NANCY LYNCH: OK so today, you're going to see something new. In fact all this week, you're going to see something that's quite different from what you've been studying in this course. These are algorithms. But they're for a completely different sort of model.

Distributed algorithms, OK, so what are they? So now instead of having algorithms that run on a typical computer, you're going to have algorithms that run on a network of processors. Or it could be on one machine that has multiple processors, multi processors that memory. Much of computing is distributed algorithms now. They solve problems like communication on the internet, data management over a network, allocating resources in a network setting, synchronizing, reaching agreement among different agents at remote locations. So these are all distributed problems, not things that you just solve on one computer.

The kinds of algorithms you design for these settings have to work under extremely difficult platforms because what you have is concurrent activity that's going on at many locations, many processors doing things at the same time. And you don't know exactly when everybody's going to perform their activities. You can have different sorts of timing uncertainty. The order of events isn't clear. There could be inputs that arrive at different locations. And then you also have to deal with failure and recovery of some of the processors or some of the channels involved in the computation. You don't think of any of this when you're just trying to run an algorithm on one computer.

So distributed algorithms can be pretty complicated. It's not easy to design them. And after you design them, you still have to make sure they're correct. So there are issues involved in proving them correct and analyzing them. A little bit of history, the field pretty much started around the late '60s. Edsger Dijkstra was one of the earliest leaders in the field. He won of the first Turing Awards. Leslie Lamport won the Turing Award last year. Although he actually started as a very young guy, way back in the early days of the field.

If you want to look at some sources, I have a book. There's another textbook by Attiya and Welch. There's a new series of monographs that basically try to summarize many of the

important research topics in distributed computing theory. And the last two lines have a couple of the main conferences in the field.

OK so I can't do that much in one week. What I'll do is just introduce the area, by showing you two common models for distributed networks. And just introduce a very few fundamental algorithms, and you'll see along the way some techniques for modeling and analyzing them. OK the two models here are synchronous distributed networks, and asynchronous distributed networks. The problems I'll look at in the synchronous setting are a simple problem of leader election, which is a symmetry breaking problem, basically. Maximal independence set problem, and then a couple of problems that should look familiar to you from the settings of this class, establishing structures like breadth-first spanning trees and shortest paths trees. In the asynchronous case I'll revisit these last two problems, setting up breadth-first and shortest path trees.

OK so I mentioned something about modeling in proofs and analysis. Turns out, getting the formal models right and getting real proofs tends to be pretty important for distributed algorithms because with all the stuff going on, they're complicated. And it's easy to make mistakes. The kinds of models that we use are interacting state machines, inputs and outputs. They send each other messages. But the kinds of proofs you do typically use invariants, a technique that you're very familiar with from this class. You can still use them in a distributed setting. And you still prove them the same way, by induction.

Something else that comes up a lot in the distributed setting is modeling and proofs using levels of abstraction. You might want to give an abstract description of an algorithm and prove that that works. And then you have a very detailed, complicated, lower level description that you can prove implements the higher level description. That's another popular technique. You use different kinds of complexity measures. For time complexity, you would measure rounds if it's the synchronous model, or some approximation to real time, if it's the asynchronous model. You also count communication, either the number of messages you send, or the total number of bits that you send in an algorithm.

So throughout these two lectures, we'll be looking at distributed networks. So you start with a graph. Let's just look at undirected graphs this week. We use n in this field for what you're calling v, the total number of nodes in the network or vertices in the graph. We use the notation gamma of u to mean the neighbors of u in the graph. So every vertex of the graph has a set of immediate neighboring vertices. That's gamma of u. And the degree of u is the

size of the neighborhood, the number of neighbors of the vertex.

OK so we start with the graph. But now we're going to plunk down a process, some kind of active entity at each vertex of the graph. So this is some kind of automaton. If you've taken automata theory, it's not really finite state machines, it's more like infinite state automata that can interact with each other. So we usually talk about vertices in a graph, processes at the vertices of a graph. But sometimes we get sloppy and just say nodes. And we could mean either the vertex or the active thing running at the vertex. Can't keep them straight all the time. OK and then with the edges of the graph, we would put communication channels, one in each direction, so that the processes can communicate over the edges. This week I'm not going to talk about what happens when you introduce failures because we just don't have time. A lot of distributed computing theory deals with what happens when some of the components in your system fail. How do you cope with that?

So we'll start right in with synchronous distributed algorithms. A source for that, if you're interested is the first technical chapter in my book. OK so you have processes at the nodes of a graph, like I just said. They communicate using messages. So think of each process as not knowing who his neighbors are, not knowing anything about the graph. So what do they have? They have ports. You could say they have output ports, on which they could send a message, and then some input ports on which messages can come in. So in general, the process doesn't know who the ports are connected to. It just has local names for the ports, like one, two, three, up to the degree.

If you have any questions just stop me and ask, if something's not clear. Otherwise I'll go pretty fast. And I know that none of this is familiar. So in general, the processes don't have to be distinguishable at all. So they don't have to have special unique identifiers so you could tell the processes apart. They could be completely identical. Well if they have different numbers of ports, they're not exactly identical. They certainly know how many ports they have, and release the local names for the ports.

Good so these are processes sitting at the nodes of the graph. What do they do? So they execute. And we talk about an execution of this network. It goes in synchronous rounds, and every round, every process looks at its state, and decides what messages it's going to send on all of the ports. So it could send different messages on different ports. So then what happens is all the messages that the processes decide to send get put onto the channels and they get delivered to the process at the other end. So the process of the other end is in some state. All

these messages come in. It updates its state, based on the arriving messages. So it changes state in response to whatever comes in.

And this is completely different from this semester so far. We're going to completely ignore the costs of the local computation. So each node can compute some complicated algorithm of the sort you've been studying in this class, and we usually don't consider that cost. We're more worried about the communication costs. And so we'll be focusing on the number of rounds that it takes, in the synchronous case, and the number of communication messages or bits.

OK so far? So let's start on the first problem. Here's a graph. The nodes start out possibly identical, but you want to somehow distinguish one of them to be a leader. So you have this arbitrary, connected, undirected graph. And exactly one process is supposed to elect itself the leader. That means it outputs a special leader signal. so exactly one should do that. So why do you want a leader? Well in practice, leaders can coordinate things. They can take charge of communication, and inform other nodes when they're allowed to send messages. They can coordinate the processing of data. Basically it allows you to centralize some of the computation. It can schedule the other processes. It can allocate the resources. It could help to reach agreement among the processes, if they start out with different opinions about what is supposed to happen.

All right so let's start out with a very simple case. You have a clique. Here's a four clique, where all the vertices are directly connected to all the other vertices, with two directional channels. And the processes are identical. So I should have asked you, instead of just giving the answer here, but are they able to elect a leader? So this theorem says that in general, that's impossible. Or it's not possible, in the most general case. If you have, no matter what n is, let's just say we have an n vertex clique for some n. It's not possible to have any algorithm that you can have all the processes run, if it's deterministic and the processes start out all indistinguishable. There's no way that they can elect a single node as a leader. So do you have an intuition for why that might be the case? Yeah.

AUDIENCE: They're all connected, and the cross-problem communication in one round is equal, then to be equal likely to select each one of them. It would be--

NANCY LYNCH: It's deterministic there's no likelihood here. And nobody is doing any selecting. You're talking as if there's somebody who's choosing a process to do something. There isn't anyone in charge. So this is a really different way of thinking.

AUDIENCE: So every node is essentially the exact same. So if it says, OK, let's assume I'm going to be leader, everyone is going to assume they're going to be leader.

NANCY LYNCH: That's exactly the right intuition. They can't distinguish themselves because they're always going to do the same thing. Let's look at a very simple case. Suppose we have just two nodes, two node clique, two nodes connected by channels. These are identical. They're deterministic. What can they do? Well you could try to design algorithms for one of them to elect itself as the leader. But you can show, by using induction, that the processes are actually going to remain in the same state as each other forever, however many rounds you execute.

So let's slow down. We can work by contradiction. Suppose you have an algorithm that solves this problem. Both of the processes, they're identical. They start in the same start state. Let's say there's a unique start state. So we could prove by induction on the number of rounds that after any number of rounds, say r rounds, the processes are still in identical states. So the inductive step is, all right, they're in identical states after some number of rounds. Let's look at the next round. They're in the same state. So they generate the same messages. So they each other the same messages. They receive the same message. And then they make the same state change. So they stay in the same state.

And you can tweak this, and say how this works for-- yeah, question?

AUDIENCE: So in what ways is the proof a contradiction?

NANCY LYNCH: I'm not finished. You're exactly right. We have to finish by using the requirements of the problem. Since the algorithm has to solve the leader election problem, the requirements say that eventually, one of them has to output leader. And what happens when he does? Anyone? Yeah.

AUDIENCE: You have node also outputting the leader signal.

NANCY LYNCH: Yeah the other one would also do the same thing. We're saying round by round, they stay in the same state. So as someone said before, when one guy outputs leader, at the same round the other guy will output leader as well. So that's a contradiction to the problem requirements. Notice we didn't assume anything at all about exactly how the algorithm works. We're just saying, however it works, it can't solve this, under the assumptions that the nodes are indistinguishable and deterministic.

So as you can see, this will extend if you have larger cliques of size n. So now the process has not just one output port, it has n minus 1 output ports to connect to all the other nodes. Let's say they're numbered 1 through n minus 1. And one of the possibilities, and one I'll use in this proof is that the ports happen to be numbered consistently. So that if you have output port number k at one node, it's connected to input port number k at the other end. So that's one way things can match up.

All right if that's the case, we could do the same proof we just did. Show by induction that all the processes in the clique remain in the same state forever. So same proof. Suppose you have an algorithm that's solves it. They all began in the same state. You show by induction that they all remain the same state. Well so now we slow down a little bit. Each process sends a possibly different message on each port. But everybody sends the same message on port k because they're all indistinguishable. And then because the way the ports match up, everybody receives the same message on port k. And then they make the same state changes.

AUDIENCE: Does this proof imply that there's a kernel for simplifying the graph when you find a clique?

NANCY LYNCH: No because if you have a graph that consists of a clique and then let's say, some other stuff, maybe the leader could be somebody outside the clique. So you can't just say because there's a clique that you can't elect a leader because you could break the symmetry of the graph with other stuff in the graph. Yeah?

AUDIENCE: What assumptions do we make to know that for each k, they receive the same message?

NANCY LYNCH: Because everybody is going to send the same message on the same numbered port, because they're identical. And one way the ports can be hooked up, and we have to tolerate all ways they could be hooked up-- say an adversary hooks them up-- is that port k, somebody's output port, is the other end's input port numbered k. So then they all receive the same message on their port number k. Yeah?

AUDIENCE: Is it actually possible to always hook up the boards that way. I mean, it's like wrapped with three vertices.

NANCY LYNCH: Well I'm just doing it for cliques. Yeah it is. Yeah you could do it. I mean you could have port one always going clockwise, and port two going counterclockwise, I mean, there's always a way to do that in a clique. I checked that. So what you've just seen is one of the very basic

problems for distributed algorithms, which is breaking symmetry among identical processes. And you see that deterministic, indistinguishable processes just can't do it. So we have to have something more. So what do you think we could add to make this problem solvable?

AUDIENCE: [INAUDIBLE] processes.

NANCY LYNCH: I can't hear.

AUDIENCE: Probability. Probability, OK, anything else? So we could have the processes actually distinguishable. The common way in this area is to say that each process has an identifier. Like, you buy a chip and it's got some identifier burned in. OK so you have some kind of unique identifiers. Or you can use randomness. OK for unique identifiers, you assume everybody has some number or some identifier that it knows what it is. It's built into its state, let's say, a special state variable. They're totally ordered, generally. They could be integers, or from some totally ordered set. When you say unique identifiers, is it means that different identifiers could appear any place in the graph. But each identifier can appear at most once. You can have a huge identifier space in a small graph. But you're Just selecting some identifiers to put in the processes in the graph.

So that's one set up. And the other one, of course, is using randomness. So let's look at the unique identifiers first. Now the problem becomes easy. Let's look at the clique again. Suppose there's an algorithm-- well, let's construct an algorithm that consists of deterministic processes with unique identifiers. And we're going to guarantee to elect a leader in the graph. And moreover, it's just going to take one round of communication. And it's only going to use n squared messages. How could that work?

Everybody in this click has a unique identifier. What would they do? Send it out, right? So you can just send it on all your ports. Everybody would send its unique identifier on all its output ports. And then they collect the unique identifiers from everyone else. So everybody sees the same set of identifiers. And so the process with the maximum unique identifier knows that it's the only one with that identifier. And it's the biggest one. So it can elect itself the leader. So all you is unique identifiers and the ability to exchange them reliably. And you can elect somebody easily.

Randomness, well, various ways to do it. But one idea is the processes could just choose identifiers randomly. You take a sufficiently large set of possible identifiers, and so if they just choose uniformly at random, they're likely to choose all different identifiers. Once you have

these randomly chosen identifiers you could use them like the really unique identifiers. The only thing is you might, there's a small chance that you'll have a duplicate. In which case, you want to be able to detect that and repeat this.

So first of all, how big the a set do you need? Well here's an example. Suppose that you have the n processes choosing at random, independently from a space of size r. Identifiers are the numbers one through r. OK and r is going to depend on n. It's going to be like n squared, but it's also going to depend on epsilon, which is the error probability that you're interested in. Turns out that n squared over 2 epsilon is good enough. OK so you have your IDs space at least that large. And then you can guarantee that with probability at least 1 minus epsilon, all the numbers that everybody chooses are different.

It's a very easy proof. The probability-- just look at two particular processes-- what's the probability that they choose the same number? It's just 1 over r, right. Because they're both choosing at random. The first one chooses something. The probability that the second one chooses the same thing is just 1 over r. But now you can take a union bound, just add up the probabilities of any pair having a duplicate. And so you have n square around n squared over 2 pairs. And so multiplying 1 over r by n squared over 2 still keeps your probability less than or equal to epsilon, your error probability. So you can choose identifiers using randomness. With large enough space, with very high probability, you can get them to be all different.

And now here's how the algorithm works. So you get an algorithm that would finish in only one round, with probability 1 minus epsilon. But it will be correct. And it will have repeated rounds, in case the first round doesn't work. But the expected time is just 1 over 1 minus epsilon, not very big. What's the algorithm? Well processes just choose the random IDs from the big space, like we just said. They exchange their Ids. And now, everybody can see everyone's ID, but they also can tell if there's a duplicate. if the maximum is not unique.

So if the maximum is unique, find the maximum wins. And everyone knows that. Otherwise you have a problem. And you have to repeat. And you just keep doing that until you succeed. So this can just continue, but it's likely to finish very fast, if you have a high likelihood of having no duplicates. Questions about the leader election? So the story was, it's impossible without something to help you distinguish some processes. You can do it with unique identifiers. You can do with randomness.

Second problem is called maximal independent set. So you have a picture of a maximal

independent set in a graph here. Let's try this. Yeah cursor. So the maximal independent set in the graph is here. But this is something I'll come back to a minute. This is actually a use of the maximal independent set to model what happens in a certain kind of biological system.

What's a maximal independence set? So you start with a general, undirected graph network. And the problem is to choose a subset of the nodes so that they form what we call a maximal independent . Set let's break that down. What does this mean? Independent means you don't have any two neighbors that are both in the set. So you don't want to get two neighbors in the set. Maximal means that whatever set you choose, you can't add any more nodes without violating independence. So now this should look something like a couple of homework problems that you had from the beginning and recently.

But I'm not saying that it's maximum independent set. I'm not saying you have to have the global, largest number of nodes. I'm just saying it has to be a local optimum, in the sense that you can't add any more nodes to your set without violating the independence property. Make sense? There's two examples, the same graph, two different maximal independent sets. The green nodes, here we have four green nodes that are independent, not neighbors of each other. And they're maximal, in that I couldn't add any of the red nodes into a set without violating the independence property. But then over here, we have a second maximal independent set for the same graph. Now we just have two nodes. And you can't add any of the red nodes without violating the independence property. In other words, every node is either in the MIS, or has a neighbor in the MIS. There's nothing else you can do to add notes to the MIS

So the notion of maximal independence, that make sense? All right, so to make this a distributed problem, let's start out assuming we have no unique identifier. Actually, for this whole problem, we're not going to have unique identifiers. They're all going to be identical. The processes do need one piece of information, which is some approximation to n, the size of the network, the total number of vertices. So we would like to have these nodes somehow cooperate to compute an MIS of the entire network graph. What that means is every process should find out whether it is in the MIS or not. If it is, it should output n. And if it's not, it'll just output out.

So you don't have to actually compute this, like you're used to solving problems like this, where somebody has to gather all the information in one place. Nobody gathers anything. Everybody just has to know whether or not they're in the MIS. So as you can imagine, this is

going to be unsolvable in certain graphs by deterministic algorithms, by the same kind of symmetry breaking problems that you saw for leader election. So we're going to move right to randomized algorithms for this problem.

Some applications of distributed MIS, well they come up in communication networks, where you want to choose-- let's say you have a very dense network of processes. You want to choose just a few nodes, which would be like an overlay network. You would choose some nodes who could take charge of communication that you can communicate on this overlay network, and then in the end, each node can take care of communicating with its many neighbors. So that's a common sort of application.

But it also comes up in other places. A great example is in developmental biology, where a couple of years ago, there was a paper in Science by Afek, Alon-- there's like eight authors on that. But Ziv Bar-Joseph was the lead author of this paper. So the idea is you have a bunch of cells in a fruit fly. And during development, some of those cells are supposed to distinguish themselves as being what's called sensory organ precursor cells. The properties that you want it that actually, you would like a maximal independent set of the cells to become distinguished in this way. So they wrote a paper about it, got published in Science. They basically designed a new distributed algorithm that closely mirrored what happened in the fruit fly during development.

So what I'm going to show you is a very well-known algorithm, a classical algorithm for MIS. This is by Michael Luby. Very simple algorithm, it executes in phases. Each phase has two realms. So you start out with all the nodes being active. They're all involved. They don't know what they're going to end up with. And at each phase, some of the active nodes are going to decide they're in the MIS. Some others will decide they're out of the MIS. And some others won't know yet. So then you just continue to the next phase, with all the remaining nodes and the edges between them. So you're basically going to settle what happens with some subset of the nodes, and then reduce the graph and continue.

So that's the algorithm. So what do you do in each phase? Here's what an active node does at a phase. Two rounds. The first round, it picks a random value in a large space, the same kind of idea as before. This time it's 1 up 2 n to the fifth. It sends that random value to all its neighbors, receives the values from all its still active neighbors, and then it just looks to see if its value is greater than all the values it received. So then it's a local maximum. It has chosen a value that's strictly greater than the values chosen by all its neighbors. So then it decides to

join the MIS and it outputs in.

But now you want to make sure none of its neighbors-- you know that none of its neighbors are going to join the MIS at round one. Because you know this guy's chosen value is larger, strictly larger, than all its neighbors. But now you want to tell them that they should not join. They should be out. So if you join the MIS you're going to announce that by sending messages to all your neighbors. And then anybody who receives an announcement can decide it's not going to be in the MIS and it outputs out. Because it knows it has a neighbor that's in the MIS.

So if you decided in or out at this phase, you're done. You become inactive. And only the remaining active guys continue to the next phase. Make sense? any questions about how the algorithm works? And animation. All right so all the nodes start out identical. They all pick IDs. So here's some numbers that they pick. So which nodes are going to now join the MIS? 16, and the one that chose 13. Good, so they're in the MIS. And then at the same phase, all of their neighbors, those for red nodes, are going to decide to be out of the MIS And now you're left with the remaining four nodes. We don't keep going with the same IDs. we start over. We want the rounds to be independent.

So if they choose again, they get new IDs. And now the guy with the 12 and the guy with the 18 going to join the MIS at this phase. And their neighbors will decide not to be in the MIS. That leaves us with just one mode, the guy who had four. Next phase, he chooses another ID. But he has no neighbors so by default, he's bigger than all the neighbors. So he just joins the MIS. So that's how this works. Very simple algorithm, and it actually works to find an MIS very quickly.

Why does this give you independence? How do we know that if this ever terminates, if everybody decides, how do we know that we don't ever have two neighbors that decided to be in the MIS? Yeah.

AUDIENCE: Because once a node joins the MIS, it broadcasts to its neighbors that--

NANCY LYNCH: Right. The only way you join the MIS is if you have the unique maximum value in your neighborhood. And when you do, all your neighbors become inactive. So you're certainly going to have independence.

Maximality, if it terminates, the final set is not going to allow you to add any more nodes. Why?

Because a node is only going to become inactive if it joins the MIS, or a neighbor joins the MIS. And we just continue this algorithm until all the nodes become inactive. So either the node is in the MIS or a neighbor is in the MIS. So you can't possibly add any more. Yes? So this has the basic correctness properties, but what you're probably wondering, is why is this efficient enough? Why is it efficient? Well we could say that with high probability, of probability 1, it will eventually terminate. More quantitative, we can state this theorem that says, with probability at least 1 minus 1 over n, all the nodes decide within four log n phases. Since n is the number of nodes, this doesn't tell us that you get probability 1 of eventually terminating. But we can repeat this and get the same sort of bound repeatedly for successive phases.

But let's just focus on getting probability at least 1 minus 1 over n that all nodes decide within about four log n phases. So let's see what this is saying. You have this big complicated graph. And in one round, for this to be like log n behavior, what has to happen at each phase? You have to reduce it by some constant fraction. The number of nodes, say, should go down. So it's sort of how the proof will go. So we start out with a Lemma saying, you're choosing these IDs at random. You want a high probability that they're all different. So we have a lemma like the one we had before. It says, the probability at least, we use 1 minus 1 over n squared, in each phase. All these phases up to four log n, everybody's choosing a different random value. All the nodes choose different values at each phase.

So this lets us ignore the possibility that you have repeats. So we'll come back to that at the end. All right, so we're going to pretend that in each phase, all the random numbers are different. So the key idea of this is to show that the graph has to shrink enough at each phase. So the way we're going to say that is not in terms of the nodes, but in terms of the number of edges. We're going to say at each phase, the expected number of edges that are live-- why is that shaking? OK.

The expected number of edges that are live at the end of the phase is at most half the number that were live at the beginning of the phase. So an edge is live, if its endpoints are still live. So instead of talking about reducing the number of nodes by a constant fraction, I'm going to reduce the number of remaining edges by constant fraction of each phase. So this is what I'm going to prove. So now I've got only three slides, but the only three slides today that have calculations on them. So probably have to pay attention, if you want to follow the calculations online. So let's see why.

But the goal is clear? We have to reduce the number of edges that remain by a factor of two.

So this is actually a new proof of this algorithm's performance. The proof in the original papers is pretty complicated. This is a very intuitive, neat proof. So the first line of the proof says if you have a node that has a neighbor that chooses a value that's bigger than all of its own neighbors-- so u has a neighbor w. W chooses a value that's bigger than all w's neighbors. But let's say more. Let's say it's also bigger than all of u's other neighbors, besides w. So w is really big, bigger than all w's neighbors, bigger than all of u's other neighbors. If that happens, then what happens to u? Well we know that w is going to decide to join the MIS. And u is going to definitely die, is not going to join the MIS. Right?

OK? I don't want to lose people in the first line. Question? Here's a picture. Here's u. And it has a neighbor w. And let's say that w's chosen value is greater than all of w's neighbors, but also greater than all of u's other neighbors. Yes? If w has that, w is going to join the MIS, and u is going to definitely not join the MIS. It's going to decide out in this phase. OK so far?

AUDIENCE: Why does you need w to have value greater than u's neighbors? Because if w is greater than all of its neighbors then it's--

NANCY LYNCH: --be in the MIS and u will not be in the MIS. And that seems like it ought to be enough. But look at the next line. Well the line after this one. What's the probability that w chooses a value like that? So if it's going to be bigger than all u's neighbors, and all of w's neighbors, and keeping in mind that they are each other's neighbors, turns out that there is degree u, at most degree u plus degree w nodes involved here. W has to have the biggest of all of those, so it's going to have the probability 1 over the number of nodes of being the biggest one. So it's just 1 over the degree of u plus the degree of w, the probability that w will choose a big enough value.

But you ask, this is pessimistic. Why don't I just say that w is bigger than its own values? Because I want to do this next step. I want to say the probability that node u gets killed by one of its neighbors, any one of its neighbors in this phase. I can calculate that as the sum. The probability that node u is killed by a neighbor is at least the sum over all of its neighbors. You look at all the vertices in the neighbor set, and you add up this fraction.

So why did I need to make that additional assumption before? That w is greater than all of u's neighbors, as well as all of its own neighbors. Yeah?

AUDIENCE: So you can add a problem to--

NANCY LYNCH: Yeah because otherwise these would be overlapping events. But this way I know they're definitely disjoint events. We can't have-- if we have w and w prime, you can't have both of those holding because the requirement for w is saying that its ID is bigger than w prime's ID. Because you have these disjoint events, you can just add the probabilities. And you know that the probability that u gets killed by some neighbor is at least this summation. OK so far?

So now I'm going to calculate. But I wanted to focus on the edges. So let's see, this tells us a way that a node can get killed. But let's look at what happens for an edge getting killed. This is the probability that a node is killed. So the probability that an edge dies at this phase is at least the maximum of the probability that either of its two endpoints die. And let's just write it as the average. The probability that an edge dies is at least the average of the probability that it's two endpoints are killed, in this way. So for an edge, an edge is definitely going to die, if one of its endpoints dies. And then the edge dies if it dies in this particular way.

So the probability an edge dies is at least the probability that one of the-- half the sum of the probabilities that the two end points die. It's the average probability. Makes sense? You might have to read this later. So now we can go from that to the expected number of edges that die. What is that? You just add up, over all, the edges, the probability that the edge dies. The expected number of edges that die is at least the sum over all of the edges of the probability that the two endpoints die. So you have the sum, over all of the edges. You add up for all the edges. The probability that one endpoint is killed, and the probability the other endpoint is killed.

So what we have is this great big summation involving now the kill probabilities for vertices. So we have the kill probability for each vertex. How many times does that occur? If you have a vertex, u, it appears once for every edge that u is an endpoint of. So you have the kill probability for each node occurring exactly it's degree number of times. So that lets me rewrite this in terms of vertices. This sum is just 1/2 the sum over all the nodes of the probability that the node gets killed times its degree.

So I'm calculating by replacing the description in terms of edges, by description in terms of vertices. More or less OK so far? So now what do I do? Well, I know the probability that u is killed. I have a bound for that up on the first line. So I'm just going to plug that in. So I get 1/2 the sum over all the nodes, the degree of the node times this summation that gives me the kill probability for that node. And now I play around with the sum. I can move the degree inside the second summation, and I get this.

So now let's stare at this again. I have the sum over all nodes of the sum over all of its neighbors of some expression. But if I'm considering a node, every note and every one of its neighbors, that's like considering all the directed edges. I look at every u, and I look at every edge that connects u to something else. So I can write it as the sum over all the directed edges of this expression. So I get half of the sum over all the directed edges of this expression.

But we were talking about undirected edges. And the undirected edges are being twice here, once for each direction. I can change this sum to a sum over undirected edges. But now I have the two endpoints to deal with. So I get the degree of u and the degree of v in the numerator because I'm looking at it from the point of view both of the endpoints of each edge. Well something drops out, so I have 1/2 the sum over all the undirected edges of 1. So that's 1/2 of the number of undirected edges.

So I don't expect you to get every step of this, but it's on three slides, so you can stare at this when you go home and make sure the steps work. But remember the point of this is to show that you reduce the number of edges by a factor of two, and it's done and sort of a clever way by counting the kill probabilities of vertices. So we get this, reducing the number of edges. And now we can just plug that back in to get our complexity bound for the entire algorithm. Remember the original theorem you're we were to prove is a probability bound for deciding within log n phases. Well you should have a pretty good idea of why that works because if at each phase, you're going to reduce the number of edges by around a factor of two, then it's going to take something like log n phases to finish.

And I just put a proof sketch. The number of edges that are still alive after four log n phases, well you divide by 2 four log n times, so you get down to practically nothing. The probability any edges are alive at the end is very small. So you get a small probability the algorithm doesn't terminate within four log n phases. There's an extra little term I threw in here. You might have forgotten. There was a term that I needed for the small probability, that somebody chose duplicate IDs. So I'm bringing them back in at the end, in a little union bound. And we get our 1 over n probability this way. But the key idea is you reduce the number of edges by half at each stage.

Enough for you to look at later, I guess to figure this out or you have any questions about this? So that's the last equations and calculation. I'm going to go onto a new idea, more conceptual

stuff. Familiar problem, breadth-first spanning trees, setting up breadth-first paths to every node, but we're going to study it in our new setting. We have a connected graph. This time, let's suppose that it has a distinguished vertex, like it already has a leader. So it has a distinguished vertex in the graph that's going to become the root of the BFS tree. And the processes don't need any knowledge about the graph for this one.

For the rest of the time today and Thursday, we'll assume the processes have unique identifiers, and I don't think we're using any probabilities. So this is just going to be using the unique identifiers to solve our problems. So everybody knows its own unique identifier. The root has a distinguished, generally known, unique identifier say i0. And the process that has i0 knows hey, I'm at the root of the graph. So the set up make sense?

We might as well assume that everybody knows the unique identifiers of their neighbors because they could easily exchange information now, and match up who's connected on which port by a unique identifier. We'll just do deterministic. There'll be a little bit of non-determinism here. I'll say more about that. But I'm not going to worry about probabilities for this. Well that told you about the general setup. What are the processes supposed to do? Well they're supposed to compute a breadth-first spanning tree, rooted at vertex v0. The branches are going to be directed paths in this undirected graph, coming from v0. Spanning means they should reach all the vertices. And breadth-first means that if a vertex is at a distance d from v0, it will appear at depth d in this spanning tree.

So everybody should get a shortest path from the root. Now how are we going to compute this in a distributed setting? Well now the output of a process is just going to be its parent in the tree. So we're not actually going to compute this tree anywhere as a whole. Everybody's just going to know its parent in the tree. Questions? Problem make sense?

So this is just an example of a spanning tree, breadth-first spanning tree. This gives you shortest paths to all of the nodes, , shortest in terms of the number of hops. So we can have a very, very simple algorithm. We're going to let the processes mark themselves as they get included in the tree. Starts out only the first process, i0, is marked. So do you want to give an idea, maybe, of how this might work? Sketch out-- yeah?

AUDIENCE: The root will send out to its neighbors. And they will then mark themselves as the parent of whoever they heard from. Then they will--

NANCY LYNCH: This is all synchronous. So that's great. They'll be doing this in synchronous rounds. So everybody will, at the certain distance, is going to get the message at the right number of rounds to mark their distance. OK so in round one, process i0 will send a special message, say search, to all of its neighbors. And anybody who receives a message in round one will mark itself, decide i0 is its parent, could output that i0 is my parent, parent i0. And then it can get ready for the next round, when it's supposed to send to continue this.

So at later rounds, if you decided you're going to send, if you know you're supposed to send from the previous round, then you send a search message to all of your neighbors. Now the process is sitting there and it receives a search message. If he's already marked, then he should just ignore the message. Once you're included in the tree, you don't care if you get other messages, search messages on other paths. So you only do anything if you're not yet marked and you receive a message. And in that case, then you mark yourself. Then you mark yourself, and then you choose one of your neighbors as to be your parent.

Now because this is synchronous, you have several nodes that could be sending at the same time. So one node could be receiving search messages from several different neighbors at once. Well, it wants to choose one of them as its parent, doesn't matter which one it chooses. So it can just choose nondeterminstically just arbitrarily. And then it decides that it will send the next round. Is the algorithm clear?

So there's, I mentioned, a little bit of nondeterministic here, only in that a process can choose arbitrarily among several possible parents. And then we could put in a default, saying that it chooses the one with the smallest ID, if we really want to make it deterministic. But it's also OK to leave distributed algorithms nondeterministic. And here I should make a remark that shows how differently nondeterminism is regarded in the distributed setting, from the way it is for sequential algorithms. For distributed algorithms, there can be many options. And maybe they're all OK. But the algorithm is supposed to work correctly, no matter how you resolve the nondeterministic choices.

So think about like np, and the other ways that you've seen nondeterminism so far. There you say you're lucky if there is a path to a choice. Here when you make a nondeterministic choice, or when the algorithm behaves nondeterministically, all the choices are supposed to work. It's like all the paths have to come up with correct answers. Do you have a question?

AUDIENCE: Yes, whenever there's a sub- [INAUDIBLE], whenever there's a race condition, we locally

assume that there wasn't a difference in local computation time. But if there is, even in the slightest, then they would get a parent [INAUDIBLE] before another one, it would still be a valid--

NANCY LYNCH: So the synchronous model is more abstract than that. You don't model the local computation time. You're moving more toward an asynchronous model, where the steps can take differing amounts of time. Here we just assume you have an abstract model, where everybody does stuff at once, in each round. But you still have nondeterminism because they can all arrive at the same round somewhere. But it's OK. You can pick any one and it still works.

So it should be not hard to see that this does give you a BFS tree because you're creating all the branches synchronously. And you're growing one hop at each round. It reaches all the nodes eventually because the graph is connected. And everybody sends messages once a node get marked. It sends messages to its neighbors. So eventually, the markings are going to reach all the neighbors, all the nodes in the graph. So here's how you get the example I showed before, simple breadth-first search. That's a search message sent by this guy. I put it to the right of the edge to indicate-- it's kind of hard to distinguish. But I put them on the right of the edge from the point of view of the sender.

So he sends a search message. it gets there. This arrow just indicates that it reached the other end. And this guy has chosen the sender, which is the other direction on the arrow, as its parent. Now the recipient is going to send some search messages. So he sends four of them. They all get to the other end. And OK, so all these guys now get marked. They're included in the BFS tree. And now the next round, they all send some messages. I'm not putting in the messages where somebody would send back to a guy who sent to him. But I put in all the others. Some of them are going to be ignored. But you do get to a few new nodes this way. That's round three. Round four, everybody sends. And now you have all the nodes included.

So this gives you the spanning tree that I showed at the beginning of this topic. This is not a very complicated algorithm. But I think you can see that things can get worse. And you want to argue about why the algorithms work correctly. So as I said before, a popular method of reasoning about the algorithms is to state invariance. So here, suppose I want to describe the state of the entire network, after some number, r, of rounds. what could you say about that? What's the case after r rounds of this algorithm? Yeah.

AUDIENCE: All nodes at distance r from the root have been marked.

NANCY LYNCH: All the nodes at distance r from the root have been marked. In fact, only those by round r, only the ones with distances up through r have been marked. So to state the invariance, if you want to state invariance, I have to say what's in the state of the processes. So all right, what can we say? So the process has a Boolean that says whether or not it's marked. It has a place to record a parent. And it has someplace where it puts information about whether it's supposed to send a message at the next round. And we also should know its UID, so I'll put that in another state variable.

So here is something I can say in invariance. At the end of r rounds, as you said, at the end of r rounds exactly the processes at distance at most r from the source node, the root node, are marked. I can say a little more. I can say a process has its parents defined if and only if it's marked. So it doesn't just get market. It also computes a parent, and the parent gets computed at the point where it gets marked. Then I should say that the parent is correct. So for any process that's at distance d from the source, if the parent is defined, then it's in fact the UID of a process at distance d minus 1 from the source. So that says it's actually getting a correct breadth-first tree. It's getting the parent on a shortest path. Yeah?

AUDIENCE: Do these invariants [INAUDIBLE] for i0?

NANCY LYNCH: Distance 0 is marked. i0 doesn't ever-- I see what you're saying. i0 doesn't have a parent. So I guess that we should say for i not equal to i0 in this case. So this would be a process other than i0. It would have its parent defined, if and only if it's marked. Well as I think you just noticed, the root node is marked but it doesn't have a parent. So it's an exception. But this should be, this doesn't involve i0. So the second one, I can fix that a bit. Other comments, questions?

So if somebody wanted to do a formal correctness proof of an algorithm like this one, you would use these invariants. You prove it by induction. In fact there's quite a few people who use interactive theorem provers to do proofs like this because the algorithms can get pretty complicated, with a lot of variables. So you have to do some bookkeeping. You keep track of all these invariants, and then you want to prove that they're all true by induction. They all hold through an inductive step. So you can use an interactive theorem prover to help you do the bookkeeping. But even a manual proof in a research paper would use invariance in this style.

OK complexity. So the number of rounds until everybody outputs their parent would be the maximum distance of any node from v0. So we can say that's at most the diameter of the

graph. It could be less. It's just is the maximum distance from this particular node.

Message complexity? Well how many messages are sent in this algorithm? So everybody is going to send messages only once on all of its edges. So that means all the edges get a message sent in each direction just once. So it's order of the number of edges. All right, so we can play around with this. So this algorithm just tells everybody who his parent is. But maybe when you're finished, you'd like to who your children are as well. For many uses of these trees, you'd like to have a parent be able to talk to its children in the tree.

So how to do that? Well you can add a child pointer because anybody who gets a search message and selects its parents could send back a message to that parents saying, hey, I'm your child. And if you get a search message, and you decide that that's not your parent, you can help that guy out by sending a message saying you're not my parent. In the synchronous case, he would just know that, if he didn't get a parent message. But things are going to get more complicated. So we'll send parents or non parent responses to the search messages.

Suppose we want to compute the distances from v0, not just to the parents are. Well that's easy. Everybody can just record its distances, as well as its parent and the mark. And then you just include your own distance value in your search message. And when somebody receives a search message, it sets its own distance to the received distance plus 1. So we can just keep track and add one to the distance. It's easy to augment this algorithm to get this extra information.

All right, now how do the processes know when this is all finished? So everybody was able to output parent. I know who my parent is. But how does anybody know when the entire tree has been produced? Not so obvious. So in some settings, you might know an upper bound on the depth of the tree. And then you could just wait for that number of rounds. But what if you don't know that? You don't know anything about the graph. Nobody knows. So let's come up with an algorithm for process i0, the root, to know definitively that the tree has been completely constructed. Ideas? You're creating this by search messages. How is i0 going to know when its done? Yeah.

AUDIENCE: Every time you mark a node, the node can send a message back to its parent, saying hi, I've been marked. Then you can probably get all the way back to the root. And then the root can count the number of-- actually, no if the root doesn't--

NANCY LYNCH: Root doesn't know the number of nodes. So that's a good idea.

AUDIENCE: If you don't have a child, you can tell your parent that you don't have a child.

NANCY LYNCH: That's a good start. Was there another? Yeah.

AUDIENCE: More generally, you just send a signal when you know your sub-tree is done.

NANCY LYNCH: When you know you're sub-tree is done, so that means you're going to be communicating something up the tree. Right, so that's the idea that you're working toward. So a termination algorithm to inform i0 when the tree is completely constructed. So let's say that the search messages get their responses. So everybody knows which nodes are their, which neighbors are its children, and which are not. So suppose a node has gotten responses to all of its search messages, knows who all its children are.

Now the leaves in this tree are going to know that they're leaves. How do they know that? Propagating all these search messages, and I'm a leaf. How do I know I'm a leaf?

AUDIENCE: You can't have children.

NANCY LYNCH: Yeah, you send all these search messages, and everybody says, sorry you're not my parent. So you know you have no children because of the kind of responses you get. So now we're going to use what we call a convergecast strategy. Broadcast is sending things out. Convergecast is fanning in information back to the top of the tree. So the convergecast would say, all right, so the leaves would send a message to their parents saying they're done.

Now if I'm some node in the middle of the tree, how do I know I'm done? Well it's what you said. You know that you can figure out when your entire sub-tree is done. Well first of all, you have to know your children are. It's kind of a two stage process. You have to know who your children are, by having received responses to all your search messages. And you wait to receive done messages from all of your actual children. So if I'm sitting in the middle of the tree, and I've got done messages from all my children, I know my whole sub-tree is done. Then I can send the done message to my parent. Got that? That's how convergecast works. And when it reaches the top, if i0 knows who its children are, and it receives done messages from all its children, it knows the whole tree is done. So it can output that the tree construction is complete. And it could tell the others by sending a message down the tree, so they all know as well. Questions?

AUDIENCE: Wouldn't i0 be the last one to know?

NANCY LYNCH: He'd be the last one. No, he'd be the first one to know that the whole tree is complete. Everybody else knows when their sub-tree is complete. So i0 still has to now send another message down the tree to tell everyone else the entire tree is complete. Is there another question? All right so this isn't showing that. This is just showing done messages, which are actually going in the opposite direction from these edges, going up the tree. But you can just see how they propagate up until the roots says done. No big deal.

Complexity for termination. Well it just takes at most diameter rounds and n messages for this done information to come up to the top, once the tree actually is finished. Because now you're just sending messages on the paths in this tree, which are only, at most, diameter in length. And this is just the process i0 can tell everybody else. It doesn't take very long either.

Applications, well suppose you construct a tree like this. And process i0 now wants to use it to communicate. It wants to send a whole batch of messages to all the other nodes. It can just send them now on the tree. It's an easy way to make sure messages reach everybody else in the network. Just send them on the edges of the breadth-first spanning tree. So now the messages, each individual message takes at most n message instances along the edges of the tree, because you only have to traverse the tree edges. No more dependence on the total number of edges in the network. And in fact, you can save time by pipelining a series of messages. So you can send them one round after the other.

The other way, suppose you want to compute something globally. Suppose everybody starts with some initial value. And process i0 is going to try to determine the value of some function of everybody's initial value, like the minimum or maximum or the sum or anything. Well you can do this while convergecasting on an already built BFS tree. So everybody can just send their information up the tree, and i0 can collect it all. In general, you can accumulate, you can do data aggregation as you go up the paths of the tree. So the message size doesn't blow up.

So if you want, for example, the sum of everybody's values, everybody just sends their values up in a convergecast. And each node computes the sum of all the values in its sub-tree. So this is pretty efficient. Make sense? I'm going to skip this. But you could do leader election in a general graph, If you don't have a leader, already, i0 by having everybody run a breadth-first search in parallel. But we'll skip that. Because I just wanted to have a couple of minutes to start the last topic, and we'll pick it up next time.

So it's the obvious extension. Instead of just breadth-first search trees, let's put weights on the

edges and try to compute shortest paths trees in terms of the total weight of the path. So we're going to add weights. It's an undirected graph. So it's just a weight for each undirected edge. I'll still have a starting node, vertex v0 with process i0. Still have unique identifiers. And I'll assume the processes know who their neighbors are. And they know the weights of the incident edges, their adjacent edges. But otherwise they don't need to know anything else about the graph.

So again, this is a familiar problem. But we're looking at it in a very different way, by distributing it. so the processes are supposed to compute a shortest paths tree, in the sense that everybody should output its parent in the tree. And let's say they output the distance as well, the weighted distance from the root node. So this is called Bellman-Ford's algorithm. Again it's got the same name in the distributed setting. The Bellman-Ford shortest paths algorithm.

So everybody is keeping track of their current best distance that they know, and their parent. And they know their unique identifier. And here's how the algorithm works. This will look familiar from when you had Bellman-Ford earlier. At every round, everybody is going to send its distance to its neighbors. Instead of just sending a search message, now it will send its actual distance information. And you receive the messages from your neighbors. And now you do a relaxation step, as you've seen before. You look at the current distance you have. And you see if you've gotten a new distance from a neighbor, such that if you add the new distance you receive to the weight of the edge between yourself and that neighbor, you get something better than what you had before. If you get that, then you're going to improve your distance. And if you improve your distance, then you're going to reset your parent to the sender of this new, better distance information.

So does this algorithm make sense? It's like what you saw before. But there's no running through all the nodes. Each node is doing its own thing. It's waiting to get better distance information and re-computing. And then it's going to be sending out its better information at the next round. Question? So this is kind of a jump in the way of thinking. All right, so now I'm just going to end basically with an animation that'll show you the kinds of things that happen here.

All right so you start out with the initial node. And what's recorded in the circle is the best distances. The rest of these, the best distance they know is infinity. So I didn't write that. So this guy knows 0 After one round, he sent two messages. The best distance each of these

guys knows is just the weight of the edge between v0 and itself. So this guy's now estimating it's distance at 16 and this guy at 1. 16 is not very good because it's actually very roundabout routes that can get there. But it's going to take us some time to make that adjustment.

After two rounds, everybody is sending their distance information. But now we get a correction here. This used to say 16. But now we have a two hop path that gives you a better distance. So you get the 1 plus the 14. So he's going to here, about the distance of 15 as a result of what 1 sends. And some new guys get their distance is calculated And then after three rounds, it gets a little bit complicated. So maybe I'm just going to flip through it quickly and let you study later. But you see that you keep getting improvements, as you perform relaxation steps. As information gets to somebody by better paths that happen to have more hops, they're going to be reducing their estimates. I'm going to flip, and you see that this guy's estimate is going down. And in the end, after eight rounds of this, you end up with a very roundabout path that actually gives this guy a much better estimate.

So you can see how that works. So the claim is that eventually, every process will have its distance being a correct minimum weight of the path, and its parent will be correct. I think maybe this is a good place to stop. We'll pick up with this algorithm and its analysis. Most of next time is going to be spent on asynchronous algorithms, which is a whole other level of complication. So I'll see you on Thursday.