PROFESSOR: OK, welcome back. This is the second half of our two lectures on distributed algorithms this week. I could turn that on, OK. All right, I'll start with a quick preview. This week, we're just looking at synchronous and asynchronous distributed algorithms, not worrying about interesting stuff like failures.

Last time we looked at leader election, maximal independent set, breadth-first spanning trees, and we started looking at shortest paths trees. We'll finish that today, and then we'll move on to the main topic for today, which is asynchronous distributed algorithms where things start getting much more complicated because not everything is going on in synchronous rounds. And we'll revisit the same two problems, breadth-first and shortest paths spanning trees.

Quick review, all of our algorithms are using a model that's based on undirected graph. Using this notation, gamma, for the neighbors of a vertex. We'll talk about the degree of a vertex. We have a process, then, associated with every vertex in the graph. And we associate communication channels in both directions on every edge.

Last time we looked at synchronous distributed algorithms in the synchronous model. We have the processes of the ground vertices, they communicate using messages. The processes have local ports that they know by some kind of local name, and the ports connect to their communication channel.

The algorithm executes in synchronous rounds where, in every around, every process is going to decide what to send on all of its ports, and the messages then get put into the channel delivered to the processes at the other end. And everybody looks at all the messages they get at that round all at once, and they determine a new state. They compute a new state based on all of those arriving messages.

We started with leader election. I won't repeat the problem definition, but here's the results that we got last time. We looked at a special case of a graph that's just a simple clique. And if the processes don't have any distinguishing information, like unique identifiers, and they're deterministic, then there's no way to break the symmetry and you can prove an impossibility result that says that you can't-- in a deterministic, indistinguishable case, you can't guarantee to elect a leader in this kind of a graph.

Just as an aside, I should say that distributed computing theory is just filled with impossibility results, and they're all based on this limitation of distributed computing where each node only knows what's happening to itself and in its neighborhood. Nobody knows what's happening in the entire system. That's a very strong limitation, and as you might expect, that would make a lot of things impossible or difficult.

Then we went on and got two positive results, a theorem that-- well, an algorithm that is deterministic, but the processes have unique identifiers, and then you can elect a leader quickly. Or if you don't have unique identifiers but you have probability, so you can make random choices, you can essentially choose an identifier, and then it works almost as well with those identifiers.

Then we looked at maximal independent set. Remember what it means, no two neighbors should both be in the set, but you don't want to be able to add any more nodes while keeping these nodes independent. In other words, every node is either in the set or has a neighbor in the set.

And we gave this algorithm-- I'm just including it here as a reminder-- Luby's algorithm, which basically goes through a number of phases. In each phase, some processes decide to join the MIS and their neighbors decide not to join, and you just repeat this. You do this based on choosing random IDs again. And we proved that the theorem correctly computes an MIS, and with a good probability, all the nodes decide within only logarithmic phases, and here is the number of nodes.

All right, then went on to breadth-first spanning trees. Here, we have now a graph that already has a leader. It has a distinguished vertex. And the process that's sitting there knows that it's the leader. The processes now are going to produce a breadth-first spanning tree rooted at that vertex. Now, for the rest of the time, we'll assume unique identifiers, but the processes don't know anything about the graph except that their own ID and their neighbors' IDs.

And we want the processes to eventually output the ID of their parent. And here is, just to repeat, the simple algorithm that we used. Basically, it's processes just mark themselves as they get included in the tree. It starts out with just the root being marked. He sends a special search message out to his neighbors, and soon as they get it they get marked and pass it on. Everybody gets marked in a number of rounds that corresponds to their depth in-- their distance from the root of the tree, [INAUDIBLE] from the tree.

We talked about correctness in terms of invariance. What this algorithm guarantees is, at the end of any number r of rounds, exactly the processes at distance up to r are marked, and the processes that are marked, have their parents defined. And if your parents defined-- it's the UID of a process that has distance d minus 1 from the root. If you're a distance d, your parent should be somebody who's correct-- has the correct distance d minus 1.

We analyze the complexity. Time is counting the number of rounds. And that's going to be, at worst, the diameter of the network. It's really the distance from a particular vertex, v 0. And the message complexity-- there's only one message sent in each direction on each edge, so that's going to be only order of the number of edges.

And we talked about how you can get child pointers. This just gives you parents. But if you want to also find out your children, then every search message should get a response either saying you're my parent or you're not my parent. And we can do a termination using convergecast, and there's some applications we talked about as well.

All right, and then at the end of the hour, we started talking about a generalization, a breadth-first spanning trees, which adds weights so you have shortest paths trees. Now you have weights on the edges. Processes still have unique identifiers. They don't know anything about the graph, except their immediate neighborhood information. And they have to produce a shortest path spanning tree rooted at vertex v 0.

You know what a spanning tree is and the shortest paths are in terms of the total weight of the path. Now we want each node each, process, to output its parent in the shortest path. And also the distance from the original vertex v 0.

At the end of the hour, we looked at a version of Bellman-Ford, which you've already seen as a sequential algorithm. But as a distributed algorithm, everybody keeps track of their best guess, their best current guess, of the distance from the initial vertex. And they keep track of their parent on some path that gave it this distance estimate, and they keep their unique identifier.

The complete algorithm now is, everybody's going to keep sending their distance around-- I mean we could optimize that, but this is simple. At every round, everybody sends their current distance estimate to all their neighbors. They collect the distance estimates from all their neighbors, and then they do a relaxation step. If they get anything that's better than what they had before, they look at the best new estimate they could get.

They take the minimum of their old estimate-- stop shaking, good-- and the minimum of all of the estimates they would get by looking at the incoming information plus adding the weight of the edge between the sender and the node itself. This way you may get a better estimate. And if you do, you would reset your parent to be the sender of this improved information. And again, you can pick-- if there's a tie, you can pick any of the nodes that sense-- the information leading to the best new guess, you can set your parent to any of those. And then this just repeats.

At the very end of the hour, we showed an animation, which I'm not going to repeat now, which basically shows how you can get lots of corrections. You can have many paths that are set up that look good after just one round, but then they get corrected as the rounds go on. You get much lower weight paths by having a roundabout, multi-hop path to a node. You can get a better total cost.

Here's where we got to last time. Now why does this work? Well what we need is eventually every process should get the correct distance. And the parent should be its predecessor on some shortest path. In order to prove that-- you always look for an invariant, something that's true, at intermediate steps of the algorithm that you can show by induction will hold-- and that will imply the result that you want at the end.

Here, what's the key invariant? At the end of some number r of rounds, what do processes have? After r rounds have passed, in this kind of algorithm, what do the estimates look like?

Well, after one round, everybody's got their best estimate that could happen on a single-- that could result from a single hop path from v 0. After two rounds, you also get the best guesses from two hop paths, and after r rounds in general, you have your distance and parent corresponding to a shortest path chosen from among those that have at most our hops. Yes? That makes sense? No? Yeah? OK.

All right, and if there is no path of r hops or fewer to get to a node, there's still going to have their distance estimate be infinity after r rounds. This is not a complete proof, but it's the key invariant that makes this work. You can see that after enough rounds corresponding to the number of nodes, for example, everybody will have the correct shortest path of any length. OK?

The number of rounds until every estimate stabilize-- all the estimates stabilize, it's going to be n minus 1. All right? Because the longest simple path to any node is going to be n minus 1, if it goes through all the nodes of the network before reaching it. Makes sense? If you want to make sure you have the best estimate, you have to wait n minus 1 rounds to make sure the information has reached you.

The message complexity-- well, since there's all these repeated estimates it's no longer just proportional to the number of edges, but you have to take into account that there can be many new estimates sent on each edge. In fact, the way I've written this, you just keep sending your distance at every round. It's going to be the number of edges times the number of rounds. You can do somewhat better than this, but it's worse than just the simple BFS case.

This is more expensive, because BFS just had diameter rounds and this now has n for n minus 1 rounds, and BFS just had one message ever sent on each edge, and now we have to send many. Comments? Questions?

Is it clear that the time bound really does depend on n and not on the diameter? For breadth-first search, it was enough just to have enough rounds to correspond to the actual distance and hops to each node, but now you need enough rounds to take care of these indirect paths that might go through many nodes but still end up with a shorter-- with a smaller total weight.

Is it clear to everybody why the bound depends on n? Actually the animation that I had last time showed how there are lots of corrections, and you had enough-- you had rounds that depended on the total number of nodes. Yes? OK. Yeah?

AUDIENCE: But could you keep track of each round if the values-- if any of [INAUDIBLE]? If you-- if everything stops changing after less than n rounds, then you might not have to--

PROFESSOR: OK, so you're asking about termination?

AUDIENCE: Yeah.

PROFESSOR: Ah, OK, well so that's probably the next slide. First, let's deal with the child pointers, and then we'll come back to termination. First, this just gives you your parents. If you want your children, you can do the same thing that we did last time. When a process gets a message, and the message doesn't make the sender its parent, this is not giving it an improved distance. Then the node can just respond, non-parent.

But if a process receives a message that doesn't improve the distance, it says, OK, you are my parent, but it might have also told another node-- it might have another parent, another node that previously it thought was its parent. It has to do more work, in this case, to correct the erroneous parent information. It has to send its previous parent a non-parent message to correct the previous parent message. Things are getting a little bit trickier here.

On the other end, if somebody is keeping track of its children, it has to make adjustments too because things can change during the algorithm. Let's say a process keeps track of its children in some set, it has a set children. If it gets a non-parent message from a child, even if that child might be from a neighbor. That neighbor might be in its set children, and this could be a correction, and then the process has to take that neighbor out of the set of children.

And suppose the process improves its own distance. Well, now, it's kind of starting over. It's going to send that distance to all of its neighbors again and collect new information about whether they're children or not. The simple thing to do here is just empty-- zero out your children set and start over. Now you send your new messages to your neighbors and wait for them to respond again. There's tricky bookkeeping to do to handle corrections as the structure of this tree changes, so getting child pointers is a little more complicated than before. Make sense?

All right, so now back to your question, termination. How do all the processes know when the tree is complete? In fact, we have a worse problem. With this problem we hit for breadth first search, but we have an even worse problem. Now what is that? Yeah?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah, before we had each individual node. Once it figured out who its parent was, it could just output that. And now, you can figure out your parent, but it's just a guess and you don't know when you can output this. How does a process even-- an individual process even figure out its own parent and distance?

There's two aspects here in termination. One is how do you know the whole thing is finished, but the other one is even how do you know when you're done with your own parent and distance? Well, if you knew something about the graph, like an upper bound on the total number of nodes, then you could just wait until that number of rounds and be done. But what if you don't have that information about the graph? What might you do? Yeah?

AUDIENCE: You want to BFS in parallel and filter down the information when you've reached the size of the graph.

PROFESSOR: Maybe. I think-- what is the strategy we use for termination for BFS? Let's start with that one. It's a little easier. You did the subtree thing that we call convergecast the information. When we had a leaf he knew he was a leaf, and he could send his done information up to his parent and that got convergecast up to the top of the tree.

Can we convergecast in this setting? Turns out we can, but since things are changing, you're going to be sending done messages and then something might change. You might be participating in the convergecast many times.

Since the tree structure is changing, the main idea is anybody can send a done message to it's current-- the node he believes is his parent, provided he's received responses to all of its distance messages so it thinks it knows who all its children are. It has a current estimate of all the children and-- so if it knows all its children and they have all sent him done messages. For all your current children, your current belief or who your children are, if they've all sent you done messages, then you can send a done message up the tree.

But this can get a little more complicated than it sounds, because you can change who your children are. What this means is that the same process can be involved several times in the convergecast, based on improving the estimates. Here's an example of the kind of thing that can happen.

Let's say you start out, you have these huge weights and then you have a long path with small weights. i 0 starts out and sends its distance information on its three nodes, to its three neighbors. And this guy at the bottom now has a distance estimate of 100, and it's going to decide it's a leaf. Why? Because when it sends messages to its children, to its neighbors, they're not able to improve their estimates based on the new information that he's sending. This guy decides that he's a leaf right away, and he sends that information back to node i 0.

On the other hand, this guy has the same estimate of 100. And he sends out his messages to try to find out if he's a leaf, but he finds out when he sends a message this way that he's actually able to improve that neighbor's estimate, because that was infinity till then. He doesn't think he's a leaf.

We have this one guy who thinks he's a leaf and responds, so this i 0 is sitting there. He has to wait to hear from his other children. OK so far? All right, in the meantime, the messages are going to eventually creep around and this node is going to get a smaller estimate based on the length of that long many hop path. Then he's going to decide he is not a child of i 0. He's going to tell i 0 I'm really not your child, which means that i 0 stops waiting for him.

But also, this guy decides he's not a child as well. He becomes a child of the node right above him. So i 0 now will realize he only has one child, but this guy believes he's a leaf again after trying again to find children. And now the done information propagates all the way up the tree, which is now just this one long path.

They start trying to convergecast, but then, oops, they were wrong. They have to make a correction, and they are forming a new tree. Eventually, the tree is going to stabilize, and eventually the done information will get all the way up to the top, but there could be lots of false starts in the mean time. It's sort of confusing, but that's the kind of thing that happens. Yeah?

AUDIENCE: There may be a process in which [INAUDIBLE] and then it just switches its mind at the very end of it. How do you make sure that, that propagation [INAUDIBLE]?

PROFESSOR: Yes, so the root node isn't going to terminate until it hears from everybody. You kind of have to close out the whole process. It's always pending, waiting for somebody, if it hasn't heard from someone. If things switch, they'll join in another part of the tree. I think the best thing to do here is sort of construct some little examples by hand. I mean we're not going to get into it how you do formal proofs of things like this.

We don't even have right now a simulator you can use to play with these algorithms. Although if anybody's interested, some students in my class last term, actually, wrote a simulator that might be available. OK, all right, so that's what you get for a synchronous-- some example synchronous distributed algorithms. Now let's look at something more complicated when you get into asynchronous algorithms.

OK. So far complications that you've seen over the rest of this course, you have now processes that are acting concurrently. And we had a little bit of non-determinism, nothing important, but now things are about to get much worse. We don't have rounds anymore. Now we're going to have processes taking steps, messages getting delivered at absolutely arbitrary times, arbitrary orders.

The processes can get completely out of sync, and so you have lots and lots more non-determinism in the algorithm. The non-determinism has to do with who's doing what in what order. Understanding that type of algorithm is really different from understanding the algorithms that you've seen all term and even synchronous distributed algorithms, because there isn't just one way the algorithm is going to execute. The execution can proceed in many different ways, just depending on the order of the steps.

You can't ever hope to understand exactly how this kind of algorithm is executing. What can you do? Well, you can play with it, but in the end, you have to understand is some abstract properties. Some properties of the executions, rather than exactly what happens at every step, and that's a jump. It's a new way of thinking.

We can look at asynchronous stuff, if you want, from my book. Now we still have processes at the nodes of a graph. And now we have communication channels associated with the edges. Now, the processes are going to be some kind of automata, but the channels will also be some kind of automata. We'll have all these components, and we'll be modeling all of them. Processes still have their ports. They need not, in general, have identifiers.

What's a channel? A channel is-- it's a kind of an automaton, infinite state automaton, that has inputs and some outputs. Here, this is just a picture of a channel from node u to node v. Channel uv is just this cloud thing that delivers messages. It has inputs. The inputs are-- messages get sent on the channel. You can have one process at one end sending a message m and the outputs at the other end are the deliveries, let's say receive message m, at the other end, node v.

To model this, the best thing is actually to-- instead of just describing what it does, to give an explicit model of its state and what happens when the inputs and outputs occur. If you want these messages to be delivered-- let's say in FIFO order, fine. You can make the state of the channel be an actual q. mq would just be a FIFO queue of messages. Starts out empty, and when messages get sent, they get added to the end, and get delivered they get removed from the beginning.

All that we need to describe-- this is a sort of a pseudo code. All we need to describe-- to write in order to describe what this channel does, is what happens when a send occurs, and when can this channel deliver a message, and what happens when it does that. A send, which can just come in at any time, and the effect is just to add this message to the end of the queue.

The recieve-- stop moving. The receive-- that cannot be construction. We'd hear noise. It's gremlins. We have-- a receive can occur only when this message is at the head of the queue, and when it occurs, it gets removed from the queue. Does this make sense as a description of what a channel does? Messages come in, get added to it, and then messages can get delivered in certain situations, and they get removed. That's a description of the channel. to be using an asynchronous system.

A process-- the rest of the system consists of processes associated with the graph vertices. Let's say pu is a process that's associated with vertex u. But I'm writing that just sort of as a shorthand, because u is the vertex in the graph, and the process doesn't actually know the name of the vertex. It just has its own unique ID or something, but I'm going to be a little sloppy about that now.

The process that at vertex u can perform send outputs to put messages on the channel, and it will receive inputs for messages to come in on the channel. But the processes might also have some external interface where somebody is submitting some inputs to the process, and the process has to produce some output at the end. There can be other inputs and outputs. And we'll model it with state variables.

Process is supposed to keep taking steps. The channel is supposed to keep delivering messages. It's a property called liveness. You want to make sure that your components in your system all keep doing things. They don't just do some steps and then stop.

Here's a simple example. A process that's remembering the maximum number it's ever seen. There's a max process automaton. It receives some messages m, some value, and it will send it out. It keeps track of the max-- that's max state variable. it starts out with its own initial values, so x for u. x of u is its initial value.

And then it has-- for every neighbor, it has a little queue-- here, it's just a Boolean-- asking whether it's supposed to send to that neighbor. This is the pseudocode for that max process. What does it do when it receives a message? Well, it sees if that value is bigger than what it had before, and if so, it resets the max. And it also makes a note that it's supposed to send this out to all its neighbors. Whenever it gets new information, it will want to propagate it to its neighbors.

You see how this is written? You just say, reset the max and then get ready to send to all your neighbors, and the last part is just sort of trivial code. It says, if you are ready to send, then you can send, and then you're done and you can set the send flags to false. Yeah?

AUDIENCE: What study?

PROFESSOR: For every neighbor. Oh, I wrote neighbor v before, and then I-- yeah. I wrote neighbor-- oh, I know why I wrote w. Here, I'm talking about if you receive a message from a particular neighbor v, then you have to send it to all your neighbors. Before, I used v to denote a generic neighbor, but now I can't do that anymore, because v is the sender of the message. Just technical-- OK?

We have these process automata. We have this channel automata. Now, we want to build the system. We paste them together. How we paste them is just we have outputs from processes that can match up with inputs to channels and vice versa. If a process has a send output, let's say send from u to v, that will match up with the channel that has send uv as an input. And the receive from the channel matches up with the process that has that receive as an input.

All I'm doing is matching up these components. I'm hooking together these components by matching up their action names. Does that make sense? I'm saying how I build a system out of all these components, and I just have a syntactic way of saying what actions match up in different components. Questions? This is all new.

When this system takes a step, well, if somebody's performing an action, and someone else has that same action-- let's say a process is doing a send. The channel has a send. They both take their transitions at the same time. The process sends a message, it gets put into the channel at the end of its queue. Make sense? OK.

How does this thing execute? Well, there's no synchronous rounds, so the system just operates by the processes and the channels just perform their steps in any order. One at a time, but it can be in any order, so it's a sequence model. You just have a sequence of individual steps.

There's no concurrency here. In the synchronous model, we had everybody taking their steps in one big block. And here, it's just they take steps one at a time, but it could be in any order. And we have to make sure that everybody keeps taking steps. That every channel continues to deliver messages, and every process always performs some step that's enabled.

For the max processes, well, we can just have a bunch of processes, each one now starting with its initial value. And what happens when we plug them together with all their channels between them? Corresponding to whatever graph they are in. They just have channels on the edges.

What's the behavior of this? If all the processes are like the ones I just showed you, they wait till they hear some new max and then they send it out. Yeah?

AUDIENCE: All processes eventually have a globally maximum value.

PROFESSOR: Yeah, they'll all eventually get the global max. They'll keep propagating until everybody receives it. Here's a animation if everybody starts with the values that are written in these circles. Now, remember, before they were all sending it once, now no more.

Let's say the first thing that happens is the process that started with five sends its message out on one of its channels, so the five goes out. The next thing that might happen is the other process with the seven might send the seven out on one of its channels. And these are three more steps. Somebody sends a 10. Somebody sends a seven. Somebody received a message and updated its value as a result, and we continue.

I'm depicting several steps at once, because it's boring to really do them one at a time, but the model really says that they take steps in some order. Everybody is propagating the largest thing it's seen, and eventually, you wind up with everybody having the maximum value, the 10. All right, that's how an asynchronous system operates. We can analyze the message complexity of this. The total number of messages sent during the entire execution, at worst, on every edge. You can send the successively improved estimate, so that's, again, what are n times the number of edges.

Time complexity is an issue. When we had synchronous algorithms, we just counted the number of rounds and that was easy. But what do we measure now? How do we count the time when you have all these processes and channels taking steps whenever they want? Yeah, so this really isn't obvious.

There's a solution that's commonly used, which is-- OK, we're going to use real time. And we're going to make some assumptions about certain basic steps taking, at most, a certain amount of time. Let's say that local computational-- time for a process to perform its next step, is little l. You just give a local time bound. And then you have d for a channel to deliver one message. The first message that's currently in the channel.

If you have assumptions like that, you could use those to infer a real time upper bound for solving the whole problem. I mean if you know it takes no longer than d to deliver one message, then you can bound how long it takes to deliver-- to empty out a queue, a channel, and how long it takes for messages to propagate through the network. It's tricky, but this is about the only thing you can do in a setting where you don't actually have rounds to measure.

Then for the max system, how long does it take? Well, let's just ignore the local processing time. Usually that's assumed to be very small, so let's say it's 0. We can get a very simple, pessimistic really, upper bound that says the real time for finishing the whole thing is of the order of the diameter of the network times the number of nodes times little d is the amount of time for a message queue to deliver its first message.

As a naive way of analyzing this, you just consider how long it takes for the max to reach some particular vertex u along the shortest path. Well, it has to go through all the hops on the path, so that would be the diameter. And how long does it have to wait in each channel before it gets to move another hop? Well, it might be at the end of a queue. How big could the queue be? Well, at worst end for the improved estimates.

Let's say it's n times the delivery time on the channel, just to traverse one channel. What I'm doing is modeling possible congestion on the queue to see how long it takes for a message to traverse one channel. Yeah?

AUDIENCE: Are we just assuming that processes process things, and messages are delivered as soon as they can possibly have them?

PROFESSOR: Good. Yeah, we normally have-- I'm sort of skipping over some things in the model-- but you have a liveness assumption that says that the process keeps taking steps as long as it has anything to do, and so we would be putting time bounds on how long it takes between those steps. That'll be the local processing time, here. I'm saying that's 0.

AUDIENCE: You do that-- wouldn't you be able to get some amount of information about what were things happening?

PROFESSOR: Ah, OK. Here is-- this is a very subtle point. What do you think is that if I'm making all these timing assumptions, the processes should be able to figure that out. But actually, you can't figure anything out just based on these assumptions about these upper bounds. Putting upper bounds on the time between steps does not in any way restrict the orderings. You still have all the same possible orderings of steps. Nobody can see anything different. These times are not visible in any sense. They're not marked anywhere for the processes to read. They're just something that we're using to evaluate the cost, the time cost, of the execution.

And you're not restricting the execution by putting just upper bounds on the time. If you are restricting the execution, like you had upper bounds and lower bounds on the time, then the processes might know a lot more. They might, in fact, be acting more like a synchronous system. These are times that are just used for analyzing complexity. They're not anything known to the processes in the system. OK? All right.

Now let's revisit the breadth-first spanning tree problem. We want to now compute a breadth-first spanning tree in an asynchronous network. Connected graph, distinguish root vertex, processes have no knowledge of the graph. They have you UIDs. All this is the same as before in the synchronous case. Everybody's supposed to output its parent information when it's done.

Here's an idea. Suppose we just take that nice simple synchronous algorithm that I reviewed at the beginning of the hour where everybody just sends a search message as soon as they get it and they just adopt the first parent they see. What happens if I run that asynchronously? I just send it and I get a search message. Then I send it out to my parents whenever and everybody's doing this. Whenever they get their first search message, they decide the sender is its parent. Yeah?

AUDIENCE: Could you possibly have a front tier that doesn't keep expanding-- not obeying the defining property of the BFS?

PROFESSOR: Yeah it could be that, because we don't have any restriction on how fast the messages might be sent and the order of steps, it could be that some messages get sent very quickly on some long path. Someone sitting at the far end of the network might get a message first on a long path and later on a short path, and the first one to gets it decides that's its parent, then it's stuck. This is not an algorithm that makes corrections. OK?

All right, well, before we had a little bit of non-determinism when we had this algorithm in the synchronous case because we could have two messages arriving at once, and you have to pick one to be your parent. Now that doesn't happen, because you only get one message at a time, but you have a lot more non-determinism now.

Now you have all this non-determinism from the order in which the messages get sent and processes take their steps. There's plenty of non-determinism, and remember, the way we treat non-determinism for distributed algorithms is it's supposed to work regardless of how those non-deterministic choices get made.

How would we describe? I'll just write some pseudo code here for a process that's just mimicking the trivial algorithm. It can receive a search message, that's the inputs, and it can send a search message as its output. It can also output parent when it's ready to do that. What does it have to keep track of? Well, it keeps track of its parent, keeps track of whether it's reported its parent, and it has some send buffers with messages it's ready to send to its neighbors, could be search messages or nothing.

This bottom symbol is just a placeholder symbol. What happens when the process receives a search message, just following the simple algorithm? Well, if it doesn't have a parent yet, then it sets its parent to be the sender of that search message, and it gets ready to send the search message to all of its neighbors.

That's really the heart of the algorithm that you saw for the simple algorithm for that is the synchronous case so that's the same code. The rest of the code is it just sends the search messages out when it's got them in the send buffers, and it can announce its parent once the parent is set, and then it doesn't-- this flag just means it doesn't keep announcing it over and over again. It makes sense that this describes that simple algorithm. It's pretty concise, just what does it do in all these different steps. OK?

Now, if you run this asynchronously, as you already noted, it isn't going to necessarily work right. You can have this guy sending search messages, but some are going to arrive faster than others. And you see you can have the search messages creeping around in an indirect path, which causes a spanning tree like this one to be created, which is definitely not a breadth-first spanning tree. The breadth-first tree is this one-- a breadth-first tree. You have these roundabout paths.

This doesn't work. What do we do? Yeah?

AUDIENCE: You could have a child [INAUDIBLE].

PROFESSOR: You're going to try to synchronize them, basically.

AUDIENCE: [INAUDIBLE].

PROFESSOR: That is related to a homework question this week. Very good. We're going to do something different. Other suggestions? Yeah?

AUDIENCE: We could keep a variable in each process that-- you can do something like Bellman-Ford.

PROFESSOR: Yeah, so you can do what we did for Bellman-Ford, but now the setting is completely different. We did it for Bellman-Ford when it was synchronous and we had weights. Now, it's asynchronous and there are no weights. OK?

Oh, there's some remarks on a couple of slides here. This is just belaboring the point, that the paths that you get by this algorithm can be longer than the shortest paths, and-- yeah, you can analyze the message and time complexity. The complexity here is order of the diameter times the message delay on one link.

And why is it the diameter even though some paths may be very long? This is a real time upper bound that depends on the actual diameter of the graph, not on the total number of nodes. Why would an upper bound on the running time for the simple algorithm depend on the diameter? Yeah?

AUDIENCE: Because you only have a longer path if it's faster than--

PROFESSOR: Exactly. The way we're modeling it-- it's a little strange, maybe-- but we're saying that something can travel on a long path only if it's going very fast. But the actual shortest paths still move along, at worst, at d time per hop. At worst, the shortest path information would get there within time d. Something is going to get there within time d, even though something else might get there faster. OK? All right.

Yes, we can set up a child pointers, and we can do termination using convergecast. It's just one tree. There's nothing changing here. And applications, same as before. But that didn't work, so we're going to-- back to the point we were talking about a minute ago-- we're going to use a relaxation algorithm like Bellman-Ford.

In the synchronous case, we corrected for paths that had many hops but low weight, but now we're going to correct for the asynchrony errors. All the errors that you get because of things traveling fast on long paths, we're going to correct for those using the same strategy.

Everybody is going to keep track of the hop distance. No weights now just the hop distance, and they will change the parent when they learn of a shorter path. And then they will propagate the improved distance, so it's exactly like Bellman-Ford. And eventually, this will stabilize to an actual breadth-first spanning tree.

Here's a description of this new algorithm. Everybody keeps track of their parent, and now they keep track of their hop distance, and they have their channels. Here's the key, when you get new information-- you receive some new information, this m is a number of hops that your neighbor is telling you. Well, if m plus 1, which would be your new estimate for a number of hops-- if that's better than what you already have, then you just replace your estimate by this new number of hops.

And you set your-- to a new parent. You set your parent pointer to the sender, and you propagate this information. It's exactly the same as what we had before, but now we're correcting for the asynchrony. We get shorter hop paths later. Makes sense? And the rest of this is just you send the message.

And notice we don't have any terminating actions. Why? Because we have the same problem that we had before with having processes know when they're done. If you keep get getting corrections, how do you know when you're finished?

And how do you know this is going to work? In the synchronous case, we could get an exact characterization of what exactly the situation is after any number of rounds. And we can't do that now, because things can happen in so many orders. We have to, instead, state some higher level abstract properties, either in variance or you'll see some other kinds of properties as well.

We could say, for instance, as an invariant, that all the distance information you get is correct. If you ever have your distance set to something, it's the actual distance on some path and your parent is correctly set to be your predecessor on such a path. This is just saying whatever you get is correct information. This doesn't say that eventually you're going to finish, though. It just says what you get is correct.

If you want to show that, eventually, you get the right answer, you have to do something with the timing. You have to say something like by a certain time that depends on the distance. If there is-- and at most our hop path to a node, then it will learn about that by a certain time, but that depends on the length of the path and the message delivery time. And the number of nodes, because they can be congestion. You have to not only say things are correct, but you have to say, eventually, you get the right result, and here it will say by a certain time you get the right result. Makes sense? This is how you would understand an algorithm like this one.

Message complexity. Since there's all these corrections, you're back in number of edges times possibly the number of nodes. And the time complexity, till all the distances and parent values stabilize, could be-- this is pessimistic again-- the diameter times the number of nodes times d, because there can be congestion in each of the links because of the corrections.

How do you know when this is done? How can a process know when it can finish? Idea? Before we had said, well, if you knew n, if you knew the number of nodes, you could weight that number of rounds. That doesn't even help you here. Even if you know-- have a good upper bound on the number of nodes in the network, there's no rounds to count. You can't tell. Even knowing the number of nodes you can't tell, so how might you detect termination? Yep?

AUDIENCE: It could bound on the diameter of [INAUDIBLE].

PROFESSOR: Yeah, well, but even if you know that, you can't count time. See this is the thing about asynchronous algorithms, you don't have-- although we're using time to measure how long it's termination takes, we-- the processes in there don't have that. They're just these asynchronous guys who just take their steps. They're ignorant of anything to do with time. Other ideas? Yeah.

AUDIENCE: Couldn't you use the same converge kind of thing--

PROFESSOR: Same thing. We're just going to use convergecast again, same idea. You just compute and your repute your child pointers. You send a done to your current parent, after you've gotten responses to all your messages, so you think you know your children. And they've all told you they're done and-- but then you might have to make corrections, so as in what we saw before, you can be involved in this convergecast several times until it finally reaches all the way to the root.

Once you have these, you can use them the same way as before. You now have costs that are better than this simple tree that didn't have shortest paths, because it now takes you less time to use the tree for computing functions or disseminating information because the tree is shallower.

Finally, what happens when we want to find shortest path trees in an asynchronous setting? Now, we're going to add to the complications that we just saw with all the asynchrony. The complications of having weights on the edges. All right, the problem is get a shortest path spanning tree, now in an asynchronous network, weighted graph, processes don't know about the graph again. They have UIDs, and everybody's supposed to output its distance and parent in the tree.

We're going to use another relaxation algorithm. Now think about what the relaxation is going to be doing for you. We have two kinds of corrections to make. You could have long paths that have small weight. That showed up for Bellman-Ford in the synchronous setting, so we have to correct for those. But you could also have-- because of asynchrony, you could have information travelling fast on many hops, and you have to correct for that as well. There's two kinds of things you're going to be correcting for in one algorithm.

This is going to-- and it's pretty surprising-- it's going to lead to ridiculously high complexity, message and time complexity. If you really have unbridled asynchrony and weights, this is going to give you a very costly algorithm. You're going to see some exponential is going to creep in there. Here's the algorithm for the asynchronous Bellman-Ford algorithm. Everyone keeps track of their parent. Their conjecture distance, and they have, now, messages that they're going to send to the neighbors. Let's say you have a queue because there could be successive estimates. We'll have a queue there.

The key step, the relaxation step, is what happens when you receive a new estimate of the best distance. This is weighted distance, now, from a neighbor. Well, you look at that distance plus the weight of the edge in between, and you see if that's better than your current distance, just like synchronous Bellman-Ford. And if it is, then you improve your distance, reset your parent, and send the distance out to your neighbors.

It's exactly like the synchronous case, but we're going to be running this asynchronously. And since you're going to be correcting every time you see a new estimate, this is actually going to handle both kinds of corrections, whether it comes because of a many hop path with a smaller weight, or whether it just comes because of the asynchrony. Whenever you get a better estimate, you're going to make the correction. Is it clear this is all you really need to do in the algorithm, just what's in this code? That's the received, and then the rest of the algorithm is just you send it out when you're ready to send.

And then we have the same issue about termination, there's no terminating actions. We'll come back to that. It's really hard to come up with invariants and timing properties, now, for this setting. You can certainly have an invariant like the one that we just had for asynchronous breadth-first search. You can say that at any point, whatever distance you have is an actual distance that's achievable along some path, and the parent is correct. But we'd also like to have something that says, eventually, you'll get the right distance.

You want to state a timing property that says, fine, if you have an at most r hop path, by a certain time, you'd like to know that your distance is at least as good as what you could get on that path. The problem is what are you going to have here for the amount of time? How long would it possibly take in order to get the best estimate that you could for a path of at most r hops? A guess? I was able to calculate something reasonable for the breadth-first search case, but now this is going to be much, much worse. It's not obvious at all. It's actually going to depend on how many messages could pile up in a channel, and that can be an awful lot, an exponential number-- exponential in the number of nodes in the network.

I'm going to produce an execution for you that can generate a huge number of messages, which then will take a long time to deliver and delay the termination of the algorithm. First, let's look at an upper bound. What can we say for an upper bound? Well, there's many different paths from v 0 to any other particular node. We might have to traverse all the simple paths in the graph, perhaps. And how many are there? Well, as an upper bound, you could say order n factorial for the number of different paths that you can traverse to get from v 0 to some particular other node. That's exponential in n. Certainly, it's order n to the n.

This says that the number of messages that you might send on any channel could correspond to doing that many corrections. This can blow up your message complexity into n to the n times the number of edges, and your time complexity n to the n times, n times d, because on every edge you might have to wait for that many messages, corrected messages, sitting in front of you. That seems pretty awful. Does it actually happen? Can you actually construct an execution of this algorithm where you get exponential bounds like that? And we'll see that, yes, you can.

Any questions so far? Here's a bad example. This is a network, consists of a sequence of, let's say, k plus 1-- k plus 2, I guess, nodes, in a line. And I'm going to throw in some little detour nodes in between each consecutive pair of nodes in this graph, and now let me play with the weights. Let's say on this path, this direct path from v 0 to vk plus 1, all the weights are 0. That's going to be the shortest path, the best weight path, from v 0 to vk plus 1.

But now I'm going to have some detours. And on the detours I have two edges, one of weight 0 and the other one of weight that's a power of 2. I'm going to start with high powers of 2, 2 to the k minus 1, and go down to 2 to the k minus 2, down to 2 of the 1, to the 0. See what this graph is doing? It has a very fast path in the bottom, which you'd like to hear about as soon as you can. But, actually, there is detours which could give you much worse paths.

Let's see how this might execute to make a lot of messages pile up in a channel. My claim is that there's an execution of that network in which the last node, bk, sends 2 to the k messages to the next node vk plus 1. He's really going to send an exponential number of messages corresponding to his corrections. He's going to keep making corrections for better and better estimates. And if all this happens relatively fast, that just means you have a channel with an exponential number of messages in it, emptying that will take exponential time.

You have an idea how this might happen? How could this node, bk, get so many successively improved estimates, one after the other? Well, what's the biggest estimate it might get? Yeah?

AUDIENCE: 2 to the k minus 1.

PROFESSOR: It could get 2 to the k minus 1? Well, let's see.

AUDIENCE: Or [INAUDIBLE].

PROFESSOR: It could do that.

AUDIENCE: Then it's 2 to the k.

PROFESSOR: And it could do that.

AUDIENCE: Plus 2 to the k.

PROFESSOR: Plus 2 to the k minus 2, plus all the way down to plus 2 to the 0. You could be following this really inefficient path, just all the detours, before the messages actually arrive on the edges on the spine.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah, so you follow all-- oh, you said 2 to the k, minus 1, parenthesis. Yeah, that's exactly right.

AUDIENCE: It's all right. [INAUDIBLE].

PROFESSOR: We can't parenthesize our speech. All right, so the possible estimates that bk can take on are actually 2 to the k-- as you said, 2 to the k minus 1, which you would get by taking all of the detours and adding up the powers of 2. But it could also have an estimate, which is 2 to the k minus 2 or 2 to the k minus 3. All of those are possible. Moreover, you can have a single execution of this asynchronous algorithm in which node bk actually acquires all those estimates in sequence, one at a time.

How might that work? First, the messages travel all the detours. Fine, bk gets 2 to the k minus 1. Then, there's a message traveling-- well this guy sends a message on the lower link. This guy has only heard about the messages on the detours up to that point. But he sends that on the lower link, which means you kind of bypass this weight of one. bk gets a little bit of an improvement, which gives it 2 to the k minus 2 as its estimate.

What happens next? Well, we step one back, and the node from node k minus 2 can send a message on the lower link, which has weight 0, to bk minus one. But that corrected estimate might then traverse the detour to get to node bk. If you get the correction for node bk minus 1, but then you follow the detour, you haven't improved that much. You've just improved by one. This way, you get 2 to the k minus 3 as the new estimate. But then, again, on the lower link, the message eventually arrives, which gives you 2 to the k minus 4.

You see the pattern sort of? You're going to be counting down in binary by successively having nodes further to the left deliver their messages, but then they do the worst possible thing of getting the information to bk. He has to deal with all those other estimates in between. If this happens quickly, what you get is a pile up of an exponential number of search messages in one channel. And then that information has to go on to the next node or the rest of the network, whatever. It's going to take an exponential amount of time, in the worst case, to empty that all out.

This is pretty bad, but the algorithm is correct. And so how do you learn when everything is finished, and how does a process know when it can output its own correct distance information? How can we figure out when this is all stabilized? Same thing as before, we can just do a convergecast. I mean this is more corrections, but it's still the same kind of corrections. We can convergecast and, eventually, this is going to convergecast all the way up to the root. And then the root knows it's done and can tell everyone else.

A moral here-- you've had a quick dose of a lot of asynchrony-- yeah, if you don't do anything about it and you just use unrestrained asynchrony, in the worst case, you're going to have some pretty bad performance. The question is, what do you do about that? There are various techniques. And if you want to take my course next fall, we'll cover some of those.

I'll say a little bit about the course. It's a basic-- it's a TQE level, basic grad course. We do synchronous, asynchronous, and some other stuff where the nodes really know about something about time. Here's some of the synchronous problems-- some like the ones you've already seen. Building many other kinds of structures in graphs, and then we get into fault tolerance. There's a lot of questions about what happens when some of the components can fail, or they're even malicious, and you have to deal with the effects of malicious processes in your system.

And then for asynchronous algorithms, we'll do not only individual problems like the ones you've just seen, but some general techniques like synchronizers, notion of logical time-- that's Leslie Lamport's first and biggest contribution. He one the Turing Award last year-- other techniques, like taking global snapshots of the entire system while it's running. In addition to talking about networks, as we've been doing this week, we'll talk about shared memory, multi-processors accessing shared memory. And solving problems that are of use in multiprocessors, like mutual exclusion.

And again, fault tolerance. Fault tolerance then gets us into a study of data objects with consistency conditions, which is the sort of stuff that's useful in cloud computing. If you want to have coherent access to data that's stored at many locations, you need to have some interesting distributed algorithms. Self stabilization-- if you plunge your system into some arbitrary state, and you'd like it to converge to a good state, that's the topic of self stabilization.

And depending on time, there's things that use time in the algorithms, and the newer work that we're working on in research is very dynamic. You have distributed algorithms where the network itself is changing during the execution. That comes up in wireless networks, and lately we're actually looking at insect colony algorithms. What distributed algorithms do ants use to decide on how to select a new nest when the researchers smash their old nest in the laboratory? That kind of question.

That's it for the distributed algorithms week. And we'll see you-- next week is security? OK, yeah.