Reading: 4.2-4.4, 5.4
(Atiya-Welch Ch. 2.4, 2.5)

Next: Ch. 6 (all)

Last time: Started
Basic algo in synchronous networks.

general
1. Simple leader election by flooding
   + used to illustrate a basic simulation rotation

2. Started BFS

Body: Continue -

- BFS
- shortest paths
- MST
- DFS

Comm: Setting up basic structures in (synchronous)
networks.

All these can be extended to asynchronous, as we'll see shortly.

And then distributed

There are all analogues of standard problems that were studied for sequential settings.
(And the algorithms have a lot in common too).

Then start distributed consensus.
Breadth-first Search

Strongly connected digraph with distinguished source node $i_o$.

Want to set up a B-F spanning tree for digraph, starting at root $i_o$.

- **Spanning** contains every node.
- **B-F**: each node at dist $d$ appears at depth $d$ in the tree from $i_o$ in the digraph.

**Ex:**

```
\begin{center}
\begin{tikzpicture}
\node (1) at (0,0) {$i_o$};
\node (2) at (1,1) {$i_1$};
\node (3) at (2,0) {$i_2$};
\node (4) at (1,-1) {$i_3$};
\node (5) at (0,-2) {$i_4$};
\node (6) at (3,0) {$i_5$};
\draw [->] (1) to (2);
\draw [->] (1) to (3);
\draw [->] (1) to (4);
\draw [->] (2) to (4);
\draw [->] (3) to (4);
\draw [->] (4) to (5);
\draw [->] (5) to (6);
\draw [->] (6) to (3);
\end{tikzpicture}
\end{center}
```

**Output:** Each node other than $i_o$ should set a parent variable to indicate the node that is its parent in the tree. (Seumed by a U.D.D.)

Output tree representation is distributed.

**Inferior derer:** Mark nodes that are already incorporated into the tree. (Marked "mark" flag in state).

Initially, only $i_o$ is marked.

**Ed:** Proc. $i_o$ sends out "search" mgs to all nbs.

At any round: If unmarked proc. receives "search" mg, marks self, chooses one of the processes that sent msg as its parent.

At next round (only) sends "search" mg to all its neighbors.
So, state needs to contain flag to say when to send, etc.

Pf: LTTB

B-F tree because created synchronously.

Complexity: Some & draw to have everyone set "parent"

actually, it's max dist (i.e., j)

for the particular i.

\[ \frac{1 \leq i}{\text{(one per directed edge)}} \]

Efficient.

(can reduce slightly: processes needn't "reflect" msg back to any

process that sent it a msg.)

Child pointers: Often, for applications, want process to know its

children in tree, not just its parent.

Can accomplish this by having each node respond to its parent, telling

it that it has been chosen.

Easy if bidirectional comm on all edges (?)

If not, must send response indirectly, on whatever paths

connect.

But this is complicated & expensive - initially, don't know

which paths these are. So have to do something like

send on all paths (flood)

and have to tell other nodes that they have not been

chosen as parents, if node is supposed to
ever figure out that it knows its entire set of

children.
Termination:
Suppose i wants to know when the whole tree is completely set up?
Let each “search” msg get a response, “parent” or “non-parent”.
After any process has received responses to all its “search” msgs, it knows who its children are, + knows they are all marked.
So, starting from leaves of the tree (these discover who they are), fan in “completion” msgs to i.
Node can send completion msg after
(a) It has received responses to all its search msgs (so it knows its children)
+ (b) It has received completion msgs from all its children.

Complexity: (Assuming bidir. comm.)
Time: \( O(diam) \)
Msgs: \( O(1 + 1) \) including responses to search msgs.
(Completion msgs just \( O(m) \)).

Applications of BFS
1. Broadcast
   Can broadcast a msg while setting up the tree (piggyback) on
   First uproot tree, with child pointers, then use the tree for broadcast.
   Can reuse the tree for msg starts; each takes only
   \( O(m) \) msgs
   \( O(diam) \) time.
2. Global computation, given a BF tree
   - Involves collecting info from everyone in the network, possibly
     the root
   - Using it in some computation.
   - E.g. sum values held by all pros.
   - Convergecast, combining at each stage (start at leaves).
   - Time: $O(diam)$
   - Msgs: $O(n)$

3. Leader election, with unknown $\gamma$
   - All pros initiate BFS in parallel, use tree to compute max rank
     in network (convergecast).
   - Then process with this max rank declares self leader.
   - Time: $O(diam)$
   - Msgs: $O(n \cdot 1E1)$ msg
     - for each BFS
     - number of different BFS trees

   (actually, $O(diam \cdot 1E1)$ is an upper bound, since
   there are only $diam$ nodes, & one msg per edge per rel.
   But there are $\gamma$ big msgs, obtained by combining
   a lot of smaller primitive msgs.)

Q: Cut down # of msgs by discontinuing participation in BFS's for sources that aren't the max one you know about.
4. Computing diameter

E.g., all i can do BFS in parallel, use tree (convergecast) to
determine max distance from i to any node. (height of tree)

Then all reuse BF trees for global comp. To determine max of
all these max distances

Complexity as for 3. above.
Shortest Paths

Also motivated by communication - setting up structure for shortest latency communication. Let where edges have associated "costs."

Generalization of BFS

Digraph, strongly connected, now with weights \( c \in \mathbb{R}^+ \) on edges

Source \( s \)

Want to determine shortest path (\( = \) path with smallest total weight )

from \( s \) to all nodes.

A tree again

Every node should output "parent in tree"

\( \text{distance from } s \) in sp. tree. (total of weights)

Node knows weights of incident edges.

Assume nodes know \( n \). (again for termination)

Bellman-Ford algorithm:

Each \( i \) remembers \( \text{dist, shortest dist it knows (so far) from } i \)

\( \text{parent, its parent in some path with total } \text{ distance} \)

Initially:

\[
\begin{array}{ccc}
\text{dist} & = & 0 \\
\text{parent} & = & \text{null}
\end{array}
\]

\( \text{for all other nodes} \)

at each \( i \), each process sends its \( \text{dist} \) to all its out-edges.

Then each process updates its \( \text{dist} \) with a relaxation step where it does:

\[
\text{dist} := \min (\text{dist, all values (dist } + \text{ wt})
\]

If \( \text{dist} \) changes, parent also updated.

Done after \( n-1 \) rounds.
After m-1 rounds, if dist contains shortest path, parent contains parent in s-p tree.

Complexity: \( \text{time} = (m-1) |E| \)
\( \text{msg} = (m-1) |E| \)

Correctness:

Key invariant says that after \( r \) rounds:
Every process \( i \) has its dist & parent corresponding to a shortest path among paths from \( i_0 \) to \( i \) that consist of \( \leq r \) edges.

(If no such, then dist = \( \infty \), parent = null.)

Proof: Induction. Key is that a shortest path to \( i \) on \( \leq r \) path must be

Key is that, after nd \( r-1 \) each pred \( j \) of \( i \) on any path for \( i \) had its best dist taken over paths of \( \leq r-1 \) edges.
And shortest path to \( i \) on \( \leq r \) path, when restricted to its pred \( j \) on this path, yields a shortest path to \( j \) on \( \leq r-1 \) path.

Complexity: \( \text{time} = (m-1) \)
\( \text{msg} = (m-1) |E| \)

Worse than BFS, which has the \( \text{time} \) diari
\( \text{msg} \) diari

You might think the algorithm is \( \text{should depend on } d \), \text{not } m.
But not so, because "shortest path" can actually be over a path with more links:
Shortest path on bottom. The complexity depends on m, not clean.

Shortest path = 4
though diam = 1.

Later will revisit Bellman-Ford in anypath networks (like old Aspen)
Minimum Spanning Tree

Another basic tree structure, this one minimizing total cost of all edges in the tree.
(Total cost, e.g., money to change, for all communications.)

Show synchronous algorithm here, namely as a stepping-stone to a more complicated asychronous algorithm we'll see later.

Assume: \( G = (V, E) \) undirected weight on each edge
weights known by adjacent pros (e.g., money, real)

So, (actually, the asynchronous algorithm doesn't need this. But the simplified version uses it (technically).)

Each node to decide which adjacent edges are + aren't in the tree.

The theory (same general theory as for sequential MST alg).

Spanning tree \( T(G) \): A tree (acyclic graph) connecting all nodes, with edges selected from \( G \).

Spanning forest: A collection of trees spanning all the nodes (again, edges selected from \( G \)).

All MST alg, both sequential & distributed, use special cases of a general strategy.

Start with trivial spanning forest of \( n \) separate nodal graphs. Repeatedly: Merge components along edges that connect components, until all are connected.

The trick is to make sure that we only merge along edges that are min weight outgoing edges of some component.

Justification for why this is OK: (Same argument as in sequential case).
Lemma 1: Let \((V_i, E_i), 1 \leq i \leq k\), be sp. trees, \(k \geq 1\)

For any \(i\), let \(e\) be an edge of smallest weight among set of edges with exactly one endpoint in \(V_i\).

Then \(3\) sp. trees for \(G\) that

(a) includes \(U \cup E\)

(b) includes \(e\)

(c) has min. weight among all sp. trees for \(G\) that include \(U \cup E\)

Proof: Like the standard arg. for the sequential theory.

Suppose false. \(T_i\) contains \(U \cup E\) but not \(e\).

Let \(T\) be the sp. tree with min. weight among those w/ \(U \cup E\).

Then argue

that there's some \(T\) that is a sp. tree for \(G\) + includes \(U \cup E\) that doesn't contain \(e\), it itself has weight strictly smaller than the weight of any sp. tree that includes \(U \cup E \cup e\).

Now construct graph \(T'\) (not a tree) by adding \(e\) to \(T\).

\(T'\) contains cycle, which must contain another edge outgoing from the same component (to \(V_i \cup E_i\)).

\(w(T') \geq w(T)\) by choice of \(e\).

Let \(T'' = T' - e\).

\(T''\) is sp. tree & has \(w(T'') \leq w(T)\).

\(T''\) contains \(U \cup E\) and \(e\).

Contradicts choice of \(T\).
General strategy for MST:
Repetitively:
- Choose some component i
  - Choose an least-weight outgoing edge from i.
    - add (merging 2 components)

Sequential MST algo be special cases of this:
Dijkstra: Shows 1 component by adding 1 new node each time
Prim:
Kruskal: Always add min wt edge globally.

Distributed version?
Would like to choose edges concurrently for several components.
Problem:
- Can get cycles.

Avoid by assuming all weights are distinct (can achieve same effect by hashing w/ UIDs).

Lemma 2: If all wts are distinct, then there is exactly one MST.
Proof: Another cycle - style argument; similar to the previous one.

This allows a concurrent strategy:
At each stage, suppose (inductively) that the forest produced so far is part of the unique MST.
Then suppose several components choose least-cost outgoing edges concurrently.
Claim each of these is in the (unique) MST, by Lemma 1. (all are, + OK to add them all.)
No cycles, since all are in the MST.
Distributed Algorithm:
assumes edge sets distinct.

Build forest in "phases" (phases).

For this presentation, for the synchronous model, the levels will be
called synchronously throughout the network.
kept synchronized (asynchronous version lets them get)
out of sync.

For each k, level k components form a spanning forest,
which is a subgraph of the unique MST.

Moreover, each level k component has \( 2^k \) nodes
with a distinguished leader node.

In this synchronous description, processes allow a fixed number
\( O(n) \) of rounds for each level.

This requires that they know \( n \). (This funny restriction is removed
later in the asynchronous version.)

At the end of

When we have level k components, each process knows:
The UID of the leader - used as ID for the component
which of its adjacent edges are in the component tree for its component.

Level 0: Single node.

Level \( k \to \) Level \( k+1 \):

Each component conducts a search (along tree edges) for MWOE of
component:

Leader transmits requests along tree edges (as before).

Each process finds the minimum weight edge among
its incident edges that are outgoing
from the component.
Do this by sending test messages along non-tree edges, that are outgoing from component, asking if other end is in same component - compare component ids.

Convergecast the min information.
Min obtained by leader is min for entire component.

After all level k components have found MWoES (want suff. many ids, O(m), to be sure)
Need to combine all components using all these MWoEs to form new level k+1 forest.

Need new leaders, etc.

So, leader of each level k component sends msg (along tree) to MWoE node (endpoint in this component), telling it to record that this edge is in the new tree at level k+1 (tells other end also).

To choose new leader: For each new component, there's a unique edge e that is MWoE of two level k components.

(See graph theory, see p. 68.)

Choose new leader to be one of the endpoints of e, e.g. largest VID.

Note new component has \( \geq 2^{k+1} \) nodes.
Terminates: When attempt to find MWOF fails. This means a
component has no outgoing edges, which means it is the entire
set of 6 nodes.

Then leader starts msg saying done.

This algorithm designed to work with component levels synchronized.

More difficulties arise when they can get out of sync (we'll see later).

In particular, test messages are supposed to determine if some
it's the same component by comparing UIDs.

Important that the node being queried has up-to-date UID info.
(That's why we needed the pros to know M).

Complexity:

Time: $O(m \log n)$

The per number of levels
level (because $2^k$ nodes in each level k
component)

Msgs: $O((m + |E|) \log n)$ if done naively:

at each level, $O(m)$ msgs sent on tree edges
$O(|E|)$ for doing all the tests
(enumerate all edges)

Reducing communication:

A major control of the async alg is reducing the comm

$O(n \log n + |E|)$.

I'll show their trick here, in the sync setting.
Idea

Process marks its incident edges as rejected when they're discovered to lead to same component — no need to restart these.

At each level, test candidate edges 1 at a time, in order of increasing weight, until the first one is found that leads outside (or until exhaust candidate edges).

Rejects all those that are found to lead inside.

\[ M_{eq} : O\left(1\varepsilon_1 + n \log n\right) \text{ total}. \]

\[ O(n) \] for \# of tree edges at each phase, for \( O(n \log n) \) total.

\[ M_{eq} \] for test, accept, + reject.

"different" "same component"

Amortized analysis.

Each edge has test-reject at most once in each direction (directed) for total of \( O(1\varepsilon_1) \).

If tested + accepted, might not be MWOE of entire component. So might be tested again. Can test + accept several times.

However, at most 1 T+A for each node at each level, for total of \( O(n \log n) \).

Back to this for asynchronous case.