Concurrent Connected Components

Robert E. Tarjan
Princeton University and Intertrust Technologies

joint work with Sixue Liu, Princeton

University of Vienna
25 January 2019 (slides revised 30 January)

Observations

Over the last 60 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.

How should practitioners respond?

Bring experts in early: get feedback while there is still time to make changes.

Build on-going relationships: short-term gains are predictable, long-term gains not, but may be much more valuable.

Open two-way feedback is critical.

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"

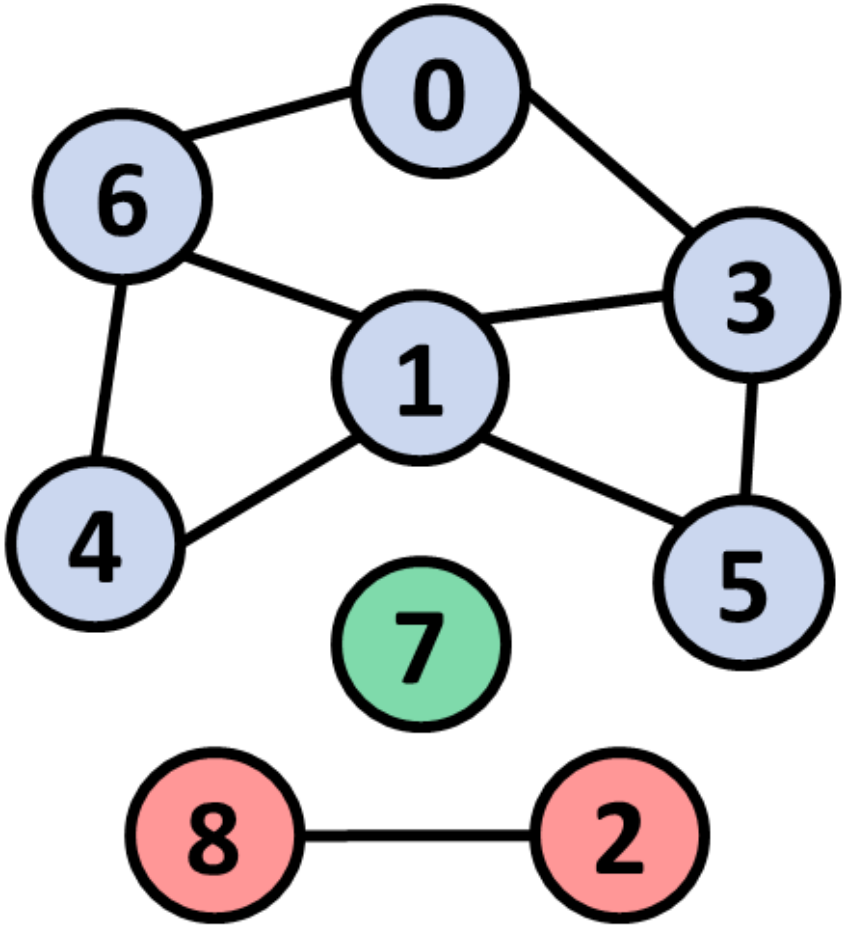
Connected Components

The most basic graph problem?

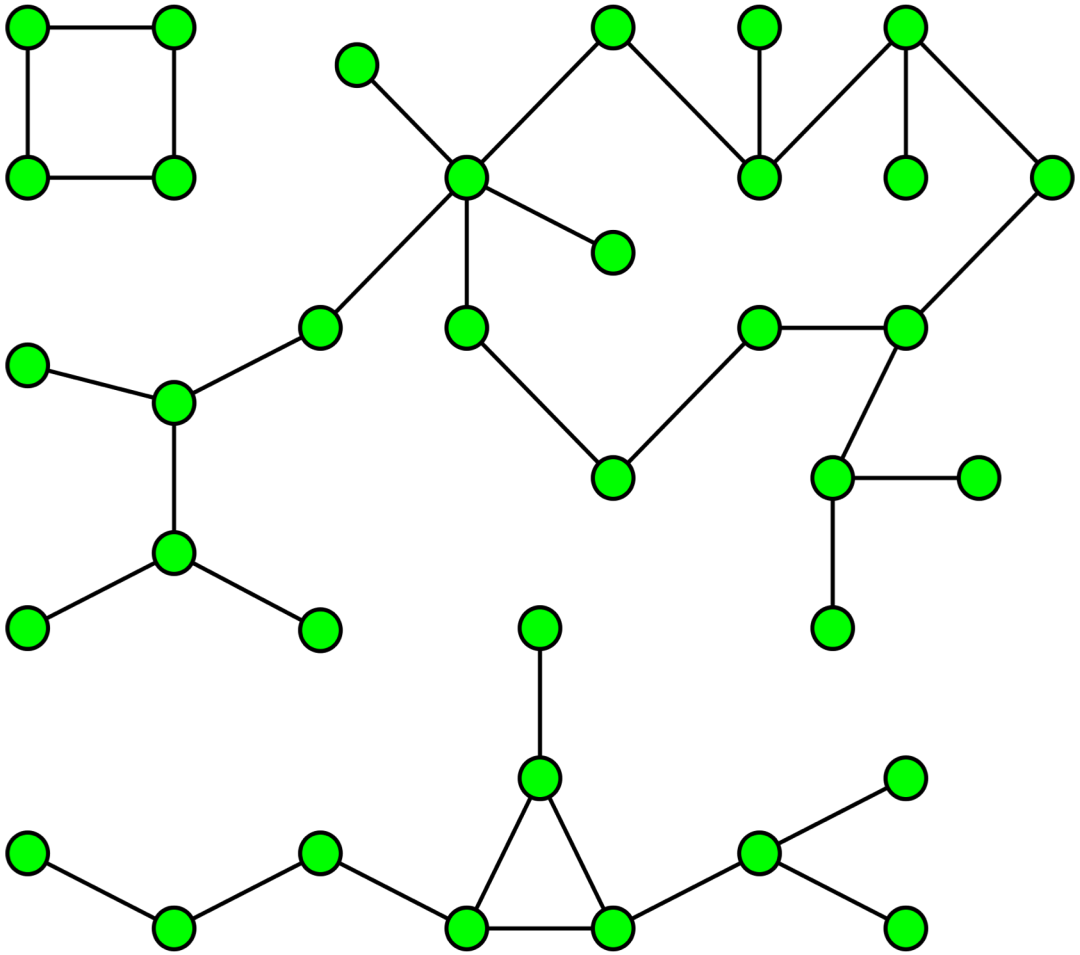
In an undirected graph, two vertices are connected if there is a path between them. A connected component (henceforth just a component) is a maximal set of pairwise-connected vertices.

Problem: Given a graph, compute its components.

[vitoshoacademy.com]

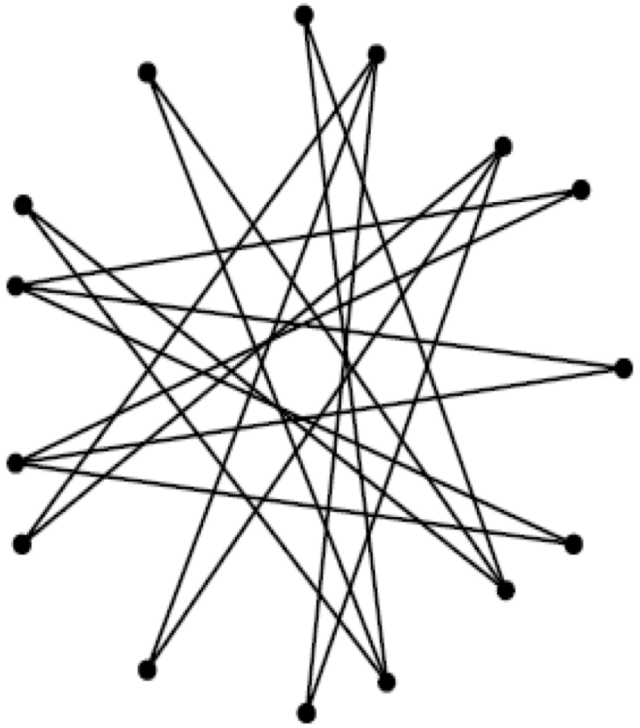


[figure from D. Eppstein]



[math.stackexchange]

A21 Is this graph connected or disconnected?



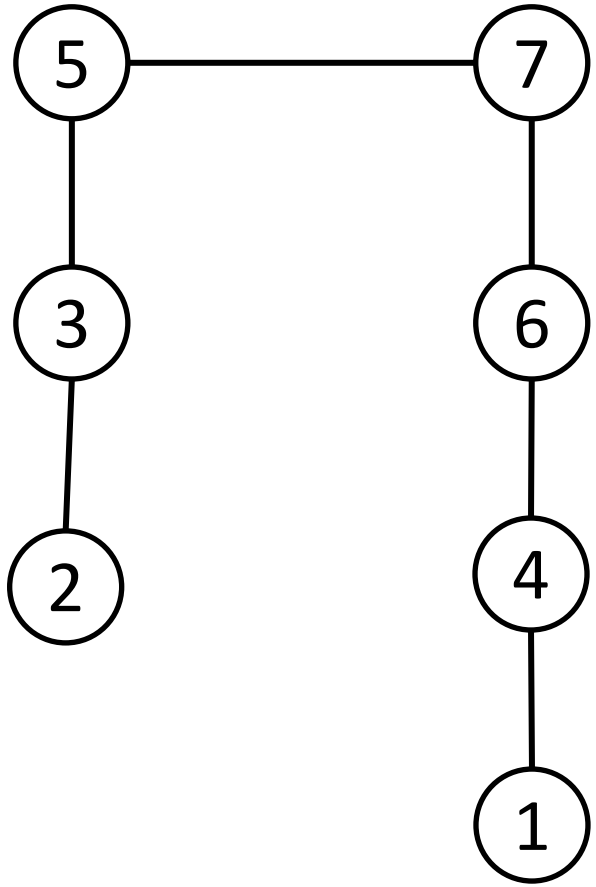
How to represent components?

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

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 - compressed trees with path compression and linking by rank:

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

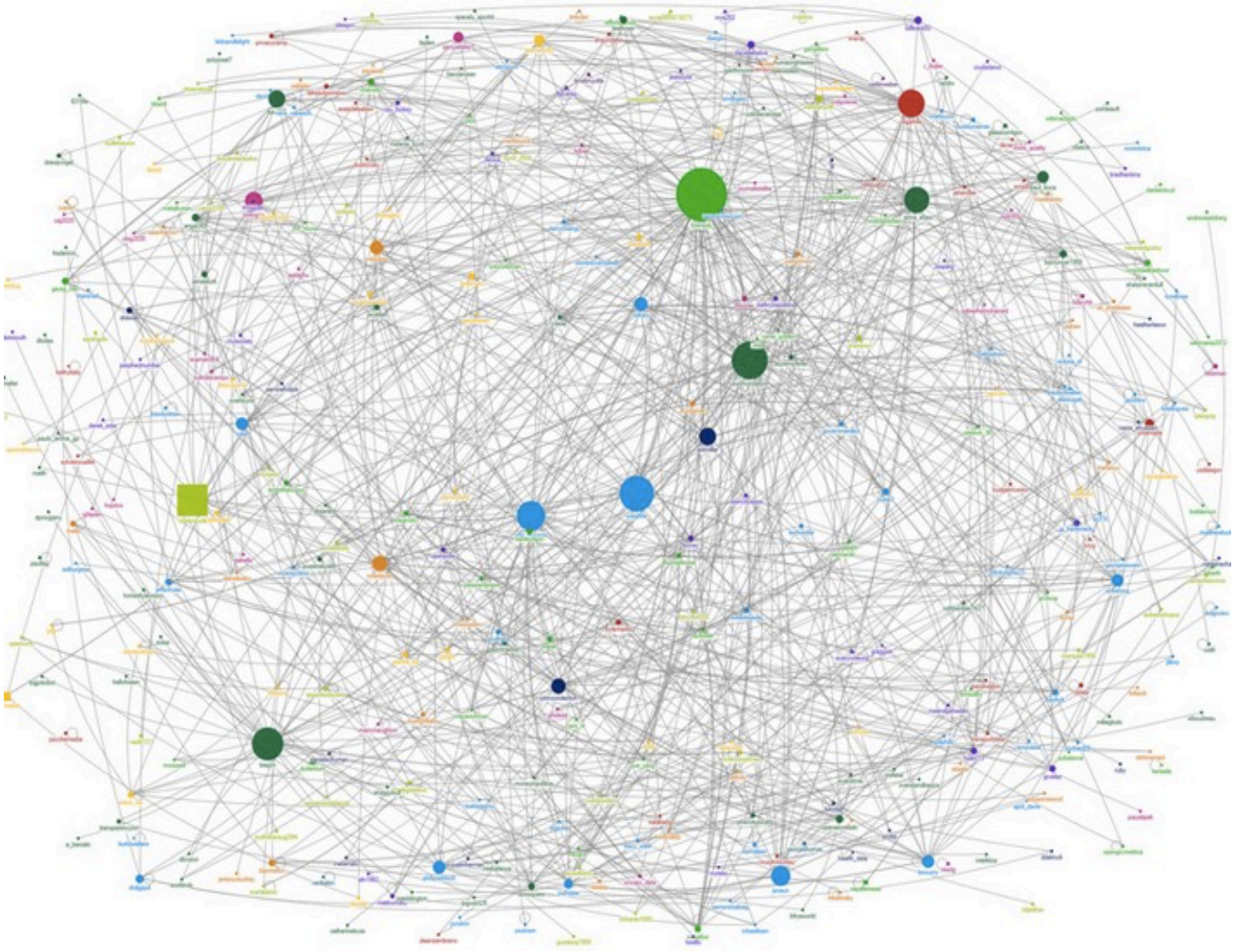
Disjoint set union uses only the edge set, supports individual and batch edge insertions, with intermixed queries

(inverse-Ackermann amortized time per unite or find, for all practical purposes constant)

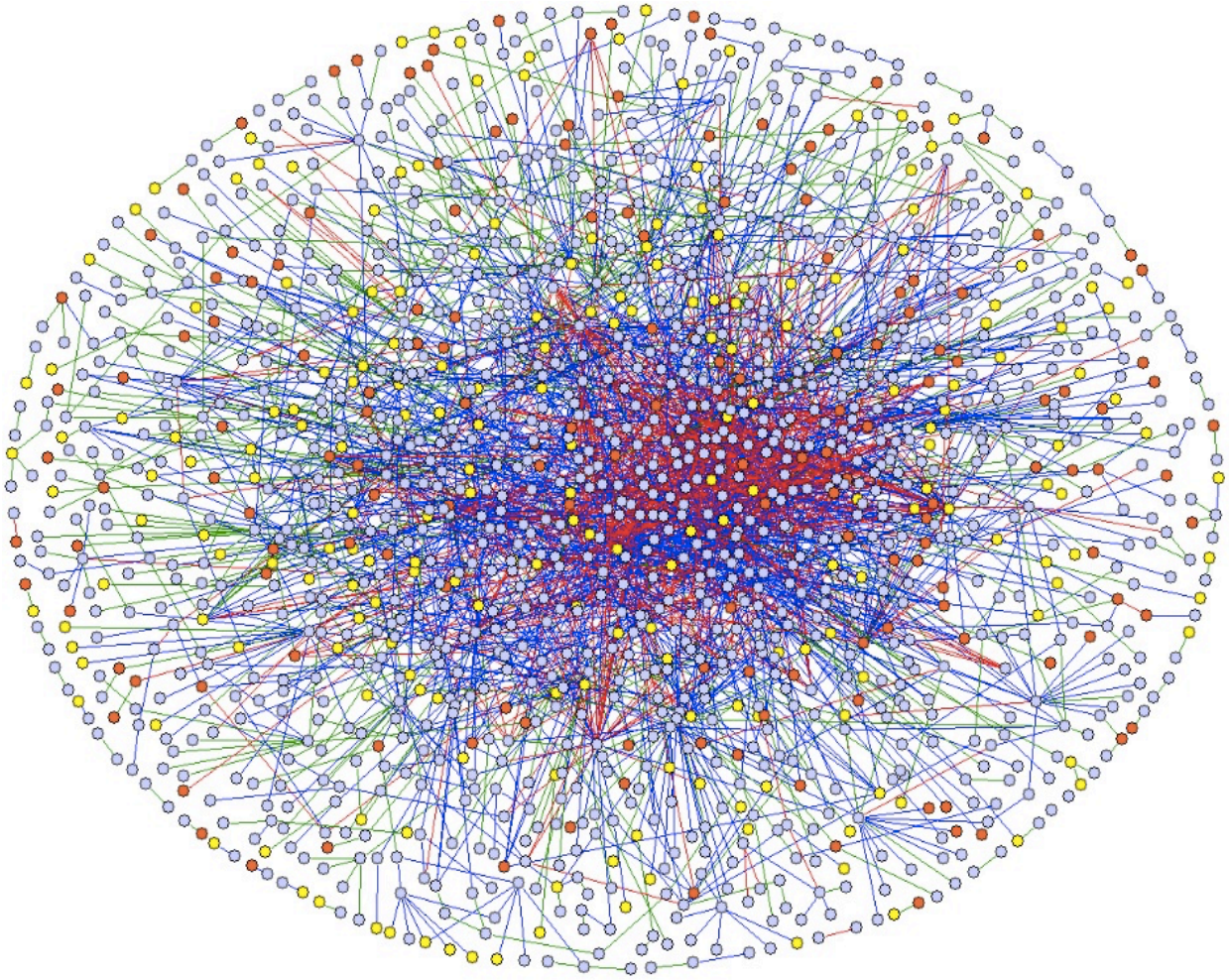
Is this the end of the story?

What if the graph is really big?

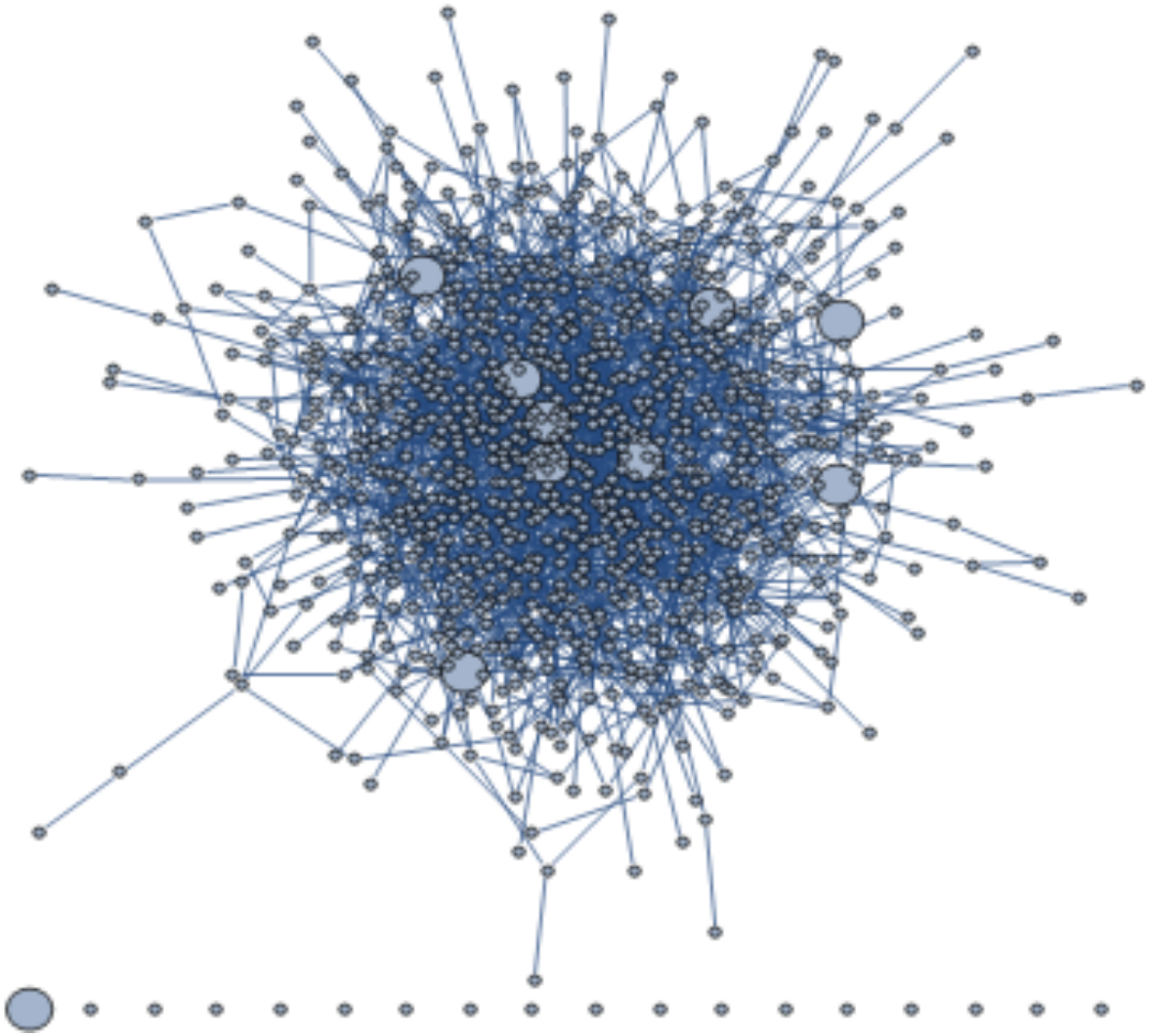
[beyondplm.com]



[Max Delbruck Center for Molecular Medicine]



[Stack exchange]



[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")

for each v do $v.p = v$;

repeat

  for each arc (v, w) do if $v.p < w.p$ then $w.p \leftarrow v.p$

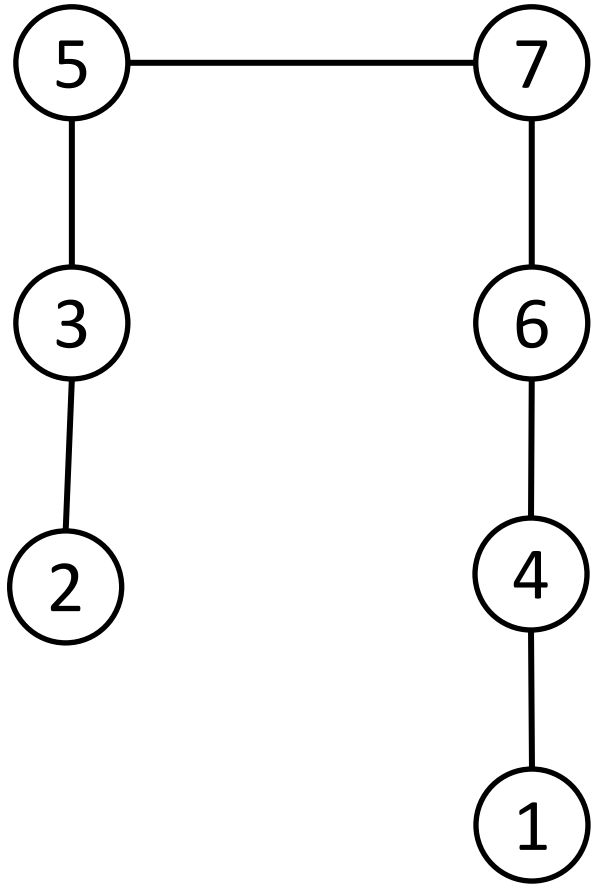
until no parent changes

Arcs (v, w) and (w, v) represent edge $\{v, w\}$

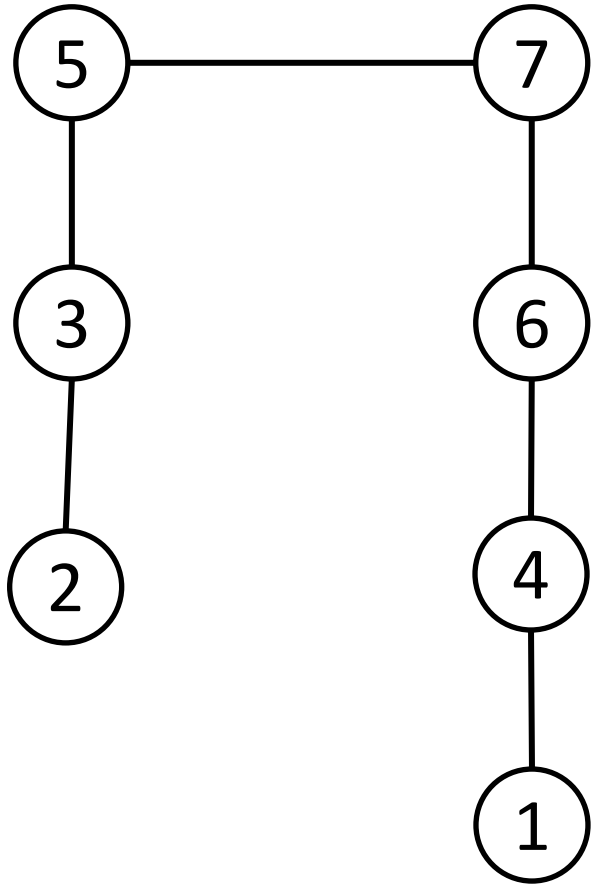
Loops run synchronously in parallel

Write conflicts resolved in favor of smallest value

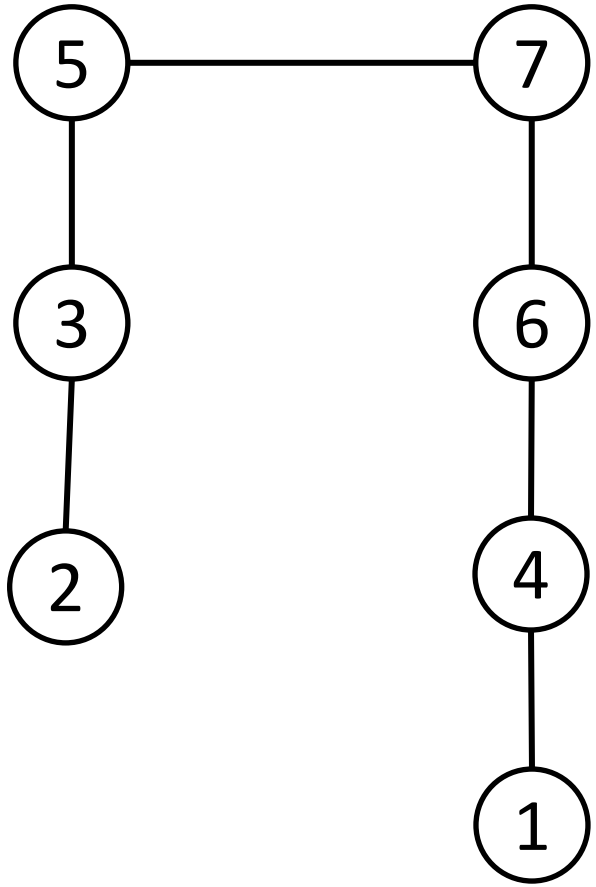
$v.p$ is the label of v ("p" for "parent")



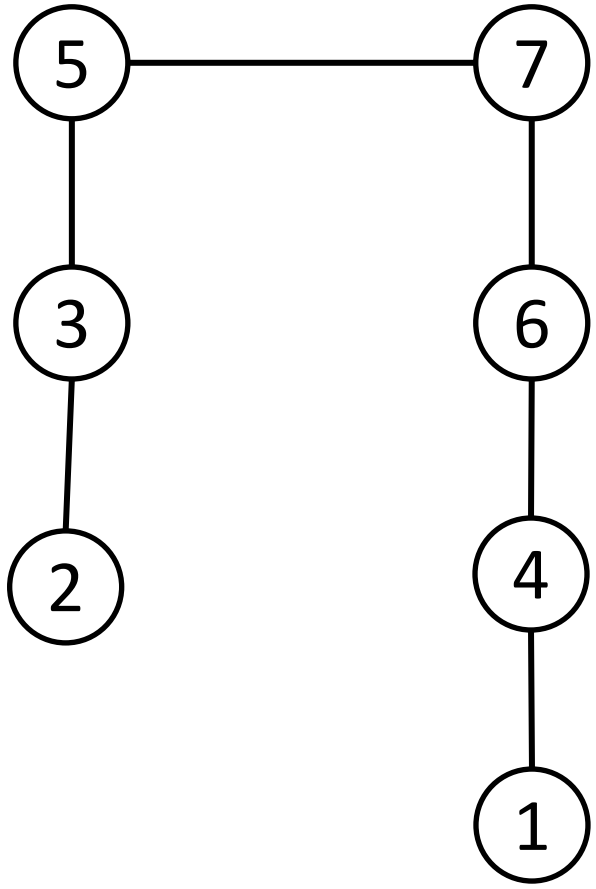
1
2
3
4
5
6
7



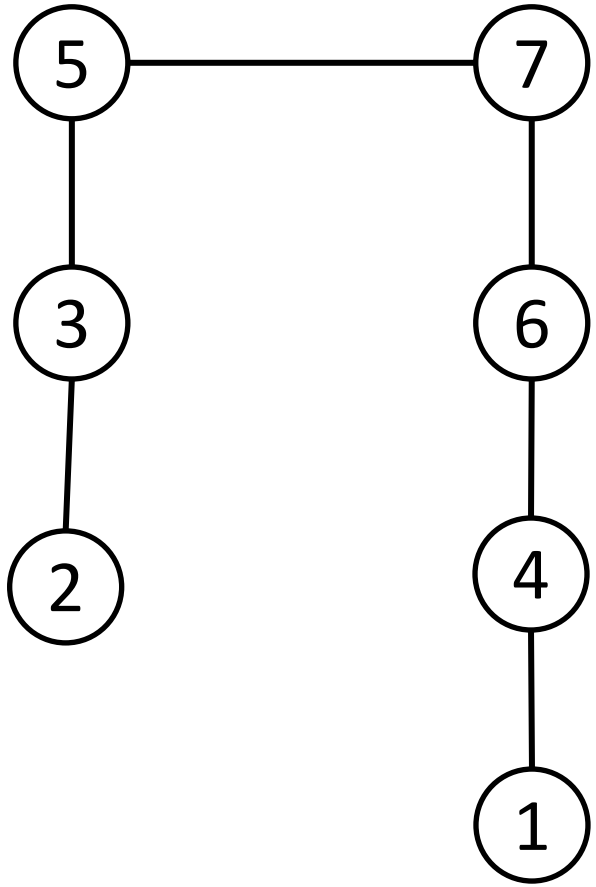
1 1
2 2
3 2
4 1
5 3
6 4
7 5



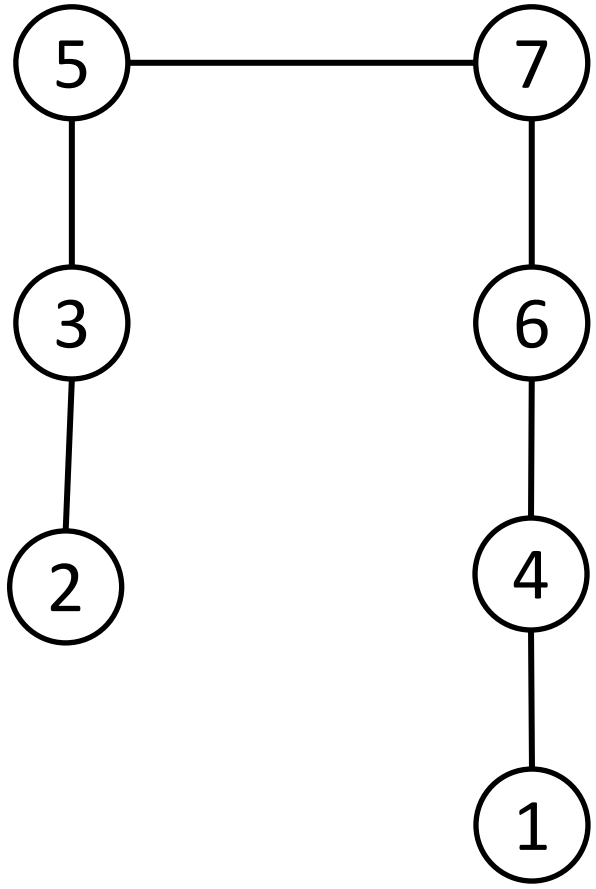
1 1 1
2 2 2
3 2 2
4 1 1
5 3 2
6 4 1
7 5 3



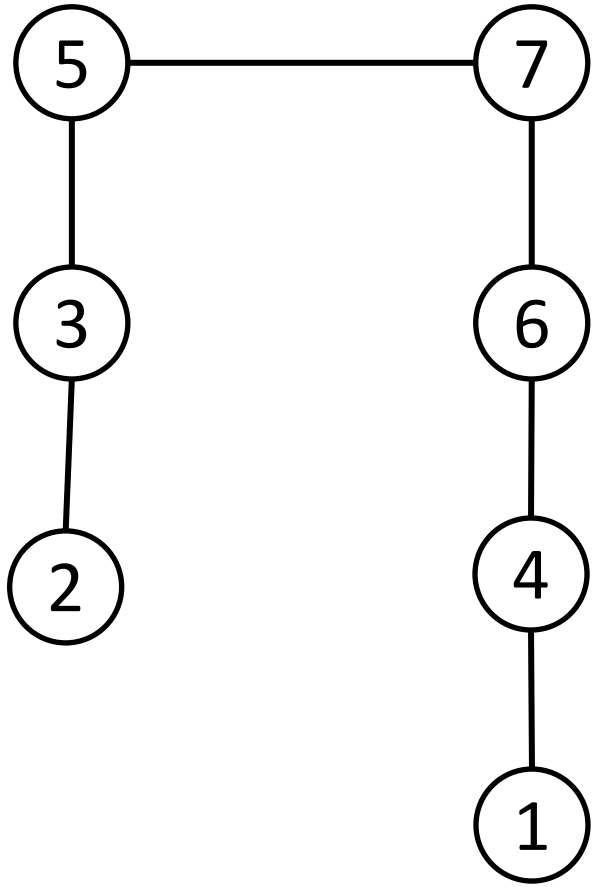
1 1 1 1
2 2 2 2
3 2 2 2
4 1 1 1
5 3 2 2
6 4 1 1
7 5 3 1



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

Why think of labels as parents?

The vertices *v* and the arcs (*v*, *v.label*) 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 $v = v.label$

v.label is the parent of *v* if $v \neq v.label$

If labels never increase, all cycles are loops

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

How many rounds?

How many rounds?

$\Theta(d)$ 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

Faster?

Shortcut (also called compress, halve):

for each v do $v.p = v.p.p$

A shortcut roughly halves the depths of all vertices

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

Algorithm C (for Connect)

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

repeat

{ C: for each (v, w) do if $v.p < w.p$ then $w.p \leftarrow v.p$;

  S: for each v do $v.p \leftarrow v.p.p$;}

until no parent changes

(not in SOSA paper)

Algorithm A (for Arc Alteration)

for each v do $v.p = v$;

repeat

{ C: for each (v, w) do if $v < w.p$ then $w.p = v$;

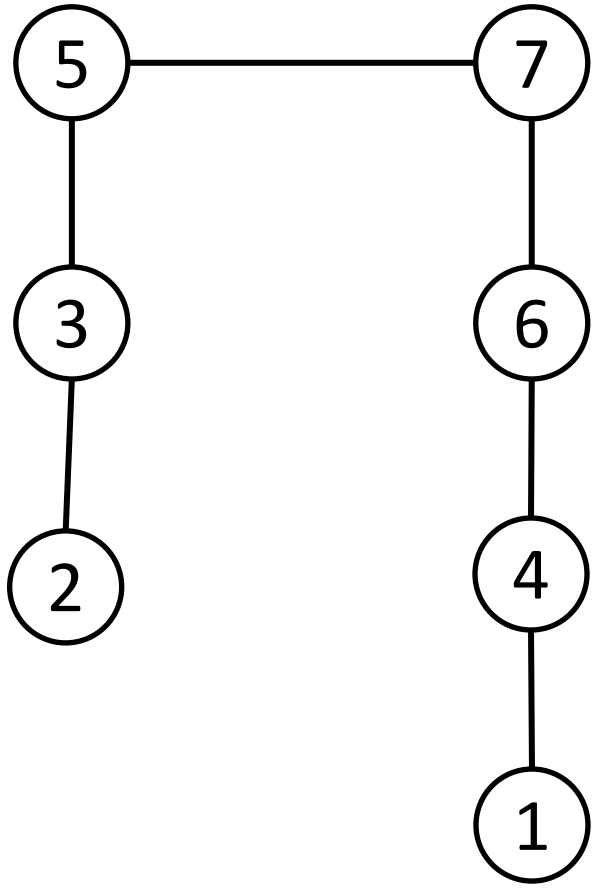
S: for each v do $v.p = v.p.p$;

A: for each (v, w) do if $v.p \neq w.p$

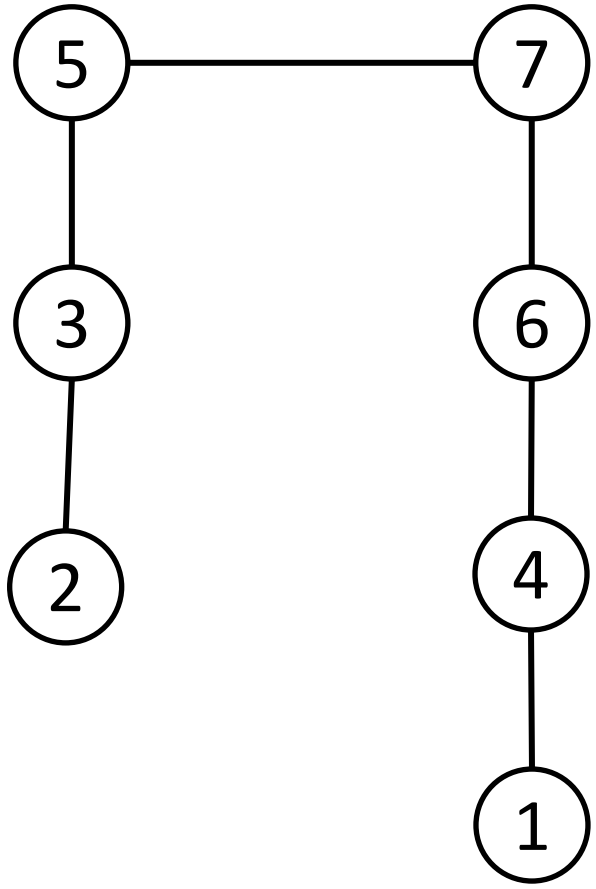
then replace (v, w) by $(v.p, w.p)$

else delete (v, w)} until no parent changes

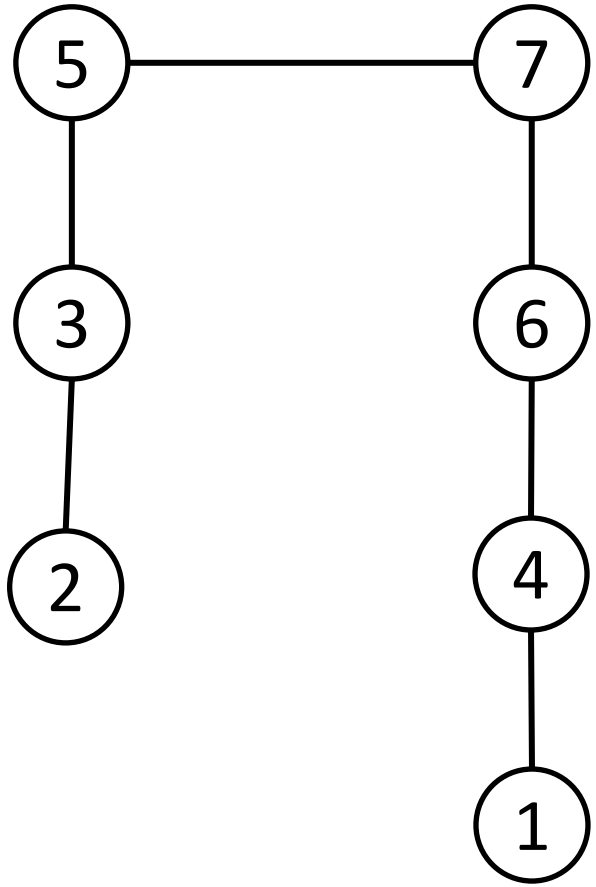
Possible advantage vs. algorithm A: #edges decreases as algorithm proceeds



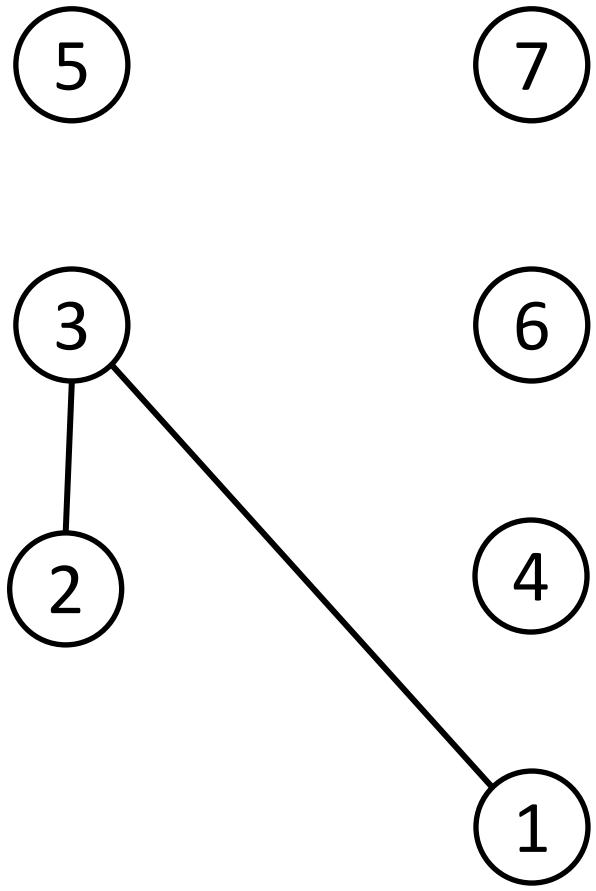
1
2
3
4
5
6
7



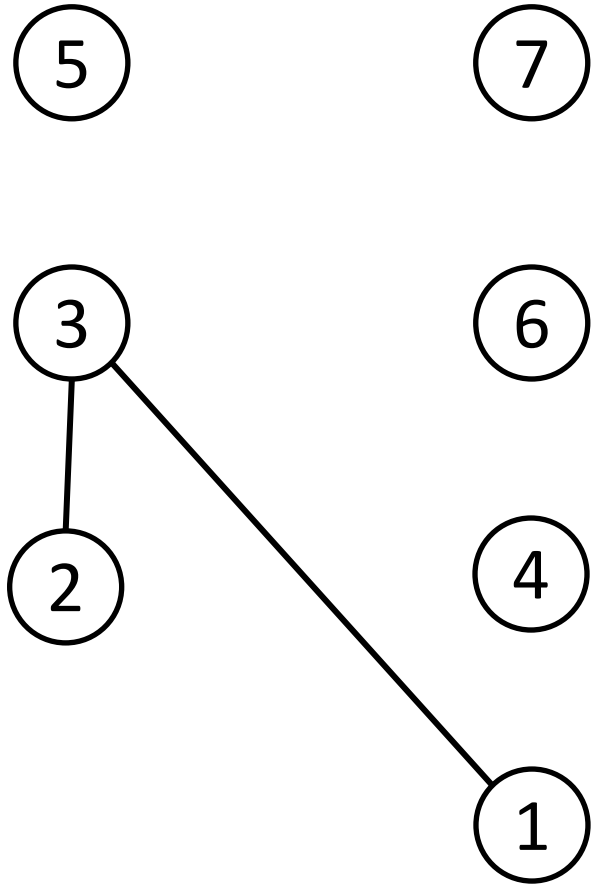
	C
1	1
2	2
3	2
4	1
5	3
6	4
7	5



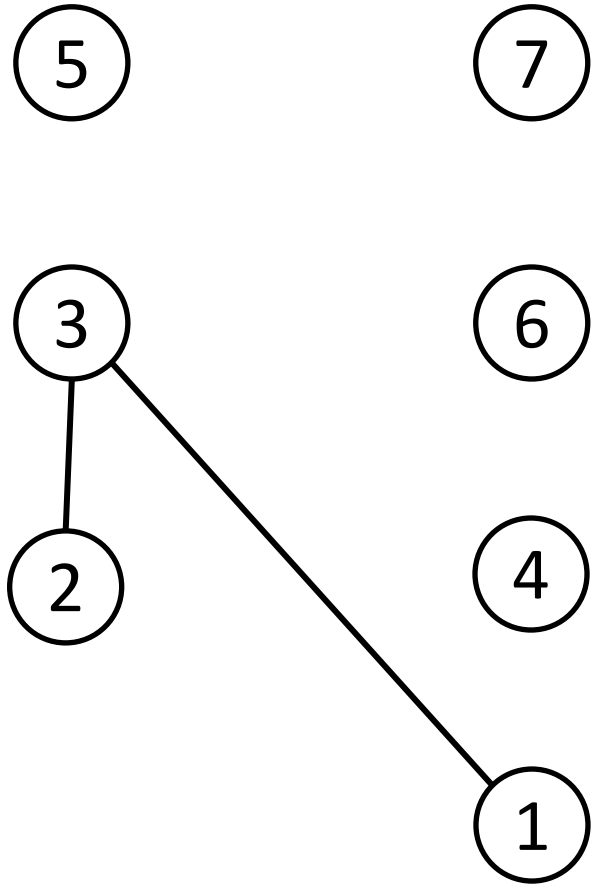
	C	S
1	1	1
2	2	2
3	2	2
4	1	1
5	3	2
6	4	1
7	5	3



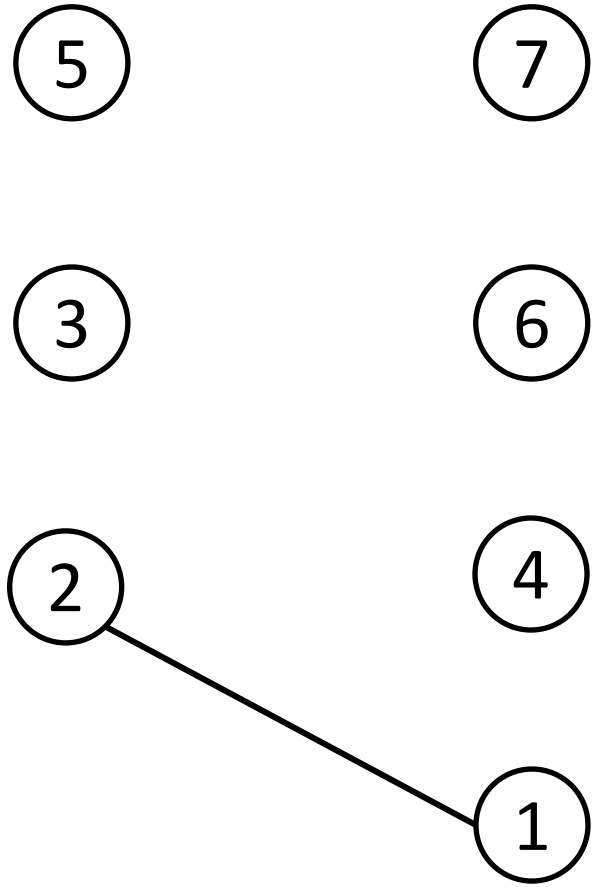
C	S	A
1	1	1
2	2	2
3	2	2
4	1	1
5	3	2
6	4	1
7	5	3



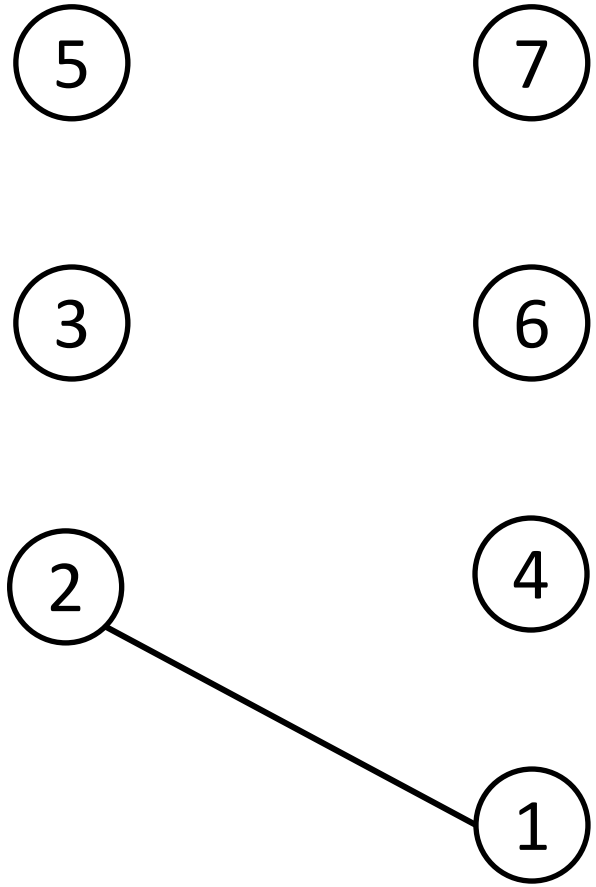
	C	S	A	C
1	1	1		1
2	2	2		2
3	2	2		1
4	1	1		1
5	3	2		2
6	4	1		1
7	5	3		3



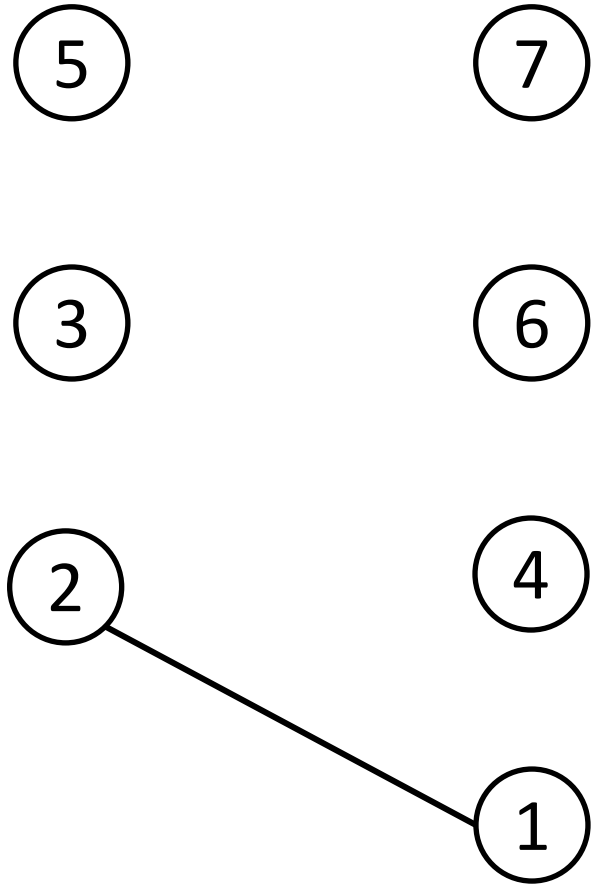
	C	S	A	C	S
1	1	1		1	1
2	2	2		2	2
3	2	2		1	1
4	1	1		1	1
5	3	2		2	2
6	4	1		1	1
7	5	3		3	1



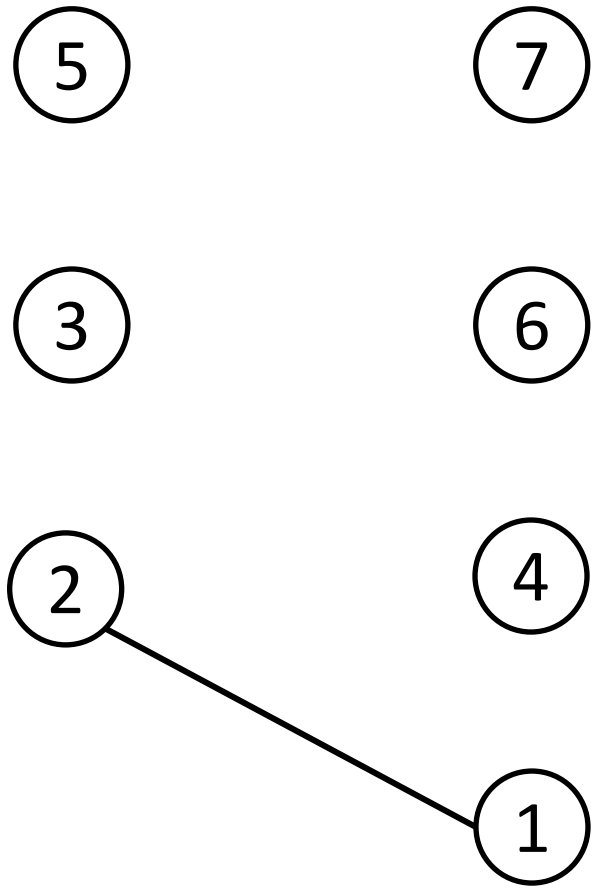
	C	S	A	C	S	A
1	1	1		1	1	
2	2	2		2	2	
3	2	2		1	1	
4	1	1		1	1	
5	3	2		2	2	
6	4	1		1	1	
7	5	3		3	1	



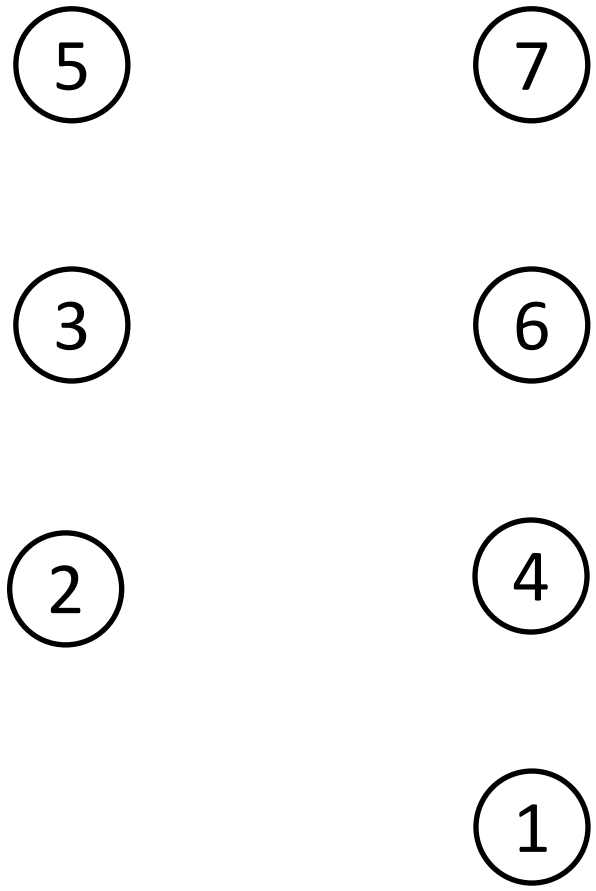
	C	S	A	C	S	A
1	1	1		1	1	
2	2	2		2	2	
3	2	2		1	1	
4	1	1		1	1	
5	3	2		2	2	
6	4	1		1	1	
7	5	3		3	1	



	C	S	A	C	S	A	C
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



C	S	A	C	S	A	C	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



C	S	A	C	S	A	C	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?

Algorithms C and A maintain trees (labels only decrease)

But they can split a tentative component (by moving a subtree)

We call an algorithm monotonic if it does not split tentative components

If non-monotonicity is a drawback, what is a solution?

Algorithm R (for root-connect)

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

repeat

{ R: for each (v, w) do if $v.p < w.p.p$ & $w.p.p = w.p$

then $w.p.p \leftarrow v.p$;

S: for each v do $v.p \leftarrow v.p.p$}

until no parent changes

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

How many rounds?

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)

PRAM

Each process has a small private memory, can access large shared memory

Processes run synchronously in lockstep, at least on reads and writes to common memory

PRAM variants handle read and write conflicts differently

PRAM variants

EREW (exclusive read, exclusive write): no concurrent reads of or writes to the same location in the common memory

CREW (concurrent read, exclusive write): concurrent reads allowed, no concurrent writes

CRCW (concurrent read, concurrent write): concurrent reads and writes allowed

Handling of write conflicts

Common: all concurrent writes to the same location must be of the same value

Arbitrary: among concurrent writes to the same location, an arbitrary one succeeds

Priority: among concurrent writes, the one done by the highest-priority process succeeds

Combining: values concurrently written to the same location are combined using some symmetric function, such as minimum or sum.

First-era work mostly used one of three models:

Arbitrary CRCW PRAM

CREW PRAM

EREW PRAM

Not a COMBINING CRCW PRAM

Notable results

Shiloach & Vishkin, 1982: Arbitrary CRCW PRAM algorithm, O(lg$n$) steps and O($(m + n)$lg$n$) work

Maintains trees and is monotone

Does not do minimum labeling

Two shortcuts per round

Extra steps to guarantee that each round combines every flat tree (height at most 1) with some other tree

Analysis is not straightforward

S & V example shows that a simpler version of their algorithm takes $\Omega(n)$ steps in the worst case

Their example works for algorithm R as well

Conclusion: To get a significantly simpler (deterministic) algorithm, COMBINING PRAM (stronger model) needed

Awerbuch and Shiloach, 1987: Variant of Shiloach-Vishkin algorithm, simpler, same bounds, simpler analysis

Reif, 1984: Simple randomized algorithm, O(lg$n$) steps and O($(m + n)$lg$n$) work, all trees flat

Johnson and Metaxis, 1997: O($\lg^{3/2} n$)-steps on a CREW PRAM; is monotonic, but does not maintain acyclicity: uses a variant of shortcutting to eliminate any cycles it creates

Many more-complicated algorithms with o(m+n) work bounds, using sparsification, edge alteration, process reassignment and other techniques

Halperin and Zwick, 1996, 2001: Two randomized EREW algorithms running in O(lg$n$) steps and O((m + n)/lg$n$) work. One of these finds spanning trees of the components.

All the PRAM algorithms we have found in the literature are monotonic

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

Distributed model

Each process has a local memory, no common global memory

Each round in lockstep, all processes can do one of two kinds of steps:

Send messages to other processes it knows about

Do arbitrary local computation

The model ignores both in-bound and out-bound message contention

Restriction to components problem

One process per edge and vertex

Initially, each edge process knows the processes of its ends, each vertex process knows nothing

$\Theta(\lg d)$ steps are needed to compute components

The $O(\lg d)$ algorithm sends many very large messages and hence is not practical (more later)

Algorithms C, A, and R use messages of $O(\lg n)$ bits, as do many of those in the literature

Three examples

Stergio, Rughwani, and Tsioutsiouliklis, 2018: algorithm like C but with an extended connect step and a variant of shortcutting that combines old and new labels

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)

This paper got us started

Yan, et al., 2014: algorithm in the PREGEL framework, simplified version of SV algorithm, claimed O(lg$n$) round bound, but on S & V example runs in $\Omega(n)$ rounds: they resolve write conflicts arbitrarily

Burkhardt, 2018: splits each edge into two arcs, alters these arcs separately, does a form of implicit shortcutting, claimed O(lg$d$) round bound, but counterexample of Andoni et al., 2018, shows false

Our bounds for C, A, and R

C & A: $O(\lg^2 n)$ rounds

Analysis combines new ideas with ideas from the analysis of disjoint set union algorithms

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

R: $\Theta(\lg n)$ rounds

Analysis uses a variant of the potential function of A & S and a novel multi-round analysis: flat trees can linger for a non-constant number of rounds

Algorithms C and A run in $O(lg^2 n)$ rounds.

The number of active vertices decreases by a constant factor every $O(\lg n)$ rounds.

Algorithm R runs in $O(lg\ n)$ time.

The sum of the heights of active trees decreases by a constant factor every constant number of rounds.

Analysis of algorithm R

After two rounds all trees contain at least two vertices (except in components of one vertex)

A tree is passive in a round if it does not change during a round, active if it does

The potential $\Phi(T)$ of an active tree T is its height plus one, plus one more if flat

The potential of a passive tree is zero

Let T be an active tree at the end of round k

The constituent trees of T at the end of round $j \leq k$ are those at the end of round j whose vertices are in T

The potential $\Phi(T_j)$ of T in round j is the sum of the potentials of its constituent trees

Lemma: $\Phi(T_{k-1}) \geq \Phi(T_k)$, and if $k - j \geq 5$,

$$\Phi(T_j) \geq (4/3)\Phi(T_k)$$

Proof sketch

A shortcut reduces the potential by almost a factor of two

A calculation gives $\Phi(T_{k-1}) \geq \Phi(T_k)$, and $\Phi(T_{k-1}) \geq (6/5)\Phi(T_k)$ if T has height at least 4

If T has height at most 3 and has at least one active constituent tree of sufficient height, or at least two constituent trees, the lemma holds

Otherwise T only has one active constituent tree

After at most four rounds, all constituent trees are combined, and T is passive, a contradiction

Fewer rounds?

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

We (Liu, Tarjan, Zhong) can simplify their algorithm and implement it on a COMBINING CRCW PRAM, a much weaker model.

Asynchronous processes?

Recent work on concurrent disjoint set union by Jayanti and Tarjan (PODC 2016 and unpublished) will (we think) translate into efficient asynchronous concurrent algorithms for connected components

Thanks!

For details see our arXiv paper
(revision of our SOSA 2019 paper)