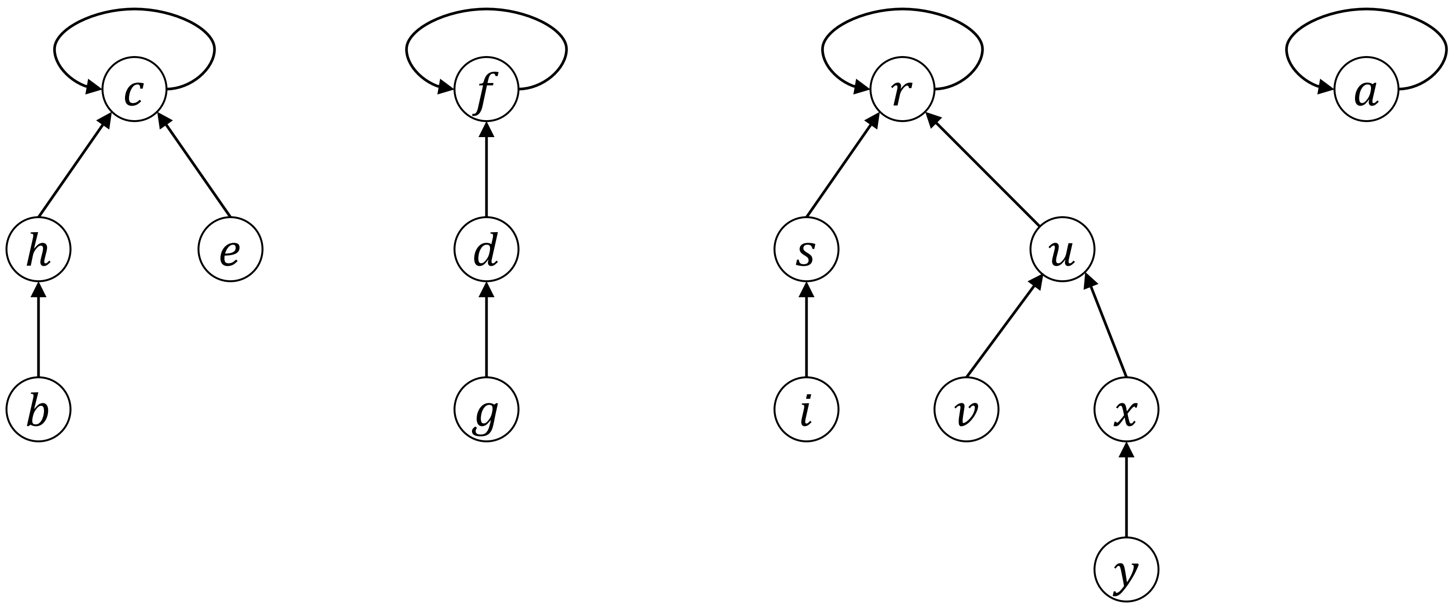
Kruskal's alg

Sorting: $O(m \log n)$

union-find part:

$O(m + n \log n)$

Disjoint-Set Forests

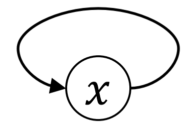


- Represent each set by a tree

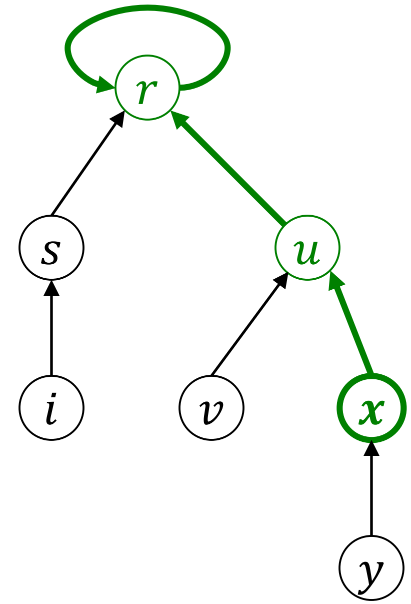
- Representative of a set is the root of the tree

Disjoint-Set Forests

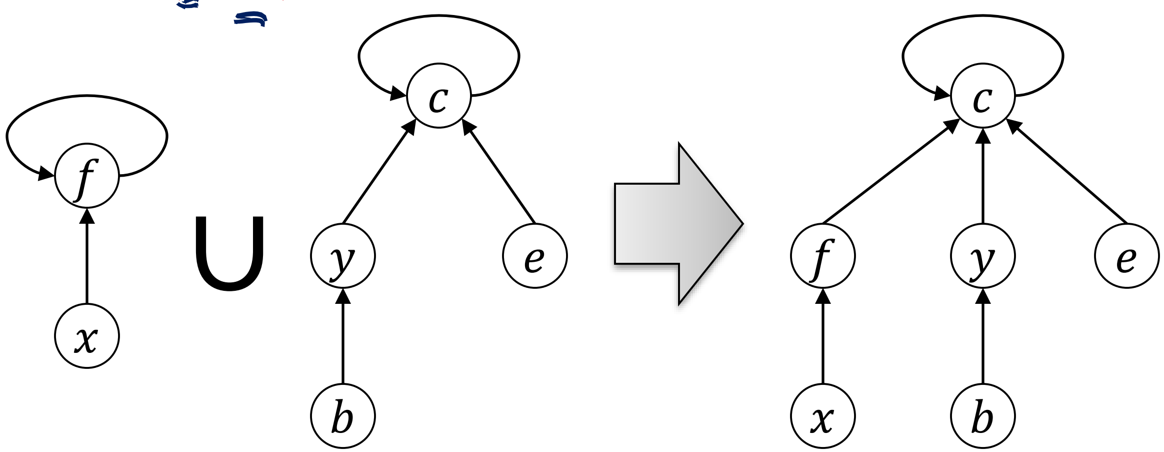
$\mathbf{make\_set(x)}$: create new one-node tree



$\mathbf{find}(x)$: follow parent point to root
(parent pointer to itself)



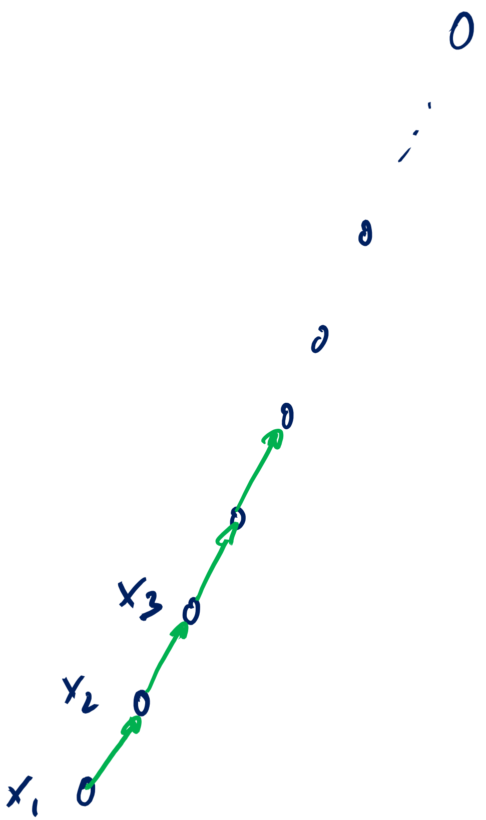
$\mathbf{union}(x, y)$: attach tree of x to tree of y



Bad Sequence

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

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



Union-By-Size Heuristic

Union of sets S_1 and S_2:

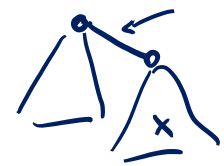
- Root of trees representing S_1 and S_2: r_1 and r_2
- W.l.o.g., assume that $|S_1| \geq |S_2|$
- Root of $S_1 \cup S_2$: r_1 (r_2 is attached to r_1 as a new child)

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

Proof: depth of tree with k nodes is $\leq \log_2 k$

depth of element x is $d_x \Longrightarrow$ size of tree cont. $x \geq 2^{d_x}$

$d_x = 0$ ✓ How can d_x grow?



when d_x grows by 1, the size of the tree cont. x grows by factor ≥ 2

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:

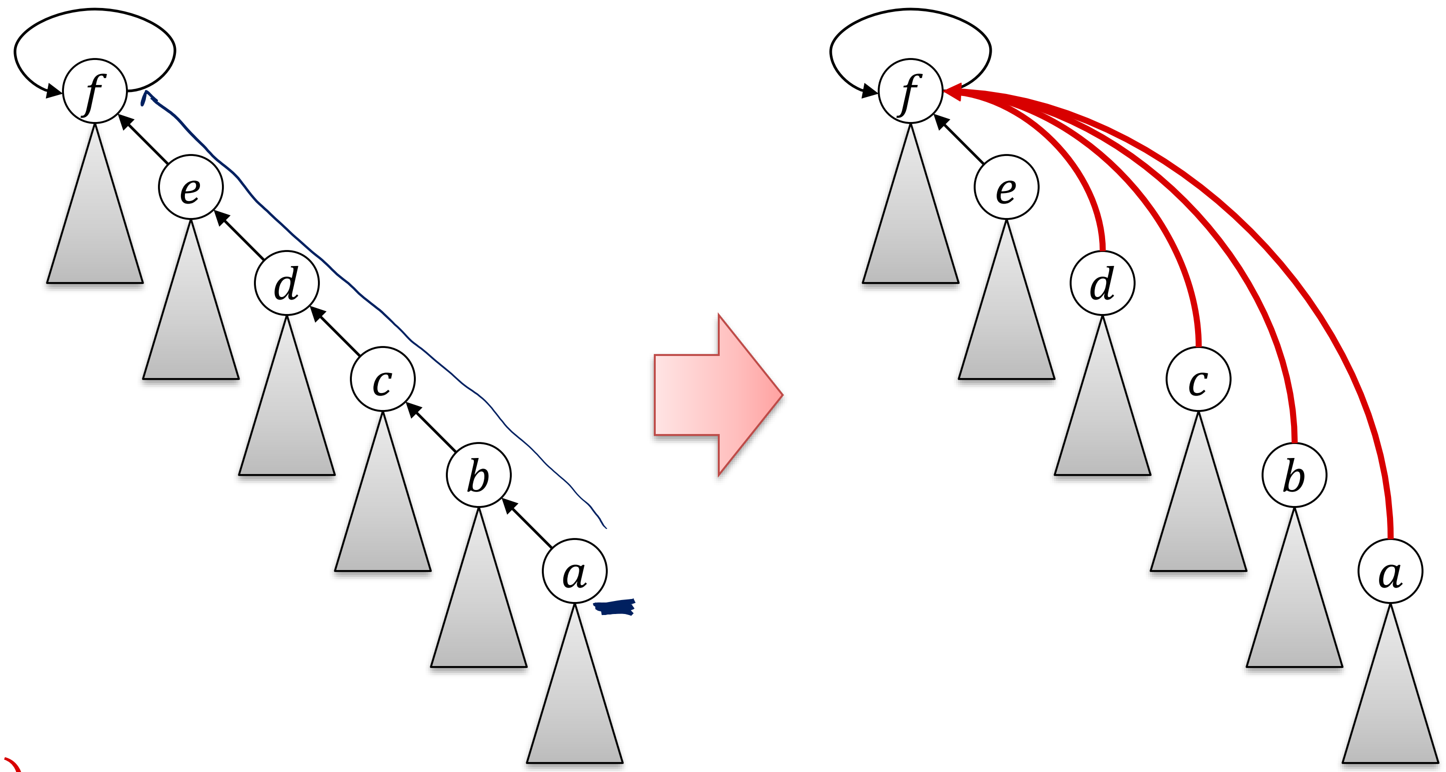
- make_set: worst-case cost $O(1)$
- find : worst-case cost $O(1)$
- union : amortized worst-case cost $O(\log n)$

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):

1. **if** $a \neq a.parent$ **then**
2. $\quad a.parent := \text{find}(a.parent)$
3. **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

$f \geq m-(2n-1)$

$f \leq m-n$

Total cost: $O\left(m + f \cdot \left\lceil \log_{2+f/n} n \right\rceil\right) = O\left(m + f \cdot \log_{2+m/n} n\right)$

if $m \gg n$

if $m \geq n^{1+\varepsilon}$ for some const. $\varepsilon > 0$

$\log_{2+n^{\varepsilon}}(n) = \frac{\log n}{\log n^{\varepsilon}} = \frac{1}{\varepsilon}$

Union-By-~~Size~~ Rank 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

$$\Theta(m \cdot \alpha(m,n)),$$

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

+

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

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:

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

Inverse of Ackermann Function

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

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

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

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

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

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

$A(2,15) = 2^{A(2,14)}$

$= 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}$ } 16

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

$10^{80} = 2^{250}$