Concurrent Connected Components

Robert E. Tarjan

Princeton University and Intertrust Technologies

joint work with Sixue Liu, Princeton

Heidelberg Laureate Forum

24 September 2019

Observations

Over the last 50 years, computer scientists have developed many beautiful and theoretically efficient algorithms.

But many such algorithms have yet to be used in practice. Some fail when used improperly, or are less efficient than simpler methods with worse theoretical efficiency.

Why?

Software developers, pressed for time, may choose the simplest solution that works, or seems to.

They may use ideas from theory but simplify them in ways that may not work. (“A little knowledge is a dangerous thing.”). Or, they may build their own solution and provide a flawed efficiency analysis.

How should theoreticians respond?

Develop and analyze simple methods. The analysis can be complicated, but the algorithm must be simple.

Apply theory to analyze and improve methods used or usable in practice.

My personal research goal

Develop and analyze reference algorithms:

algorithms from “the book”

a la “proofs from the book” (Erdős)

Algorithms as simple as possible, with provable resource bounds for important input classes, and efficient in practice

Systematically explore the design space

Einstein: “Make everything as simple as possible, but not simpler”

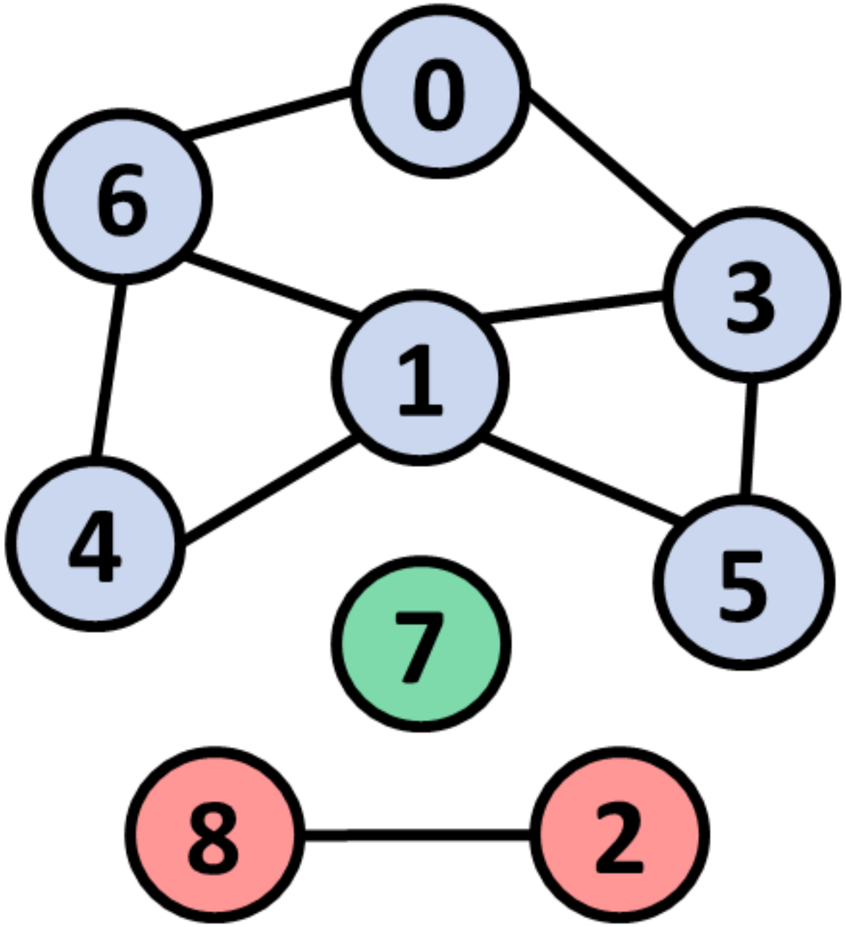
Connected Components

The most basic graph problem?

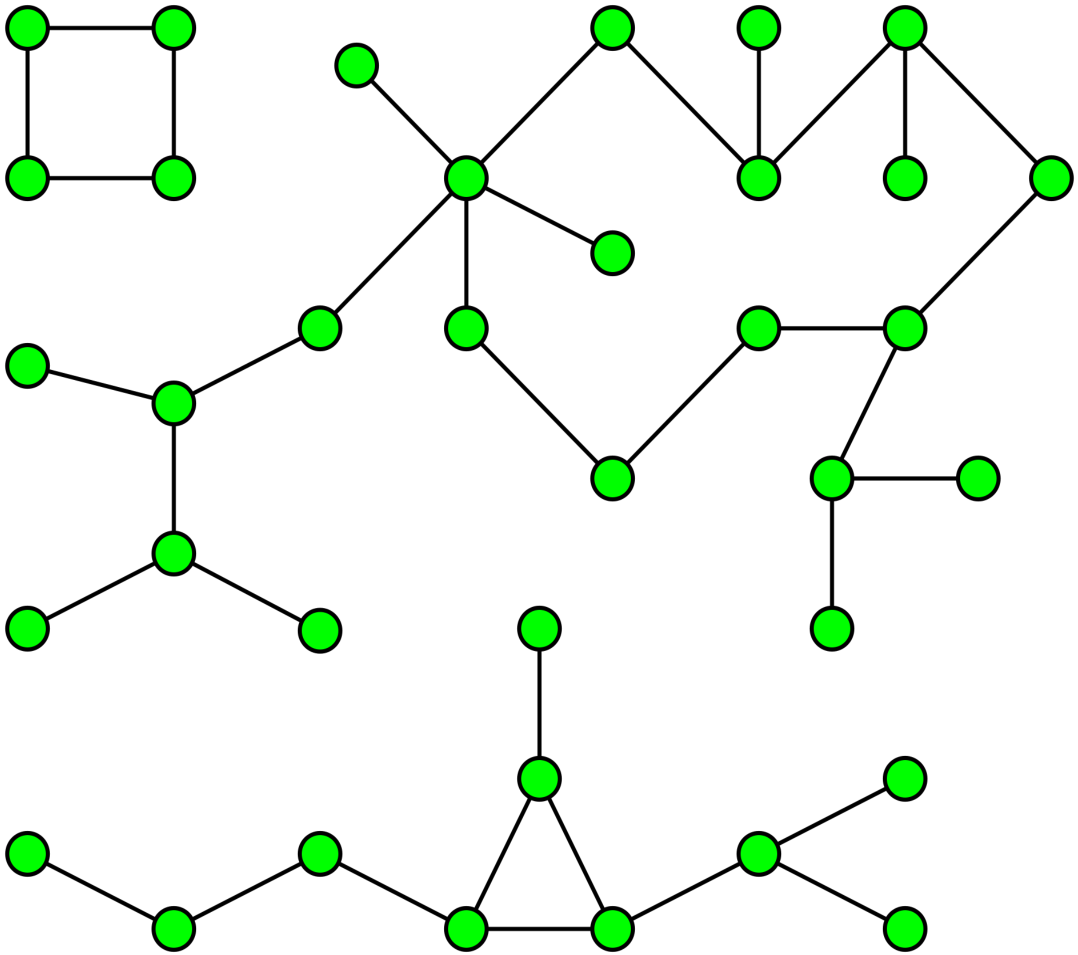
In an undirected graph, two vertices are connected if there is a path between them. A connected component (henceforth just a component) is a maximal set of pairwise-connected vertices.

Problem: Given a graph, compute its components.

[vitoshoacademy.com]

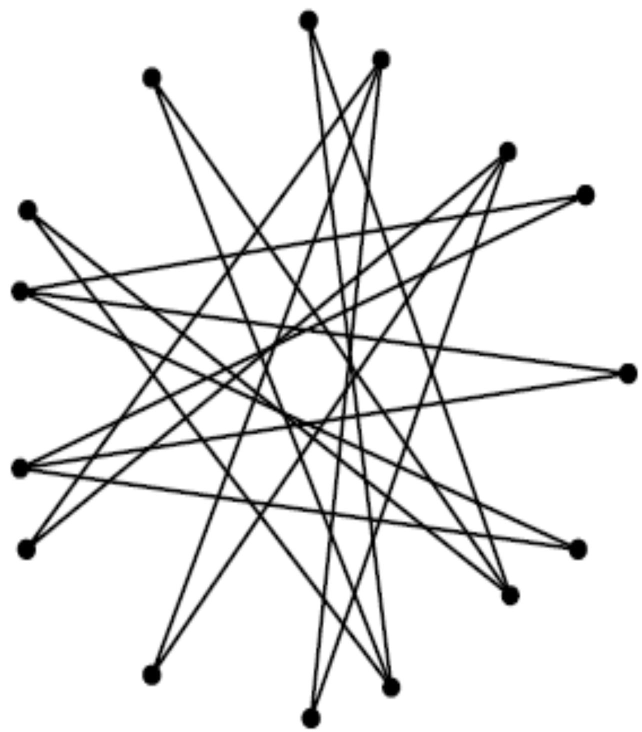


[figure from D. Eppstein]



[math.stackexchange]

A21 Is this graph connected or disconnected?



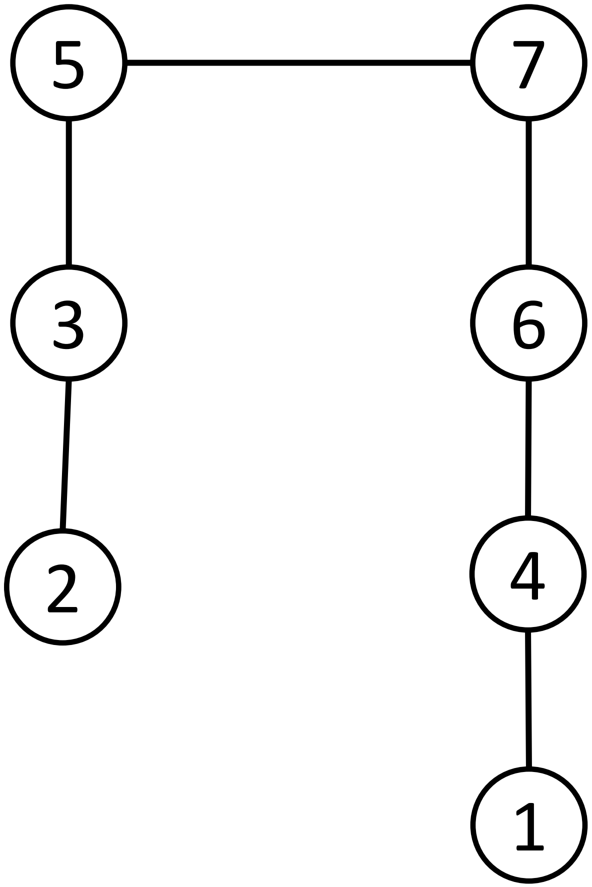
How to represent components?

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

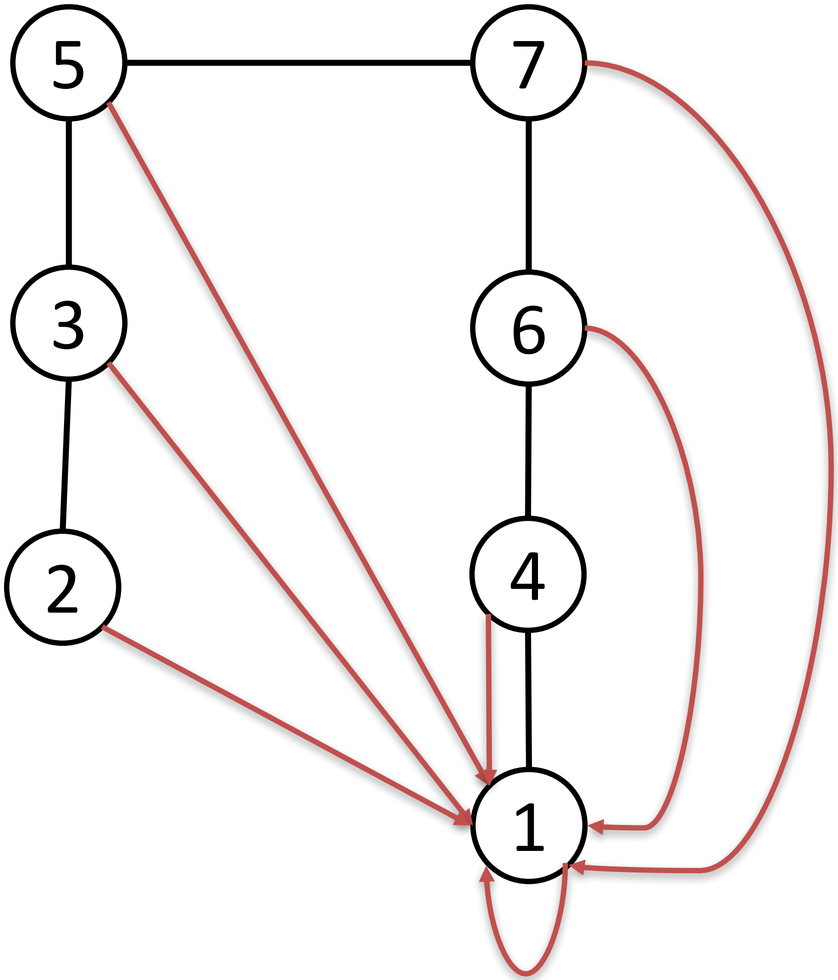
4 1

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, initially singletons, each with a unique canonical element, subject to two operations:

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

find(x): Return the canonical element of the set containing x.

Components via disjoint set union

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

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

Need not actually execute the second loop, just use *find* as needed: *v* and *w* are in the same component iff *find*(*v*) = *find*(*w*)

Running time

Graph search: $O(m + n)$

Disjoint set union via compressed trees:

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

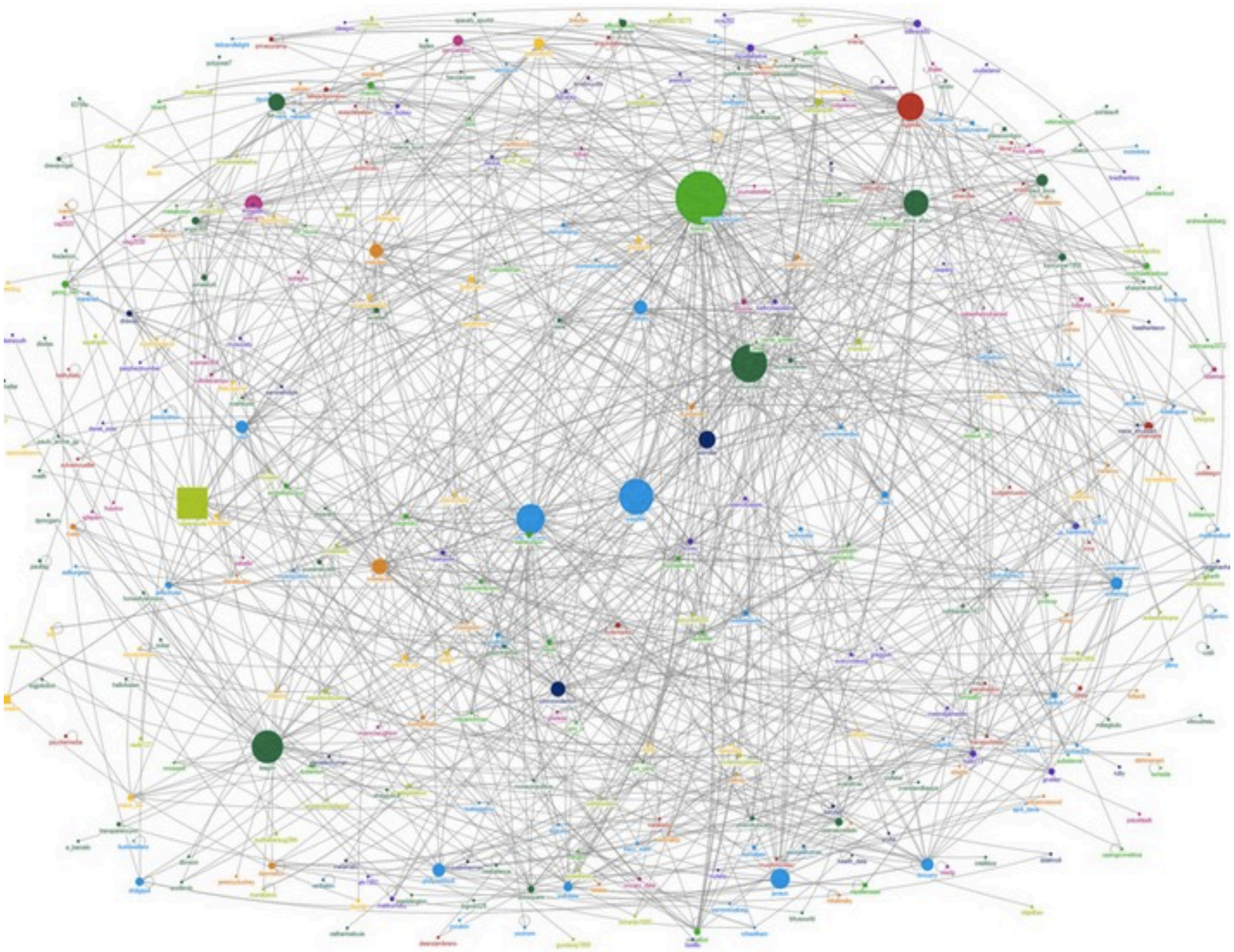
Disjoint set union uses only the edge set, supports individual and batch edge insertions

inverse-Ackermann amortized time per edge insertion or query

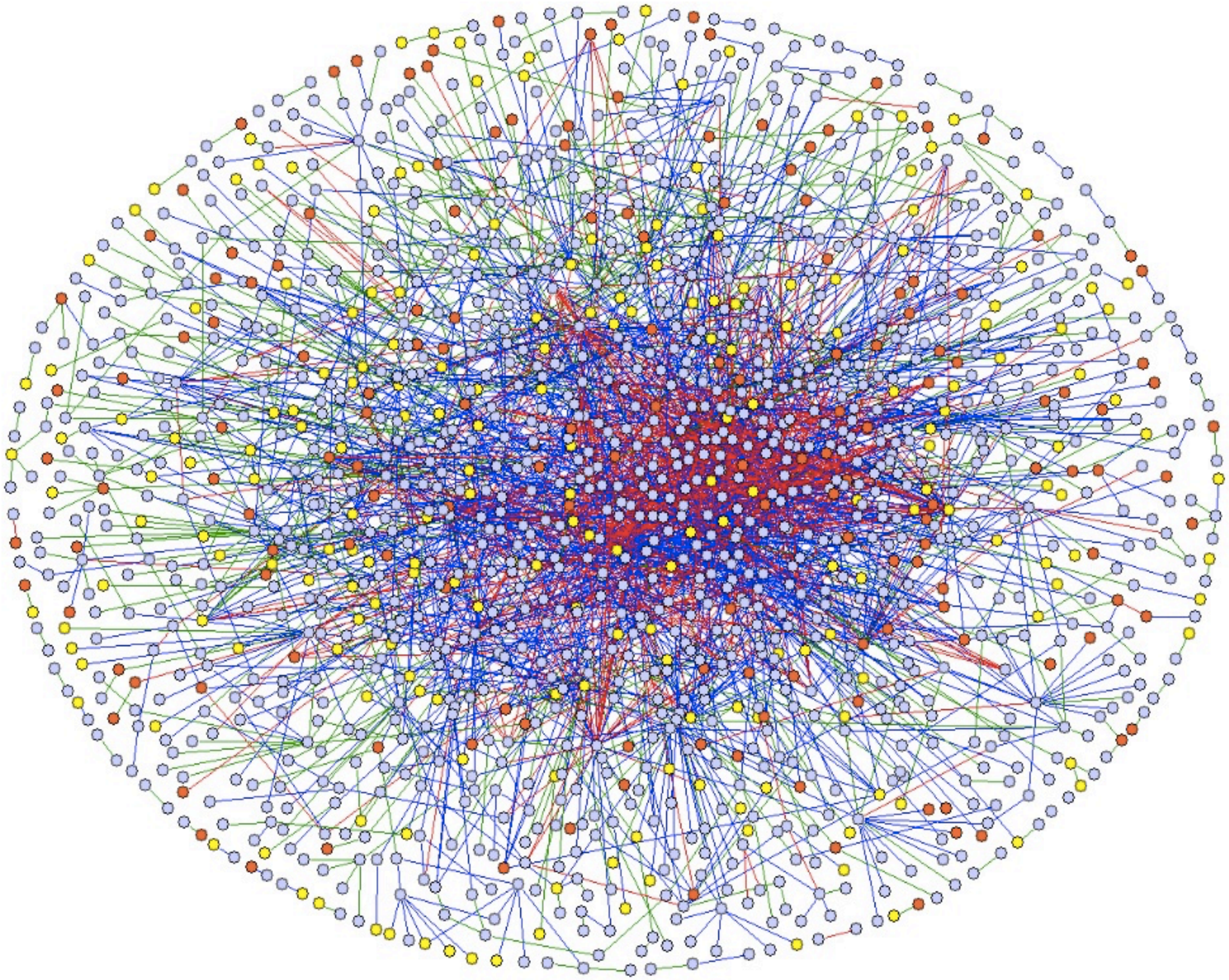
Is this the end of the story?

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 $O(1)$ per edge?

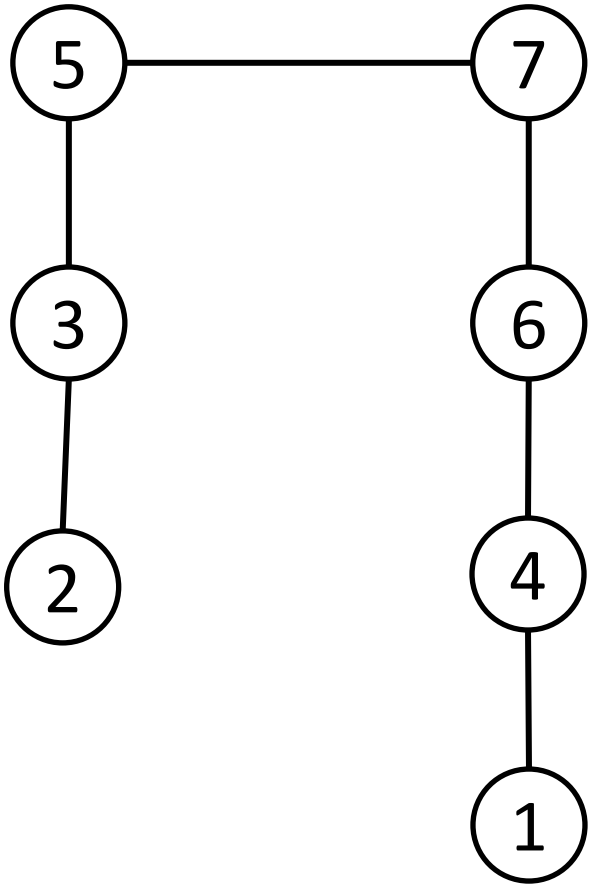
Computation models:

- Common memory (PRAM)
- Distributed memory (message-passing)

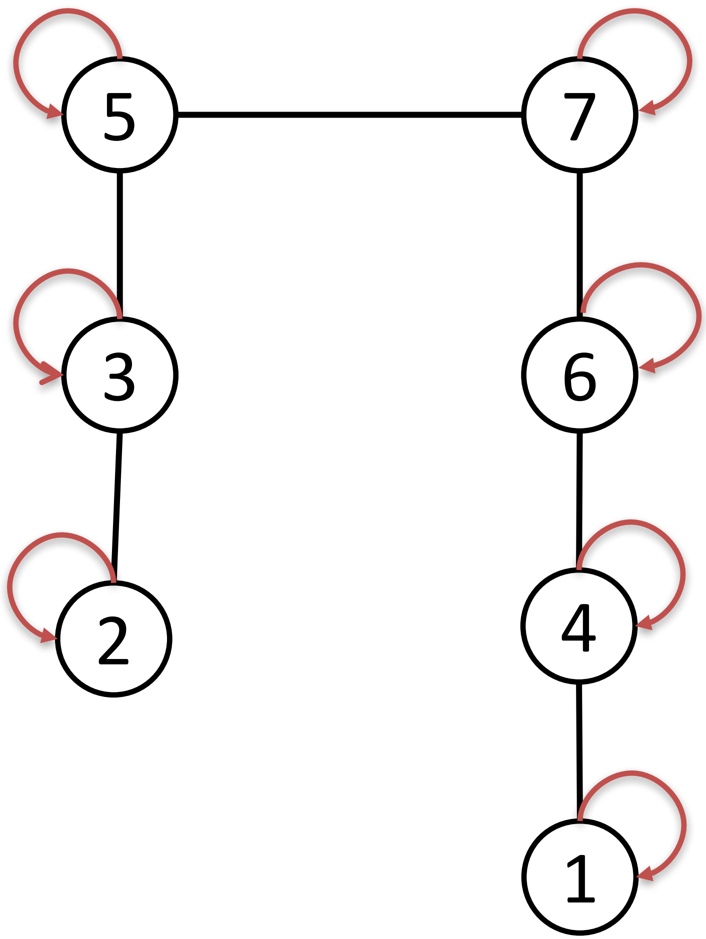
Naïve algorithm ("label propagation")

replace each edge $\{v, w\}$ by arcs (v, w) and (w, v)
for each vertex v do $v.p_0 \leftarrow v$
$i \leftarrow 0$
repeat
    for each arc (v, w) do $v.p_{i+1} \leftarrow \min\{v.p_{i+1}, w.p_i\}$
    $i \leftarrow i + 1$
until no parent changes

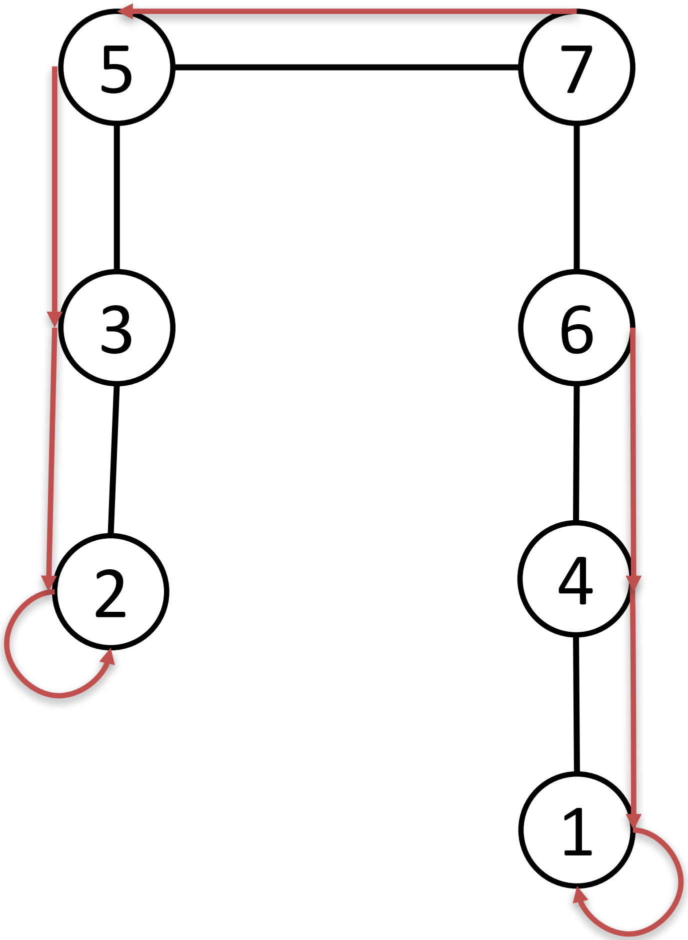
- First three lines are initialization
- $v.p$ is the label of v ("p" for "parent")
- Loop runs synchronously in parallel
- Write conflicts resolved in favor of smallest value



1
2
3
4
5
6
7



1
2
3
4
5
6
7



1 1

2 2

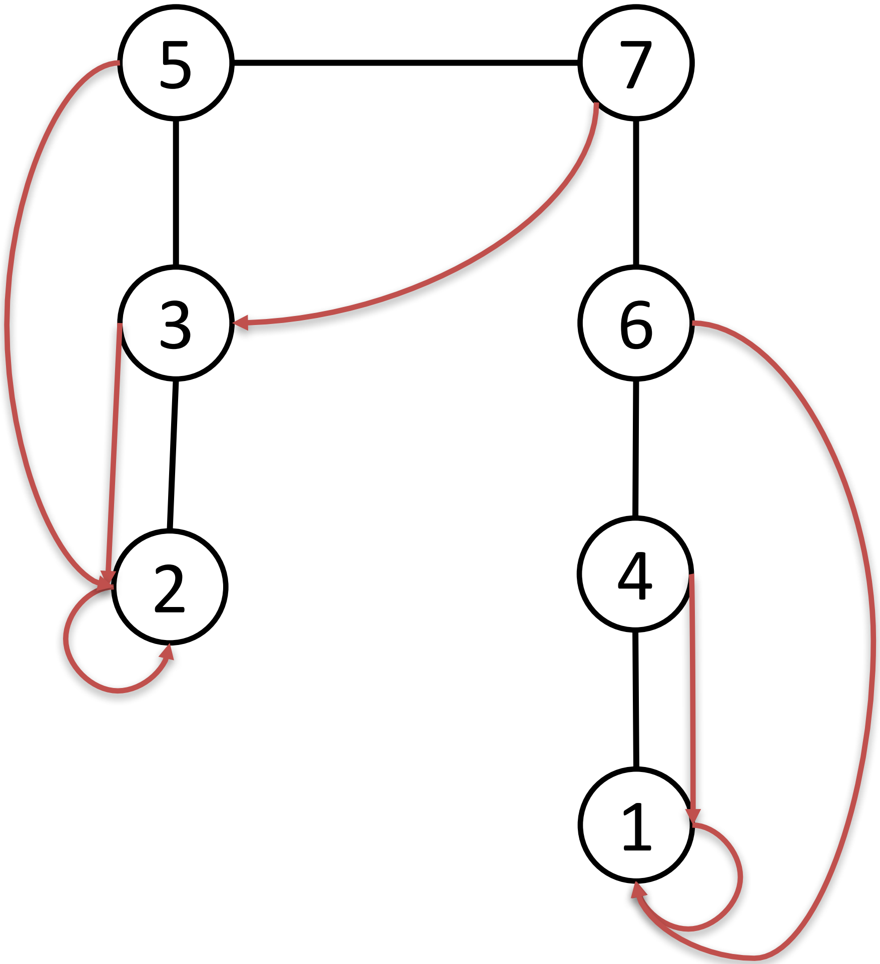
3 2

4 1

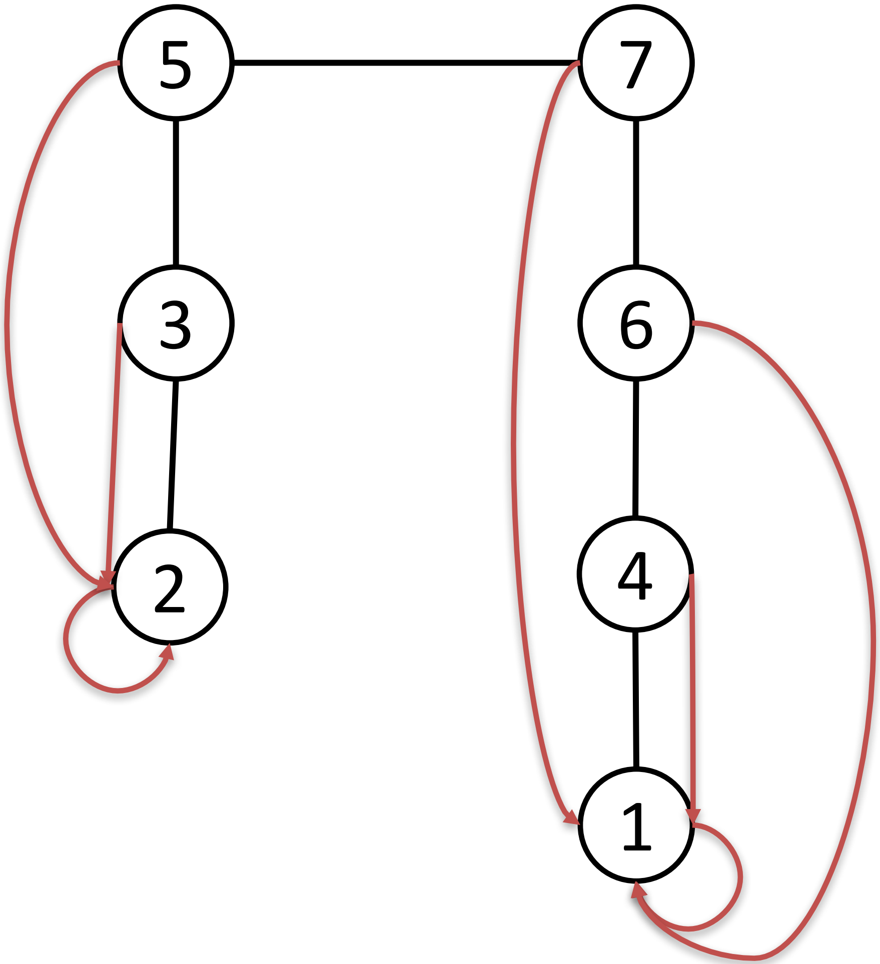
5 3

6 4

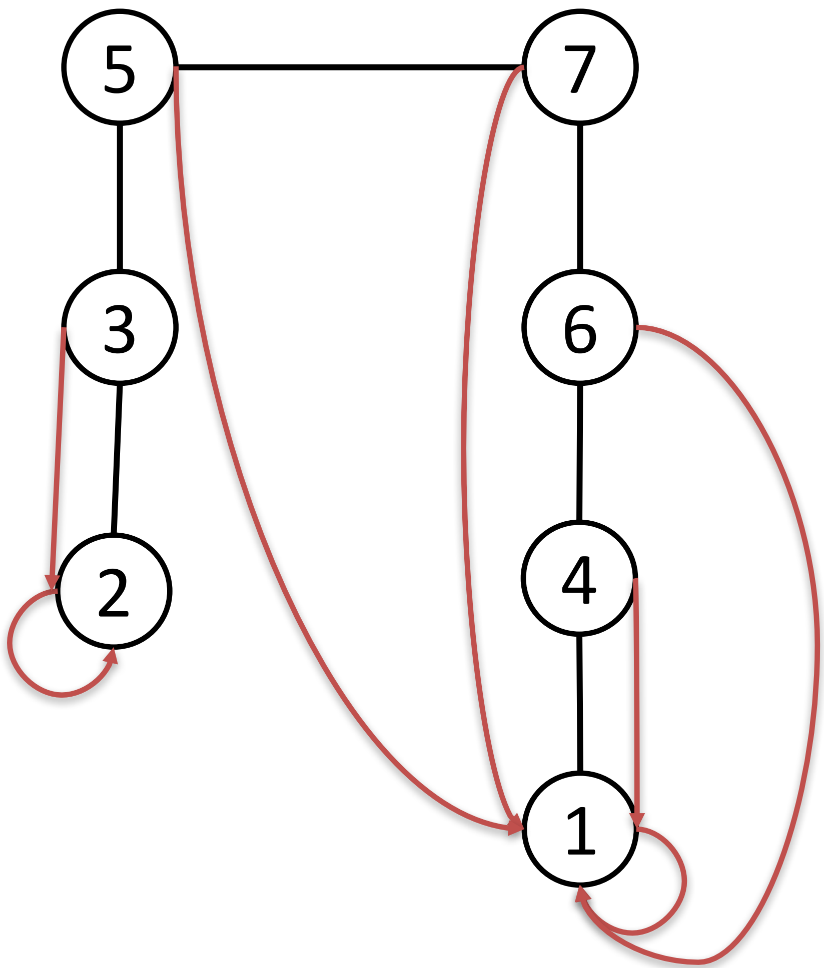
7 5



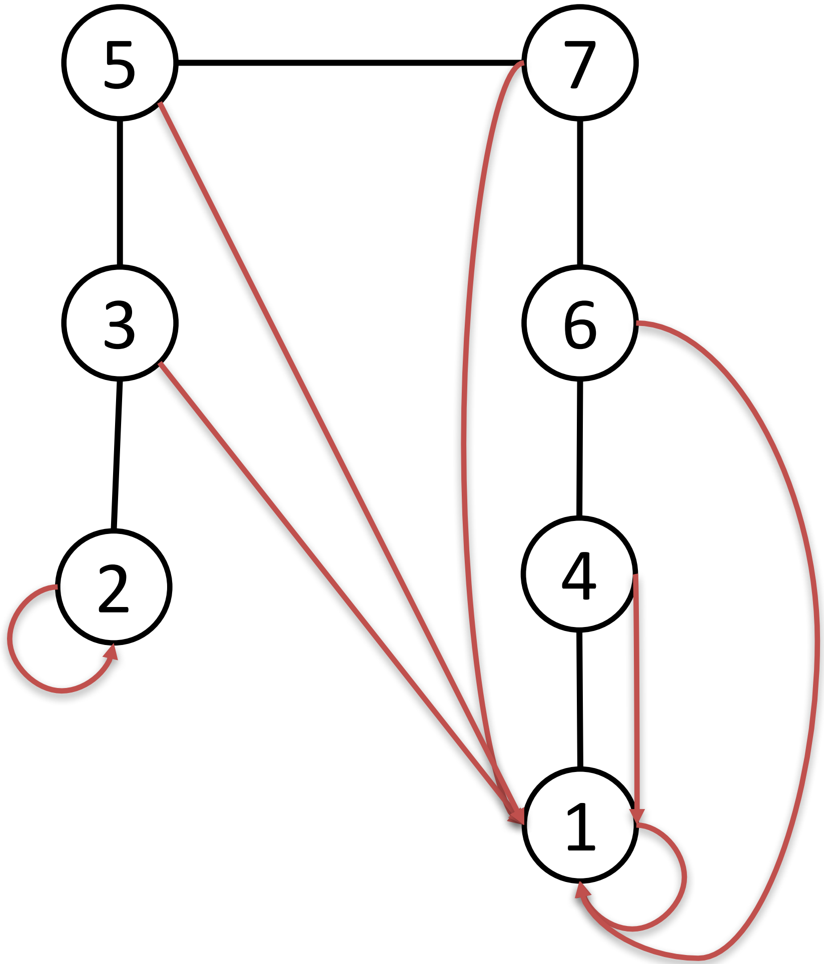
1	1	1
2	2	2
3	2	2
4	1	1
5	3	2
6	4	1
7	5	3



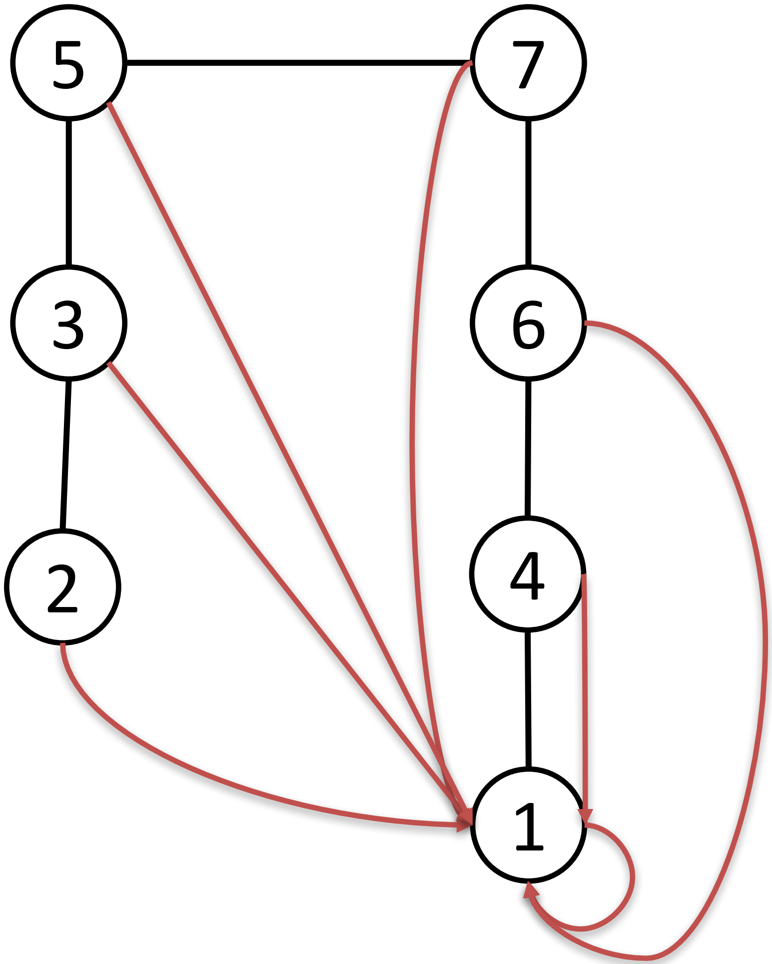
1	1	1	1
2	2	2	2
3	2	2	2
4	1	1	1
5	3	2	2
6	4	1	1
7	5	3	1



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
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

How many steps?

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

This algorithm does concurrent breadth-first search from smallest vertices in components (plus extra work)

Slow on high-diameter graphs

Why think of labels as parents?

The vertices v and the arcs $(v, v.p)$ define a directed graph (digraph)

If the only cycles are loops (arcs of the form (v, v)), the digraph consists of a set of rooted trees:

v is a root iff $v = v.p$

$v.p$ is the parent of v if $v \neq v.p$

If labels never increase, all cycles are loops

Flat tree: the parent of each vertex is the root.

Faster?

Shortcut (also called compress, halve, pointer jumping):

for each v do $v.p_{i+1} \leftarrow v.p_i.p_i$

$i \leftarrow i + 1$

A shortcut roughly halves the depths of all vertices

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

Simple labeling algorithms

Initialization followed by rounds, each a connect, one or more shortcuts, and possibly an edge alteration, repeated until no parent changes

connect: update parents based on arcs

shortcut: replace parents by grandparents

alter: replace arcs

Ways to connect

Given (v, w), replace $v.p$ or $v.p.p$ by w or $w.p$ (if smaller than current value).

direct-connect:

for each (v, w) do $v.p_{i+1} \leftarrow \min\{v.p_{i+1}, w\}$

$i \leftarrow i + 1$

parent-connect:

for each (v, w) do $v.p_i.p_{i+1} \leftarrow \min\{v.p_i.p_{i+1}, w.p_i\}$

$i \leftarrow i + 1$

Arc Alteration

alter:

  for each arc (v, w) do

    if $v.p_i \neq w.p_i$ then replace (v, w) by $(v.p_i, w.p_i)$

    else delete (v, w)

Other kinds of arc updates

sparsify: delete arcs

densify: add arcs

Algorithm C (for connect)

C: repeat

    parent-connect

    shortcut

  until no parent changes

Algorithm A (for alter)

A: repeat

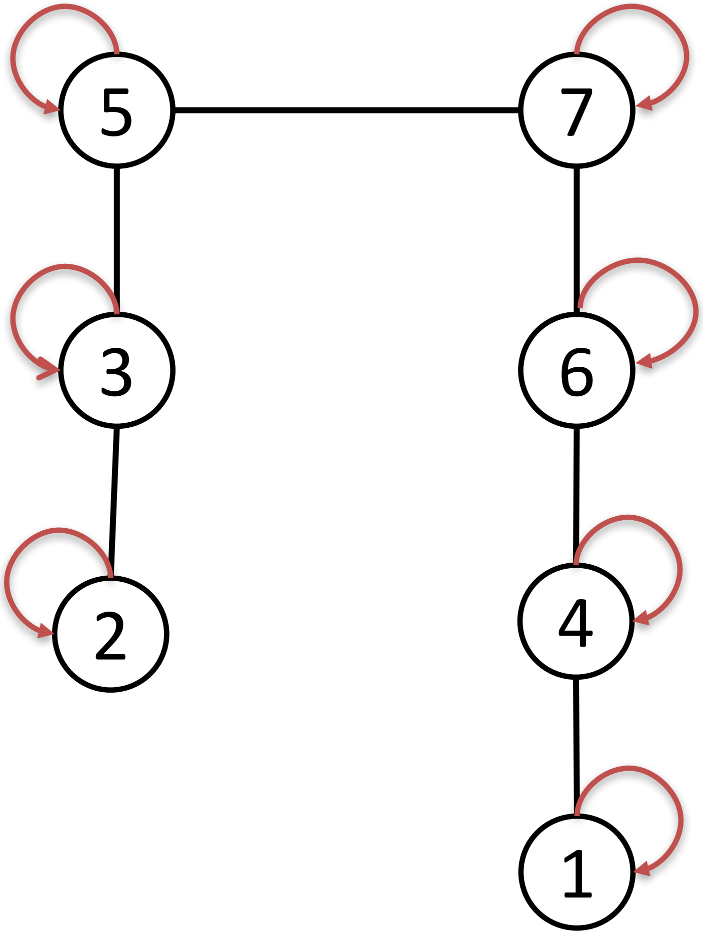
   direct-connect

   shortcut

   alter

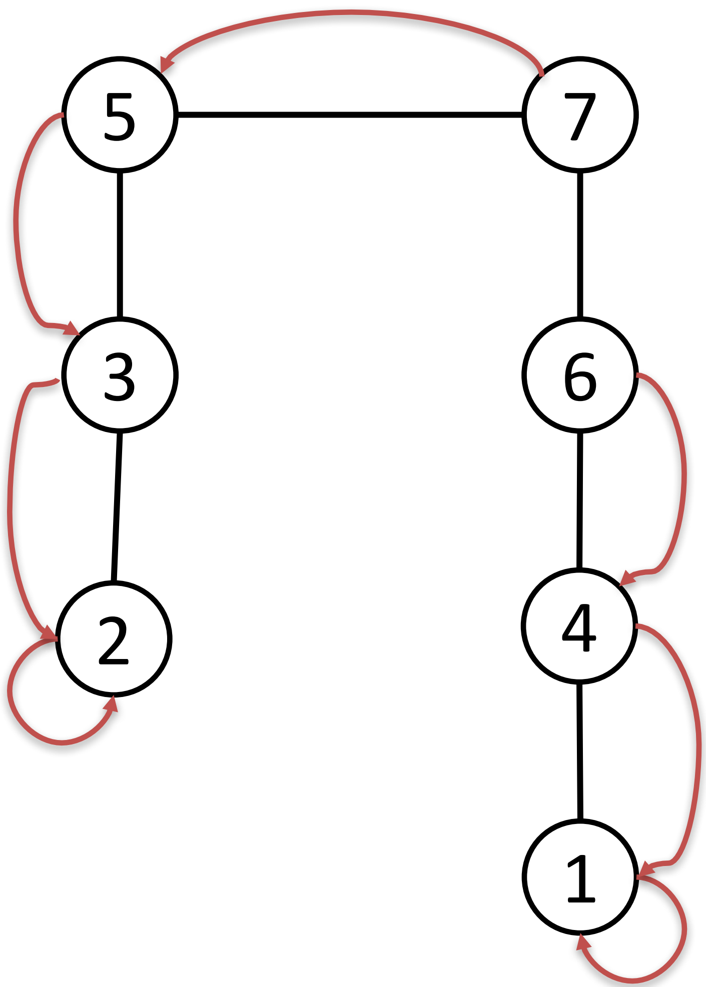
  until no parent changes

Algorithm A



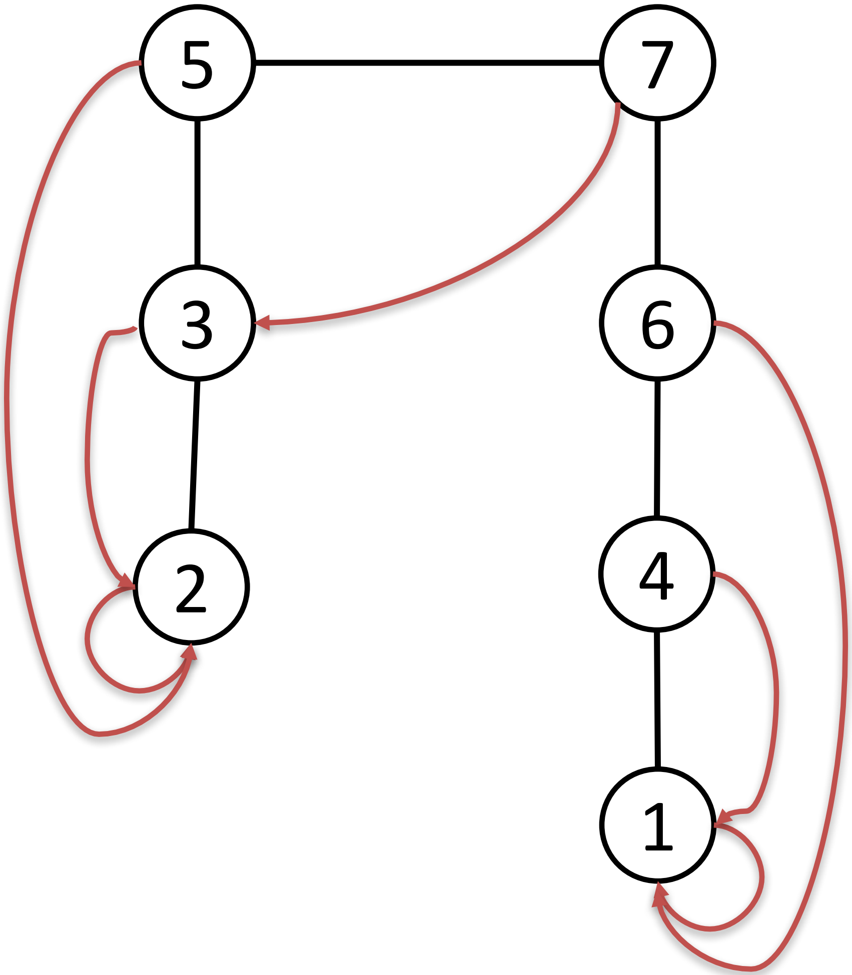
1
2
3
4
5
6
7

Algorithm A



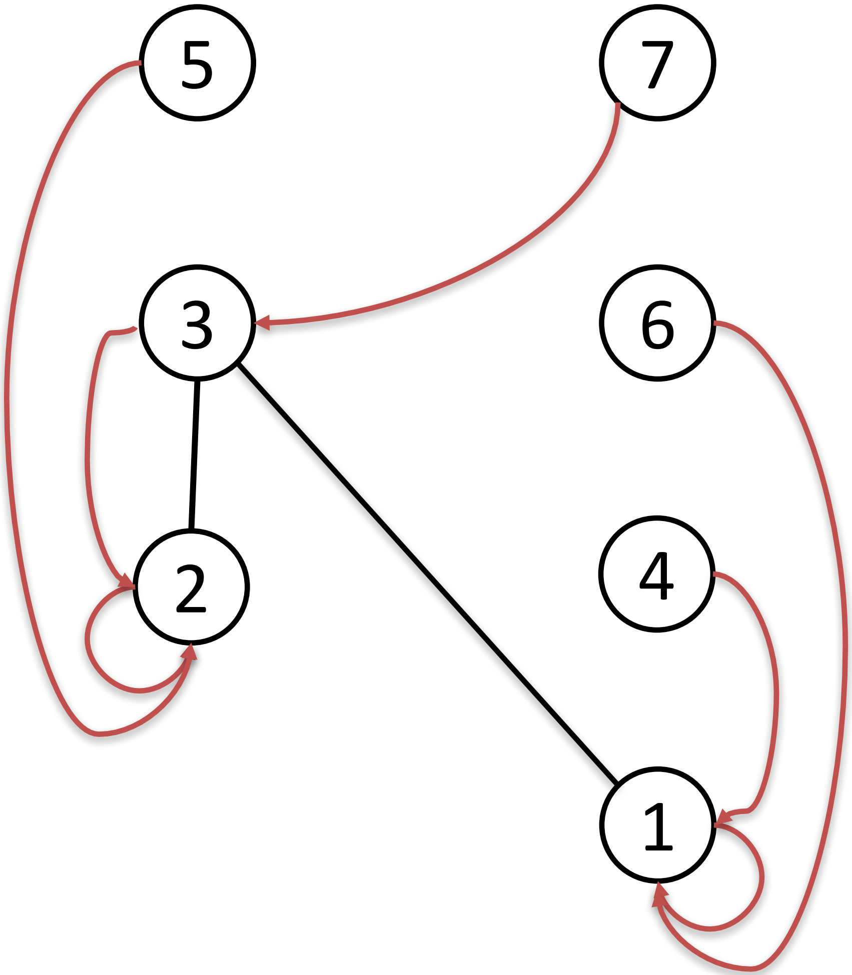
	D
1	1
2	2
3	2
4	1
5	3
6	4
7	5

Algorithm A



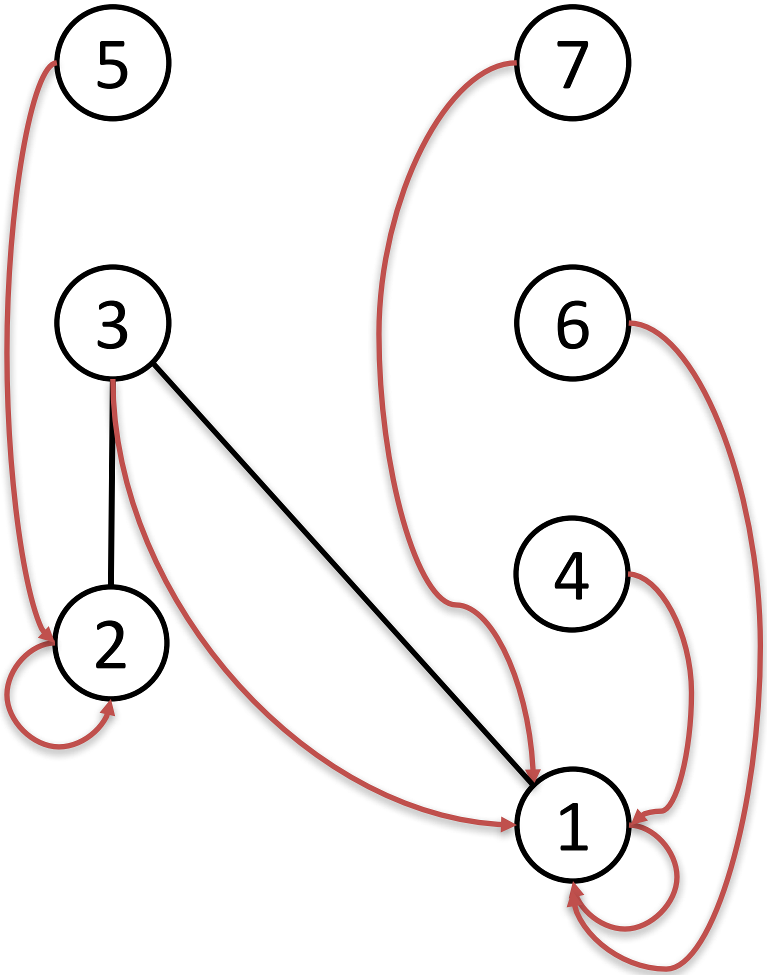
	D	S
1	1	1
2	2	2
3	2	2
4	1	1
5	3	2
6	4	1
7	5	3

Algorithm A



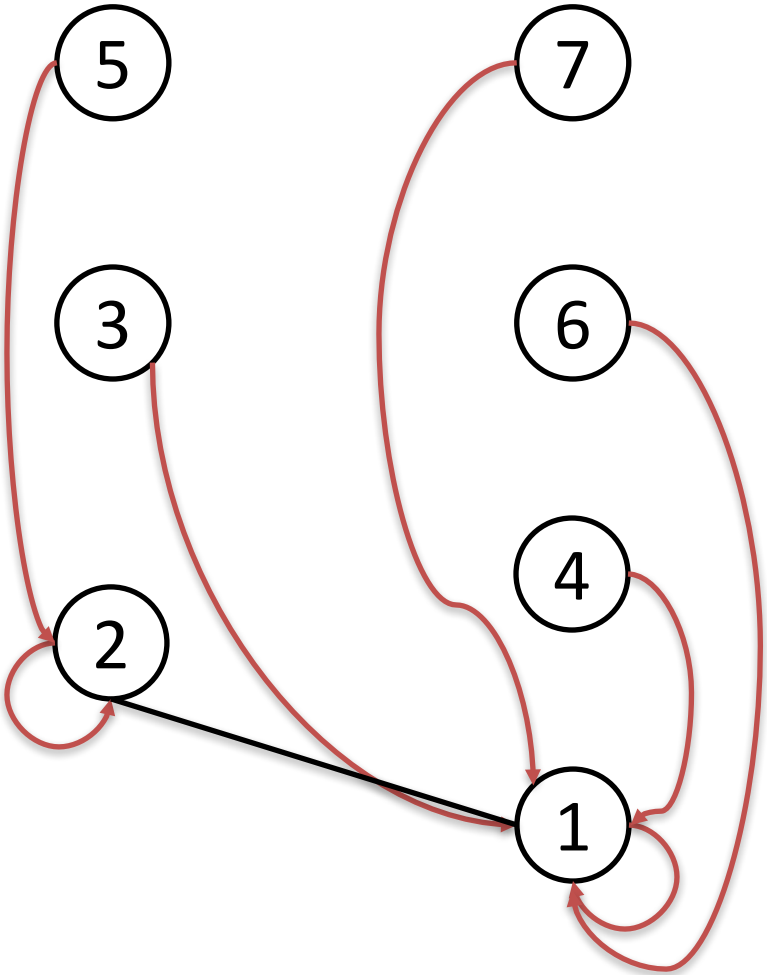
	D	S	A
	1	1	1
	2	2	2
	3	2	2
	4	1	1
	5	3	2
	6	4	1
	7	5	3

Algorithm A



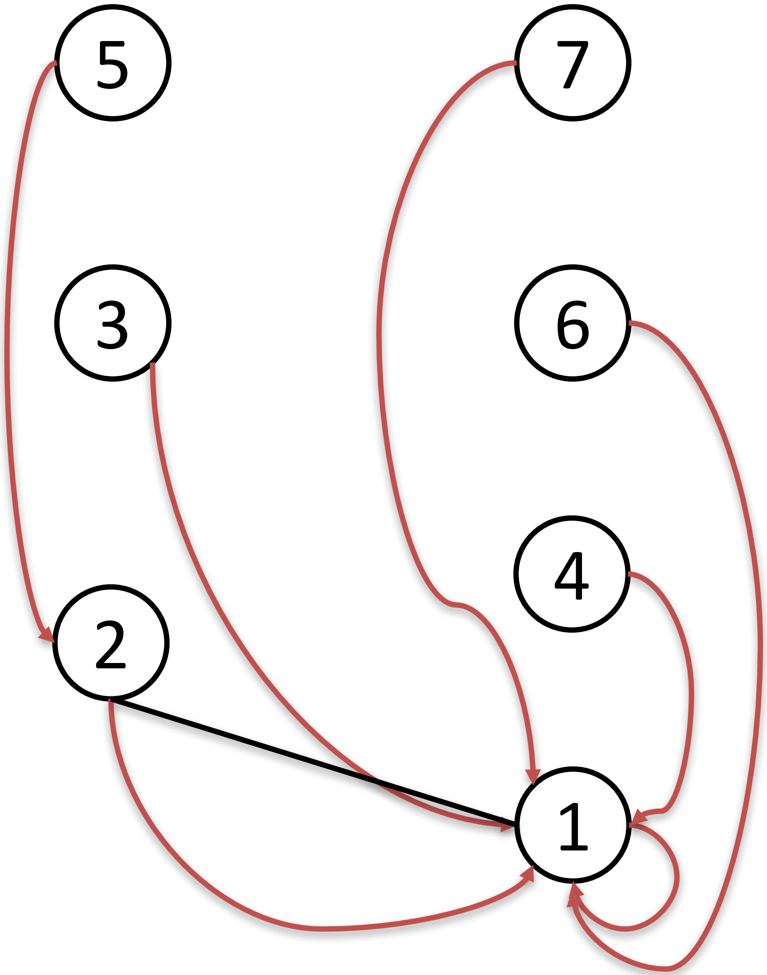
	D	S	A	D	S
1	1	1		1	1
2	2	2		2	2
3	2	2		1	1
4	1	1		1	1
5	3	2		2	2
6	4	1		1	1
7	5	3		3	1

Algorithm A



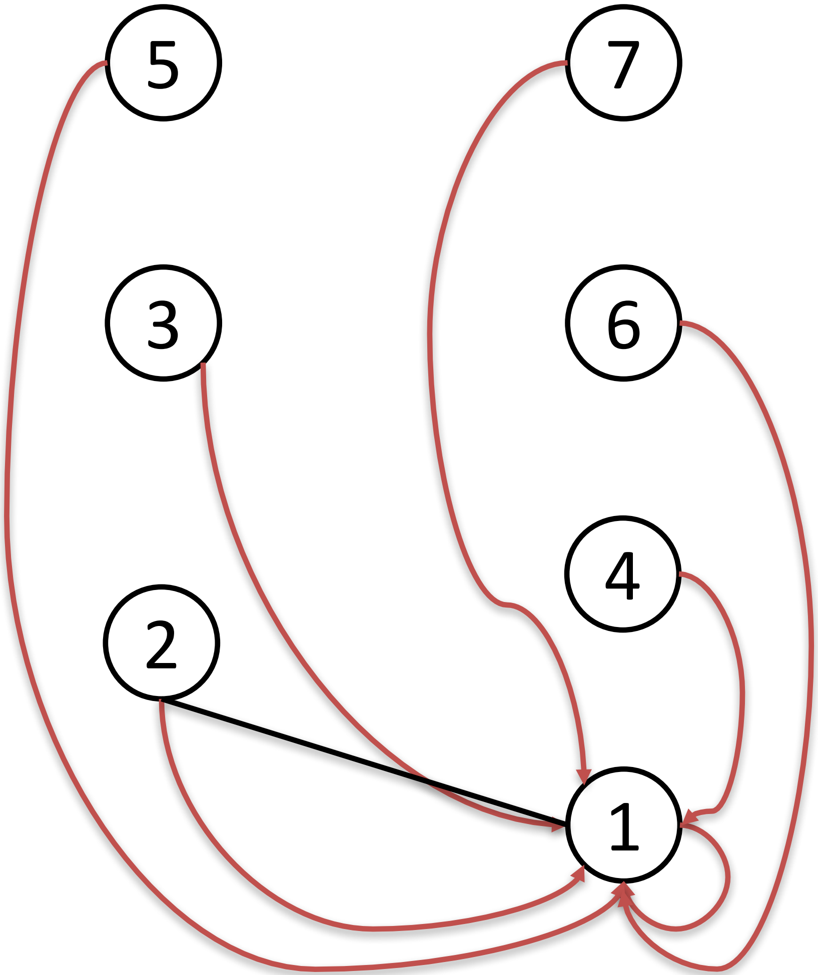
	D	S	A	D	S	A
1	1	1		1	1	
2	2	2		2	2	
3	2	2		1	1	
4	1	1		1	1	
5	3	2		2	2	
6	4	1		1	1	
7	5	3		3	1	

Algorithm A



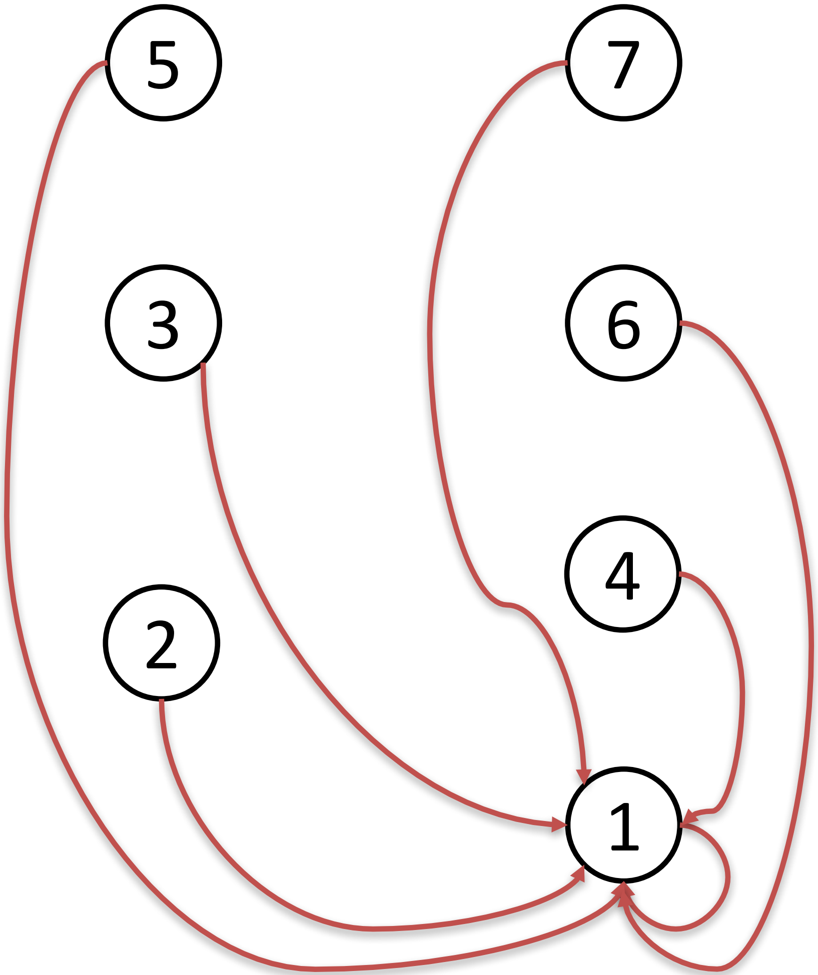
	D	S	A	D	S	A	D
1	1	1		1	1		1
2	2	2		2	2		1
3	2	2		1	1.		1
4	1	1		1	1		1
5	3	2		2	2		2
6	4	1		1	1		1
7	5	3		3	1		1

Algorithm A



	D	S	A	D	S	A	D	S
1	1	1		1	1		1	1
2	2	2		2	2		1	1
3	2	2		1	1		1	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	1		1	1

Algorithm A



	D	S	A	D	S	A	D	S	A
1	1	1		1	1		1	1	
2	2	2		2	2		1	1	
3	2	2		1	1		1	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	1		1	1	

Possible drawback?

Algorithm A maintains trees (labels only decrease)

But it can split a tree (by moving a subtree)

We call an algorithm monotonic if it does not split trees

Possible solution: when connecting, only change parents of roots

Root connection

When connecting, only change parents of roots

parent-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

    parent-root-connect

    shortcut

  until no parent changes

Possible drawback?

Connects can produce deep trees, delaying further connections being until shortcuts flatten the trees

Possible solution: repeated shortcuts

Algorithm S (for repeated shortcut)

S: repeat

parent-connect

repeat shortcut until no parent changes

until no parent changes

Surprisingly, algorithms C, A, R, and S are new (as far as we can tell)

How many steps?

A little history

First era

1980's – 2000's

Theoreticians

PRAM (parallel random access machine)

Goal: minimize time and total work (even if at the expense of algorithm complication)

Best: O(log$n$) steps, m/log$n$ processors,
randomized (Halperin & Zwick 1996, 2001)

Second era

1990's – present

Practitioners

Distributed (message-passing) model or a variant, based on new distributed computing frameworks: MAPREDUCE, HADOOP, etc.

Goal: speed in practice - algorithm needs to be implementable by a competent programmer

Dismissal of existing PRAM algorithms as too complicated or not implementable on distributed model

Invention of “simpler” algorithms, but with flawed proofs of resource bounds

Origins of our algorithms

Algorithm R simplifies a classical PRAM algorithm of Shiloach & Vishkin, 1982:

- Arbitrary resolution of write conflicts
- Maintains trees and is monotone
- Does not do minimum labeling
- Two shortcuts per round (not needed?)
- Extra steps guarantee that each round combines every flat tree (height at most 1) with some other tree
- O(log*n*) steps, analysis not simple

S & V show that a simplified version of their algorithm takes $\Omega(d)$ steps if write conflict resolution is arbitrary

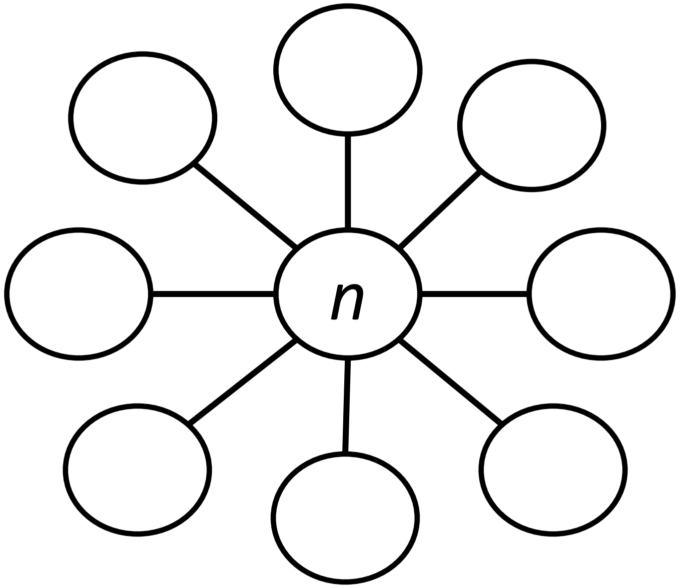
The same example is bad for algorithm R with arbitrary write conflict resolution

➡ To get a simpler algorithm, need stronger write conflict resolution

Algorithm S simplifies Greiner's Hybrid algorithm (1994)

- Each round repeats shortcuts until all trees are flat
- Uses direct-connect and alter, but alternate rounds use max value, not min value, to update parent
- Greiner claimed an $O(\lg^2 n)$ bound but in fact $\Omega(n)$, not even $O(d)$

Bad example for Greiner's algorithm



Algorithm A simplifies an algorithm of Stergio, Rughwani, and Tsioutsiouliklis, 2018:

- Extended connect step (implies O(d) rounds)
- Variant of shortcutting combines old and new labels
- No arc alteration

Their “proof” of O(lg$n$) steps is incorrect.

Solves problems on huge graphs fast in practice, on Hronos platform (clever handling of message contention, other optimizations)

Their paper got us started

Our bounds

S: $O(\lg^2 n)$ rounds worst-case

$O(\lg n \lg\lg n)$ average (random vertex numbers)

Correct expected bound $\Theta(\lg n)$?

R: $\Theta(\lg n)$ rounds worst-case

Analysis uses a variant of the potential function of A & S, novel multi-round analysis: flat trees may not change for many rounds

A: $O(\lg^2 n)$ worst-case

Correct bound $\Theta(\lg n)$?

Fewer stepss?

Andoni et al., 2018 give a complicated algorithm with an $O((\lg d)\lg\lg_{m/n} n)$ round bound on a powerful distributed model

We (Liu, Tarjan, Zhong) can simplify their algorithm and implement it on a PRAM with arbitrary resolution of write conflicts

Their key idea: careful densifying with random connect steps, flat trees

Our contribution: very sparse hash tables with very few collisions

The Latest

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

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

Asynchronous processes?

Recent work on concurrent disjoint set union by Jayanti, Tarjan, and Boix (PODC 2016, 2019) gives efficient asynchronous concurrent algorithms for connected components

Thanks!

For some details see our arXiv paper
(revision of our SOSA 2019 paper)
Maybe wait for next version (in process)