presented last time a rather crude collection of algo's for setting up various kinds of spanning trees in asynchronous networks. One we didn't do: DFS

Algorithms done earlier in term in synch case, essentially sequential, work fine in asynchronous setting.

Minimum Spanning Tree

A good illustration of the problems introduced by asynchrony.

Nice simple synchronous alg needs a bit more mechanism to work in asynch setting.

**Problem:**

\( G = (V, E) \) connected, undirected, weighted edges (distinct sets)

Generate MST, where each node knows which of its adj edges belong

to known to endpoint nodes

Nodes don't know m, d, a, etc.

Wakeups: at one or more times, asynchronously, via input

makeup signals.

Can make each other up with ordinary mags.

**Sync alg:**

Recall, based on growing in levels

Run synchronously

At each level, spanning forest

→ next level by having all components find MWOE + join along all these

Find MWOE by least-cost from component's leader.

After merge, have to choose new leader, based on unique crossing edge (MWOE of both components)

[Diagram of a spanning forest with a new leader and edge]

Complexity good: \( O(n \log n + E) \) mags, \( O(n \log n) \) time

In beast-coast, each mode determines its own MWOE by testing its edges in T-A-R protocol.
Good msg complexity depends on doing this bookkeeping carefully, testing edges in order of net, + not retesting rejected edges.

Running this asynchronously?

Problems:
1. In sync version, when mode queries nbr to see if on same or different component, knows nbr is up to same level.
   In async version, nbr might be lagging. Might be in same component, but not yet know this.

2. In sync version, levels kept synchronized, so component sizes grow more or less in balance.
   Here, some components could get way ahead of others, which can lead to more msgs.

E.g.

![Diagram]

Component might keep adding 1 node at a time.

Each time, # of msgs in component's tree is proportional to component's size. Leads to $2n^2$ tree msgs (bad).

3. When modes are out of sync, not clear what other unexpected interference might occur. (come back to this)

So must adapt:
The Modified Algorithm

Same basic ideas as before, form into components, combine as before.
Within any component, proceed as before to find component MWOE.
Boost from border, crank back as before.
Combine as before.

But need to introduce synchronization to prevent nodes from getting ahead of their roots.

Associate a level with each component (as in synch case).
Number of nodes in level \( k \) component will still be \( \geq 2^k \).
Level \( k+1 \) components will be formed out of (exactly) two level \( k \) components.

Level numbers can be used for synchronization - help determine who's in same component.

More detail:

Combine pairs of components in 2 ways:

**merging:**

\[
\begin{array}{c}
\text{C} \\
\text{level } k
\end{array} \quad \text{MWOE} \quad \begin{array}{c}
\text{C'} \\
\text{level } k
\end{array}
\]

2: with same level, \( k \), & common MWOE
Result is new merged component, \( C + C' \) + the new edge, at level \( k+1 \)

**absorb:**

\[
\begin{array}{c}
\text{C} \\
\text{MWOE}
\end{array} \quad \text{MWOE} \quad \begin{array}{c}
\text{C'}
\end{array}
\]

level \( (C') \) > level \( (C) \)

Just absorb \( C \) into \( C' \).
Keep level \( (C') \).

(Sticky C catch up, + absorbing it in.)
A key is that absorbing will be cheap — we don’t need to explore all of C in order to absorb C in along C’s MWOES.

**Handle Difficulty 2**

Any component at level k has $\geq 2^k$ modes.

**Proposition** They’re Merge, absorb allowed because special cases of what’s allowed by the general theory of MSTs.

Why are they enough to ensure the final MST gets completely constructed?

**Lemma**. After any allowable finite sequence of merges + absorbs, either the forest consists of one tree (we’re done) or some merge or absorb is enabled.

**Pf. Consider the current component digraph.**

- nodes = components
- directed edges correspond to MWOES

Then there must be some pair whose MWOES point to each other.

Claim in this case must be the same edge (else...)

Claim can combine using either a merge or absorb:
- if same level, merge
- if ≠ level, absorb (in one direction)
Finding the MWOE in a component

Each node must determine local MWORE
  + convergent to leader
Need to know who leader is,
  + direct way to tell if edge is outgoing.
  
  for every component at level \( \geq 1 \), identify core edge.
  Defined in terms of the merge + absorb ops.
  After merge, the common MWOE.
  After absorb, keep the old core edge of higher-level component
  "The edge along which the most recent merge occurred."
  Use (core, level) to identify the component.
  
  actually, the weight of the core edges.

Endpoint with higher id is the leader.

Determining if edge is outgoing:

Suppose i wants to know if edge is outgoing from i's current component.
In this case, it will happen that i's component identifier is up-to-date.
If j's current (core, level) is same as i's, then i knows j is in
  same component.
If j's core is \( \neq i \)'s and j's level is \( \geq i \)'s, then i knows
  that j is in different component
  (component has only one id per level and i is)
  up-to-date.
So that leaves the case where \( j \)'s core \( = i \)'s + j's level is < i's.

In this case, don't know, so C doesn't yet respond - wants to catch up to i's level.

This handles Difficulty 1: (telling if edge is outgoing)

But extra delay could affect the progress argument - might cause alg to become blocked where it couldn't before.

Redo the progress argument, this time considering only those components with current lowest level, say k.

All these proc must succeed in their determinations, so these components succeed in determining MWOE.

If any of the level k components' MWoes lead to higher level, abort.

If not, then all lead to other level k, so as before, must have a 2-cycle of these

So may fail.

Difficulty 3: Possible interference among concurrent searches for MWoes by adjacent components at different levels.

In particular, suppose C absorbed into C' while C' is making on determining its MWOE.

2 cases
1. if not yet determined its local MWoe when the absorb occurs

Then it is not too late to include C in the search - if sends the heart message into C too.
2. \( j \) has already determined its \( \text{mwoe} \).

Then it might be too late to include \( C \) in the search, because \( j \) might have already responded in the ceast.

But claim it doesn't matter - the MWOE for the combined component can't be outgoing from a node in \( C \) anyhow.

Claim 1: \( \text{mwoe}(j) \) for the \( C^1 \) search cannot be the edge \((i, j)\).

This is because \( \text{mwoe}(j) \) already determined, so it leads to component with level \( \geq \text{level}(C') \).

But level of \( i \) is still \( \lt \text{level}(C') \),

because absorbed.

Therefore, the weight of this \( \text{mwoe}(j) \) is \( \lt \text{weight}(i, j) \).

Claim 2: MWOE for combined \( C - C^1 \) not outgoing from a node in \( C \).

This is because \((i, j)\) is MWOE for \( C \).

So if any edges out of \( C \) with \( w(t) \lt w(t) \),

so no edges... \( \lt \text{the already determined} \text{mwoe}(j) \).

More details:

Messages: initiate - the host sends then the components
report - the ceasts back
request - to query if edge is outgoing
accept, reject - answers
changeover: sent from leader to location of the determined MWOE
connect: sent across the MWOE, to connect: maps who map
   goes both maps on edge, also which go to higher level
I-A-R involves the same sort of bookkeeping as in the synchronous case:

Each node keeps incident edges in order of weight, classifies as

- `known` (in the MST)
- `rejected` (goes to same component)
- `unknown` (not yet classified)

Only test "unknown" edges, and do so sequentially, in order of weight.

Send test msg with core + path, recipient compares.

- If core + rejected, (sender reclassifies edge as rejected)
- If core +, but level of recip >= level of sender, send accept (sender doesn't reclassify edge)
- If core + but level ...

  just delay responding, waiting for that for your level to become >= level of sender.

Retracing possible for accepted edges.

Reclassify as branch only as a result of change in req msg.

Merge, absorb, ...

Based on whether S2 connect msg cross

- 1 connect msg only, from lower to higher level component
- Output branch edges when found (or at end, when all local edges classified).

Complexity: As for Synchronous, comm: O(n log n + |E|)

Amortized: O(|E| for all trees)
$O(m \log n)$ for all common tree edges

# of levels

$O(1E1)$ for T-R

$O(n \log n)$ for T-A, at most one such pair per node per level

Time: $O(n \log n (d+1))$ LTR

Correctness

Hard to do proof - complex

Several attempts, but all too complicated.

Proofs use invariants - but there are a lot, because there are lots of variables & subalgorithms.

Some proofs use levels of abstr. with graph manipulation (merge, absab, etc.) at higher levels.

Could try other things:

Break up into composed subalgorithms for Short, TAR, CCAST, etc.

Relate anything carefully to such version.

Orig. paper has only informal proof (plausibility arg.)

Several have their published piece more ready nice.

Using MST to elect leader

General, unknown, under. network.

$O(m \log n + 1E1)$ common to elect leader

Set up sp. tree, then fan in (converge) from leaves until find "center"

E.g. in ring, this gives $O(m \log n)$ common alg. for leader el
Simulating synchronous alg's in asynclx networks

Complications in GHS arise from the fact that different parts of the network can be at different levels.

Another approach: try to keep levels of nearby nodes close (restrict the asynclx)

Each process keeps level variable, keeping track of level of component it currently belongs to (acc to its local knowledge).

Process at level k delays participating in alg to find level k MWOE until it learns that all its suby's have level ≥ k.

(just as good as global synch, as far as anyone can tell, but cheaper.)

Discovers this by: each node informs sbns about new levels simpler than GHS, more like Synch GHS (synch at each level)

true complexity same

but

Msg complexity suffers somewhat: O(1E1 log n)

all edges get msgs at each level

Worse than asynclx.

This idea suggest a strategy for designing asynclx dist. alg's:

First design a synchronous alg, then transform it by local synchronization into an asynclx version.

In general, might have to synchronize at every round, not just every level.

Strategy is only for fault-tolerant algorithms

In fact, as we will see, some fault-tolerant problems, like agreement protocol, aren't even solvable in asynclx network model.
Will show a general strategy and some implementations.

And just for fun, will illustrate present the strategy using
compilation + abstraction
Successive refinement
Service specs

Finally, a lower bid on tire for a strong (global) kind of
synchronization, vs. the local kind represented here.

Problem:
Reconsider synch modeled in terms of I/O automata.

\[ \text{user - send} \left( T_{ij} \right) \]

\[ \text{user - rec} \left( T_{ij} \right) \]

\[ \text{set of} \ (m, j) \text{ pairs} \]

\[ \text{msg} \text{ destination (sends)} \]

\[ \text{msg} \text{ source} \]

The U's might have other actions too, for interacting with their spec
(model their inputs + outputs with actions).
The way this works:

Users send entire seq of all req msg (1 msg after the other)
MkSynch collects all centrally & sorts by destination.

Delivers to users 1 at a time.

Users send for req 2, etc.

This is supposed to be like the synchronous model.

Has limited asynchrony, e.g. GS can deliver req 1 msg to i,
get req 2 msg from i, before delivering req 1 msg to j.

But shouldn't be hard to see this behaves like the synch model.

MkSynch synchronizes between each pair of reqs.

Requirements on V_i: "well-formed" (sends the right types of msg)
in the right order.

Linearize: eventually submits msg for each req
(forever), after it receives msg from preceding round.

MkSynch: see code, p. 534
Tasks: one for each dest. + each req.

Synchronization Problem: "Implement" MkSynch so it "looks the
same" to each V_i:

∀ z, fair exec of Us × Impl. ICA
∃ V_i = Us × MkSynch
∀ i, x (i, x) = y

Project to give identical local exec.
Looks same to each - but note this allows global rendering of events at
different users.
First “refinement” - local synchronizer

Allow local synchronization rather than global; argue this still looks the same locally.

Only difference is in *user-seq*: only check if your subs have already sent their round $r$ msgs - don’t wait for all nodes to get that far.

**Lemma:** If $s$ is fair exec of Loc-Synch system

then $E_{d'}$, fair exec of Glob-Synch system, $A_i \subseteq v_i d'$.

If: *Six mapping* can’t quite work here, since external event order not preserved.

Instead, can argue about partial order of events & dependencies:

- actions of same $V_i$
- receive $(T, r)_i$ + send $(T', r)_j$, $j \in mbrs$

+ take transitive closure.

Claim: if you start with a fair exec of Loc-Synch system & render events while preserving these dependences, the result is still a fair exec of Loc-Synch system.

Then using claim, take a *render* preserving dependences, to line up events by rounds.

This gives the extra precondition needed for Glob-Synch

(*preserves events at each user*)
Simple Sync

Crucial dist. alg. to implement

Loc Sync:

Processes, pt-pt reliable channels.

After receiving input for rd \( r \), process sends msg to each \( u \) containing rd \( H \) \( r \) and ag msg for that \( u \).

Wait to receive rd \( \neg m \) msg from all \( u \).

Then output \( r(e, r) \).

\[ \text{code p. 537-8} \]

Lemma: \( \forall u \) fair exec of Simple Sync system

\( \exists u' \) fair exec of Loc Sync system,

\( \forall i \in V \text{ } \forall u' \).

\( u' \in u \).

\( \text{Proof:} \) No reasoning this time, so a sim mapping \( p_f \) will work

\( \text{(for safety part)} \)

(technical: just preserves traces. Have to do a bit more work

to preserve everything about each \( V \): internal events + states.

Can argue with putting + projection lemmas.

Also need to argue fairness - essentially based on fairness of channels + automata.

LTIR: Main idea is high level trick \( (i, r) \) is just the union

of everyone's output \( (i, r) \), incoming to \( i \).

Combining results: \( \forall u \) fair exec of Simple Sync system

\( \exists u' \) fair exec of GC

\( \forall i \in V \text{ } \forall i' \).

\( u' \in u \).

Complexity: \( \text{msgs:} \) each rd needs \( 2 |\text{set of reds}| \)

\( \text{time:} \) Assume use does send immediately

\( \frac{1}{2} \) rd on task time for any process

\( \frac{1}{2} \) rd on time to first msg in any channel
Lower communication algorithms

Based on a general "safe synchronizer" strategy.

Idea: If msg $V_i \rightarrow V_j$ at rd r of synchronous alg, try to avoid
sending msg in simulating async alg.

Can't just omit, since each proc needs to know when it has already
received all the msgs it ever will, for each round r
(Needs this to know when to do recv (r).)

Decompose for synch into pieces:

- Communicating the msgs
- Determining when done

Decompose for synch into pieces:

- Front ends + channels
- OK(r) i , Go(r) i

FEs send & collect all the msgs & want to receive acks that the
msgs actually arrived.

(acks add extra msgs, but if actual msgs are "sparse" this is still)
a win.

When FE receives acks for all its rd r msgs, it's safe
- means all its msgs received by its nbs.

Then sends OK i to SafeSync.

FE wants to know if has received all its nbs' msgs for rd r.
Suffices to determine all nbs are safe.