Finding Connected Components in Massive Graphs

Robert E. Tarjan
Princeton University

Joint work with Sixue (Cliff) Liu, CMU;
Siddhartha Jayanti, MIT
Thanks to Siddhartha for some slides

DB&IS 4 July 2022

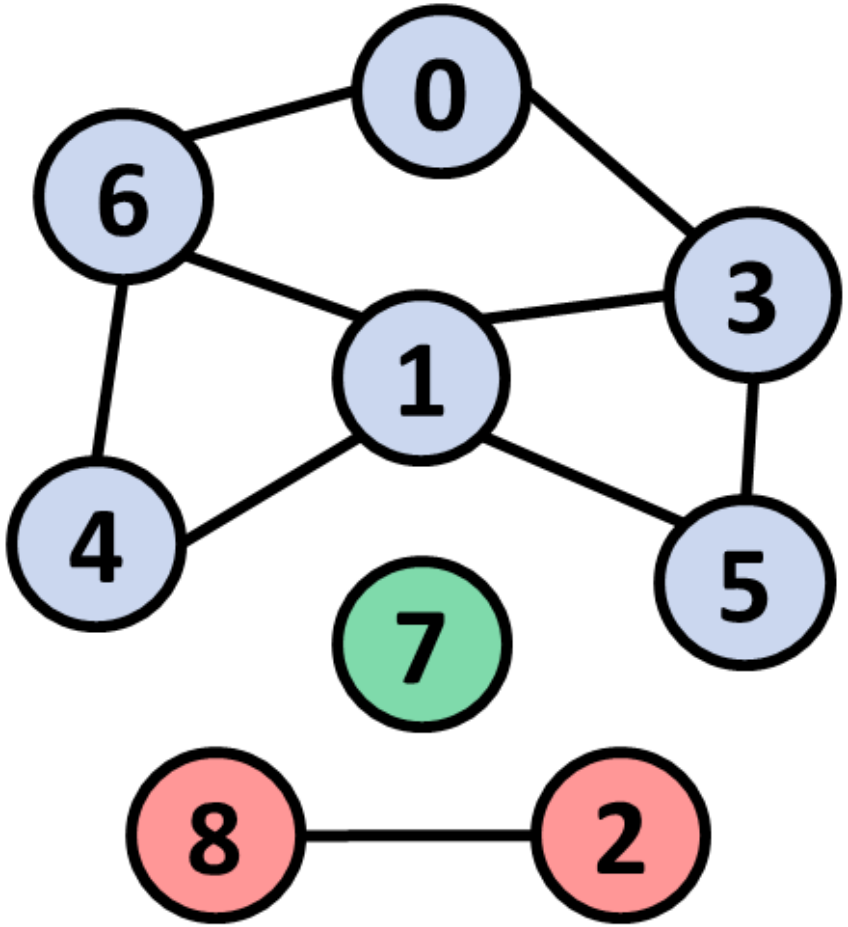
Connected Components

The most basic graph problem?

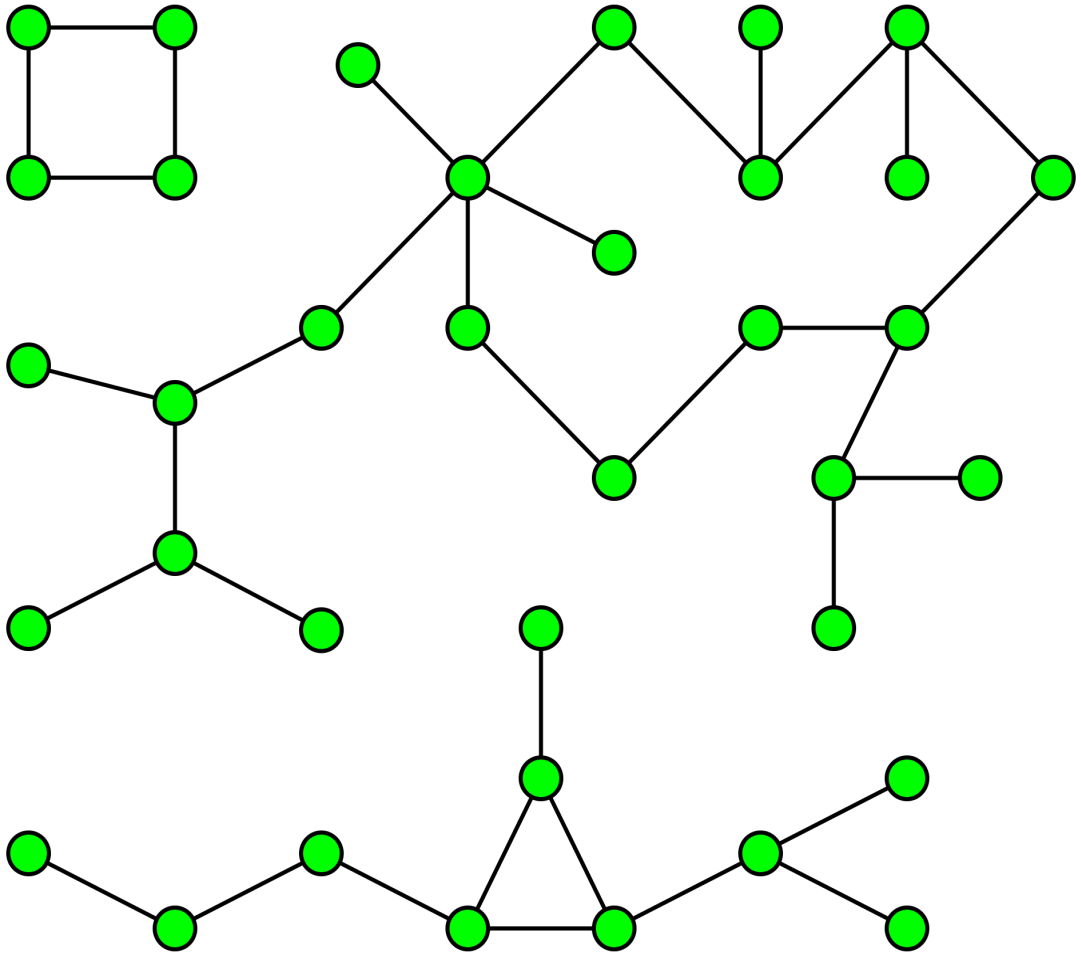
In an undirected graph, two vertices are connected if there is a path between them. A connected component (henceforth just a component) is a maximal set of pairwise-connected vertices.

Problem: Given a graph, compute its components.

[vitoshoacademy.com]

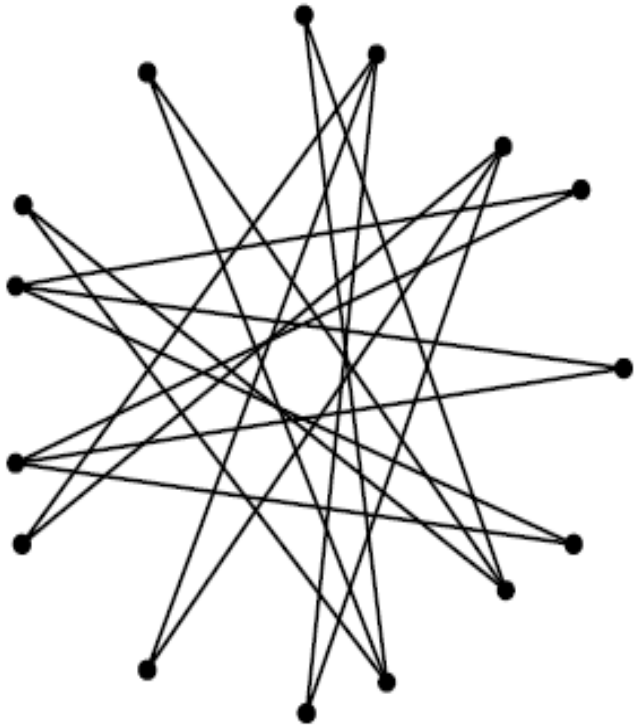


[figure from D. Eppstein]



[math.stackexchange]

A21 Is this graph connected or disconnected?



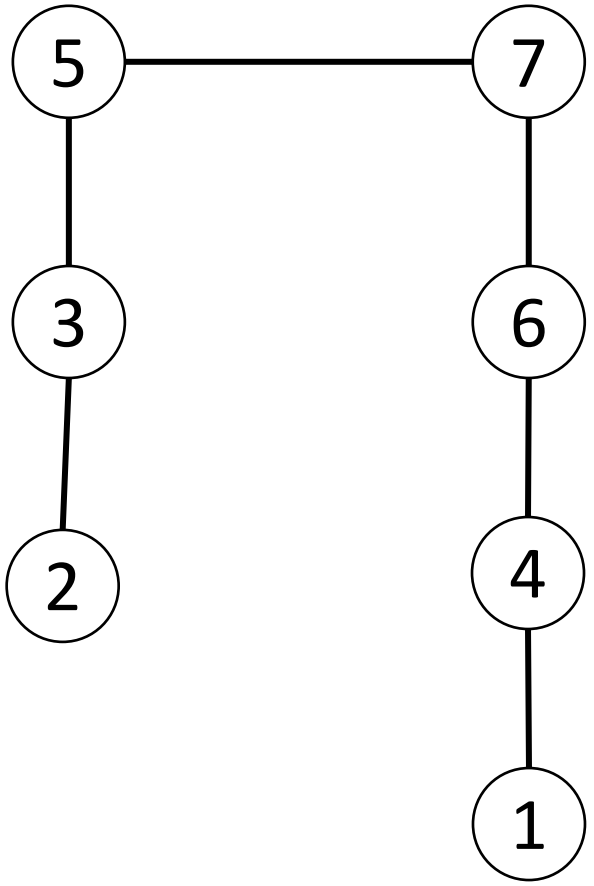
How to represent components?

Label all vertices in each component with a unique vertex in the component: can test if two vertices are in the same component by comparing their labels.

Assume n vertices, 1,…, n; m edges

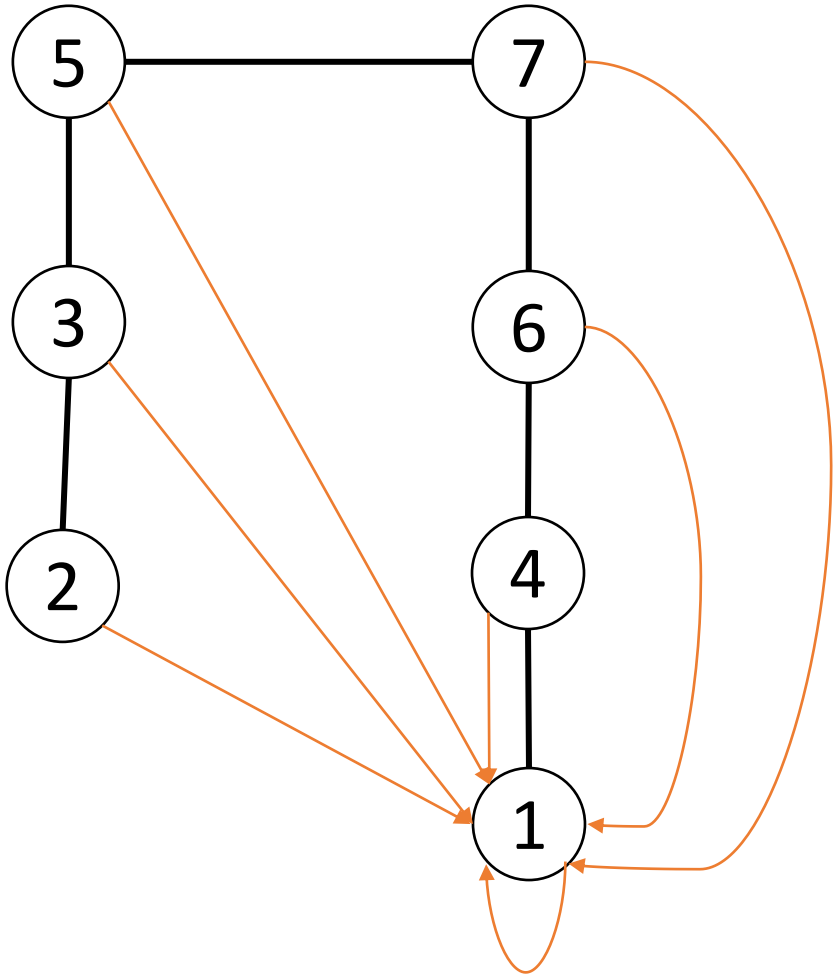
Minimum labeling: Minimum vertex in component.

Minimum labeling



1 1
2 1
3 1
4 1
5 1
6 1
7 1

Minimum labeling



1 1
2 1
3 1
4 1
5 1
6 1
7 1

Classic sequential algorithms

Graph search: breadth-first, depth-first or any other kind of search.

Disjoint set union: Use a disjoint-set (union-find) data structure.

Disjoint set union

Maintain a collection of disjoint sets, each with a unique element called its leader, subject to three operations:

make-set(x) (x in no set): Create a set $\{x\}$ with leader x

find(x): Return the leader of the set containing x

unite(x, y): If x and y are in different sets, unite these sets and choose a leader for the new set

Components via disjoint set union

for each vertex *v* do *make-set*(*v*)

for each edge {*x*, *y*} do *unite*(*x*, *y*)

for each *v* do *v.label* = *find*(*v*)

Need not do the third loop, just use *find* as needed:

v and *w* are in the same component iff

find(*v*) = *find*(*w*)

Running time

Graph search: $O(m + n)$ m = #edges, n = #vertices

Disjoint set union via compressed trees:

$O((m + n)\alpha(n, m/n))$

Disjoint set union uses only the edge set, supports individual and batch edge insertions

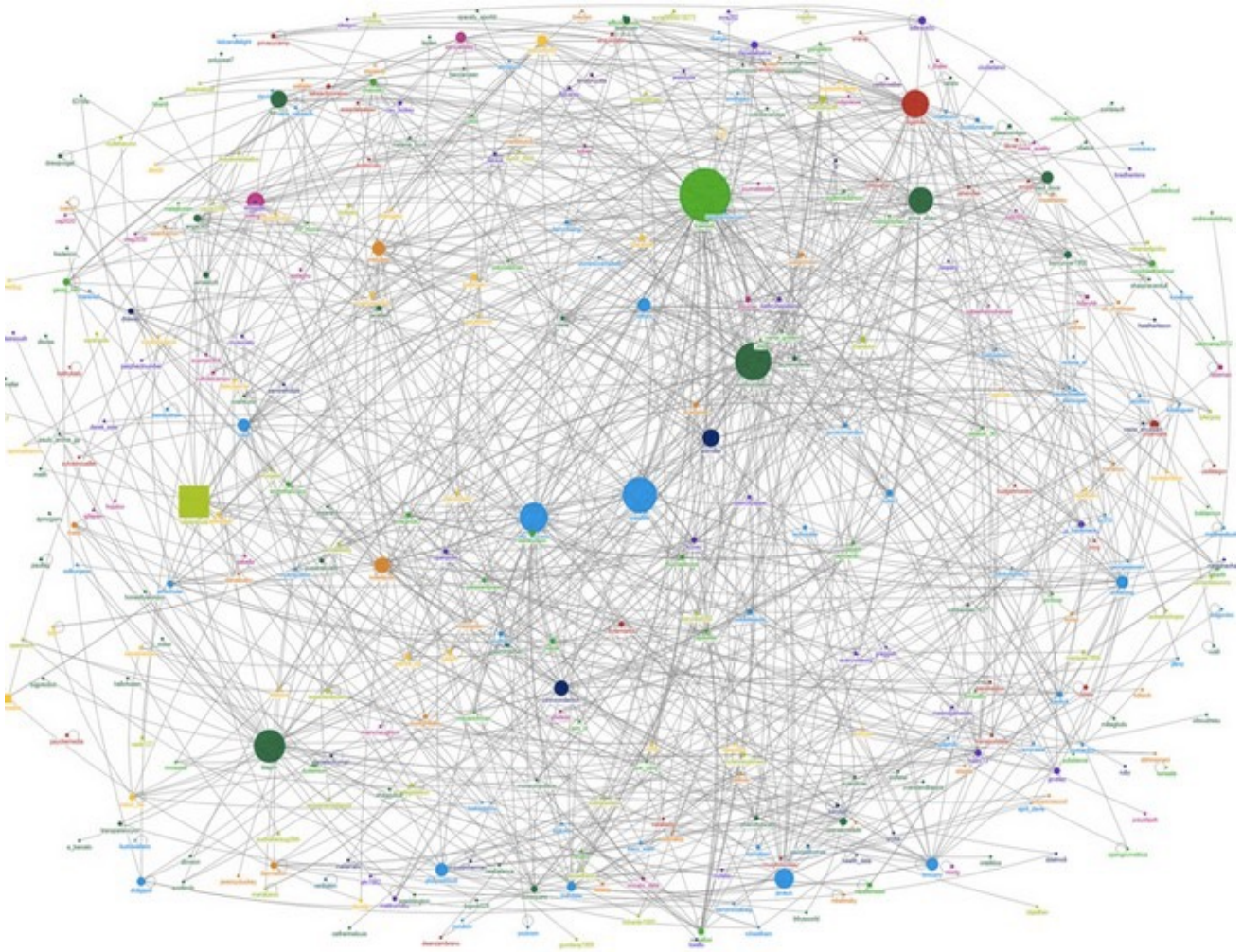
Inverse-Ackermann amortized time per

edge insertion or query

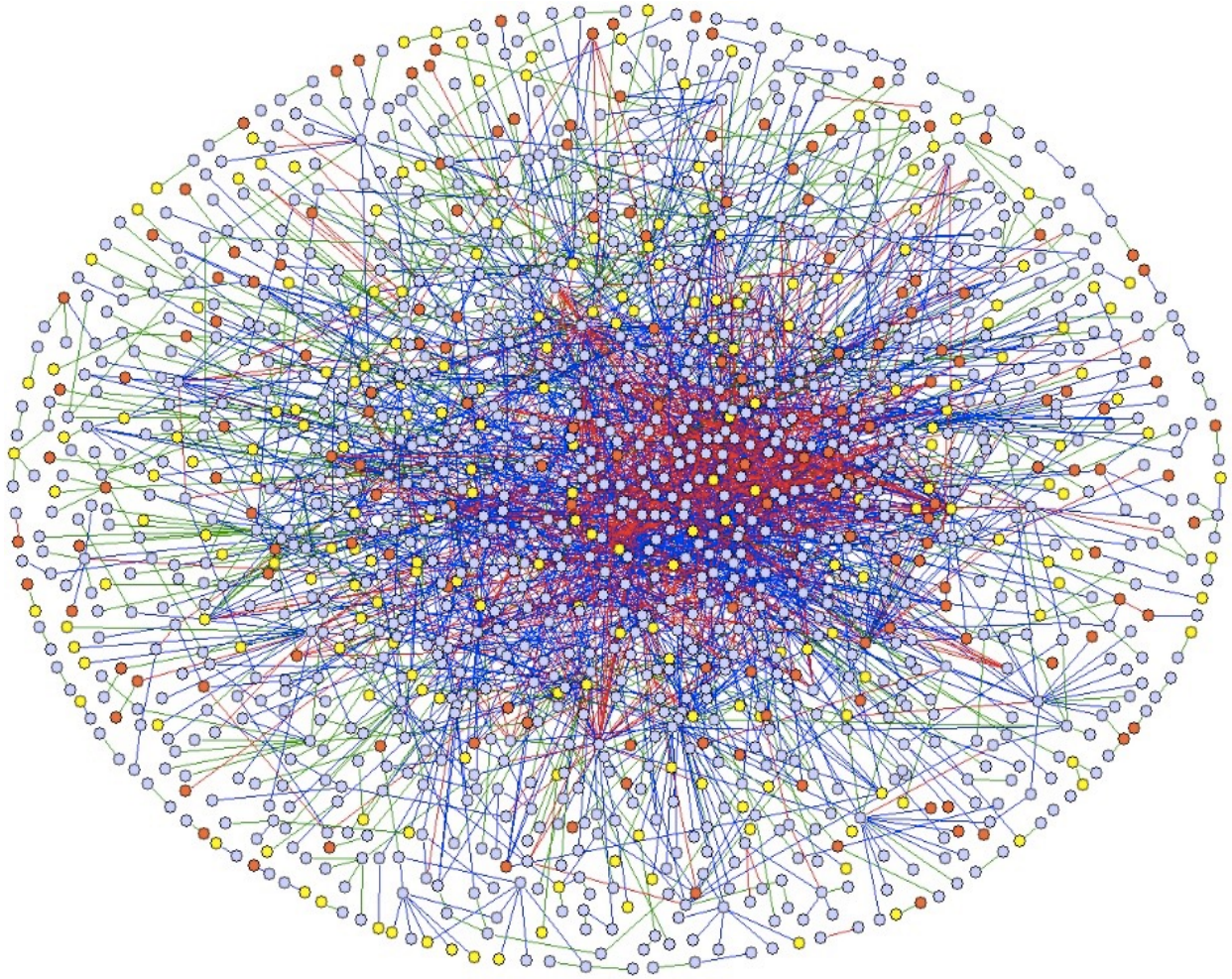
Is this the end of the story?

What if the graph is really big?

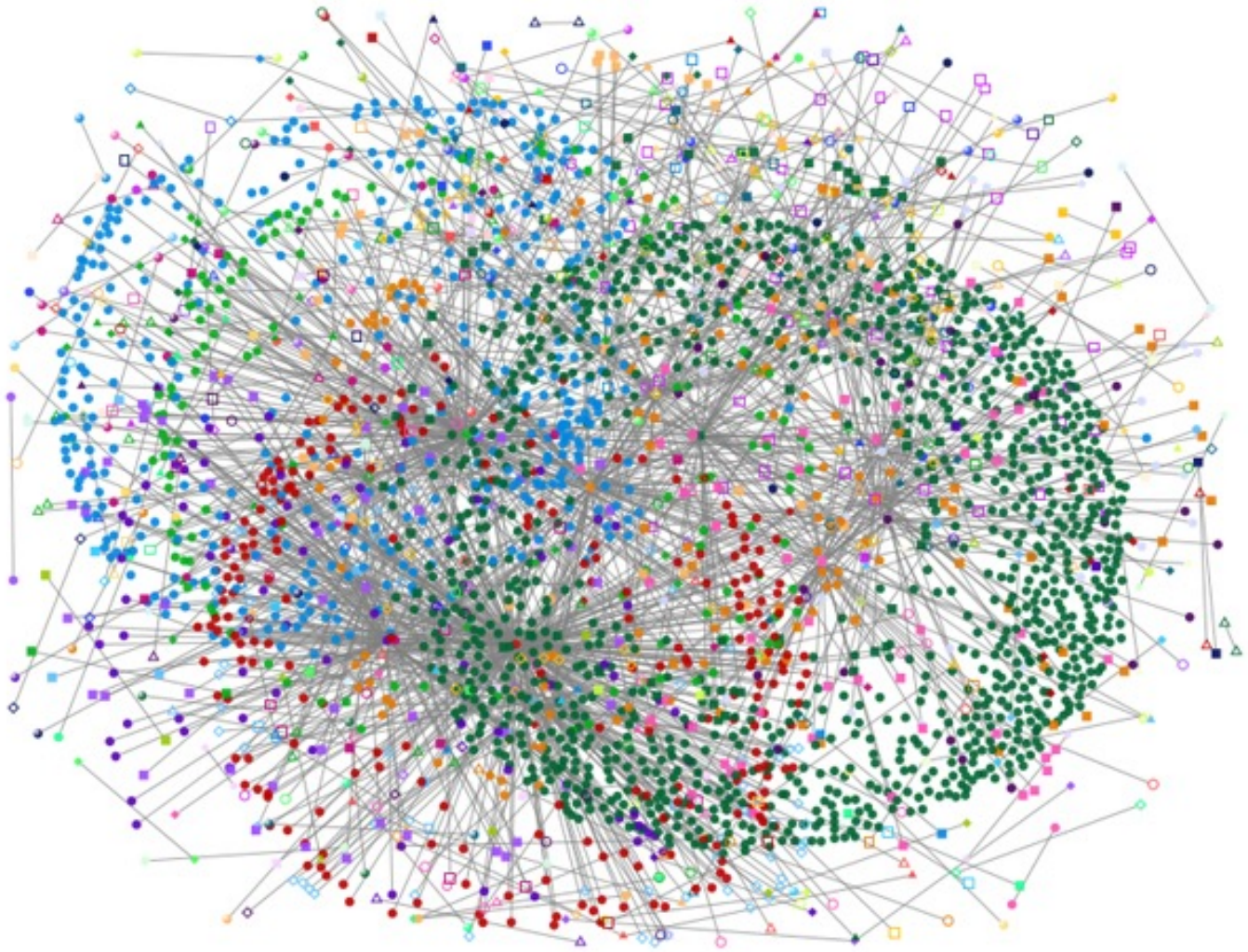
[beyondplm.com]



[Max Delbruck Center for Molecular Medicine]



[hub.packtub.com]



How big is “big”?

Billions of vertices, trillions of edges

Concurrency

Can we speed up the computation using lots of processes, as many as one or two per edge or vertex?

- Computation models:
  - Common memory (PRAM)
  - Distributed memory (message-passing)
- Critical issue: how much synchronization?

Synchronized model (with S. Liu)

Many processors, all executing one step at a time synchronously

(global clock)

In the distributed model, a “step” can be a large amount of local computation

Naïve algorithm ("label propagation")

replace each edge $\{v, w\}$ by arcs vw and wv

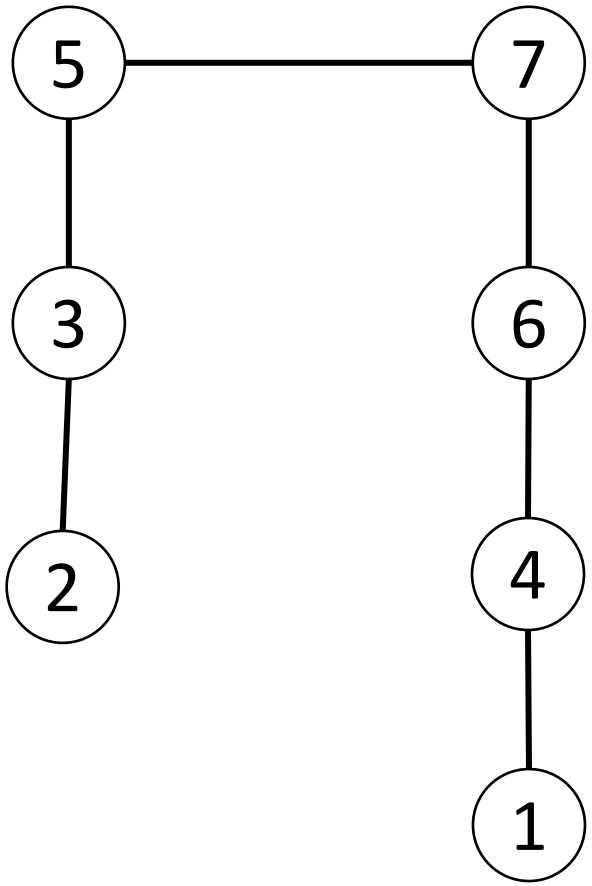
for each vertex v do $v.p \leftarrow v$

repeat

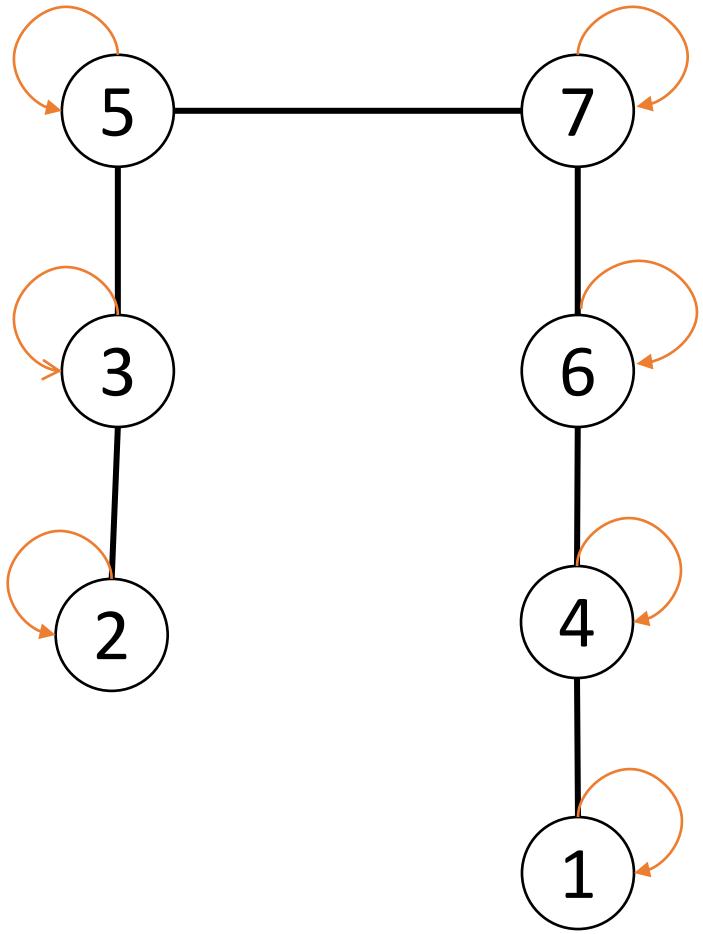
  for each arc vw do $v.p \leftarrow \min\{v.p, w.p\}$

until no label changes

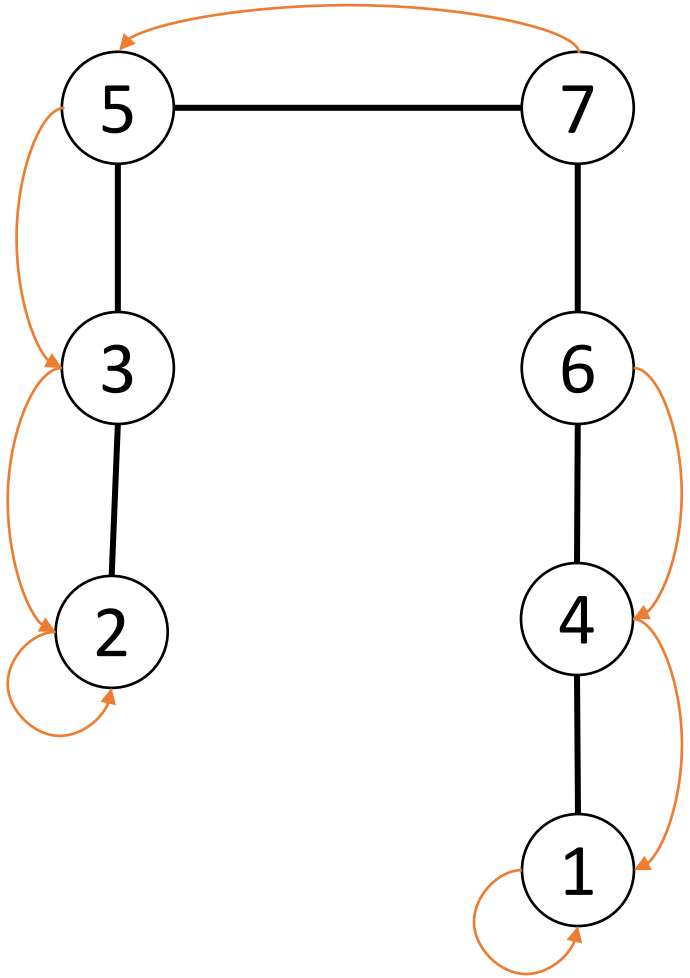
- $v.p$ is the label of v (v points to $v.p$)
- Repeat loop runs synchronously in parallel
- Write conflicts resolved in favor of smallest value



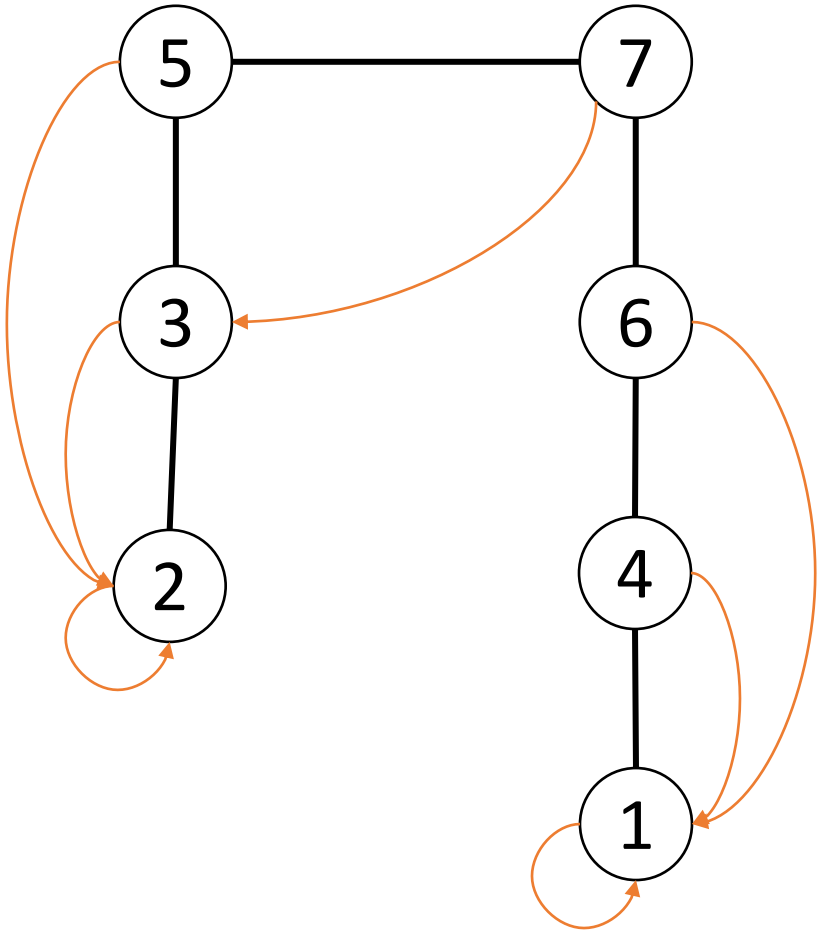
1
2
3
4
5
6
7



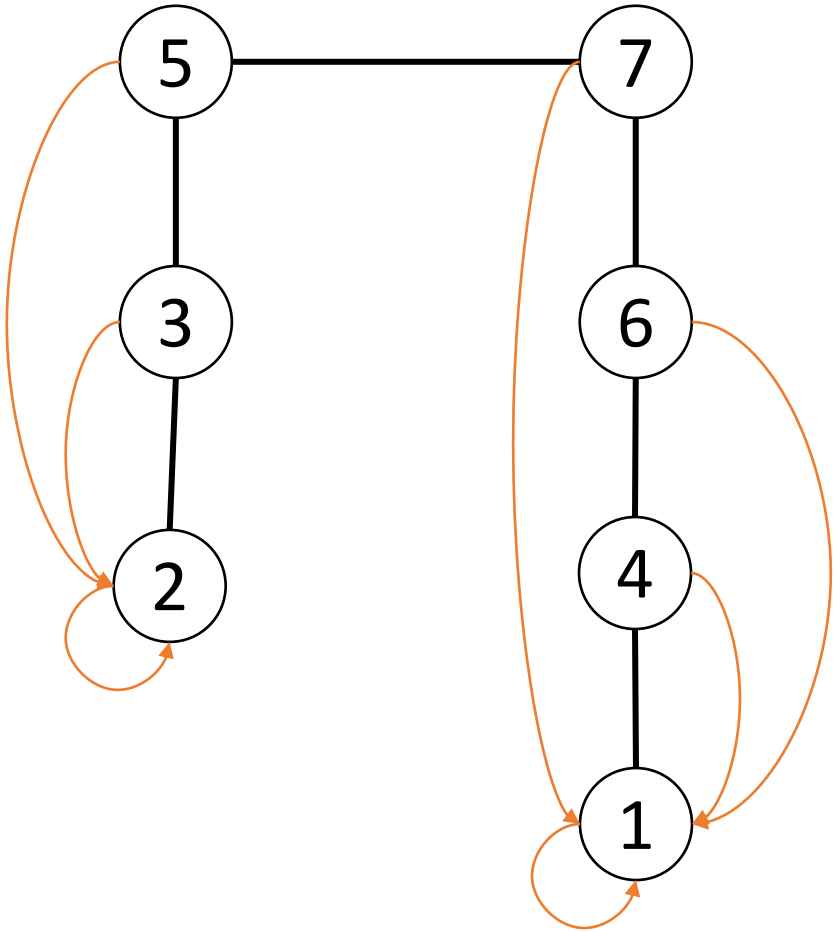
1
2
3
4
5
6
7



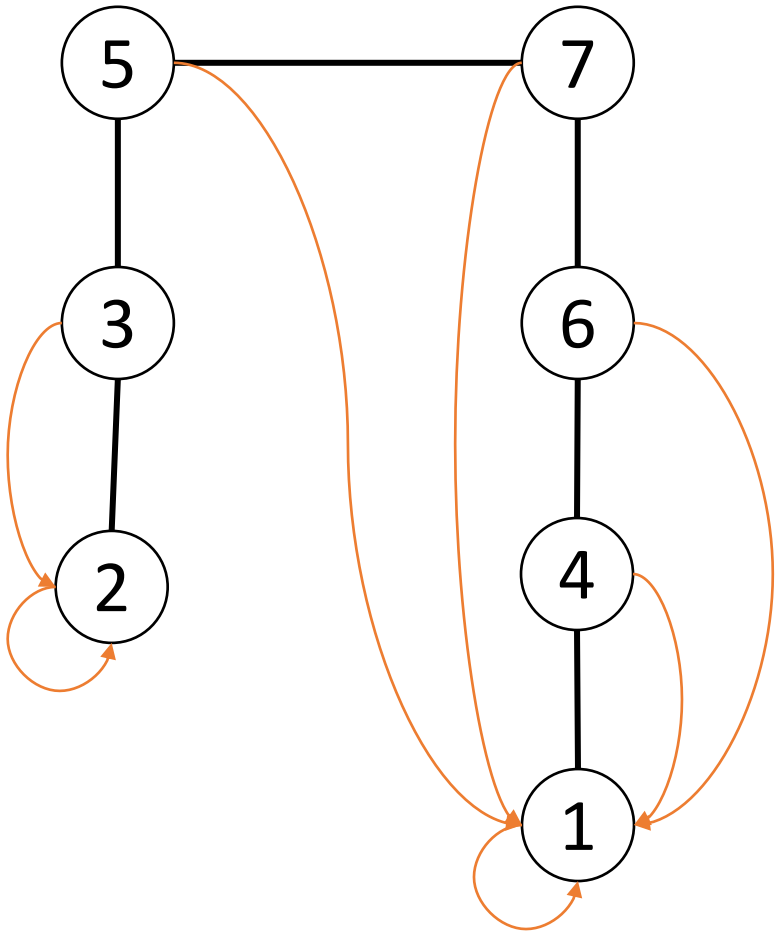
1 1
2 2
3 2
4 1
5 3
6 4
7 5



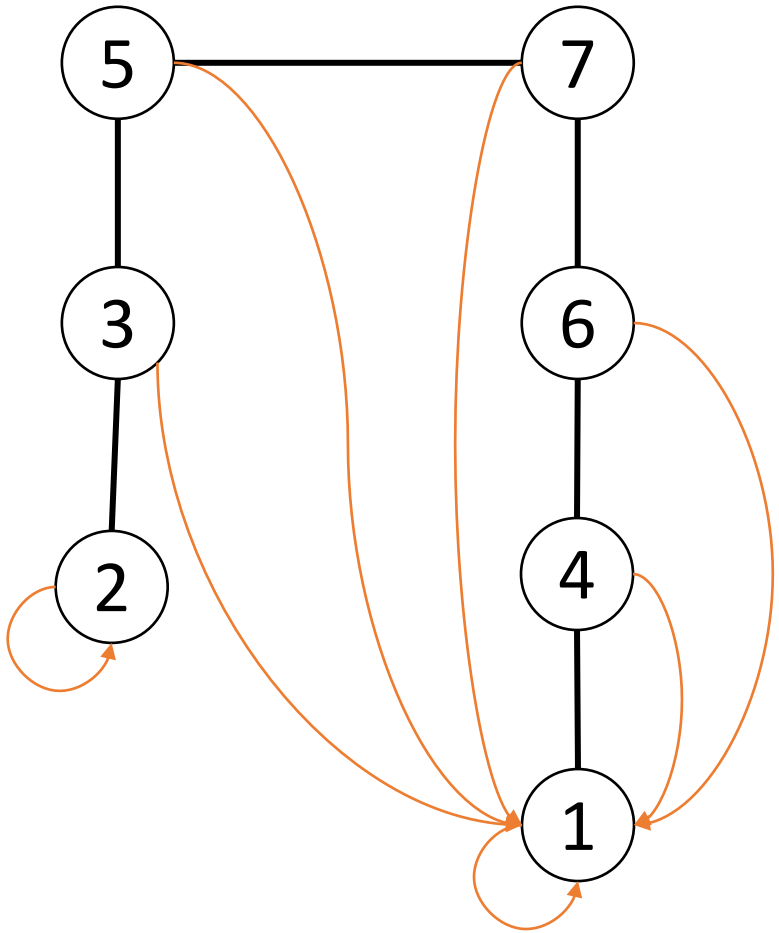
1 1 1
2 2 2
3 2 2
4 1 1
5 3 2
6 4 1
7 5 3



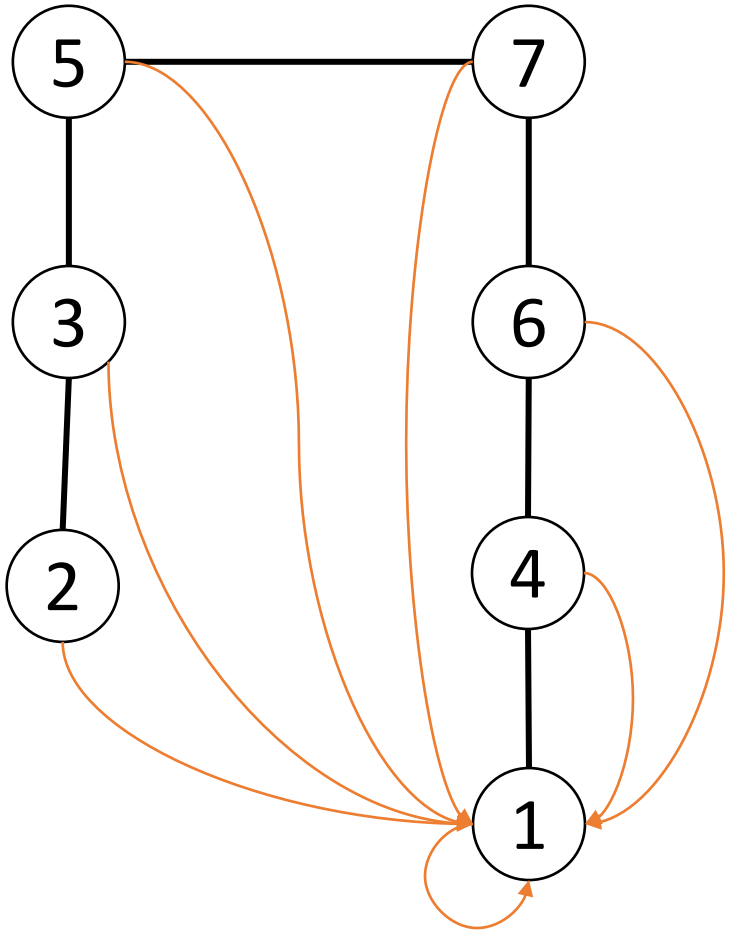
1 1 1 1
2 2 2 2
3 2 2 2
4 1 1 1
5 3 2 2
6 4 1 1
7 5 3 1



1 1 1 1 1
2 2 2 2 2
3 2 2 2 2
4 1 1 1 1
5 3 2 2 1
6 4 1 1 1
7 5 3 1 1



1 1 1 1 1 1
2 2 2 2 2 2
3 2 2 2 2 1
4 1 1 1 1 1
5 3 2 2 1 1
6 4 1 1 1 1
7 5 3 1 1 1



1 1 1 1 1 1 1
2 2 2 2 2 2 1
3 2 2 2 2 1 1
4 1 1 1 1 1 1
5 3 2 2 1 1 1
6 4 1 1 1 1 1
7 5 3 1 1 1 1

Notes

Each vertex points to itself or a smaller vertex

No pointer cycles except loops (self-pointing vertices)

The pointers define a set of (in-)trees, each rooted at a self-pointing vertex

We call a self-pointing vertex a root

Roots are locally minimum

We call the label of a non-self-pointing vertex its parent

The roots in a component are the current candidates to be its leader

Each component is partitioned into one or more trees

How many steps?

$\Theta(d)$ where d is the maximum diameter of a component

This algorithm does concurrent breadth-first search from the smallest vertices in the components

Slow on high-diameter graphs:

Paths of pointers can be long

We need a way to shortcut long paths

Faster?

Shortcut (also called *compress, split, pointer jump*):

for each v do $v.p \leftarrow v.p.p$

A shortcut roughly halves the distance from a vertex to a candidate for leader

Might lead to an algorithm that takes O(lg$n$) steps

One more idea: root connection

When connecting, only change the parents of roots

We need one level of look-ahead

root-connect:

for each (v, w) do

if $v.p.p = v.p$ then $v.p.p \leftarrow \min\{v.p.p, w.p\}$

Algorithm R (for Root-connect)

R: repeat

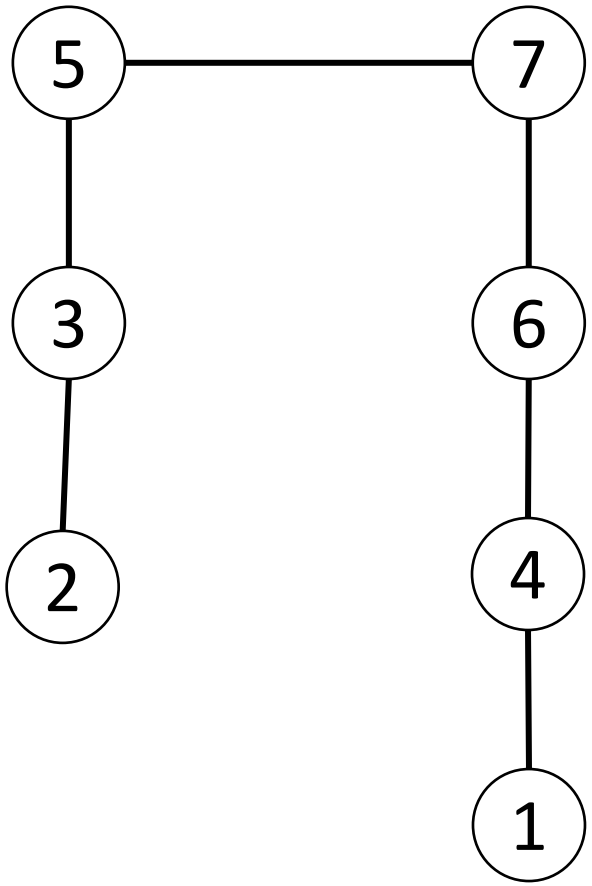
root-connect

shortcut

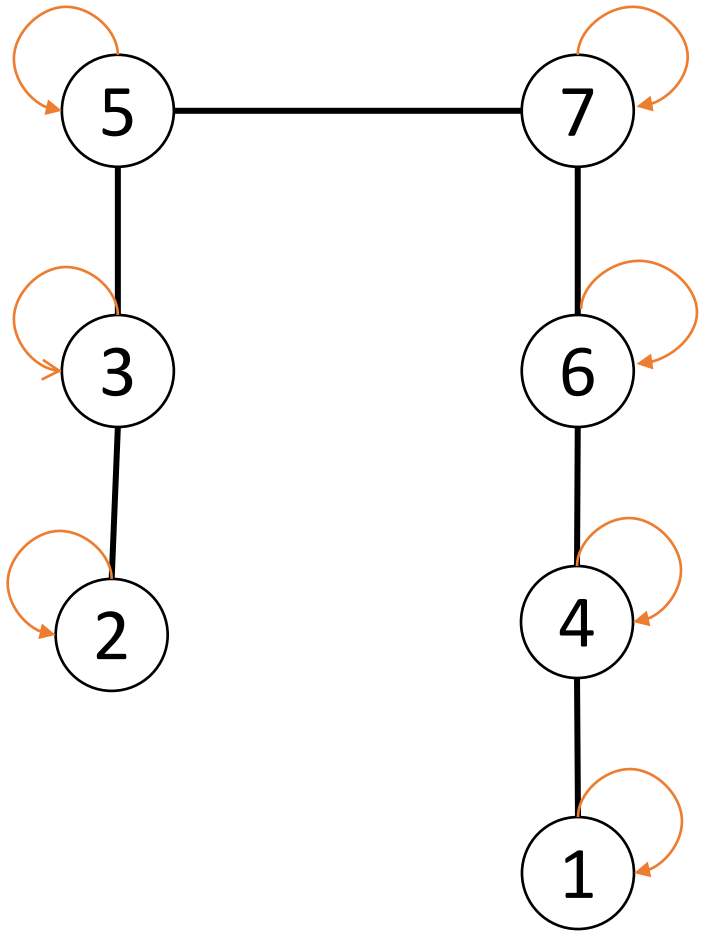
until no parent changes

Based on Shiloach-Vishkin 1982 but simpler

Crucially, breaks ties in favor of smallest label

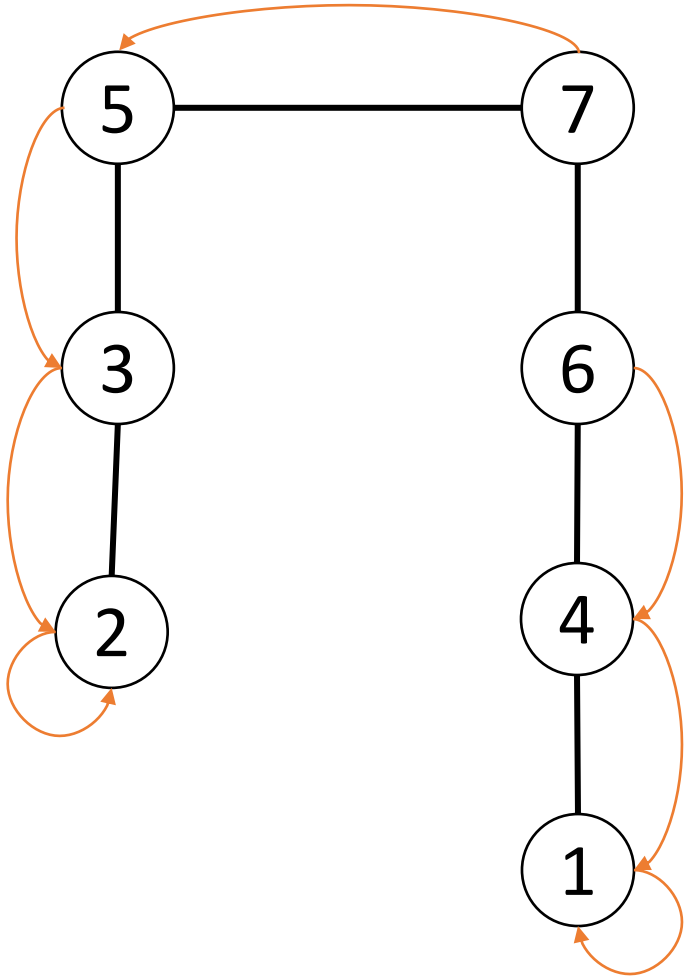


1
2
3
4
5
6
7



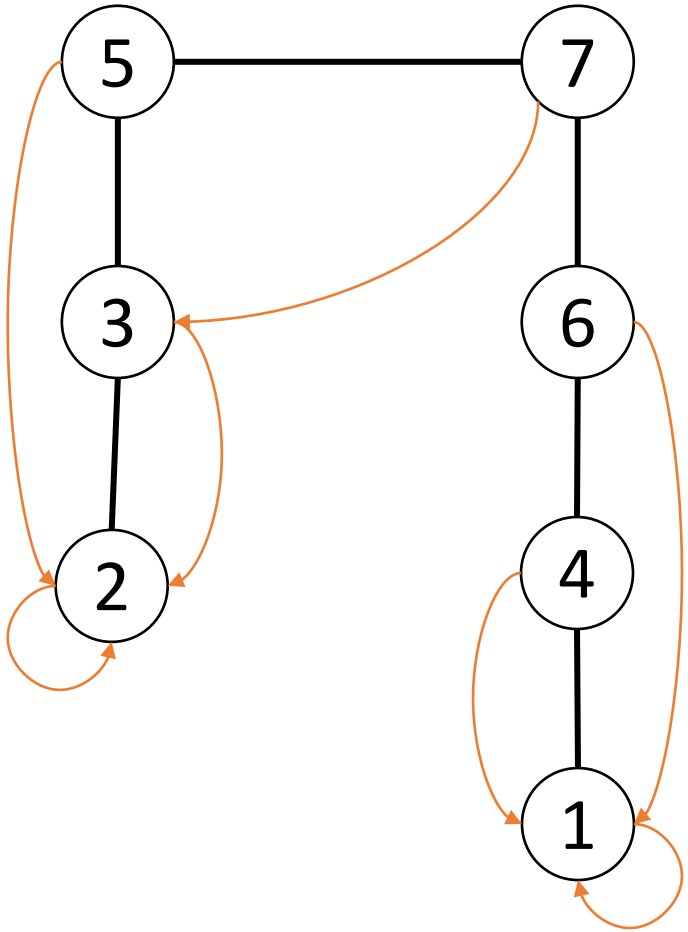
1
2
3
4
5
6
7

root-connect



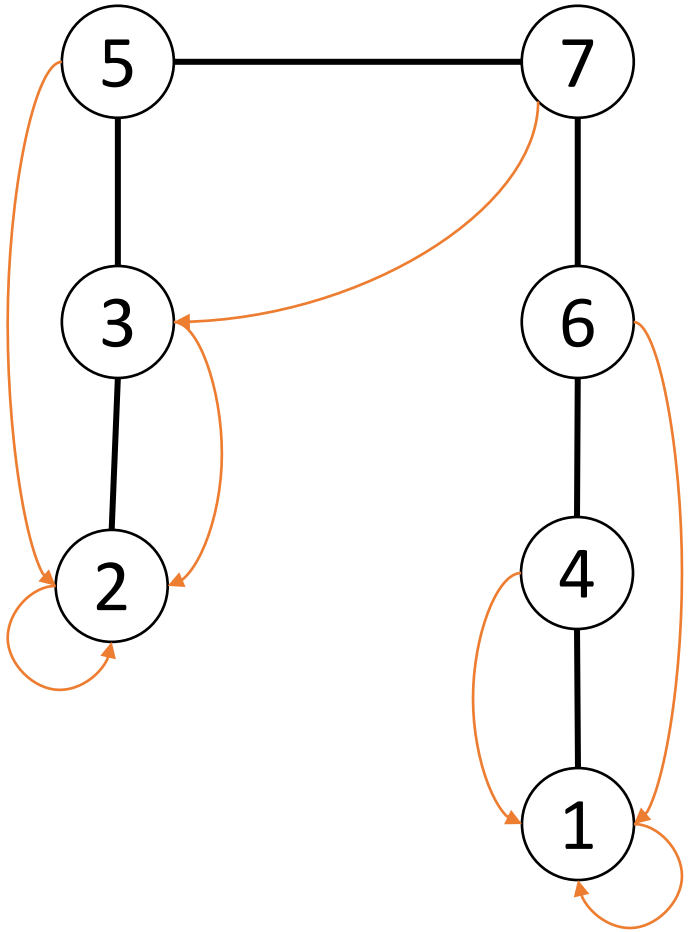
1 1
2 2
3 2
4 1
5 3
6 4
7 5

shortcut



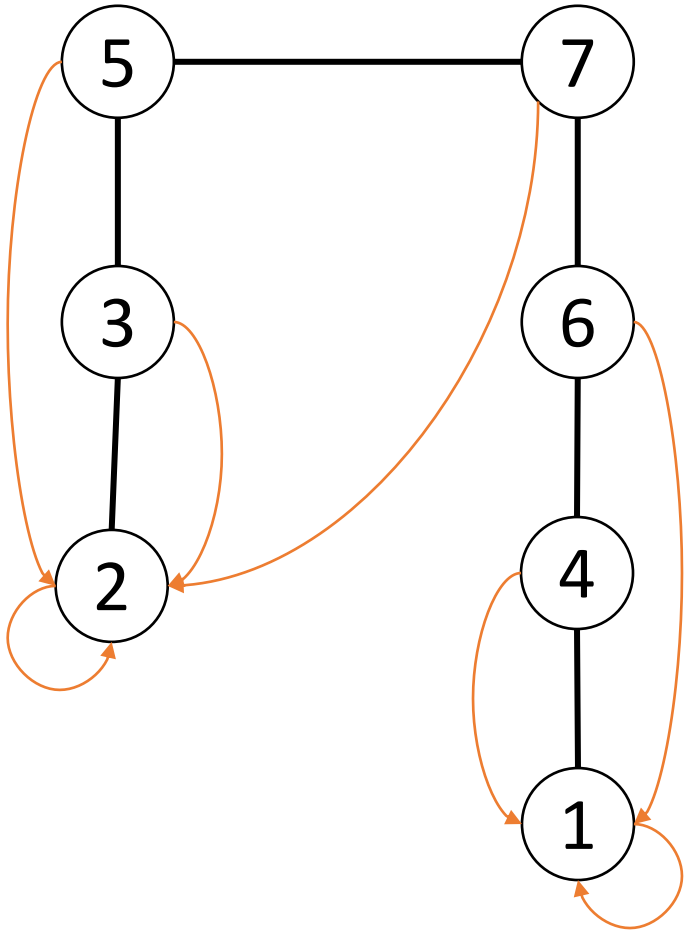
1 1 1
2 2 2
3 2 2
4 1 1
5 3 2
6 4 1
7 5 3

root-connect



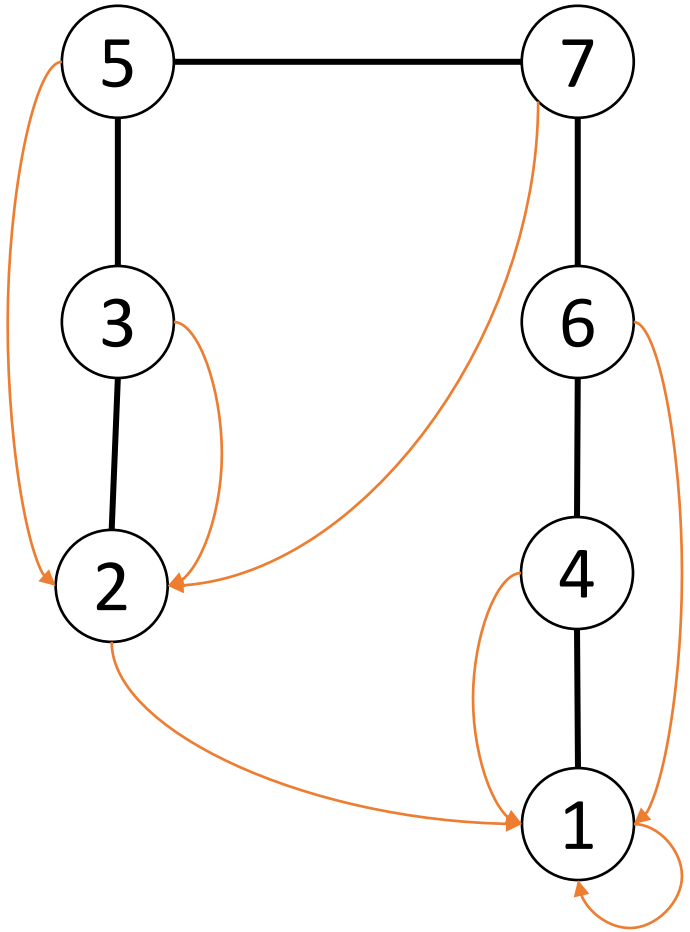
1 1 1 1
2 2 2 2
3 2 2 2
4 1 1 1
5 3 2 2
6 4 1 1
7 5 3 3

shortcut



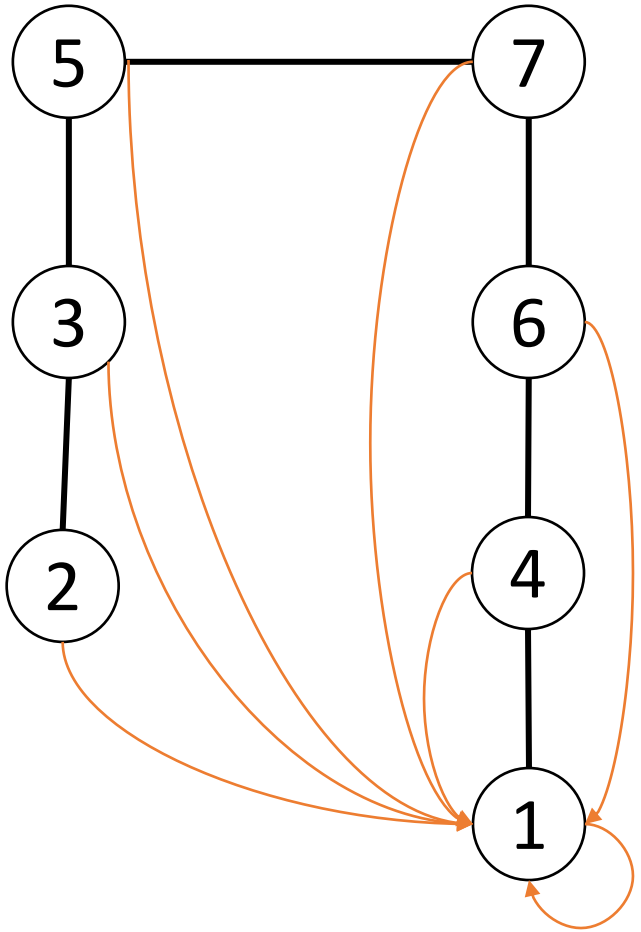
1 1 1 1 1
2 2 2 2 2
3 2 2 2 2
4 1 1 1 1
5 3 2 2 2
6 4 1 1 1
7 5 3 3 2

root-connect



1 1 1 1 1 1
2 2 2 2 2 1
3 2 2 2 2 2
4 1 1 1 1 1
5 3 2 2 2 2
6 4 1 1 1 1
7 5 3 3 2 2

shortcut



1 1 1 1 1 1 1
2 2 2 2 2 1 1
3 2 2 2 2 2 1
4 1 1 1 1 1 1
5 3 2 2 2 2 1
6 4 1 1 1 1 1
7 5 3 3 2 2 1

How fast?

$\Theta(\lg n)$ rounds worst-case: Liu and T

Analysis uses a variant of the analysis of Awerbuch and Shiloach 1983 plus a novel multi-round analysis to handle flat trees

A tree is flat if every vertex points directly to the root

A flat tree may not change for many rounds

The Latest

Behnezhad, Dhulipala, Esfandiari, Łącki, and Mirrokni 2019:

$O(\lg d + \lg\lg_{m/n} n)$ steps

in a powerful distributed model

complicated!

Uses randomization, but can be made deterministic:

Coy and Czumaj 2022

Algorithm R is fast in practice

BUT assuming global synchronization is unrealistic in practice

What if we assume NO synchronization?

Disjoint set union

Maintain a collection of disjoint sets, each with a unique element called its leader, subject to three operations:

make-set(x) (x in no set): Create a set $\{x\}$ with leader x

find(x): Return the leader of the set containing x

unite(x, y): If x and y are in different sets, unite these sets and choose a leader for the new set

The power of laziness

Use indirection: do not store with each element the leader of its set, but provide a function to compute it (*find*)

Maintain a pointer field $x.p$ in each node x, with each leader pointing to itself and such that following pointers from any node eventually leads to the leader of its set

To unite the sets containing v and w, find the leaders of their sets, and if they are different make one leader point to the other. The "other" becomes the leader of the new set

Tree representation

Each set forms a rooted tree, whose nodes are the elements in the set, with each node x having a pointer to its parent $x.p$; if x is a root, $x.p = x$

The root of the tree is the set leader

The shape of the tree is *arbitrary*. The shape is determined by the execution of the operations

This is exactly the data structure used in concurrent label propagation and in Algorithm R

Unite via *link*

link(*x*, *y*): Unite the trees with roots *x* and *y*

Implementation of *link*:

make *y* the parent of *x* (or *x* the parent of *y*):

$x.p \leftarrow y$ (*or* $y.p \leftarrow x$)

Links take O(1) time, finds take O(path length) time

Implementation of disjoint sets

make-set(x): $x.p \leftarrow x$

find(x): if $x.p = x$ then return x

else return *find*($x.p$)

unite(x, y): if *find*(x) $\neq$ f*ind*(y) then

link(*find*(x), *find*(y))

A bad sequence of links can create a tree that is a path of n nodes, on which each *find* can take $\Omega(n)$ time, totaling $\Omega(mn)$ time for m *finds*

Goal: reduce the amortized time per *find*: reduce path lengths

Improve links: link by *rank*

Improve finds: *compact* the trees

Linking by rank: Maintain an integer *rank* for each root, initially 0. Link root of smaller rank to root of larger rank. If tie, increase rank of new root by 1.

make-set(x): $x.p \leftarrow x$; $x.r \leftarrow 0$

link(x, y): if $x.r = y.r$ then $y.r \leftarrow y.r + 1$;

if $x.r < y.r$ then $x.p \leftarrow y$ else $y.p \leftarrow x$

The rank of a root is the maximum length of a path to it (in arcs)

Linking by rank locally minimizes this maximum length

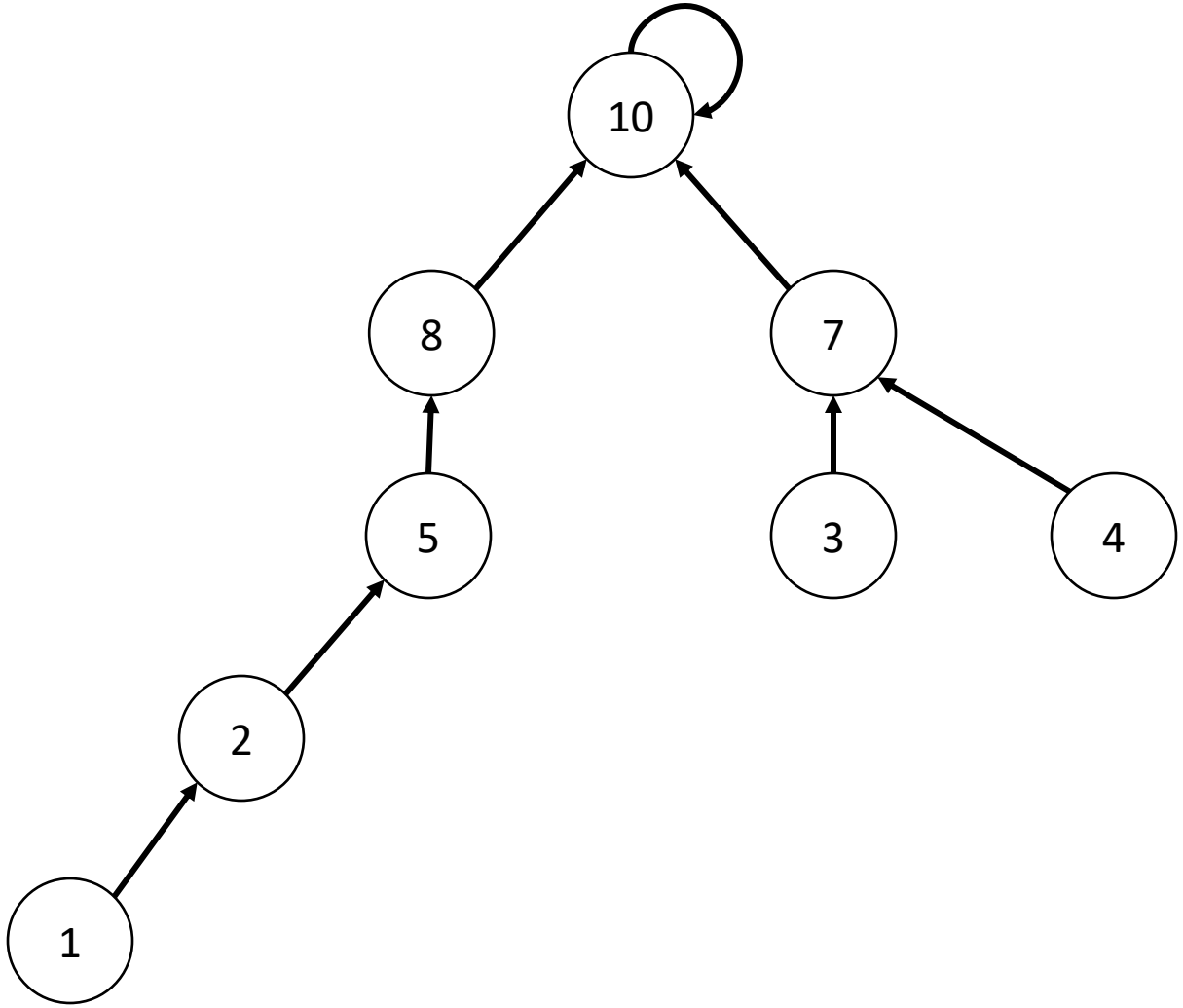
Shortcutting during a find

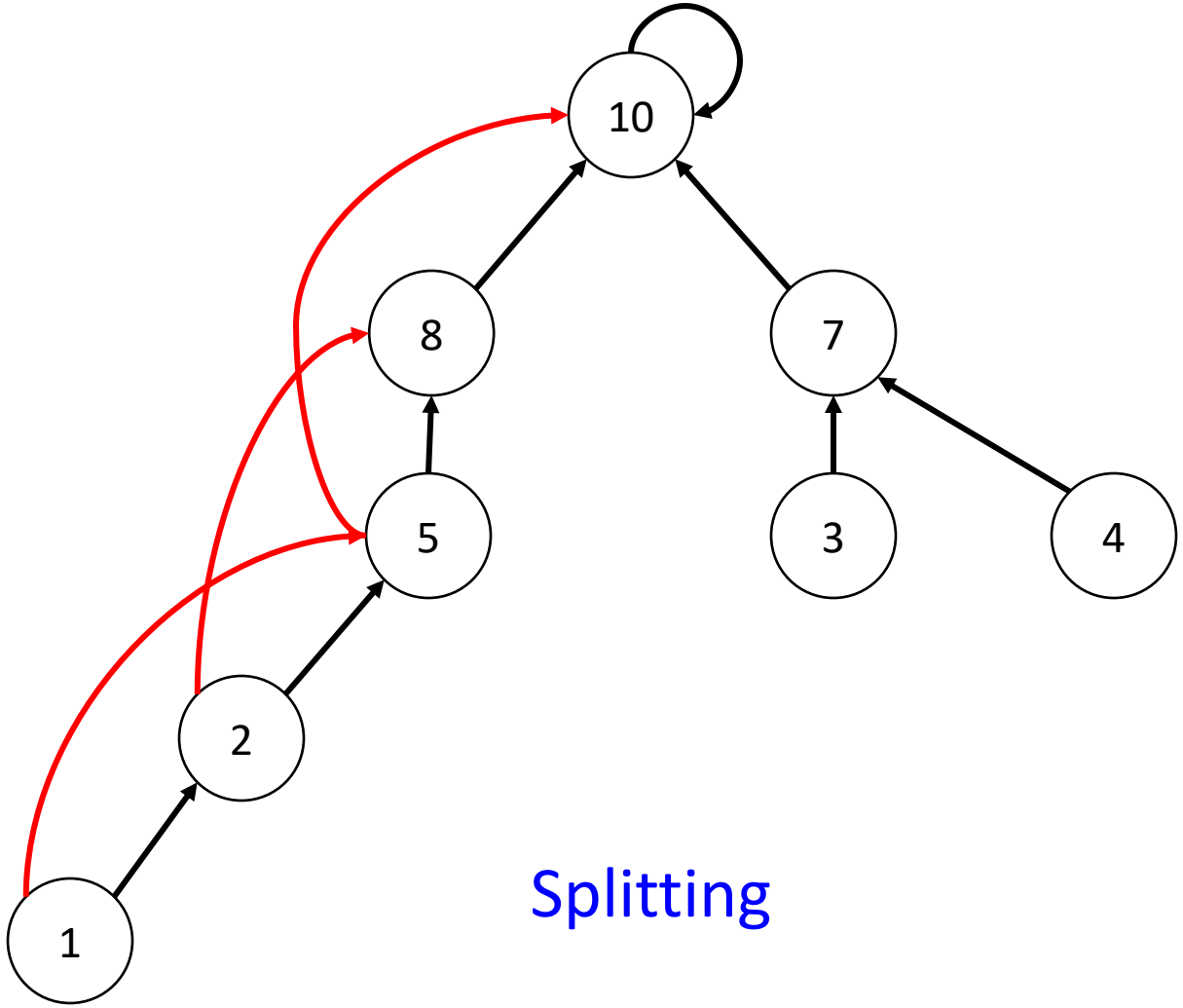
Splitting: replace the parent of each node on the find path by its grandparent

find(x): while $x.p.p \neq x.p$ do

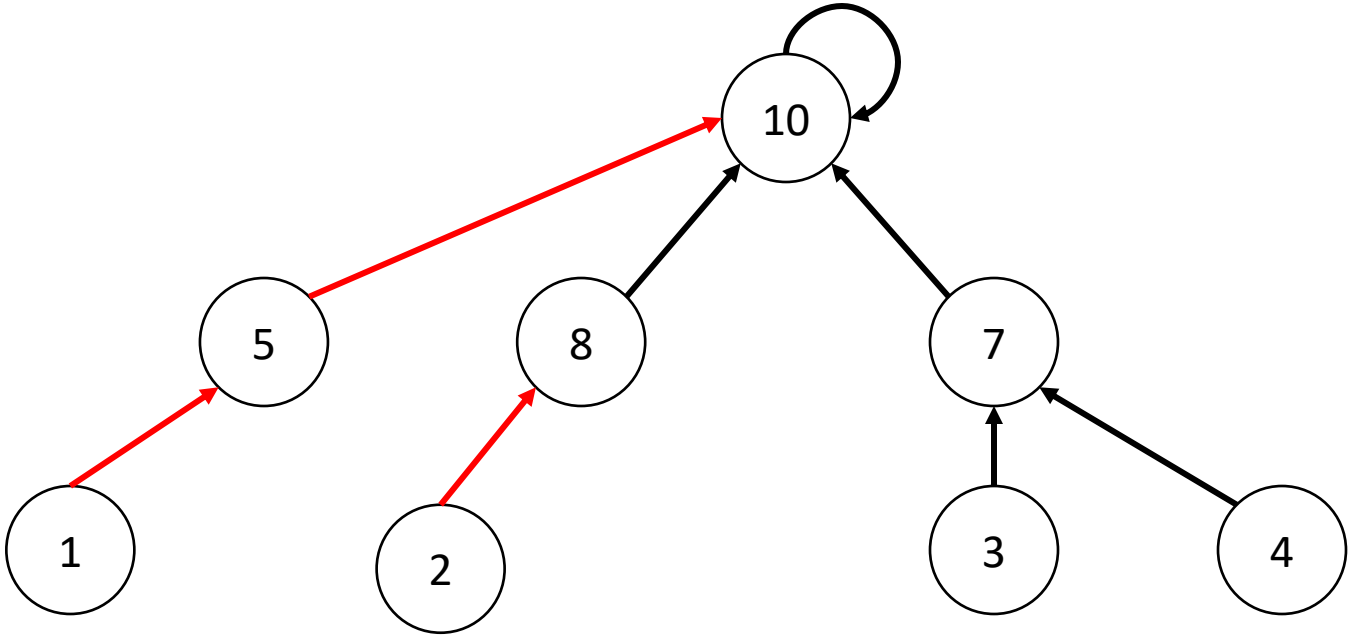
$$y \leftarrow x.p;\ x.p \leftarrow y.p;\ x \leftarrow y$$

This is shortcutting applied only to the nodes on the find path


10
8
7
5
3
4
2
1



Splitting



Splitting

How efficient is splitting, with or without linking by rank?

Compression with any linking rule

$\Theta(m\log_{m/n}n)$ for m operations on sets with n elements total

This is $\Theta(\log n)$ amortized time per *find*

Compression with linking by rank

$\Theta(m\alpha(n, m/n))$ for m operations on sets with n elements total (T 1975)

$\Theta(\alpha(n, m/n))$ amortized time per *find*

Ackermann's function (Péter & Robinson)

$A_0(n) = n + 1$

$A_k(0) = A_{k-1}(1)$ if $k > 0$

$A_k(n) = A_{k-1}(A_k(n-1))$ if $k > 0$, $n > 0$

$= A_{k-1}^{(n+1)}(1)$

$=$ A_{k-1} applied to 1, $n + 1$ times

$\alpha(n, d) = \min\{k > 0 \mid A_k(\lceil d \rceil) > n\}$

Application to connected components

For each vertex *v* do *make-set*(*v*)

For each edge {*v*, *w*} do *link*(*find*(v), find(w))

For each vertex *v* do *v.label* = *find*(*v*)

Can we run the main loop concurrently but asynchronously?

Efficiency

Total work: total number of steps taken by all processes, as a worst-case function of *n*, *m*, and *p* (the number of processes)

Goals: Total work not too much bigger than the sequential time bound and sublinear in *p;* number of steps per operation small

Then concurrency may help

We allow concurrent reads but not concurrent writes

Synchronization Primitives to avoid locks

Compare & Swap

CAS(*x*, *y*, *z*): **if** *x* = *y* **then** {*x* ← *z*; **return true**}

else return false

Double Compare & Swap

DCAS(*x*, *y*, *z*, *u*, *v*, *w*): **if** *x* = *y* **and** *u* = *v*

then {*x* ← *z*; *u* ← *w*; **return true**}

else return false

A CAS succeeds only if the value did not change since the process last read it; the process knows whether the CAS succeeds or fails

Previous work: Anderson & Woll 1994

Concurrent version of linking by rank with splitting using CAS.

Big problem: CAS seems too weak: linking by rank requires changing a rank in one node and a pointer in another.

Their algorithm does not avoid rank ties.

Work bound is $O(m(\alpha(n, 1) + p))$: not so good, and "proof" is buggy: they did not account for interactions between different processors doing splitting along overlapping paths.

Our goal (Jayanti and T 2016-2022)

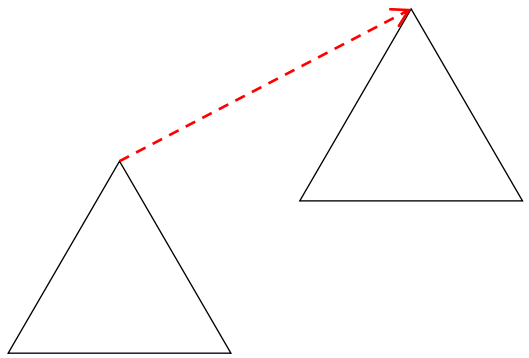
Simple algorithms with good work and step bounds, work bound sublinear in p if possible

Anderson & Woll gave a simple wait-free implementation of find with splitting using CAS, but their analysis is not correct

What about linking by rank?

Link: CAS or DCAS?

CAS okay for links not using ranks, DCAS needed for links by rank



Linking by rank via CAS

link(*v*, *w*): if *v* and *w* have equal rank, first change the parent of *v*, or the rank of *w*?

Our answer: flip a fair coin to decide:

randomized linking by rank

In concurrent linking, must allow for failure, retry until success

Concurrent Find(x)

Find(x)

u = x

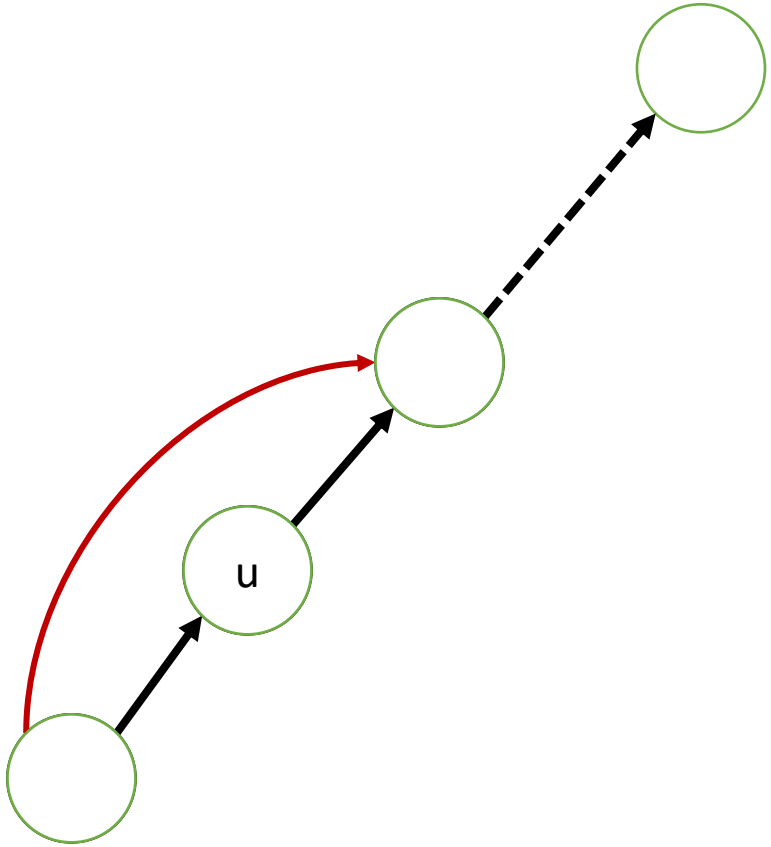
while (u not root)

v = u.parent, w = v.parent

CAS(u.parent, v, w)

u = v

return u



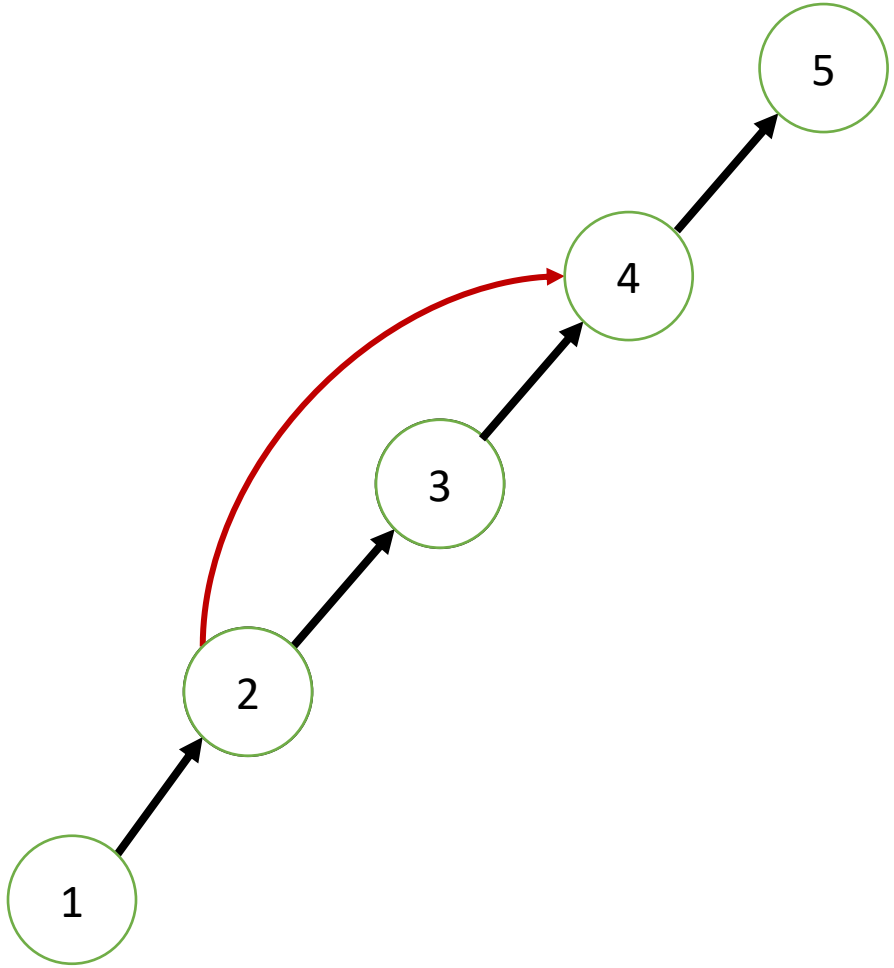
Interfering splits threaten efficiency

Sample Run

- π_A visits 1,2,3 and stalls
- π_B visits 2,3,4 and changes parent of 2 to 4
- π_C visits 1,2,4 and **tries** to change 1's parent to 4
- **but** π_A wakes up and changes 1's parent to 3

- π_C **tries** to split, **but** witnesses **no improvement**

- Anderson and Woll overlooked these possibilities



Concurrent Two-Try Find(x)

```
Find(x):
    u = x

    while (u not root)
        v = u.parent, w = v.parent
        CAS(u.parent, v, w)
        v = u.parent, w = v.parent
        CAS(u.parent, v, w)
        u = v

    return u
```

Our Results

DCAS gives a simple wait-free implementation of linking by rank.

Worst-case time per operation

$O(\log n)$ with or without splitting

Total work with "1-try" splitting

$O(m(\alpha(n, \lceil m/(np^2) \rceil) + \log(np^2/m + 1)))$

Total work with "2-try" splitting

$O(m(\alpha(n, \lceil m/(np) \rceil) + \log(np/m + 1)))$

Our Results

CAS gives a simple wait-free implementation of linking by randomized linking by rank

Worst-case time per operation (w.h.p.)

$O(\log n)$ with or without splitting

Expected work with 1-try splitting

$O(m(\alpha(n, \lceil m/(np^2) \rceil) + \log(np^2/m + 1)))$

Expected work with 2-try splitting

$O(m(\alpha(n, \lceil m/(np) \rceil) + \log(np/m + 1)))$

Assumes a benign scheduler

Randomized linking by rank is more robust against an adversarial scheduler

Versions of these algorithms are very fast in practice

One of the few examples where one can prove good speedup in an asynchronous setting

Thanks!