**PROFESSOR:** We are still in chapter 12 on the sum-product algorithm. I hope to get through that today, fairly expeditiously, and to start with chapter 13. The handouts for today will be chapter 13. And, as usual, problem set 9 with problem 8.3 has been moved up into problem set 9. Problem set 8 solutions for 8.1 and 8.2.

So I'm going to start off on a different tack of presenting the sum-product algorithm. Where is everybody? 1, 2, 3, 4, 5, 6, 7, 8, 9. Is there any competitive event that I should know about? I know the traffic was bad and it's a rainy, late morning. Problem getting here myself. Well, this is improving.

The write up of the sum-product in the notes is in terms of equations. And I've never been terribly satisfied with this write up. It's concise, but we have to invent some new notation, which I don't consider to be completely transparent. I've never found it particularly easy to present in class, and it's been a little frustrating to me. So I'd like to try a different approach. I'd like in class just to go over through a very explicit example, our favorite 844 code. What we're trying to achieve with the sum-product algorithm and how the sum-product algorithm helps us to achieve it in an efficient way. So it's proof by example. It won't be very satisfying to those of you who like to see real proofs, but that you can get by going back and re-reading the notes. So I'm going to try doing it this way and you will let me know by your looks of gladness or of frustration whether you like it or not. So let's start at the beginning again.

What are we trying to do? We're trying to compute the a posteriori probability of every possible value of every symbol, internal and external, that appears in a graphical realization for a code. And we will consider these to be APP vectors. In other words, if a symbol is eight valued, there will be eight APP values in the vector. And we're going to assume that we have a cycle-free graph realization, so that we can proceed by the cut-set idea to solve independent parts of the graph. And at the very end, we'll relax that and see whether we can still do OK.

All right, one concept that seems very simple in theory but that some people don't get till they really try to do it in practice is that an APP vector is really a likelihood

weight vector normalized so that the sums of all the probabilities are equal to 1.

For instance, if we send plus 1 or minus 1 through an additive white Gaussian Noise channel with sigma squared equal to 1 or whatever, then the probability of-- we call this y and this r, the probability of r given y equals whatever. This is probability of r given y, is 1 over square root of 2 pi sigma squared e to the minus y minus r-squared over 2 sigma-squared. And so we can compute that for y equals 1 and y equals minus 1. And we'll get a vector of two things. The APP vector would then consist of this evaluated for y equals 1. Let me just take this part of it, a to the minus r minus 1 squared over 2 sigma squared and e to the minus r plus 1 squared over 2 sigma-squared.

Something proportional to these two vectors normalized and that's why I continue to use this proportional to symbol, which you may not have seen before. Or maybe you have. So the APP vector in this case would be proportional to-- well, actually 1 over square root of 2 pi sigma-squared times each of these two things. But this would basically tell you how much weight to give to plus 1 and minus 1 as possible transmitted symbols. And all this matters -- there's one extra degree of freedom in here. You can scale this, the whole vector, by any scale factor you like and it gives you the same information. Because you can always normalize the two terms to be equal to 1.

So if this, for instance, was 0.12 and this was 0.03, that's equivalent to 0.8 and 0.2. Two probabilities. Actual a-posteriori probabilities that sum to 1, all you've got to do is keep the ratio right. This is four times as large as that, so what that really means is 0.8 and 0.2.

And in implementation, so the sum-product algorithm, we never really worry about constant scaling factors. As long as they appear in every term we can just forget about scaling. I mean we don't want things to get too large or too small, so we can apply an overall scaling factor. But as long as we've got something, the relative weights, then we have what we need to know.

Let's go to the 844 code since we know that very well and ask ourselves what it is

we're trying to compute. We're given APP vectors for each of the eight received bits, let's say. We transmit y, we receive some r, perhaps over an additive white Gaussian noise channel. So we get a little 2 term APP vector for each of these y's, telling us the relative probability that a 0 was sent or a 1 was sent. That's called the intrinsic information. Let me just call that. So for each one we get p0 and-- let me make it p0 and q0, the two terms for y0; p1 and q1 for y1 and so forth. So we have that information for each of the received symbols.

What's the likelihood of each of these code words then given these individual bitwise likelihoods, the probability of the all zero code word. It's likelihood, let's say, is then proportional to p0, p1, p2, p3, p4, p5, p6, p7. Whereas the probability at 1, 1, 1, 1, 0, 0, 0, 0 is q0, q1, q2, q3, p4, p5, p6, p7 and so forth. Each one is a product of the eight corresponding terms. Yes?

**AUDIENCE:** You're assuming that the bits are independent.

**PROFESSOR:** Yes, it's a memoryless channel. So the noise in every symbol transmission is independent. And that's important too.

**AUDIENCE:** So basically once again, the second bit is independent of the first bit in transmission?

**PROFESSOR:** I'm sorry, I was writing and I missed what you said.

**AUDIENCE:** Well, I mean, if the first bit is 0, maybe that is a code where if the first bit is 0, the second bit has to be 0.

**PROFESSOR:** Yeah, there's certainly dependencies among the code words. That's what we're trying to take into account. This is how to take it into account. So I'm assuming a memoryless channel. But I'm of course, assuming dependencies within the code. How do I do that?

One way is maximum likelihood decoding. What do we do when we do maximum likelihood decoding in principle? We exhaustively computed each of these 16 products, each of these 16 likelihoods for each of the 16 code words. We'd simply

pick the one that's greatest and that's the code word that is most likely to have been sent. So that's what exhaustive maximum likelihood decoding consists of, finding the maximum likelihood word. And that clearly takes into account the dependencies. We're only considering valid code sequences when we do that.

In APP decoding we're doing something else, but it can be characterized quite simply. What's the a-posteriori probability that say, the first bit, is equal to a 0 or is equal to a 1? We can get the APP by summing up the likelihoods of all the code words that are associated with the value of the bit that we want to see. So we take the eight code words that have a 0 in this place and we'd sum up these likelihoods for those eight code words, and that would be the weight, the likelihood weight of y0 being 0. And the likelihood weight of y0 being 1 would be the sum of the other eight. And again you see that scale factors aren't going to matter here.

All we want is the relative weights of these things. So that's what we're trying to do in APP decoding, but we're trying to do it now for every single variable. At this point, I've only shown you the symbol variables, the external variables. So a brute force way of doing that would be to fill out this table, all 16 likelihoods, do these sums and then we'd have the APP vector for each of these.

Now we're looking for a more efficient way of doing this. The efficient way of doing this is going to be based on the fact we have a cycle-free graph realization. Which in this case, is a trellis realization. I'm going to use this two section trellis where each section is a four-tuple.

We've drawn it two different ways. This is a very explicit way showing the two parallel transitions that go from the initial node to the central state, and then two more down here. The set of code words is the set of all possible paths through-- corresponds to the set of all paths through this trellis.

So for instance, includes the all zero word, (0,0,1,1,1), (1,1,1,0,0,0), (1,1,1,1,1,1,1,1), and so forth. And these I happened to have listed as the first four here. Or we now have this more abstract way of writing this same thing. This says there are four external variables and two state variables or a vector space of dimension 2 here

and a vector space with dimension 4 here. That the dimension of this constraint is 3. That means there are eight possible values for these combinations of 4 and 2. They're shown explicitly up here. And they form a linear vector space. Either way, this is our trellis realization.

Now, we know that to do maximum likelihood decoding, it's useful to have this trellis realization, because, for instance, we can do Viterbi decoding. And that's a more efficient way of finding what the maximum likelihood sequence is. Basically, because we go through and at this point we can make a decision between 0, 0, 0, and 1, 1, 1, 1. We only have to do that based on this four-tuple and then we can live with that decision regardless of what else we see in the other half. And likewise in the other direction.

So now we'd like to use this same structure to simplify the APP decoding calculation, which looks like a more complicated calculation. So how do we do that? We introduce two state bits here. All of these have state bit 0, 0, 0, 0. Sorry, this point, we've called this state 0, 1. Just to keep it visually straight, 0, 1, 0, 1, 0, 1, and then there are eight more corresponding to the other two possible values of the state bits over here.

This is a quaternary variable. We want it to assume that it's a four-valued variable and to make the realization cycle-free, I don't want to consider it to be two independent bits. So there are four possible values for this state variable. So maybe I just ought to call this my state space -- which is a four-valued state space.

Now, what's the a-posteriori probability of the state being say, 0, 0? I compute that in the same way. I compute that simply by summing up the four likelihoods of the code for the external variables, the code words that are associated with state variable 0, 0 -- the first possible values of this quaternary variable. So it'd be the sum of these four things. Yes?

**AUDIENCE:** [UNINTELLIGIBLE PHRASE]

**PROFESSOR:** I've only listed half the code words. This is only 8 of the 16 code words. You want

me to write the other 8?

And by the way, there's at least an implicit assumption here that the code sequence determines the state sequence. As we found when we did minimal trellis realizations, the code sequence always does determine the state sequence. So if I know the entire code word that also tells me what the values of all the state variables are. There's a 1:1 correspondence between code words and trajectories in a minimal trellis. And I believe I'm sometimes implicitly using that assumption. But anyway, it holds whenever we have a cycle-free graph. In a cycle-free graph, we have a well-defined minimal realization. And the minimal realization does have this 1:1 property. So the state variables are determined by the symbol variables.

Suppose I want to compute the values of these state variables here. Let me write this out. This will be p0, p1, p2, p3, q1-- sorry --- q4, q5, q6, q7. And then there's one that's all q's. q0, q1, q2, q3, q4, q5, q6, q7. So basically, what I want to do is take the sum of these four things, and that's just a sum of products and I could just do it. But the Cartesian product lemma gives us a simpler way of doing it.

We notice that I can have either of these two four-tuples as the first half and either of these two four-tuples as a second half. In fact, I see over here, explicitly, these four code words are the Cartesian product of two four-tuple codes, the repetition code on 4. So I can use the Cartesian product lemma to write this as -- again, let me write it very explicitly. p0, p1, p2, p3, which you recognize as the probability of this four-tuple plus q0, q1, q2, q3. The probability of this four-tuple times p4, p5, p6, p7 plus q4, q5, q6, q7.

Now do you agree that the product of those two terms is equal to this, the sum of these four terms? Yes, there are four terms in this sum and they're what we want. And it's because we have this Cartesian product structure, which we always have in the state space. This is the basic Markov property of a state variable. So that's a simpler calculation. This requires four multiplications, each one is seven-fold multiplication.

This only requires one multip -- well, it requires one, two, three, four-fold

multiplications, plus one overall multiplication. And it's certainly a simpler way to compute that. And I can similarly do that for each of these down here.

What this leads me to is what I called last time the past future decomposition. We have one APP vector over here that corresponds to the-- let's call it the APP vector of the state given the past received symbols. We had something like that. In other words, given what was received for y0, y1, y2, y3, what's the probability, the partial a-posteriori probability for each of the four possible state variables given this past?

And similarly, we have another vector which I will consider to be a message coming in this direction, a message consisting of a vector of four values. Which is what's the probability the state vector given these four received simple, which is computed in the same way? That's this here.

And the past future rule is that just the overall probability for any of these internal state variables, the overall APP vector, is just obtained by a component-wise multiplication. This would be the APP of the state vector being equal to 0, 0 given the past. This would be APP of the state vector given 0, 0 given the future. We'd have the same thing for 0, 1, 1, 0, 1, 1. And to get the overall APP vector given all of r, the rule is just multiply them component-wise. We take the likelihood weight for 0, 0 given the past and multiply it by likelihood weight of 0, 0 given the future. And that gives us the total likelihood weight up to a scale factor.

The notes derive this in equation terms. You see it basically comes from this Cartesian product decomposition, which we always get for any internal state variable. The possible code words consistent with that state consist of a certain set of possible past code words consistent with that state. Cartesian product with a set of possible future code words consistent with that state. And as a result of that Cartesian product decomposition, we always get this APP vector decomposition. That's what the proof says in the notes. Do you like this way of arguing? A couple people are nodding at least. It's an experiment.

All right, so that tells us what we're going to want to do, at least, for the internal variables. And actually, the same is true up here. The variables going in are easy.

That's just p0, p1, p2, p3, or whatever. The ones coming out are-- this is called the intrinsic variable, the one that we get from direct observation, the intrinsic APP vector, excuse me. The extrinsic APP vector is the one that we get from all the rest of the inputs and symbols and the other parts of the graph. And we combine these component-wise to get the overall APP vector.

All right, but we now have to do this for every part of the graph. Let me go through another calculation, which will-- how do we, for instance, find the APP vector for y7, the extrinsic APP vector for y7, given that we have a trellis like this?

In general, for y7, we're going to be adding up all the zeros, these guys. And four more down here, and we'll make that the extrinsic APP vector of 0. So we want to add up this and this and this and this and so forth, eight of those. We want to do that in an efficient way.

This defines a code word with eight possible components. Let me move this up now. So let's talk about this 6, 3 code. What does it look like? With state vectors 0, 0, I can have y4, y5, y6, y7 be 0, 0, 0, 0. Or I can have it be 1, 1, 1, 1. The state vector is 0, 1. I can have 0, 1, 0, 1 or I could have 1, 0, 1, 0. With state vectors variable 1, 0, I can have 0, 0, 1, 1 or 1, 1, 0, 0. And with 1, 1, I can have 0, 1, 1, 0 or 1, 1, 1, 0, 0, 1. All right, so that's precisely the 6, 3 code I'm talking about.

Now suppose I have the APP vector for each of these. The APP vector for each of these, 0, 0 we've already calculated. It'd be p0, p1, p2, p3 plus q0, q1, q2, q3. That's the likelihood vector for 0, 0 coming from the past. So that's one part of this message here.

For either of these I get p0, q1, p2, q2, q3 plus q0, p1, q2, p3. Reflecting the two possible ways I can get to this value. I'm going to get four terms that I've computed for the past message. I'm going to have four likelihood weights corresponding to each of the four values of this state variable. So I've got that vector. I already computed it as part of the computation to do the APP for here. It's the past part of it.

So now, I want to compute the probability that y7 is a 0. The rule here is to take

every combination in this code and again, I'm going to get a-- well, let me write this part of it. This is going to be p4, p5, p6, p7. That's the probability of this four-tuple. This is q4, q5, q6, q7. This is p4, q5, p6, q7, q4, p5, q6, p7 and so forth. That's the weights I want to assign to each of these. So now I get a weight for each of these six-tuples.

I multiply this by this to get the weight, the likelihood weight, for this code word. This by this to get the likelihood weight for this code word and so forth. And I add these to buckets for-- I simply go through this and I'll add this one to the bucket for y7 equals 0. And I'll add this to the bucket for y7 equals 1. And this to the bucket for y7 equals 1. And this to the bucket for y7 equals 0 and so forth. It's clear that in this bucket I'm always going to get the same value for p-- I'm always going to get a p7 for 0 and a q7 for 1.

So just looking at y7, I don't have to compute the seventh value here. So if I did this separately as a little y7 variable, then the incoming message, the intrinsic information, would just be this p7, q7 likelihood weight vector. And what I really want to compute is the outgoing part, which has to do with y4 through y6. So let me draw it that way. This I don't think is very good exposition. Bear with me.

So now I'm going to put this times this in my bucket for 0. This times this in my bucket for 1, and so forth.

The question here is, does this method give me the same answer as my exhaustive method up here? And again, because of the Cartesian product character of the states, you can convince yourself that it does. In other words, I can use this as a summary of everything in the past that got to the state value equal to 0, 0, 0. Any time I got to 0, 0, 0, I got through one of these two things. And so for any possible continuation over here, I could have got to it in one of these two ways. And I'm going to find terms, e.g., these two terms. Let's see.

What do I want? I want two ways of getting to 0, 0, 0. I'm sorry, these two. I claim that this times these is equal to the sum of these two. And it is. I can get to 0, 0, 0, 0, 0 either through this or this. And I get to 1, 1, 1, 1, which is this one, either

through this or this. And again, it's a fundamental property of the state that whenever I get one of these, I'm going to get both of them in equal combination. So I can summarize things in this way, which leads to an efficiency of computation.

So what I'm basically explaining to you now is the sum-product update rule, which is how to propagate messages. Propagating messages, which are APP vectors. In general, my situation is going to be like this. I'm going to have some constraint code, (n,k), I'm going to have some incoming messages over here which tell me about everything that's happened in this past, p prime, and in this past, p prime. Give me a summary of all the weights corresponding to that. I'm going to have some output symbol over here and I want to compute now what the APP vector is for the possible values of this output symbol, which is going to be some aggregate of this down here.

What I do is I go through all 2 to the k code words, call this constraint code. Sorry, I'm using k again. That's 2 to the k code words. And compute the product of the inputs. Of input APP vectors according to the possible combinations of the input variables and the output variables that are allowed by this constraint. That's explicitly what I'm doing here. Here are all the possible combinations of state variables and symbol variables that are allowed by this 6, 3 constraint code. There are eight of them.

For each one, I'm going to compute the relevant product of APP vectors and I'm going to add it to a bin. So add to a bin. Bins which correspond to-- draw it like that-- output. I'm calling this the output variable values. OK, and that'll give me the summary APP vector of the outputs for now the past, which consists of everything that could have happened in this side of the graph.

Again, I'm using the cycle-free assumption in several ways. I'm saying that everything that happens up here is independent of everything that happens here. Everything that happens completely in this past is just the sum of what happens here. What happens here are subject to this constraint and has nothing to do with what happens over here in the future.

Eventually I'm going to do the same thing in the future. I'm going to compute a future component that's based on all this stuff. I'm going to get messages going in each direction and I'm going to sum them up.

OK, now I'm seeing a fair amount of puzzlement on people's faces. Do you get abstractly what I'm trying to do here? Does anyone want to ask a clarifying question?

So now--

**AUDIENCE:** [UNINTELLIGIBLE PHRASE] the bottom right, output variable?

**PROFESSOR:** Output variable values, I'm sorry. I think there's a little bit of confusion here because sometimes I've considered this output to be a 16-valued output. Here, in fact, well, it has eight actually valid values. And if I was considering it that way, then I would just have one bin for each of these likelihoods. But now if I consider these as four binary variables, then I would aggregate these things according to the values of each of the binary values. And I'd get four vectors of length 2 for the individual APP. Which is ultimately what I want here, so I'm sorry I've gone back and forth between subtly different realizations.

I do find this very hard to explain. Perhaps there's someone else who can do a better job. You have a homework problem which asks you to do this, and I think having done that you'll then get the idea. But it's a matter of going back and forth between the gross structure and the actual equations and just working it out.

All right, so the key things in the sum-product algorithm. I've already explained some level to them. We have this sum-product update. That basically explains given messages coming into a node, how do we compute the message going out of the node? We take contributions, products corresponding to what the code allows. We dump them into the appropriate bins in the output vector and that's the sum-product update rule.

We have the past future decomposition or combination rule. And that says at the end of the day, you're finally going to get messages going in both directions. And

you just combine them component-wise as we started out with.

And finally, we need to talk about the overall schedule of the algorithm. So let me draw an arbitrary cycle-free graph.

OK, suppose I have an arbitrary cycle-free graph. It consists of external variables out here, internal variables and constraint nodes.

How am I going to schedule these computations in order to do APP decoding of this graph? Well, what do I have at the beginning? At the beginning, I measure the received variables corresponding to each symbol, and that gives me the intrinsic information, which is incoming message you can consider from every symbol variable. Basically, what did I see on the channel? That's what that likelihood weight vector corresponds to.

All right. What do I need in order to do the sum-product update rule? I need the inputs. Think of this now as a directed graph. I need the incoming messages on all of the incident edges on this node except for one. If I have all but one, I can compute the last one as an outgoing message. So that's the structure of that update.

So at this point, can I compute anything here? No, I would need the inputs. But I can compute this outgoing message. I can compute this outgoing message. I can compute this outgoing message. I haven't drawn these of very high degree. Normally we have degree higher than two.

If I had another input here, however, I could still get that output.

All right, so that's time one. Think of there being a clock and at time one I can propagate messages into the graph to depth one if you like. There's going to be a systematic way that we can assign depths.

All right, now at time two what can I compute? I can compute this guy because I have this input. Now I have both inputs coming in here, I can compute this guy. Anything else? So maybe 0, 1, 2. I should put times on each of these, so I don't get

confused. 0, 0, 1, 0. So this is 2 at the output.

Now I'm at time three, what can I compute? At time three, I now have two inputs coming in here. I can compute this output. Actually, from these two inputs, I can compute this output at time three. And from these two, I can compute this outgoing message at time three.

Are there other things I can compute? Can I compute this? No, I don't have this yet. That's probably it. Does anyone else see anything else I can compute at time three based on the time zero, one, and two messages? No.

All right, but I'm making progress. And should be clear that I'm always going to be able to make some progress. At time four I can compute this message in that direction. Let's see, I now have everything I need to compute this message in that direction. And I'm almost done.

And finally, I can compute this message in this direction at time four, the extrinsic information for those variables. I guess I can do this at time four. I had that at time three.

And now at time five, I can get everything else that I need. Time five I get that out. I get this out based on this input and this input. And I get this out based on this input and this input.

So in a finite amount of time, in this case, five computational cycles, I can compute all of the messages in both directions for every variable in the graph.

Now as a final cleanup step, I simply component-wise multiply the vectors in the two directions, and that gives me the APPs for all the variables. This is a kind of parallel computation of all the APPs for all variables in the graph. And I claim two things. It's finite and it's exact. And the exactness is proved from the equations and the cycle-free assumptions. So we can always decompose things as Cartesian products. Independent components.

The finiteness, well, I assumed this is a finite graph. What is this number 5? It's

really the maximum length of time it takes me to get from any of these external symbol variables to any other external-- from leaf to leaf in graph theory terms. And that's called the diameter of the graph. And again, some people count the last little symbol node and some people don't count the last little symbol node. But it's clear that a finite graph is going to have finite diameter and that the propagation time is basically going to be equal to the diameter. That's how long it takes to get across the graph. So each of these messages in each direction has a certain depth. That's the distance in the graph to the most distant leaf node. And the maximum sum of these is always going to be 5 in this case. Notice that it is. A tiny little graph theory theorem.

**AUDIENCE:** [UNINTELLIGIBLE PHRASE] you see why if you sum these messages flying through those past and future of the single edge, you would get the overall APP because you're summing the contribution from both the set of outputs and those [UNINTELLIGIBLE PHRASE]. How would you prove that that should be overall the APP of the symbols?

**PROFESSOR:** Recall the state space theorem. Let's take any edge in here. Suppose we think about cutting it. A cut-set through that edge. Suppose we consider agglomerating everything back here and calling that the past. Agglomerating everything out here and calling that the future. This is past and future.

The idea of the state space theorem was that this is exactly the same situation as a two section trellis. Because everything is linear you get the state space theorem basically, which says that you get a Cartesian product of a certain past, a certain future. You get a minimal state space, which is kind of the total code. The modulo, the part of the code that lives purely on the past, and the part that lives purely on the future. That quotient space is really what it is, corresponds to the state space at this time. Because every edge in a cycle-free graph is by itself a cut-set, you can do this for every edge in any cycle-free graph. You can make this same argument. So you get a well-defined minimal state space. You get a Cartesian product decomposition, which has a number of terms equal to the size of the state space, due to the dimension of the state space over binary field. And so everything goes

through just as it did in the trellis case. That was the argument.

So mushing all this together, you get the sum-product algorithm. This is the fundamental reason why I took the time to go through minimal trellis realizations of block codes. Not because that's actually the way we decode block codes all by themselves, or even use block codes. We use convolutional codes. It's so that we could prove these very fundamental theorems about graph decoding. And we aren't even going to decode cycle-free graphs. But that's the intellectual line here. Thank you for that question, it was very well timed.

**AUDIENCE:** [UNINTELLIGIBLE PHRASE]

**PROFESSOR:** Let's take a look at the structure of this. Where do the computations occur? Every node there's a computation. What's the complexity of that computation? It's somehow approximately equal to the size of the constraint code. We do one thing for every possible combination of legitimate combination of variables, which is what we mean by the size of the constraint code.

So if this is a 6, 3 code, we need to do eight things here. Basically, eight multiplications. So this, in more general cycle-free graphs, that's what corresponds to branch complexity and trellis graphs. We found that the branch space corresponded to a constraint code in trellises. So the computation is basically determined by the constraint code complexity and really, overall what we'd like to do is minimize the constraint code, the maximum size of any constraint code because that's going to dominate computation. So we try to keep the k in these things as small as possible. Yes?

**AUDIENCE:** [UNINTELLIGIBLE PHRASE] and you get larger dimension constraint for [UNINTELLIGIBLE PHRASE].

**PROFESSOR:** Yeah, we're actually not too concerned about diameter. Because things go up exponentially with constraint code dimension, just that single parameter tends to be the thing we want to focus on. But we remember that's hard to minimize in a trellis. We showed that basically our only trick there after ordering the coordinates was

sectionalization and that we could not reduce the constraint code complexity, the branch complexity, by sectionalization.

And then again, we build on that and we went through this argument that said, gee, this corresponds to some trellis. We could draw a trellis where this, literally was the past and this was the future. And the minimal size state space in this cycle-free graph would be the same as in that trellis. What I guess I didn't show is you can't reduce the constraint code complexity by significantly -- actually, I'm not sure there's a crisp theorem. But the fact that you now are constrained to size of state space, this is certainly going to be at least as large as state space sizes. Tends to indicate that you're not going to be able to do a lot about constraint code complexity by going to more elaborate cycle-free graphs than just the chain trellis graph either. That's, I think, a missing theorem that would be nice to have.

I had a paper about a year or two ago, I'm not sure it ever quite gets to that theorem. But that's the theorem you'd like to see. You really can't improve this.

So ultimately, we're bound by what you can do in a trellis. And in a trellis you can't do too much about this branch complexity parameter once you've ordered the coordinates. A bit of hand-waving here, but we're going to need to go away from cycle-free graphs.

One reason I like normal graphs is that you get this very nice separation of function. You get the idea that nodes correspond to little computers. You could actually realize this with little computers. Put a computer in here for each node. Its job is to do the sum-product update equation.

And then, well, what are the edges? They're for communication. These are the wires that run around on your chip. And so when we talk about state space complexity, we're really talking about something like bandwidth. How wide do you have to make these wires? Do you have 6 bits, or 9 bits, or whatever? So state space really has to do with communication complexity. Constraint or branch has to do with the computational complexity, which sometimes one's more important than the other. Computational tends to be the thing we focus on. But we know more and

more as we get to more complicated chips, communications complexity tends to become dominant.

And this, of course, is the I/O. These are your paths which go to the outside world, your I/O paths. So it's really a good way to think about this. And in fact, some people like Andy [UNINTELLIGIBLE] and his students have experimented with building analog sum-product decoders.

The general idea here is that you can compute an output as soon as you have the two corresponding inputs.

Well, imagine another schedule where this thing just computes all the time based on whatever inputs it has. In a cycle-free graph, say you're in the interior. Initially, what you have on your inputs is garbage. But eventually, you're going to get the right things on all your inputs. And therefore, you're going to put the right things on all your outputs. So again, up to some propagation times, just putting in non-clocked little computers here, whose function is to generate the right outputs given the inputs in each direction, is eventually going to be right in a cycle-free graph. And in fact, there are very nice little analog implementations of this where it turns out the sum-product update rule is extremely amenable to transistor implementation nonlinear rule.

So you just put these in, little computing nodes, and you put whatever you received on here as the received symbols, and you let it go. And it computes incredibly quickly for a cycle-free graph.

Where we're going to go is we're going to say, well, gee, we got this nice little local rule that we can implement with local computers for the sum-product algorithm. Maybe it'll work if we do it on a graph with cycles. And on a graph with cycles this kind of parallel or flooding schedule, where every little computer is computing something at every instant of time, is probably the most popular schedule. And again, you build an analog implementation of that and it goes extremely fast and it converges to something. It just relaxes to something, there's a local equilibrium in all these constraint nodes. And therefore, some global APP vector that satisfies the

whole thing. I think this is a nice attribute of this kind of graphical realization.

All right, a particular example of how this schedule works. The cycle-free realization that we care most about is the trellis. The BCJR algorithm is simply the implementation of the sum-product algorithm on a trellis. SP on a trellis. So generically, what does a trellis look like? I guess I should allow for some nontrivial code here. So all trellises have this same boring graph.

What is this schedule? It's a cycle-free graph, so we could operate according to this nice controlled schedule, compute everything just once.

And how does that operate? We get intrinsic information that's measured at each of these symbols here. What did we receive on the channel?

We put that through a constraint code and we get here, let's give this a name. This is called intrinsic 0. Here is alpha 1, which is just a function of the intrinsic information at time zero. And we combine that with the intrinsic information at time one and we get alpha 2, and so forth. So we get a forward-going path down the trellis.

If any of you knows what a Kalman filter or a smoother does, this is really exactly the same thing. This is finding the conditional APP probabilities of this state vector given everything in the past. In Kalman filtering or smoothing, it's done for Gaussian vectors. Here we're talking about discrete vectors. But in principle, it's doing exactly the same thing. It's computing conditional probabilities given everything observed in the past. And so the diameter here is just the length of the trellis basically. So this is i n minus 1, i n. And here we have alpha n.

When we have alpha n finally in here, we can compute the extrinsic information based on alpha n as just extrinsic n. We combine these two and we're done for that guy.

Meanwhile, we have to compute the backward-going information on each of these branches. So we compute first b n based on i n. Then we get b n minus 1 based on i n minus 1 and b n, and so forth. We get a backward-going propagation of

conditional information. Each of these betas represents the a-posteriori probability vector given the future from it.

And finally, when we get down to here we can compute extrinsic 0 from data 1. And by this time, when we have both alpha 1 and beta 2 coming into here, we can, of course, get extrinsic 1.

When we get both the alpha and beta coming in here, we can get extrinsic 2, and so forth. Of course, you can write this down as equations.

If you want to read the equations, you look in the original Bahl, Cocke, Jelinek, and Raviv article in 1973. This was again, a kind of lost paper because just for decoding a trellis, the Viterbi algorithm is much less complex. And gives you approximately the same thing. I mean, I talked last time how probably what you really want to do in decoding a block code is maximum likelihood. That's minimum probability of error in a block-wise basis. What APP decoding gives you minimum probability of error. If you then make a decision based on each of these extrinsic information, or on the combination of these two to get the overall APP vector, that's called a map algorithm, maximum a-posteriori probability.

If you make a decision based on that, that will give you the minimal bit error probability, but it may not necessarily give you a code word, and it will not minimize the probability of making any error, which is generally what you want to minimize. So for trellis decoding of block codes this was not favored.

However, a block code as a component of a larger code, you want the APP vector to feed off to something else. And so with the advent of much larger codes, of which block codes are components or convolutional codes, BCJR is the way you want to go. And some people say it's about three times as complicated as the Viterbi algorithm. All right, so you'll have an exercise with doing that in the homework.

There is a closely related algorithm called the min-sum algorithm, or the max-product algorithm. And the idea of that is that you can really carry through the same logical computations except when you get to one of these combining operations,

rather than doing a sum of products, you can do a max of products. And that the Cartesian product law still holds, the same kind of decompositions that we have still hold.

For instance, in this example, to get the state 0, 0, 0 before what we said we're going to get an APP vector where this is the past APP for the 0, 0 value of the middle state vector. We simply combine the likelihood weights of the two possible ways of getting to that state and we sum them.

Instead of doing sum, let's just take max. The maximum probability of getting-- in other words, what's the best way of getting here? Same question as we ask in the Viterbi algorithm. We'll let that be the likelihood weight for the state vector. Otherwise, the logic is exactly the same.

Similarly, coming this way, what's the max of these two? OK, what's the best way of getting to the state vector in terms of likelihood from the future? And what do you know, if we combine these things at this point, we now have a past vector and a future vector. Let's take the max of this times the max of that. That gives us the maximum likelihood way of going through the 0, 0 value of the state.

If we do exactly the same logic for decoding this trellis, we get a two-way algorithm. It's like this. It's like the BCJR algorithm, where at every point we're computing the maximum likelihood. It's not hard to show that what this gives you is that each of these symbols out here, it gives you the symbol that belongs to the maximum likelihood code word.

How does it differ from the Viterbi algorithm? It gives you the same answer. It gives you the symbols, one by one, of the maximum likelihood code word. It doesn't have to remember anything. It doesn't store survivors. That's its advantage. It's disadvantage is it's a two-way algorithm. You have to start from both ends and propagate your information in the same way. And for block codes, maybe that's OK. For convolutional codes that's certainly not something you want to do because the code word may go on indefinitely. So the Viterbi algorithm gets rid of the backward step by always remembering the survivor. It remembers the history of how it got to

where it is. So that once you get the max, you don't have to-- you simply read it out rather than having-- this backward part amounts to a trace back operation. That's probably much too quick to be absorbed.

But I just wanted to mention there is a max product version, which if you take log likelihoods becomes a max-sum. Or if you take negative log likelihoods it becomes min-sum. So this is called the min-sum algorithm. And it does maximum likelihood decoding rather than APP decoding. And sometimes it's used as an approximation or very intermediate approximations between these two.

**AUDIENCE:** [UNINTELLIGIBLE PHRASE]

**PROFESSOR:** No, if you use the min-sum, basically everybody will converge on the same maximum likelihood code word. So all these constraints will be consistent with-- you'd be taking a single max at each point. But at each point you will solve for the maximum likelihood code word. And what you'll see up here is the symbol that belongs to the maximum likelihood code word. This is maybe a bit of a miracle when you first see it, but you'll independently get each of the bits in the maximum likelihood code word. Check it out at home.

**AUDIENCE:** Why not always do this?

**PROFESSOR:** Why not always do that? It turns out that we want softer decisions when this is part of a big graph. We actually want the APPs, the relative probabilities of each of the values. The max, in effect, is making a hard decision.

**AUDIENCE:** [UNINTELLIGIBLE PHRASE]

**PROFESSOR:** Yeah, all right. It does give reliability information. I'm not sure I can give you a conclusive answer to that question except APP works better. That's a more empirical answer. Yes?

**AUDIENCE:** [UNINTELLIGIBLE PHRASE] Where does that flexibility of the input symbols being [UNINTELLIGIBLE PHRASE]. We should be able to take that into account [UNINTELLIGIBLE PHRASE].

**PROFESSOR:** Yeah. OK, well, if the input symbols are independently not equiprobable, you can feed that in as-- you know, once they came out of the channel, once we see the channel information, the two symbols are not equiprobable anymore. So if somehow they a priori independently were not equally probable, you could just feed that in as part of this intrinsic information vector. That's almost never the case in coding.

The intrinsic information is sometimes regarded as a-priori and this has a-posteriori, or the combination of the two of them is then a-posteriori when this goes off somewhere else. But it doesn't really mean that the symbol itself was-- it means that the evidence is biased one way or the other. And when this appears somewhere else in the graph, this will now appear somewhere down in some other graph. It's called the a-priori information, but what it is is the a-posteriori information given all of the received symbols in this part of the graph.

**AUDIENCE:** [UNINTELLIGIBLE PHRASE] Does the i0 of that other graph contain this table sharing information added to it?

**PROFESSOR:** Yeah, the i0 for another graph is just the e0 of this graph. I mean, if you consider it all as one big graph, that's what you want. You want the message that goes in this direction, which basically summarizes all of the inputs from this graph. Generally, you'd have a little equals node here. You'd have the actual intrinsic information coming from the outside world there. You would compute the e0 from this graph. And at this point, you would combine e0 and i0, and that would go around here.

So the very last page of chapter 12 just says, OK, we've got a nice, clean, finite exact algorithm that will compute all these APPs on a cycle-free graph.

Now, suppose we're faced with a graph that has cycles. And here's the situation. Suppose the graph has cycles in it. Then first thing that might occur to you, at least after this development is, well, we now have a local rule for updating at each of these computational nodes, each of these constraint nodes, in this representation. Why don't we just let it rip? See what happens.

We're going to put in some intrinsic information at the outputs as before. We can

compute some output message here based on-- well, eventually we're going to get some input message from here. And we can compute some output message here based on all of these and we'll pass that over. Let's assume what's called a parallel schedule or a flooding schedule, where in each cycle every one of these little guys does its thing. It takes all the inputs that it has available at that time and it generates outputs.

Well, hope for the best. It will compute something. You can sort of intuitively see that it'll settle. It'll converge, perhaps, into some kind of equilibrium. Or hopefully it will converge into some kind of equilibrium. There is a basic fallacy here, which is that the information that comes in here is used to compute part of this message. Which is used to compute part of this message. Which ultimately comes back and is used to compute part of this message. And then part of this message again, so that information goes around in cycles and it tends to be used again and again and again. Of course, this tends to reinforce previously computed messages. It tends to make you overconfident. You heard some rumor and then the rumor goes all around the class and it comes back and you hear it again. And that tends to confirm the rumor. So it's that kind of telephone situation, and it's not really independent information.

Intuitively, makes sense that if all of the cycles are very large-- in other words, the rumor has to go to China and back before you get it. That it probably is highly attenuated by the time you get. It's mixed up with all kinds of other information, so maybe it doesn't hurt you too much if the cycles are very large. And in fact, that's the way these capacity approaching codes are designed. Whereas, as I mentioned last time, if you're in a physical situation where basically your graph looks something like this, you don't have the freedom to make your cycles large, then this is probably a bad idea.

And this isn't why in fields, like vision, for instance, people try to use this-- the sum-product algorithm is the belief propagation algorithm, if you've ever heard of that. Widely used in inference on Bayesian networks. And the religion in belief propagation used to be that you simply can't use belief propagation on graphs that

have cycles.

Why? Because most of the graphs they dealt with look like this. And in fact, it works terribly on that. It was a great shock on that field when the coding people came along and said, well, we use belief propagation on graphs with cycles and it works great. And this, as I understand it, really had an enormous impact on the belief propagation community. But it was realized that gee, coding people have it nice. You make your own graphs. And you make them so this is going to be a very low order, low impact event. And that's right. But coding we do have this freedom to design our own graphs.

We realize from the basic arguments that I put up before that we're really going to need to have cycles. As long as we have cycle-free graphs, we're going to get this kind of exponentially increasing complexity, which is characteristic of trellis graphs. So we're going to need cycles, but as long as the cycles are very long and attenuated, as long as the girth of the graph is great, maybe we can get away with it. And that, in fact, is the way the story has developed. And Gallager basically saw this back in 1961, but it took a long time for it to catch.

So I didn't even get into chapter 13. Chapter 13 we're going to first just introduce several classes, the most popular classes of capacity approaching codes. And then I'm going to give you some real performance analysis, how you would design, simulate, in these codes. And it's what people actually use.

OK, see you next time.