PROFESSOR: We're into chapter 11. I've asked Ashish to hand out today a revision of the previous version of chapter 11, plus chapter 12 on the sum-product algorithm, in case we get to it. I'm giving you a complete copy of the problem seven solutions for both weeks, and to get back on track, we're handing out problem set eight. There probably will only be one more problem set that you hand in.

All right. We've been getting to the centerpiece of the second half of the course which is codes on graphs, which is the way we're going to get to capacity-achieving codes by building them on graphs of linear complexity and decoding them on the graphs with iterative decoding. I frame this in the language of behavioral realizations. I say a graph is a realization of a code via a behavior, which is the set of all the possible trajectories on the graph. So this is, again, putting it in system theory language.

I want to distinguish between two types of variables. The external ones, which communicate with the outside world-- in our case, these will be the symbols in a code word. These are the ones that are actually involved in the code. So these are more or less pre-specified for us when we're trying to realize a code. But we can adjoin auxiliary variables, internal variables, we often call them state variables, hidden variables, latent variables, which are really there for our convenience. They're to make the realization simpler, or more elegant in some way, or to have properties, or whatever. But these are free for us to have because they're inside the box and the world never sees them.

Now, we describe the behavior by a set of constraints on the variables, which in our case, these are local constraints that only involves a few of the variables, each one. The idea is to express a global behavior via local constraints on variables. And in our case, the constraints will be expressed by equations, as we've seen, or in a little bit more generality by little codes, which will go in to make up this big code. So in some sense, you can think of codes on graphs as being making big codes out of little codes, making global codes out of local codes.

The behavior, then, is simply defined as the set of all combinations of variables, both external and internal, that satisfy all the constraints. If we can find a set of values for all the variables such that all these local constraints are satisfied, then that's called a valid trajectory, or whatever. And that's the entire behavior. And the code is simply the set of all external variable n-tuples that occur as parts of valid behaviors-- in other words, the projection of the behavior just onto the external variables. All right, so it's an elegant and rather implicit way of describing the code, but we'll see it's very, very powerful.

And last time, we talked about various styles of realizations. I gave you examples of generator representations and parity-check representations. This is really where it all started, with parity-check representations, where this style of graphical representation is very natural. You've got the symbols in the code, those are the only symbols you need, and you've got parity checks, and the parity checks constrain the symbols. If you've got n minus k parity checks, you get n minus k constraints. And it's a kernel representation in the language of linear algebra. It's simply the set of all code n-tuples that satisfy all the parity checks. That's it, we don't need any hidden variables.

In a generator representation, this is a more natural representation, when we're trying to generate or simulate a code. We want to run through a set of inputs that in effect create all the outputs, all the code words. In this case, we don't actually see the inputs as part of the code words, necessarily. We might, we might not. But we distinguish them in this representation and we regard the inputs as hidden variables, or state variables, if you like. They're part of the realization, but they don't actually occur in the code words, so they fit into the hidden category.

OK, and we talked about two styles of graphs. Tanner graphs, which are older and more established. Tanner really wrote the foundation paper in this subject in 1981. I regard him as the founder of the subject of codes on graphs. However, like a lot of good papers, his was completely ignored-- that's almost true-- for 15 years. And then it was rediscovered around 1995, when people began to wake up to this subject again.

And Tanner graphs really come out of this idea. They were closely associated with that. We can generalize them to be more like this. But it's this simple idea. We have a bipartite graph, we make one class of vertices into the variables, one class into the constraints, and we simply tie a variable to a constraint when it's involved in the constraint. And we can generalize it by making internal and external variables, which we have to distinguish in the picture in some way.

And the constraints, initially, they were just zero-sum constraints and parity-check codes. One of the things Tanner did was he said, well, they could be any codes. A single parity check is like a single parity-check code. And we can make any codes the constraints.

And the edges don't work very hard in the Tanner graph. They implicitly carry the variables to the constraints so we could label them with the variables. And we showed how a Tanner graph could always be transformed to this normal graph, which I think is certainly in more complicated situations, a cleaner way of explaining things, and cleaner in a variety of senses.

Here is our taxonomy. Vertices indicate constraints, half edges indicate external variables. A half edge is something you can tie to another half edge to form an I/O pad. Edges tie together two constraints. They're internal, and so these represent the internal variables. And this is cleaner both analytically-- there's this nice duality theorem about normal graphs that I won't be presenting to you, and a couple other things. They're cleaner when you come to implement the sum-product algorithm. I think they're just generally cleaner, but I may be biased. I want you to see both because these still predominate the literature, although more and more people seem to be using this style of graph, and that's what I will use in the course. All right? So that's a review of where we are.

OK. Today we're going to go on to other important styles of realizations, and particularly the next one we'll talk about is trellis realizations. So this will tie back to all the work we did in chapter 10. And what did we discover in chapter 10? The key to a trellis realization was a trellis-oriented generator matrix. And again, I will put up our favorite matrix on the board, the generator of the 8, 4, 4 Reed-Muller code. And in the form we've come to know and love, it looks like this.

And what makes it trellis-oriented is that all the starting times are different and all the ending times are different. Therefore these generators are as short as possible. The main feature of this generator that we want to look at is the spans, the active parts of the generators, which we've made as short as possible.

And let me explicitly write out what they are. The first generator is active from symbol time 0 to symbol time 3. This one is active from 1 to 6. This one is active from 2 to 5. And this one is active from 4 to 7.

OK, so those are the active spans. And we know from that, we can find the dimensions of minimal state spaces and of minimal branch spaces. OK, we just count the number of active generators at either of the state times, which are between the symbols, or the symbol times themselves for the branches. OK, again, quick review, especially because we have a few guests here who we haven't seen previously in the course. So that's where we've been.

All right. Once you have any generator matrix, or in particular a trellis-oriented generator matrix, the code is simply described as the set of all linear combinations of the matrix elements. And in particular, the code symbol at every time is a linear combination of the matrix elements in a particular column.

All right, so let's use that to get a trellis realization. I've done this in various ways. I'm still not satisfied, though the way that I do it in the notes is quite notation heavy and mathematical. I hope this picture of what we're going to do is going to give you a totally transparent idea of what's going on.

OK, we're going to have a trellis realization. Let's write down the symbol times. And I'll write them all the way across the board. 0, 1, 2, 3, 4, 5, 6, 7. And let me save the coefficients, ui, here. And I'll write them as equality constraints, but you can also think of them as a little binary shift register.

I'm going to need u1 at four different times. I'm going to need u1 during its span at time 0, 1, 2, and 3. So I'll set up a set of constraints here to propagate u1 between these three times. And u1 is going to be an internal variable. Strictly, there are three replicas of this internal variable here. And I can pull it out at each time, too. So this is where I'm going to keep u1, in effect.

Similarly, u2 I'm going to keep around for all the times that I need it, which is these six times. OK, so is that six? That's only five. These are all the times that u2 is actually involved in the output symbols, so I'm going to have it when I need it.

u3 I need for four times. I've been inconsistent in whether I write the variables or the values of the variables. In this case, I'm writing values. And finally, u4. I see I'm going to need more than one board here. So now I've got everything I need at the time that I need it.

Now, this is going to be tricky. I'm simply going to have a linear combination-- call it g0, g1, and so forth-- which is going to create my output at time 0, or time 1, or time 2, and so forth.

And you'll agree from this generator matrix that the output at time 1 is some linear combination of u1. And the output at time 2 is some linear combination of these two guys. OK, and the output at time 3 is some linear combination of these three guys, and so forth. You see what I'm doing?

So I could describe these as simply a linear constraint. What are the possible combinations of inputs and outputs in this constraint? And similarly-- and don't make me draw the lines in. OK, so I claim this is a realization directly from this trellis-oriented generator matrix. Yes?

AUDIENCE: In this example in the generator, g has a 0 [UNINTELLIGIBLE].

PROFESSOR: Right.

AUDIENCE: So before, the component had to be 0 in the linear combination? Or are you--

PROFESSOR: I can do it that way. Clearly, I don't need this input at this time. So I could erase that.

Would that make you feel better? Actually, that gives an even better picture. It's not going to make any fundamental difference, but I'm doing this for a generic set of generators that cover these spans. All right? For the particular ones we have here, yes, I can make that further change and eliminate those two inputs. But I'm shortly going to aggregate this and put it in a more aggregated form. And what's happening internally here then won't matter to us very much.

OK. And notice that I can regard this as the branch space at this time. And it has the right dimension for the branch space at each of these times. And let me take it a little more slowly. First, I claim it's a realization. If I built that thing with the appropriate linear combinations here, then the behavior of this is basically the set of all u's and y's that satisfy all these constraints. And if I just look at the outputs, that's the set of all code words.

So I've constructed a behavioral realization in a certain way. OK, and now I'm going to start to aggregate. Aggregate just means drawing lines around sub-graphs. Or sometimes in the notes, I call this agglomeration. So I'm going to regard all this as one constraint on the variables that come out of this. The constraints that are affected by this agglomeration-- the variables that are affected are y0 and u1.

OK, so this I can draw-- let me draw it this way-- as a little constraint code which operates at time 0. And it's a linear 2, 1 code that basically ties together y0 and u1, whatever the relationship between them is. It probably is the y0 equals u1. The first generator starts at this time. That's about the only thing it could be.

Similarly, I'm going to aggregate all this part of the realization here and consider it a big block, or a big constraint. At time 1, there's some constraint that affects y1 and u1 and u2. So this is a little 4, 2 code on the four incident variables. One of the constraints is that u1 on this side equals u1 on this side. These are really two replicas of the same thing. So there's an equality constraint propagating through there.

Then I have some function of y1 as a function of u1, which I'm going to regard as the state at this time. And u1 and u2, let me call that state space at time 1. Let me call this at time 0, let me call this the state space at time 1. And I did go through a development where I showed that these u's could be regarded as the components of the state spaces. This is a more constructive way of seeing the same thing.

And notice they have the right dimensions. So the states are what carry the information from time 0 to time 1, just as we would want. They kind of embody the Markov property. Yes?

AUDIENCE: Why is c0 length 2?

PROFESSOR: Why is c0--

AUDIENCE: Length 2.

PROFESSOR: Length 2. Because it affects two bits.

AUDIENCE: Thank you.

PROFESSOR: It affects these two bits. Let's see, how did I know that there's only one possible constraint here? Because that's the dimension of the branch space. It's really all determined by u1. Over here, however, it's determined by u1 and u2. So the dimension is 2. u1 and u2 are free, And the others are fixed once we know those. So I was a little ahead of myself there.

And similarly, let's keep drawing in this fashion, here's a constraint code at time 2 which relates these possible variables, u1, u2, u3, which I'm going to call the state space at time 2. So my times are not what I expect. But it's always really a problem to keep the indexes consistent. We want state time 1 to occur after the first symbol. So this should be state time 2, this should be state time 3 to be consistent with what's in the notes. I'm sorry.

And so forth. What is the length of this code? It's simply the number of bits that it controls, which is 6. 1, 2, 3, 4, 5, 6. What's the dimension? It's 3, because u1, u2, u3 are free if I only look at this constraint independent of everything else.

OK, and continuing this way, at this point, you notice I only have u2 and u3. So my state space at time 4 has gone down in dimension. And here I have c3, which is, again, a 6, 3 code, and so forth, as I go ahead. Let me just draw it. This is also a 6, 3. This is time 5. I have the state space at time 6. This is back down to 1, 4. It's symmetrical on this side, as you know.

So that's my trellis realization. And I claim this, too, is a realization, in the sense that any combination of u's and y's that satisfy all of these constraints is a legitimate trajectory. And the set of all y's that are part of those legitimate trajectories form the code. In fact, it's pretty explicit here that the dimension of the code is 4, corresponding to u1, u2, u3. So I have 8 outputs, I have 16 possible behaviors. I pick off the combinations of u's and y's that could make that, which is basically determined by this matrix. And I just pick off the y's and that's my 8, 4 code.

OK, so that's what trellis looks like. The constraint code really embodies the branches. This encapsulates everything that can happen at time 0. This encapsulates everything that could happen at time 1. The u's are really my state spaces. This constrains state, output, next state, in nice linear system theories style. This tells me what combinations of state, output, next state, I can have at time 2, and so forth across the board.

So the constraints are local constraints-- local in time, in this sense-- that constrain what can happen as you get to the next state. Once you get to the next state, this state has the Markov property. This is all that the memory you need of the past in order to determine the future, or the whole set of future possibilities. Each of these is a linear vector space over the ground field. In fact, it's just the space of 1-tuples, 2-tuples, 3-tuples, 2-tuples, and so forth. And it has certain dimension, in this case, equal to its size.

And if you calculate the dimensions of-- call this the state space, call this the branch constraint code if you like, or the branch space, it's isomorphic, the dimensions are minimal here by construction from the trellis-oriented generator matrix. If we want to know the minimal size of the state space at time 4 here in the center, we just calculate the number of generators that are active at time 4, and that's precisely what we're going to get over there, too, by our construction.

If we want to compute the dimension of the branch space at time 3, it's the number of active generators at symbol time 3. There are 3 active ones, and we've just forced this to have that structure. So this is a normal graph of a minimal trellis realization of the 8, 4 code. OK?

So we can clearly do that for any code, right? For any code, we can find a trellis-oriented generator matrix. We can go through these steps, and we'll always come up with something that looks like this down here.

OK, now, what are the properties of this graph? Does it have cycles? Does this graph have cycles? Anybody? "No," is one answer I get. What's the other possible answer? How many people say no, and how many people say yes? I want to see a show of hands. How many people think this graph is cycle free? OK, and how many people think that it has cycles?

OK, the majority is wrong. This graph clearly has cycles. Here are two edges here, and here's a cycle. OK, so as a graph, it has cycles because it has multiple edges going between these constraints. All right, so a good thing to do is to make this a cycle-free graph, and that's just a matter of regarding this as instead of two binary variables, we regard it as one quaternary variable.

All right, so we're going to regard this as a single binary variable of dimension one. This is a quaternary state space. There are 4 possible states here. So the dimension of the state space at this time is 2. So when I draw abbreviations like this, you can think of them as being decomposable into binary variables, if you like.

But the advantage of drawing them as larger variables, higher-valued variables, is that now does this graph have a cycle? No, this graph is cycle free. I've just gotten rid of all the possible cycles. How can you tell if a graph is cycle free? Various ways. I think the most elegant one is to say if every edge is by itself a cut-set, then the graph is cycle free. If I remove any edge, then I decompose the graph into two parts. And that's clearly true of this graph. We don't really have to test the half edges.

In fact, this is kind of the defining concept of a state space. A state space is kind of a cut between the past and the future. When we asked about the minimal state space, we asked what's the minimal dimension of the information, the state, that we need to pass from the past to the future? And so states correspond to cuts. If we make each state space, if we consider it to be a single variable, then we get just a chain graph on a sequential time axis, as is conventional in system theory, as your conventional integer-time axis for discrete time systems. In this case, it only has a finite number of times where anything happens. But you can think of this as part of the infinite set of integers. And so this is everything that happens.

And a trellis is always going to be very boring. It's always going to look like this. But the advantage of now we have this cycle-free graph realization, we'll find out that we can do exact decoding of cycle-free realizations using the sum-product algorithm, or the min-sum algorithm, which in this case kind of reduces to the Viterbi algorithm with an asterisk on it.

And the cost of doing that is that now have to instead of considering two binary variables here, I have to consider a single variable that has four possible states. So basically, I'm going to be carrying messages which are vectors indexed by the state space across here. And in this case, the vector is going to have to have four elements, which are going to be likelihoods or metrics or weights of some kind. It's basically telling me what's the weight of each of the four possible survivors, if you can think of the trellis that goes with this. Or here, I'm going to need to carry a vector with 8 elements. So it takes more to specify a vector with 8 elements than 3 vectors each with two elements.

So by aggregating these, I've created a sort of exponential complexity situation as these state spaces get very big. But on the other hand, I get cycle freedom. All right? Yeah.

AUDIENCE: There's no real difference between this and what you had before, right?

PROFESSOR: As President Clinton might have said, it depends what the definition of "real" is. What do you mean by real difference? I mean, sure. We think of something u1, u2, you can think of that as two binary variables or a single quaternary variable. Is there any real difference between that? Well, when we actually go to the implement decoding algorithms, we'll find there is a real difference in which attitude you take. But mathematically, it's a distinction with hardly any difference.

AUDIENCE: So then there's no fundamental difference between the cycle freeness and the cycle--

PROFESSOR: No, you'll see this real difference-- there is another real, significant difference. If I drew this as two binary variables, I have a graph with cycles. And we'll find that I can't do exact decoding with a sum-product algorithm on a graph with cycles. So if I tried to apply that algorithm to this, I'd sort of iterate around this little cycle.

Whereas if I agglomerate them into a single variable, I get rid of that behavior. And I can summarize everything in four values. So in that sense, it's a huge difference.

AUDIENCE: Can I always transform this whole graph into cycle-free graph using this kind of technique?

PROFESSOR: I didn't get the first part of your question.

AUDIENCE: Can I always transform the graph with cycles to a cycle-free graph using this-- combining those--

PROFESSOR: Yeah, OK. So this agglomeration technique of drawing lines around sub-graphs and then considering everything inside there to be a constraint, and all the variables coming out-- well, it depends on their topology, but I can group the variables coming out however I want-- yes, I can always do that.

AUDIENCE: So we can always do that for things [UNINTELLIGIBLE] transform the graph into cycle free.

PROFESSOR: Right. But we may find that to make it cycle free, I then have to aggregate all the edges into a single edge between any two parts of the graph. And that may radically increase the complexity. And I'll give you several examples of that today. So these little fine points all of a sudden loom large when we actually come to build something.

Let me give you another example of that. A very fine example would be to say, what does sectionalization consist of? We talked about sectionalization. Suppose we want to get a trellis graph for a 4-section trellis, where we take pairs of variables. Well, the graph realization of that is simply obtained by agglomerating pairs of these blocks, like that.

OK, so let me do that, and then let me see what I've got. I've now got here, a constraint that affects two variables. Well, first of all, I've now got only two visible bits in each of the state variables. Here I have u1, u2, here I have u2, u3, here i have u3, u4. So we get rid of some of this state complexity. We did this trick in another way before, by sectionalization.

So we get rid of a lot of state spaces, including ones that are big. Let's see, what do we have here? This is now a code which has got two bits coming in, two bits coming out. It's a 4, 2 code. It's basically controlled by u1 and u2. Probably y0 and y1 are some simple linear function of u1 and u2. So this is a 4, 2 constraint code. It controls these two bits and these two state bits, these two symbol bits, and these two state bits.

What is this over here now? We've got 2, 2, 2. So this is going to be a code of length 6. And what dimension? This is, remember, u2, u3. So this whole behavior here is affected by u1, u2, and u3, it has dimension 3.

Or there's a shortcut I can do here, because this is a self-dual code, all of these little codes are going to be self-dual. Well, they're not going to be self-dual, but they're going to be rate 1/2. Half as many bits here as they do here. But that's another duality theorem that we won't prove.

OK, and symmetrically, we get another code here, which is a 6, 3 code, and another one here. So we can do this. The reason I call this styles of realization is there's obviously a lot of freedom in how we want to depict the code. And depending on how we depict it, when we get to decoding algorithms, it may affect the complexity of the algorithms. So we want to find a nice way of depicting it.

AUDIENCE: I don't understand why you-- so you want the ... freedom ... is [INAUDIBLE], because you would only have two [UNINTELLIGIBLE] to be used [UNINTELLIGIBLE]. And then you have two [UNINTELLIGIBLE] u1 and u2. [UNINTELLIGIBLE] u2 and 3.

PROFESSOR: There are two bits here, two bits here, two bits here.

AUDIENCE: But that's 6. That's 6. [INAUDIBLE]

PROFESSOR: Weight 6, all right.

AUDIENCE: [INAUDIBLE] freedom that you talk about.

PROFESSOR: How many possibilities are there for these 6 bits? I've got to consider all possible combinations of u1, u2, and u3 to drive what's happening in these two times. If I go back here and look at those two times, I see that there are three generators that I've got to consider, these three. All right, so for all eight possible linear combinations of those three generators, I'll get different combinations of these six bits here.

And I can go further. Actually, at the end of the day, I concluded here, we said the idea of sectionalization was to sectionalize as far as possible without increasing the branch complexity, which we've now translated into this constraint code complexity. And so it's even better to just keep aggregating. Consider the first half. And this is 1, 2, 3, 4, two bits there is still only a 6, 3 code. And the second half is still only a 6, 3 code.

When we get to the sum-product algorithm, that means we have to compute eight things when we get to this node. Maybe it's a slightly more complicated thing, but as long as we keep the dimension down to 3, we're only going to have to compute eight things. So we haven't really increased the decoding complexity at all by doing this. So that we consider to be our best sectionalization.

And if we tried to aggregate these two, what would we get? We'd simply get the eight bits here, we get a constraint that says it's an 8, 4 constraint, it says these eight bits have got to be in the code. So we only go that far, because that would increase the complexity to 16. All right, so that's our minimal trellis. We call it a two-section trellis where the symbol bits have been grouped into 4-tuples. And for decoding, that's the simplest one.

So you see what kind of games we can play here. Yeah?

AUDIENCE: At the beginning of this process, you started at the [UNINTELLIGIBLE].

PROFESSOR: Well, you remember in more detail how we concluded this was the optimal sectionalization. We looked at time 3, in particular. And we said there are three generators that are active at time 3. So I'm going to expand that as far as I can without bringing any more generators into the picture. So I can't expand it over here, because I'll hit u4. But I can expand it as far as I want over here. And that's the way we sectionalize.

AUDIENCE: So do you always start with [UNINTELLIGIBLE]?

PROFESSOR: Well, there's some art in this. Here it's kind of obvious where to do it. And I give two heuristic algorithms in the notes on sectionalization, which we'll come up with basically the same thing. But for simple codes, you can just eyeball it.

OK, so let's see. That's another general and very useful style of realization. It's a trellis realization, or a sectionalized trellis realization. Is there anything else I wanted to say about that? We can do it so that we get minimal branch and state spaces, or constraint complexities. It's cycle free if we aggregate the states into a single state space. Sectionalization, we talked about that. OK, so let me leave that up, because we're of course not done with that.

Now let me talk about some general properties of graph realizations. And the most important thing I'm going to talk about here is the cut-set bound. I want to get across the idea that a graph really captures dependency relationships. If variables are incident on the same node, they're obviously all dependent. Or if you think of larger sets of variables aggregated in this way, then they're somehow dependent. Variables that are very far away on the graph are less dependent on each other than ones that are close to each other on the graph. This is all kind of vague and woolly statements.

Let's see if we can make them more concrete. First of all, there's a very simple but important observation that -- I'll put it this way-- disconnected if and only if independent. What does this mean? If I have a disconnected graph, let's suppose I have one graph over here with certain external variables. Let me aggregate them all into y1 and some constraint 1 over here, and a completely separate graph, some different code over here, c2, constraining some separate set of variables, y2.

OK, that's a disconnected graph realization. It has to look like that, right? So we're aggregating the two disconnected halves. All right, what can I say? What code does this graph realize? This graph, the code that it realizes is simply the Cartesian product of c1 and c2. In other words, this means the set of all pairs, c1, c2, or I should say y1, y2-- y1, y2 such that y1 is in code 1 and y2 is in code 2. That is the set of all y1, y2 that satisfy all these constraints, right? So it's a behavioral realization of a Cartesian product code.

Now this is really a notion of independence, right? This independently realizes a code word from c1 in this part and a code word from c2 in this part. What would the generator matrix look like? The generator matrix from this would look like a generator for c1, 0, this is a 0, and a generator for c2. So there's a generator matrix notion of independence. In other words, these are just two independent codes that for some reason, we choose to regard as one code.

And the same thing is true that if we have a Cartesian product code, simply a code made up of two independent components, then we can always realize it in this way, right? We just realize c1, realize c2. So that's elementary, but it begins to get across the idea that graph properties have to do with dependence properties. And here's the most radical and simple form of that. If we have a disconnected graph, then it really realizes two independent codes.

Now, more important than that is the cut-set bound. What is a cut-set? It's a set of edges. Actually, in graph theory, it's defined in various ways. It could be vertices, it could be edges. Here, we're going to say it's a set of edges whose removal disconnects the graph.

Probably there are people in this room who know more graph theory than I do? Does anyone want to quibble or refine that in any way? That's my idea of what a cut-set is. Any elaboration? No? All right.

All right, so we have some large graph realization. It's got vertices which I always draw as blocks, its constraint codes, they have various interrelationships. I'm just showing one. We have various external variables that we bring out of that, and that's the general graphical realization of the code.

What are some cut-sets in here? I've drawn this, I'm thinking of this as a cut set. If I take these two edges and remove them, then I've disconnected the graph. So they form a cut-set. I've already mentioned the very close connection between cut-sets and cycle-free graphs, which is the graph is cycle-free if and only if by removing any single edge, I disconnect the graph. So every single edge is itself a cut-set. That's a very elegant characterization.

All right, so I'm thinking of a cut-set. I'm going to write it as chi for scissors. And it's a set of edges. So in our cases, what are the edges? It's a set of state spaces for some index set. OK, so I'm going to select some minimal set of state spaces that disconnects the graph. All my edges are internal variables. We don't really need to worry about these half edges out here, as I've said before. They're always going to trivially separate their variables from the rest of the graph, and it's just tedious to try to keep them in the explanation. So we're only talking about state-edges here.

All right, once I've done that, let me now do my agglomeration trick. Sorry, I'm trying to draw a dotted line around a part of the graph that I'm going to leave outside the external variables that I still want to be external. I'm going to leave the state variables, which I'm going to remember were once connected. And let me reconnect them now.

So I'm taking my original graph and I'm dividing it according to this cut-set into two parts. And I'm arbitrarily going to call this the past, p-- arbitrarily but suggestively-- and the future, f. And it doesn't matter which one I call p and which one f.

And I'm going to redraw this, so this is just a re-drawing of that in a nicer form. I'm going to aggregate all of these variables over here. And I'm going to call that the set of past external variables, y projected on the past. And similarly over here, I'll get the set of all external variables projected on the future. It was just the ones that occur when I make this kind of cut through state edges. Some of the external variables wind up in the past side, and some wind up connected to the future side. But there are obviously none connected to both because the definition of a cut-set. We've disconnected the graph.

All right, so there's the past. There's the future. And here are the state variables. Now I'm going to define these all together as a super-state space as I did in the trellis graph over there. So let's see, here I labeled the edges by a set of states. But what I mean here is that the super-state space is just the product of the component state spaces. It's elements s chi are just the set of sj, j, and sigma-j, is that clear? In other words, I have a vector of states here. And I consider the super-state space to be just the Cartesian-- this, again, is the Cartesian product of all the state spaces.

OK, and then I have constraints. So we'll call this the aggregate past constraint, and the aggregate future constraint. This constrains these variables in this state. This constrains these variables in that state.

OK, so that's where figure 5 comes from. Every year, people say, how did you get figure 5? That's how I get figure 5. Is there any confusion about that? Today I've been talking about agglomeration quite a bit, so maybe this year it'll come through better than in past years.

AUDIENCE: [INAUDIBLE] that we see projected on it. Or is it just--

PROFESSOR: It's just the aggregate of all these constraints. What does this say? It's the set of all the possible values of these variables, and these variables that can actually occur. That forms a linear space which we call little code. Anything consistent with that satisfies this constraint. Anything else doesn't.

AUDIENCE: [INAUDIBLE] for the c's projection of the code on the past?

PROFESSOR: No. The projection of the code on the past would be you realize the code and you just project onto these variables here. So no, this is just an index. This is a constraint whose index is the past.

OK, well, all right. Oh, but this looks like a trellis, like a two-section trellis. It looks very much like this thing right here. It has the same general form. And we can think of this super-state variable here, it sort of has the Markov property. It, again, tells everything about the past that is needed to specify what possible futures there could be. We're really interested in these up here, eventually. But this is the whole set of constraints that determine the futures along with-- there may be some more free variables over here. But these are the constraints we get from this half of the code on this half of the code and vice versa.

So again, a cut-set is related to a Markov property. Let me try to state this property more explicitly. The code by the behavioral realization is simply the set of all combinations of -- let me say, the behavior is the set of all past states and futures that satisfy the constraints.

OK, so this gives me a certain set of pasts that are consistent with each possible value of the state variable. Let's call that y projected on the past that's consistent with a particular value of the state variable, the super-state, the vector of states here. And these are the ones in the future that are consistent with that state variable. We define that so the set of all possible past and future y's is simply this Cartesian product.

And we say that in a more graph theoretic way. If I specify the state here as some particular value, sx, and I fix that, I've really disconnected the graph. For a fixed sx, I can just put that over here, put that over here. And this realizes, this connected graph that realizes the Cartesian product of whatever -- yp of sx. This just comes through and affects that with yf of sx.

So for any particular value of the state, I have a certain Cartesian product of pasts and futures they can occur. Any past that's connected with that state can be connected to any future that's connected with this state. All the possible future continuations of any particular past in this equivalence class are the same. This should sound very much like the kinds of things we were talking about when we did the state space theorem.

And we get a sort of state space theorem out of this. Just to draw this out, we kind of get a two section trellis realization where we have the various states in the state space here, s0, s1, and so forth. We have the pasts that are connected with s0, s1, and so forth. And we have the futures that are connected to s0, s1.

So we could get from this, this picture of the two sectioned trellis. This really represents a set of parallel transitions here. These are all the possible pasts that are consistent with that state. These are all the futures that are consistent with that state. We can tie them together, and that realizes the code. I should say explicitly, the code now is the union over the states in the state space, the super-state space of the pasts there times the futures. They're consistent with the state.

So it's a union of Cartesian products. And that's expressed by this diagram. This is one of them. This is another one. These are all of them. How many are there? The size of the super state variable, which by the way is the product of the sizes of the state spaces of each of these individual state spaces that I had coming across here.

OK. What's the cut-set bound then? For linear codes, it's basically the same as the state space theorem. The state space theorem said that the dimension of the state space at any time -- well, let me consider now this state space, the super-state variable. The dimension of this super-state variable has got to be greater than or equal to the dimension of the code that we're realizing minus the dimension of the --

this is going to be bad notation now. Now I mean the sub-code, so I should have written something else for the constraint here.

Minus the dimension of the sub-code whose support is entirely on the past, minus the dimension of the sub-code whose support is entirely on the future. And the state space theorem still applies for this partition. So that gives me a lower bound on the total dimension of the states in any cut-set.

So for example, in the 8, 4 code, suppose we have any realization, so some graph here, and any cut-set such that the size of -- I should say the dimension. I really mean that there are four external variables in the past and in the future. This is dimension, dimension. In other words, we have four variables sticking out here, and four variables sticking out here, and we have a cut set somehow going through there. So our picture now is going to look something like this.

What's the minimum possible dimension of the total dimension of the edges that cross this cut-set, the state spaces that are in this cut-set. We know this code very well now. Let's do the Muder bound. What's the maximum dimension? The dimension of the code is 4. What's the maximum dimension of any code that lives on the past? It's 1, because such a code has to have minimum distance 4. So it can't have dimension greater than the repetition code of length 4. So 4 minus 1, same argument for the future, equals 2.

So it says that however you do it, you're going to have to have at least two binary edges or one quaternary edge crossing a cut-set that divides the external variables into two equal sized parts. So let's go further. Let's suppose we want a cycle-free graph. OK, if we have a cycle-free graph, then all of the cut-sets are single edges.

All right, so in a cycle-free graph, when we make a cut, it's through a single edge. And we're saying the state space if there's a way you can make a cut that divides this into two equal parts, the state space has to have dimension 2, it has to have size 4 for any cycle-free graph.

Similarly, if we have the 24, 12, 8 code, we've proved that the minimum dimension of any central cut has to be at least 6. No, in the center, it has to be at least 8, right? Yeah, because the dimension of the code is 12, and a sub-code that has support on 12 can't have dimension more than 2. This can't have dimension more than 2. So that would be 8. So for the 24, 12, 8 code, the set of edges that cross the cut have to have total dimension at least 8. And if we want it to be cycle free, then there has to be a single edge that has dimension 8.

So the general conclusions -- we first talked about trellis realizations. We can generalize that any cycle-free realization-- and we're going to see that we have strong motivation to keep our realization cycle-free, because then we can do exact maximum likelihood decoding. So we'd like to have cycle-free realizations.

But what have we just seen? We've just seen that the state complexity cannot be less than that of a trellis realization with a comparable cut. In other words, one that divides the past and future in the same way, divides the external variables into the same size.

OK, so we really can't beat the trellis bounds, the Muder bound, in particular. Now again, there's some games we can play. Let's ask about the 24, 12, 8 cut. We've said if we want to realize it with the cycle-free realization, this says, well, if we have a cut that divides the external variables into two parts of size 8, then we're going to be stuck with a state space of dimension 8 and a size 256. No way around it.

Here's a slight way around it. Let me hypothesize a realization that looks like this. This is a 14, 7 code. So is this. So is this. We're going to divide the external variables into three parts, each of size 8. And we're going to have three internal state spaces, each of dimension 6. And we're going to constrain those by an 18, 9 code. And again, these all have rate 1/2 because this code is dual.

Now, is there anything in what I've done that prevents this kind of realization? Let's test the cut-sets. Where are the cut-sets in-- well, A, is this cycle-free realization? Yes, OK, good. Let's test the cut-sets. It's kind of a three-way symmetrical thing. The internal cut-sets are here, here, and here. Each of these cut-sets divides the coordinates into one set of size 8 and one set of size 16. In that case, what does the

Muder bound give us?

In that case, 8, 16, the dimension of the code is still 12. The minimum dimension of a code of length 16 and minimum distance 8 is 5. And one of length 8 and minimum distance 8 is 1. So as we did on the homework, we find that we are permitted to have a state space-- the minimal state space here could have dimension 6. So this possibly could occur.

Again, we're kind of camouflaging some of the state space. You see there is no central state space in this. There is no cut-set that partitions it into 12 and 12. So by being a little bit clever, we've got it down to something where instead of a 256 central state space, we've got three 64 state spaces. And this is what we saw on the trellis realization too, that if we sectionalize into three sections, then we get a 64 state and a 64 state at the boundaries between the three sections, but we still got the same 512 branch complexity in the middle.

And what do you know? We still have a constraint code with complexity 512 here, too. So if you consider that the more valid measure of complexity, the minimum dimension of any constraint code or branch space, then we haven't improved.

So subject to these qualifiers, this is a pretty strong argument that by going beyond trellis realizations to general cycle-free, graphical realizations, we can't gain very much. So the moral is to significantly reduce complexity, we're going to need to go to graphs with cycles. We must go to graphs with cycles. OK, so you don't get much for free in this life. So when we get to our sum-product decoding algorithm, we're going to have to apply it to graphs with cycles where it's iterative, where it's not exact, where it's just a lot less nice.

So what do we do when we go to graphs with cycles? Where is the potential gain? The potential gain is that now this super-state variable could be made up of a number of simpler variables. We've got a certain minimum dimension that we're going to need any cut-set, like 8 for the central state space of the Golay code.

But now we can spread it around over two or more sub-state variables. So for instance, maybe we can find something where we now have a graph with cycles where we have two state spaces here. Here's the cycle again. And the two state spaces each have size only 16. That would still satisfy this cut-set bound. So we greatly reduce the size of the state spaces, which is exponential in their dimension, but at the cost of introducing a cycle. I insist that's a cycle.

Let's go over to this trellis here. The next thing I might talk about is tail-biting trellis realizations. This is very simple. What's the first realization you would think of on a graph with a cycle? Well, here's a trellis realization. It's always going to look like this. And then let's just loop this around.

In other words, we make the last state space equal to the first state space. We don't require they used to be trivial state spaces of size 1. We allow them to have some dimension. And here's a potential realization on a graph with a cycle.

Cut-sets. Where are the cut-sets in this graph? Well, the cut-sets now look like that. All the cut-sets involve at least two edges, two state spaces. OK, so we potentially might get this kind of benefit.

Here's an example. Suppose instead, go back to our very favorite example, suppose we let u2 go across here, and we bring out u3 out of here, and we bring it all the way back to here. Everything still good? It's obviously another realization. Now I've made it clear that when I draw u2 and u3 separately here, I've really got a cycle. I've made a big cycle.

And now what's the number of states in this two-section realization of the 8, 4 code? This is a two-state tail-biting trellis, because each of these is just a little binary variable. So I get some kind of two-state trellis. If I were actually to draw out the trellis, I would have two states at time 0, two states at time 4, some kind of -- what have I got-- 6, 3. So I've got 8 branches that go back and forth, a couple of parallel branches. It's going to look like that.

Then same thing out to time 8. I don't have a time 8. This is really time 0 again, where I identify these two state spaces. This is really the same state as this, and this is the same one as that.

So that's if I really drew out the trellis in the style I originally drew trellises, this would be a picture of all the-- there's a 1-to-1 map between all the paths through this trellis that start and end in the same state, because that's what I mean when I identify the beginning and end states. So if I start in this state, there's a four-way branch here, another four-way branch there.

Is that right? No, there's only a two-way branch such that I can get back to the initial state. So there are 8 possible paths that start here and get back to the same state, 8 possible paths that start out from here and get back to the same state. And they together correspond 1-to-1 to all the code words.

OK, well, that's not a very big deal. I was able to reduce a four-state trellis to a two-state trellis. But suppose I do the same thing with the Golay code. In the Golay code, there's a very beautiful tail-biting trellis realization. It looks like this. It has 12 sections, so maybe I won't draw all the sections. It groups each of the output variables into pairs and comes around like that. And each of these state spaces has dimension 4, or size 16. And I give the generator matrix for this in the notes.

So this is for the 24, 12, 8 code, and now you test me. Is this a possible-- use the cut-set bound. See if this violates the cut-set bound anywhere. Does it for this code? No, it doesn't. Because every cut-set, no matter how I draw it, we're going to get two edges, each with dimension 4, adding up to a super-state of dimension 8. And the state spaces at all the even times, notice I've sectionalized here, so I only need to look at even times. The state spaces at all even times in this code could have dimensions as small as 8. Remember, at the odd times, they go up to 9.

And these are all little-- 4, 4-- these are all little 10, 5 codes. So the branch complexity is only 32. So this is very much simpler than any of the conventional trellises that I've-- than the minimal conventional trellis that we were able to draw for this code. Or if this funny little pinwheel with three arms on it, which is also a cycle-free realization, this code.

OK, so they're going to tail-biting. That's a significant advance. An aside note-- if we just break this, it turns out we have a generator matrix that has period 4, and let this go on infinitely, this is a realization of a rate 1/2, 16 state, branch complexity 32, as we expect-- perfectly conventional, except it's periodically time varying-- linear convolutional code with absolutely phenomenal performance. Like the Golay code, it has d equals 8. Therefore it has a nominal coding gain of 4, 1/2 times 8, or 6 dB.

This is with only a 16 state, rate 1/2 code. Look at the ones on the table. This is significantly better than that. And it's effective coding gain, I forget. But it's still very good, excellent, for the complexity of this code. So this is called the Golay convolutional code. And it's an interesting example of the interplay between this block coding stuff and the convolutional coding stuff, in this case.

Actually this code was first discovered in computer searches, but nobody realized how wonderful it was. And it was only when its connection with this Golay code was recognized that people realized this was really a special code, both from a performance versus complexity point of view, and also algebraically.

OK, well that brings us to the end. There's actually one more style of graph realization that I want to mention to you, for the Reed-Muller codes. We know that they're basically based on Hadamard transforms. It's a very nice kind of Fourier transform like graph realization of Hadamard transforms. So that gives us another style of realization of Reed-Muller codes that again, we can aggregate to lead to a lot of nice structures. Again, at the end of the day, unless we allow cycles, we don't really gain anything.

So I'll do that the first part of next time and then do the sum-product algorithm. I hope I can do it in one lecture.