--solutions for problem set 8. I found some problems as I went through them.

These are all new because we've never been in this territory before this early where we could actually offer problems. So it's a little bit of a shakedown.

8.2, simply, I don't think you have enough background to actually do it. If you want to try it, be my guest. I'll give you a sketch of a solution, anyway. But really, for the same reason that I didn't do exercise 3 in Chapter 11, we really shouldn't have done exercise 2, either.

For 8.3, which is a decoding exercise using the BCJR algorithm. I don't know if we'll get to the BCJR algorithm today. If we do, then please do it. If we don't, then we can defer it.

And you also need to know that the noise variance is 1. Or, pick something, but I suggest you just pick sigma squared equals 1, even though that's a large value.

OK, any questions? Has anyone tried to do the homework yet?

Not yet. Good strategy.

OK, Chapter 11. We have one topic to go in Chapter 11, which is just the last little bit on Hadamard transform realizations, which I do want to cover. We've really been talking about styles of realizations on codes on graphs, graphical realizations. We had the generator. And we had the parity check and we had the trellis-style. And we even had a tail-biting trellis. And we also developed the cut-set bound which doesn't exactly prove, but strongly suggests that we're going to need cycles in our graphical representations.

And one reason I want to talk about the Hadamard transform realizations, the Reed-Muller codes, is that they really demonstrate this point I think very nicely. If we allow cycles, we get very simple representations. If we demand cycle-free, then we get still pretty nice realizations, but much more complicated. They meet the cut-set bound everywhere. Reed-Muller codes in general are pretty good anyway for trellis

realizations, but they're certainly much more complicated than the realizations with cycles. So that's what we're going to do today is-- at least the first part is Hadamard transform realizations of Reed-Muller codes.

OK, what's the Hadamard transform over the binary field F2? It's related to but a little different than the Hadamard transform over the real field. So people seeing the latter, you'll see similarities, but you may see differences as well.

So the Hadamard transform. I think you remember when we defined Reed-Muller codes, one of the ways we defined them was in terms of a universal matrix Um, which I defined as the m-fold tensor product, very briefly written like that, of a 2 by 2 matrix, u1. What does u1 look like?

u1 looks like-- over the binary field, it just looks like this. It's the set of generators for the Reed-Muller codes of length 2. What you would expect, it's a lower triangular matrix.

And if we want to take-- I don't know if you've run into tensor products before. If we want to take the 2 by 2 tensor product like that, we simply replace each one by the matrix itself and each 0 by a matrix of 0's. So we get a 4 by 4 matrix which looks like this 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0. And what do you know?

That's what we called our universal Reed-Muller generator matrix for m equals 2. In other words, for Reed-Muller codes of length 4. And you see how all the Reed-Muller codes are generated. We can find generators for all the Reed-Muller codes from this set.

And similarly, u3 was this matrix that we've now seen many times. So all these matrices have common properties. They're all lower triangular. They have many properties.

One of them is that all of them are self-inverse. And that's very easy to prove from the fact that u1 is clearly self-inverse. u1 times u1 is the identity matrix. And because these are all tensor products, that makes all of these their own inverses.

So what's the 2 to the m by 2 to the m Hadamard transform is if we have a vector u, then u times u m, a vector y is called the Hadamard transform of u. What's the inverse Hadamard transform if you like?

Well, since u m is its own inverse, if we just multiply both sides by u, we get u equals y u m. So to get u back, we just transform y again by u m. So the inverse Hadamard transform is the Hadamard transform again in this case. So we get the typical kind of transform relationship. You can think of this time domain, frequency domain. Anyway, this is the set of variables from which we can get this set of variables. They're somehow in dual domains.

OK. Well, now recall one definition of the Reed-Muller codes. The Reed-Muller codes are-- Rm of mr is-- sorry, it's always rm, which I've always found curious notation. Since m is more important than r, it would seem that you would say m first, but maybe it was for this, I don't know.

This is generated by-- so let's say that. The rows of u m of weight 2 to the m minus r [? for ?] greater.

So for instance, the 0 order Reed-Muller code of length 8 is simply the 8, 1, 8 repetition code and it's generated by this last row. It's the only one of weight 8.

If we want the Reed-Muller code of length 8 and minimum distance 4, we take this, this, this, and this as our four generator rows and we get the 8, 4, 4 code that we've been mostly using for examples.

If we want the single parity check code of length 8, the 8, 7, 2 code, we take all these 7 generators, everyone except for the unique weight 1 generator up here.

And as a hint for doing the homework, it's helpful to notice that if we write the indices of these rows as 0, 1, 2, 3, 4, 5, 6, 7, then notice there's a relationship between the Hamming weight of the m bit binary expansion of these indices and the weights of these rows. Is that clear instantly?

I won't prove it, but if the Hamming weight is 3, then the row has weight 8. If the

Hamming weight is 2, then the row has weight 4. If the hamming weight is 1, then the row has weight 2. And if the hamming weight is 0, then the row has weight 1. And that's the easiest way I've found to do problem 8.1.

So these-- there are infinity of relations, nice relations, you could prove for these things. They're very important combinatorially and so forth.

OK, so from this observation, we can therefore define a Reed-Muller code as the set of all u times u m where we simply let some of the ui free for some rows. In other words, it's just a binary variable. And 0 for other rows as I just described in words over here.

So basically, where we're going is we're going to define some kind of graph which realizes u m. We're going to put the 8-- do it either way. I'm going to put y0 y7 over here. If we're doing a code of length 8, I'll put u3 in here. We're going to have 8 other things sticking out over here. I shouldn't draw these because they're really going to be internal, implicit variables. So this will be u0 through u7 in some order. It turns out to be necessary to play with the order to get a nice-looking graph.

And we're going to fix some of these to 0. For instance, for the 8, 4, 4 code, we're going to fix those to 0. And we're going to allow these coefficients corresponding to rows of weight 4 or more be free arbitrary binary variables. So the set of all code words will be generated as the set of all Hadamard transforms of 8 tuples, which are fixed to 0 in these 4 positions, and which are free in these 4 positions. Is that clearly stated?

I think I'm reasonably proud of that. So that's what I mean by this notation here. And it's clear that the dimension of the code is the number of free coefficients here. Basically, because the u m matrix is invertible, so we're going to get an isomorphism between 4 tuples and code words. So that's a way of constructing Reed-Muller codes. Now, how do I get a nice graphical representation of u m?

Let's work on that for the next little bit. Well, as always with Reed-Muller codes, we should do things recursively. We should start with u1. That's a pretty simple

relationship. How do we draw a graph for the relationship y equals u1 u?

OK, that means y0 equals u0. Let's see. Is that right?

I've been putting u on the left here to imitate what we do with generator matrices. So that equation is y0, y1 equals u0, u1. Sometimes I have problems with basic things like this.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah, I want row vectors. Sorry. OK, so that's the equation, which we can see that says y0 equals u0 and y1 equals u0 plus u1. Do I agree with that?

y1 equals-- no, still not right.

AUDIENCE: y0 is [INAUDIBLE].

PROFESSOR: Plus u1. And y2 is-- y1 is u1. All right. Sorry, now we're in business.

OK, so let's draw a little graph of that. We have y1 equals u1. And if I make an equals down here, I'll get a little u1 in here. So I can put u0 in here and get out y0 equals u0 plus u1.

OK, so that's a graphical realization of this Hadamard transform. If I put 2 bits in here for u, then I'll get the Hadamard transform out here for y. But by the way, notice there aren't any arrows here. This is a behavioral realization. And I could equally well put a 2 tuple here on the y's and I'd get out the Hadamard transform at the u's. It goes either way.

And this I find somewhat interesting is called a controlled NOT gate. The idea here is that this bottom variable is the control and it determines whether this variable is inverted or not as it goes across here.

And this is a basic gate in quantum mechanics. Maybe the basic gate in quantum mechanical realizations. [INAUDIBLE] at least [INAUDIBLE] cubits.

But anyway, that's actually neither here nor there. All right, so that's how we realize

u1.

All right, now recursively there must be some way of putting together little blocks like this to make bigger blocks, say, for u2. To realize u2 or u3.

OK, so for u2, let me just try to steam ahead and hope this will work. We build another gate to do a set of transforms to an intermediate variable here. Intermediate variables, and then we sort of do a butterfly.

We're going to build this out of 4 little blocks like this. We're going to write like this. And how are we going to tie them together?

The outputs here are going to go to the equals signs. The outputs here are going to go to the plus signs. And we get something that looks like that for the 2 by 2 Hadamard transform. And y0, y1, y2, y3.

Now, probably this should be u2 and this should be u1 just from the way these works. And if we do that, we get what?

This is u3. This is u1 plus u3. This is u2. This u0 plus u2. So we get that-- doing it backwards, y3 equals u3, y2 equals u2 plus u3, y1 equals u1 plus u3, and y0 equals u0 plus u2 plus u1 plus u3. And that's probably correct, but it's something like that. Done in the notes.

And those of you who have ever looked at the fast Fourier transform will see some resemblance here to the butterflies that occur in a fast Fourier transform. And it's not surprising because these are all based on groups of-- in this case, z2 squared. And so they have the same algebraic structure underlying them. But that's, again, a side comment that I won't take time to justify.

And similarly for 8, if we want to do the 8-- there's a picture in the notes-- we build 8 of these gates. Maybe I actually have to do it because I want to reduce it also. So 1, 2, 3, 4. Sorry, this takes a little time.

This is where making up slides ahead of time is a good idea.

All right, so we have three levels of 2 by 2 Hadamard transforms as you might expect. At each level, we do four 2 by 2 Hadamard transforms. And we connect them together. At the first level, we stay within the half. We connect them together in the same way.

And at the next level, we have to spread out over the whole thing, so we get-- this goes up to this equals. This goes up to this equals. This goes up to that equals. This comes down here. This comes down here. This comes down here and this goes across. That gives y0, y1, y2, and so forth. y3, y4, y5, y6, y7.

And again, we have to jigger with the order here. Probably something like this. OK, in any case, it's done correctly in the notes.

All right, so this executes an 8 by 8 Hadamard transform. You put y's on here and you get the Hadamard transform out on the u's, or vice versa. You with me?

OK. Notice that the components here are all very simple. I've got a graphical realization of the Hadamard transform relationship that involves only binary variables, first of all. All the internal variables here are just bits. They're all binary.

And all of the constraint nodes are these simple little degree 3 constraint nodes. The only two nontrivial ones you will probably ever see. Namely, the zero sum type of node and the equality type of node. 3, 2, 2 and 3, 1, 1 if we consider them as codes.

So it's made up of extremely simple elements. And in that sense, we can say it's a simple randomization. But of course, this graph is full of cycles, isn't that right?

Yes. These little things cause cycles. So we get cycles as we go around like that, for instance. Nonetheless, it's a perfectly-- you put something over here and you get something deterministic out over here.

All right. So to realize the 8, 4, 4 code-- are you all with me on this? I should check again. Is there anybody who doesn't understand this? That's always the way to put the question. Yes?

AUDIENCE: You count the output when you're counting degree?

PROFESSOR: Do I count the output when I'm counting degree? In this case, yes. I reserve my options on that. Sometimes I do, sometimes I don't. It's whatever is convenient. My choice.

Sorry, I think I just whacked the microphone. OK.

So now, let me go to my Reed-Muller code realization, the 8, 4, 4 code. How are we going to realize that?

We've got these internal variables here, some of which we're going to hold to 0. These four we're going to hold to 0. And the rest we're going to let go free.

And so I claim that already now I have a realization of the 8, 4, 4 code, if I regard these as internal variables, these as my external variables over here. OK, does everyone find that plausible at least? OK.

But having done that-- now, watch this part closely because this part you're going to have to imitate on the homework. We can do some reductions. All right.

If I have a 2 by 2 Hadamard transform of 0, 0, what's the result?

0, 0. OK. I can do that in several steps, but that's a good way to do it. So I don't really need these gates here. I can just put 0's over here.

Similarly, down here if I have a 2 by 2 Hadamard transform of 2 free variables, internal variables, what do I get?

I get two free variables. One of them happens to be u3 plus u7 and the other one is just u7. I could call this u3 prime, let's say. And since it's an internal variable that I'm just using to generate the code, I can delete that, too.

So what can I do about these here? When I have one free variable and one zero, what's going to happen?

This free variable is going to come through here and here. The 0 doesn't count

here, so a plus just becomes a repetition in effect. It's got to have the same parity as this input, so it is that input.

So by another major sweep, I'll just-- this is an internal variable. I'll just call this u6. OK, I don't need to actually draw a gate to generate it. It has to be the same here and here, so I have to tie it together. Similarly, down here I can just call this u5.

And now, continuing in here-- I hope this is going to come out all right. It doesn't seem quite right so far, but let's see how it comes out. What do we have coming in here?

We have two different free variables. So again, let me just call this u3 prime again. And this is maybe u5 prime. But in any case, I can just draw this. In this case, I'm coming around this way. And u3 prime was dangling anyway, so I can leave that over here.

And similarly down here, I get two free ones, so let me just call them like this. And this is equal to y7. Well, it's a little strange. I did something wrong there. I did something wrong.

Well, maybe not. Maybe not. This is not going the way it's gone in the past, so maybe I drew the graph differently in the first place. This is u6 again. This is still u6 up here. But this is a 0. So this is u6 also coming out here. But I need this little-- I do need to replicate u6 three times. And maybe that's what I needed down here too, was to replicate u5 three times.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Excuse me?

AUDIENCE: The bottom line.

PROFESSOR: Say again, I just didn't hear it.

AUDIENCE: The bottom line. You remove that [INAUDIBLE] shortcut.

PROFESSOR: I removed-- this equaled a shortcut.

AUDIENCE: Yeah. But the [INAUDIBLE].

PROFESSOR: There was an [INAUDIBLE] on top of it, right. And--

AUDIENCE: That cannot be replaced.

PROFESSOR: That cannot be replaced by a shortcut. OK. Thank you. So it should look like that?

I don't know. I think I've botched it, so you're going to do it right on the homework. Let me just-- yeah, good. This is what it should look like. You do need the equals there.

It's because I drew the graph backwards in the first place. Anyway, presto change-o. Going through that kind of manipulation, what we eventually come up with is something that looks like this. And I apologize for botching it.

And coming around, this is tied to this and this is tied to that. All right. And I asked you to imagine that if I-- what I did was since I can draw the Hadamard transform-- either way, I drew it the wrong way. And therefore, I wasn't going to get what I intended to get, which is this.

But it's done correctly in Figure 10. And the types of reductions that I explained to you are the types of reduction that's necessarily to get from Figure 10A to 10B. Please do try this at home.

Now, so I claim that this is a realization of the 8, 4, 4 code. And you can verify that either by putting-- you'll find there are four free internal variables here and you can either set them to 1 or you can try this trick that I discussed before, I think. That we can regard these as an information set. And by fixing y0, y1, y2, and y4, you can trace your way through the graph and find out that y3, y5, y6, and y7 are determined.

Again, do try this at home. So you can convince yourself that this is a graph for 6-- that has 16 possible solutions for all the internal and external variables. 16 possible

trajectories that are valid that satisfy all the internal constraints. And that the external 8 tuples that are parts of these trajectories are the 8 tuples of the 8, 4, 4 code. I just assert that now. Please verify it.

All right, so now let's step back and take a look at this realization. I believe it's the simplest representation of the 8, 4, 4 codes if you account the complexity of the constraints and of the variables. Notice that all the internal variables, the states if you like, they're all binary. So it's a two-state representation if we want to use that language, that all the constraints are simple, either 3, 2, 2 or 3, 1, 3 constraints. We count that as the branch complexity in effect. This isn't a trellis, but this is what's analogous to the branch complexity as we'll see when we get to the sum product algorithm.

The complexity is proportional to the number of code words in each of these constraints. So there are either 2 or 4 code words. It's a very simple realization. And there are only 12 of these constraints.

On the other hand, it does have cycles, like this cycle here. When we get to the sum product algorithm, we'll see that there's a fundamental difference between decoding on a graph with cycles and a graph without cycles. Without cycles, we get an exact decoding algorithm. It does maximum likelihood decoding, like the Viterbi algorithm. Or actually, a posteriori probability decoding.

And in any case, it gives us the optimum, in some sense. When we decode on a graph with cycles, we get some degradation from the optimum. Sae-Young Chung tried decoding with this. And as I remember, it was a few tenths of [? db ?] sub-optimum. It wasn't bad. But it wasn't optimum either. So that's the trade-off we make.

Suppose we want to make this into a cycle-free graph. We can probably do that by agglomeration. Let me suggest the following agglomeration.

Let's make all of this-- let's suppress that cycle into one big block. Let's suppress this cycle into one big block. So we're going to consider this one overall constraint.

What is that constraint?

We have two outputs. Or I'm sorry, two variables over here. We have four variables over here. So it's going to be some code of length 6. And from the fact that you have three inputs coming in here, you might guess it's going to be a 6, 3 code, which it is.

So if we do that, we simply get 6, 3. And similarly, a 6, 3 constraint down here. There are 8 possible valid behaviors just for this constraint. And again, since everything is self-dual here, it's going to be 6, 3. So let me, again, assert that. Does anybody recognize what this realization is?

Would it help if I drew it like this? This is four variables, four variables. Anybody? Come on, you've seen this before.

AUDIENCE: [INAUDIBLE].

PROFESSOR: No. This is a realization of a different style. This is now a cycle-free realization, as you can easily see either from this or from-- well, it's cycle-free now. Sorry, I should do one more thing.

I should make this into a quaternary variable. So this will be 2. It's basically u5, u6 in this notation. Without doing that, there's still a cycle. This little cycle going back and forth there as we said before.

So somebody I heard say it. Say it again.

AUDIENCE: Trellis.

PROFESSOR: Trellis. It's our two-section trellis realization of the 8, 4, 4 code where we divide a code word into two sections of length 4. And simply, this is the trellis at the first half. Eight possible lines, each one containing two parallel transitions.

Sorry. Four lines each containing two parallel transitions going to one of eight states. And that's what the actual trellis looks like.

So this is, indeed, cycle-free. What's the difference between this realization and the other one?

Well, first of all, we've got four states in this cycle-free realization and we've got-- certainly, a much larger constraint code here, which will turn into complexity, computational complexity, when we go to decoding.

Now, we've got a branch complexity of 8. Whereas, we never had anything larger than 4 in the zero sum constraints of the previous diagram. So in that sense, this is more complicated. Its state complexity, its branch complexity is higher than the cycle-free one that we did. Than the one with cycles, but it's cycle-free.

You remember also we did this one. We have both of these going down here and we consider them each to be a binary variable, then we get our tail-biting realization, which is only two states. But it now has a cycle.

And if we're going to go to cycles, I would go all the way to that other one. But maybe not.

If you decode this tail-biting realization, maybe its performance is a little bit better because it more faithfully represents what's going on inside the circuit than the other one. How many different ways have we had of realizing the 8, 4, 4 code now?

We've had at least half a dozen, maybe more than that. And this is all to illustrate the various styles that we have. We've now had at least one of each style. And now we have one of Hadamard transform. Or, as I'm going to call it, we have a reduced Hadamard transform, the one with only the 12 simple constraints in it. Or, we also have the agglomerated Hadamard transform, which gets us to cycle-free. This has cycles. But this is simple and this is, at least, more complex. So there's a trade. What do you want?

OK. Just one more side comment on the subject. For exercise 2, which I didn't assign, shows that in general an agglomerated Hadamard transform realization of a Reed-Muller code, you can always get one that looks like this. Let's see. Which is sort of a four-section realization where each of these has length 2 to the m minus 2.

This is like the four-section realization of the Reed-Muller code.

Each of these state variables has dimension which meets the cut-set bound. And the cut-set bound for a cut that divides the trellis into one quarter and three quarters or one half and one half. You remember all of these dimensions were the same as simply the state dimension of a four-section trellis.

And I hope you remember that the state dimensions were the same at all four boundaries. So we always get the same dimension for each of these, the same as in a four-section trellis. And we get the minimal trellis complexity here. So this is going to turn out to be length equals 3s and k equals t, where t is the branch complexity, the minimal branch complexity parameter for a trellis that we had in a table back in Chapter 6.

So this actually turns out to be a nice-- the nicest I know-- cycle-free realization of a Reed-Muller code. For instance, for a 32, 16, 8 code these dimensions are 18, 9, 18, 9. These are all 6. It's a 64-state realization with 8 symbols at each of these points. And each of these is a 14, 7 code. Is this fundamentally simpler than a four-section trellis realization of the 32, 16, 8 code?

Not really. The branch complexity is the same as it is in the central sections of this code. This is very close to being a-- it is a four-section realization if we draw the four sections like this.

Except we have to draw a little additional thing up here, which is slightly helpful. So it's sort of a expanded trellis that looks like this.

OK, so maybe think of that as a quote, "explanation" for this. And I give explicit expressions for s and particularly for t, which perhaps you can derive or perhaps you don't care. You won't be held responsible for it, anyway.

You might even remember that the Golay code, the 24, 12, 8 realization, which looked very much like this but contained only three sections, each of length 8 and a single 18, 9 constraint in the middle. These were all 6. Maybe you remember that, maybe you don't. But that's a realization of the Golay code. And it shows that these

two codes are very close cousins. There are many ways of showing that they're very close cousins. They are.

OK, so now I'm just going off into side comments. You obviously won't be held responsible for any of the side comments. But I hope they give you something of a flavor for how the subject can be developed.

OK, that's the end of Chapter 11. There is an appendix in Chapter 11 that talks about other flavors of graphs. Factor graphs is a more embracing philosophically, conceptually notion where each of the constraints, instead of representing codes represents the factors, local factors, some global function factors into a product of local functions. The local functions are represented by constraints. It's a nice way of looking at things. It's not strictly necessary for this course, so I haven't gone into it. But most of the literature now is in terms of factor graphs.

In other fields, we have related topics, like Markov graphs or Markov random fields. And slight differences in choices have been made in these areas.

Here, you see we're putting all the variables on the edges and the constraints as nodes in a graph. And that's the normal graph style. In Markov graphs, they make exactly the opposite convention.

They make the variables into nodes, which is probably what most people would do if they sat down to start doing a graphical model of something. And they make the constraints into edges. But there are several problems with that.

One is that if you have a constraint of larger degree than 2, then you really need a hyper-edge, which is represented by a clique. And cliques sometimes introduce artifacts that you don't really want. So I think it's actually an inferior style, but it's a very popular style in some fields. Physics, particularly, for indicating dependencies. All of these graphical models are methods of indicating dependencies and what's related to what.

There's another style. It's very popular in statistical inference, which is called

Bayesian networks. And this style is similar to ours except the graphs are directed. Everything is a cause and effect relationship. So you have one variable that causes another variable.

We've generally stayed away from cause and effect here. We've used the behavioral style, undirected graph. But Bayesian networks have been hugely popular in statistical inference. And so the appendix is just to explain the relationship between these various styles. And obviously, you can work with any of them. It's a matter of preference.

OK, any other questions about Chapter 11? This was the basic introduction to codes on graphs.

Chapter 11 is, what are codes on graphs? Chapter 12 is, how do we decode codes on graphs? Which is the sum product algorithm. So we'll move into that now. And then Chapter 13, we'll actually talk about classes of capacity-approaching codes. So these three chapters certainly form a closely related set that should all be read together.

OK, what are the basic facts about sum product algorithm? I'm going to describe it as a method of doing A Posteriori Probability decoding, APP decoding, initially on cycle-free graphs. I'll define-- what do I mean by a posteriori probability decoding?

Then, we'll develop the algorithm for cycle-free graphs where the theory is kind of complete. There are various theorems that you can easily prove about the performance of the algorithm here. The algorithm completes in a finite amount of time. Basically, equal the amount of time it takes to get from one side of the graph to the other, which in graph theory language is called the diameter of the graph. And it does exact a posteriori probability decoding. So if that's actually what you want to do, this is a good way of doing it.

If you do this on a trellis for instance, it's the, by now, well-known BCJR algorithm, for Bahl, Cocke, Jelinek, and Raviv, who published this algorithm back in 1973. Probably known in other fields before that. Actually, you can sort of find it in

Gallagher's thesis. You can certainly find the sum product algorithm there.

Anyway, the BCJR algorithm is now widely used as a component of these decoding algorithms for turbo codes, in particular, and capacity-approaching codes in general because it's an exact way of decoding a trellis. And probably fast. Although, we'll find it's more complex than the Viterbi algorithm. So if you are just interested in decoding a trellis, you would probably use the Viterbi algorithm. But you use the BCJR algorithm when you actually want to compute the a posteriori probabilities. And that's what you want when it's part of a larger algorithm.

So we'll do all the development on cycle-free graph. Where we really, eventually, want to use it is on graphs with cycles. And there are hardly any theorems.

In this case, you find yourself going around the cycles. So there are all kinds of new questions. How do you start? When do you stop? What are its convergence properties?

It becomes an iterative and approximate algorithm. For instance, when I said that decoding of that simple, little reduced Hadamard transform realization for the 8, 4, 4 code, which had cycles in it, I said it was a couple of tenths of [? db ?] sub-optimum. That was obtained by running this algorithm on that little graph with cycles until it seemed to have converged to something and plotting its performance. And it's not quite as good as the performance would be on a cycle-free graph like this one.

All right, so you give up something in performance. And you give up-- it becomes an algorithm that runs indefinitely. And you'd have to figure out some stopping criterion and so forth. All together, it's much more, what are we doing here?

We can't really say with any precision what we're doing here in most cases. Although, in Chapter 13, I'll give some cases where we can say quite precisely what it's doing. But it becomes much harder to analyze. But in coding, this is actually what you want to do. This is what works. Just try it on a graph with cycles and hope for the best.

And because we design our own graphs in coding, we can design them so that the

algorithm works very well. Basically, we want to design them to have cycles, but extremely large cycles, large girth. And then the algorithm tends not to get confused and to be near optimal. Because it takes a long time for information to propagate around the cycle and it's pretty attenuated when it comes back in.

Whereas, with the short cycles that you saw here, that's not good. And in many of these other fields, like physics for instance, image processing, you're dealing with graphs that look like grids. Something like that.

You have lots and lots of short cycles. And then just applying the sum product algorithm to a graph that looks like that won't work well at all because you have these little, short cycles. So if you can design your own graph, this is a good thing to do.

OK, so let's get into it. First of all, let me describe what I mean by a posteriori probability decoding. This means simply that for every variable, both internal variables and external variables, we want to compute the probability of that variable. Let's call it probability that some external variable y, k takes on some particular value y, k given the entire received sequence.

OK, so you can think in terms of the 8, 4, 4 code. We're going to take this code. We're going to transmit it over some kind of a noisy channel. We're going to get a received sequence. And from that, we want to figure out the probability that any particular input bit is equal to a 0 or a 1.

That might be more precisely what we're going to do. So we really want to develop a vector consisting of the values of this a posteriori probability for all possible values of the variable. This is sometimes called a message. But this is what we're going to try to compute.

If we're actually decoding a code and we get the a posteriori probabilities for all these variables, then at the end of the day, what do we want to do?

We want to make hard decisions on each of these and decide what was sent. It's quite easy to show. Actually, I'm not sure I do this in the notes. That a posteriori

probability decoding is what you want to do to minimize the probability of bit error. What's the probability of bit error?

It's the probability that y is not what you guessed it was. In other words, it's 1 minus the max of all of these. You would choose the max. The probability of it actually being the max is given by that a posteriori probability. And 1 minus that is the probability that it's anything else. And that's the bit error probability.

So doing maximum a posteriori probability decoding on a bit-wise basis is the way of minimizing the bit error probability. Now, that isn't actually what we've been doing in the Viterbi algorithm, for instance, or in general in our decoding. We've said, we want to minimize the probability of decoding any code word on a sequence basis. We want to minimize the probability of error in decoding the whole sequence. So these are not exactly the same thing.

In fact, if you do max APP decoding, you may not even get a code word. You'll make individual decisions on each of these bits, and that may not actually be a code word. There's nothing that requires it to be.

Usually, it is. Usually, there's no difference. These algorithms tend to-- at least in relatively good signal noise, reach ratio situations. They tend to both give the same answer. And the probability of bit error or word error is not markedly different whichever one you use as a practical matter. But it's certainly possible in principle that since maximum likelihood sequence decoding gives you the lowest word error probability, maximum APP bit-wise decoding has got to give you a higher word error probability.

So in any particular situation, you've got to decide which is more important to you, minimizing the bit error probability or minimizing the probability of any error in the whole block. Actually, generally, the latter is what you prefer. That governs your minimum time between decoding errors and things like that.

So that's a general discussion of what APP decoding is. Why do we want it here?

Because when we get to capacity-approaching codes, we're going to talk about big codes that are made up of smaller codes. The great thing about this graphical realization-- this is realization of linear systems. You realize a big system by putting together blocks representing little systems. That's a good way of designing things for all kinds of reasons.

It's going to be the way we design capacity-approaching codes. And we're going to want a decoding algorithm for a part of a system. Say, this 6, 3 part, or say something else. Say this whole thing is a part of the system. This whole thing is just 8, 4. That yields soft probabilistic information in the output that we can then feed into decoding algorithms for other parts of the system. So we won't want to make hard decisions.

We'll want to feed the whole message, the whole vector of a posteriori probabilities from one-- a partially decoded part of the system. We'll feed that into some other part and then go decode that.

For instance, in turbo codes we'll decode-- we'll make it up out of several convolutional codes. There will be several trellises. We'll do BCJR decoding of one trellis. That will give us a bunch of a posteriori probabilities about the bits that are involved in that trellis, then we'll pass them back to some other trellis decoder that's decoding-- apparently, a different trellis, but involving some of the same bits.

So maybe I shouldn't go on too long with these generalities, but that's why APP decoding is going to be a good thing for us to do. And it's not because ultimately we want to do APP decoding of a single code. It's because we want the soft decisions to feed into other decoding.

All right. How do we compute this?

Probability that yk equals yk given r is going to be a basic probability theory. It's just the sum over all, let's say, code words in which yk equals yk of the probability of those code words. So sum over all y. I'm just making up notation here.

In our code, we're going to have some code words in the code that have yk equals

0 and some code words that have yk equals 1. So we'll divide the code words into two sets, the ones that are compatible with yk equals 0 and the ones that are compatible with yk equals 1.

The a posteriori probability of yk equals yk is simply the sum of the a posteriori probabilities that we get of any of those code words. I don't have to make proportional sign there.

But by Bayes' law, each of these-- Bayes' p of y given r is proportional to. Here's a sign that a lot of people use. I'm not completely comfortable with it yet, but that means proportional to. Prop 2 in [INAUDIBLE]. Probability of r given y. In other words, the a posteriori probability of a code word is proportional to the probability of r given y. Why?

Because by Bayes', it's actually equal to that times probability of y over probability of r. But this is the same for all of them. If the code words are all equiprobable, these are the same for all of them. So it's just proportional to the likelihood of getting r given that you transmitted y. I'm sure you've seen this many times before, so I won't do in detail.

So this is proportional to the same sum. y in this code word, where y-- a portion of the code where yk equals y. p of r given y. But that's easy. I'm assuming that I have a memoryless channel here. That's been implicit throughout this course, like the Additive White Gaussian Noise channel or the Binary Symmetric channel. So this just breaks up into a component-wise product.

This is just the product of p of-- over the k. p of rk given yk. And because yk has the same-- I should say k prime here. Sorry, that's very confusing.

So this is over all k prime. But I can break this up. One of these terms is going to be k. For that, yk is fixed at little yk, at this particular value. So I can write this in turn as this for k prime not equal to k. And then I get a common term which is p of rk given yk. OK, so this is the expression that I want to compute. Two comments.

When I say "sum product," it's really because this is just a sum of products here. We

see we want to compute this actually rather nasty sum of products. We're looking for an efficient way of computing the sum of products. And we're going to show how this could be computed by tracing through the graph. That's one comment.

The second comment is that in the literature, particularly in the turbo code literature, this has a name. This part, the direct likelihood of the symbol given what you received corresponding to that symbol, is called the intrinsic information about y, k. And this part, which has to do with, basically, the likelihood of yk given all the other yk prime, for k prime not equal to k, is called the extrinsic information.

We basically take these two parts, which we can compute separately, and we multiply them together. And we get the overall probability. So the probability consists of a part due to rk and a part due to all those other rk prime, which we're going to compute by going through the graph.

And I just introduced this language because if you read papers, you're going to see this throughout the literature. OK. Let me start moving back. I never used this.

Well, let me introduce one more thing. So we want to compute all these messages. In particular, we want to compute the probability of yk equals yk given r. This part here is really the r on k prime not equal to k. Terrible notation. The other r's. So that'll be a message.

We also want to compute similar things for all the internal variables, which we have sometimes called state variables. And it breaks up in the same way. And I won't take the trouble to write out the expression. But again, now we really want to go over all configurations which have a particular state variable in them. So we're really going over all behaviors. And we can compute probabilities this way. I won't belabor that right now.

So we want messages, really, indicating the likelihood of each of the external variables and each of the internal variables. The whole set of possible likelihoods, the APP vector, the message, for both the symbol variables and the state variables. And that's what the sum product algorithm is going to do for us. OK.

So the sum product algorithm is based on-- I discuss it three parts. One is the past future decomposition of the code. Another is the actual sum product update rule, which is based on another decomposition. And finally, we need an overall schedule.

OK, so let me discuss this first. Let's take a particular state variable. I only mean internal variable. And here it is. Somewhere in our big graph, it's an edge.

Our big graph is cycle-free. So what's the special property of edges in cycle-free graphs?

Every edge is a cut-set. So if I take this edge, it divides the whole graph-- we've done this before-- into two parts, which we can call past and future. And if we were to take out this edge, it would disconnect the graph from these two parts. So there's no other connections between the past and the future because the graph is cycle-free.

And there's some set of past observed variables, rp, and there's some set of future observe variables, rf. So before, I called this yp and yf. These are the received observations corresponding to all the past variables, the received observations corresponding to all the future variables.

And the past future decomposition is basically like this here. Let me write it out. It basically says that the probability of-- that state variable has a particular value given r is proportional to the probability that the state variable has a particular value given r-- that part-- times the probability that it has a particular value based on the future part.

And that we can simply-- when we take the vector, we can simply multiply the two components that both have a common value sk and we get the overall a posteriori probability that this state variable has a particular value.

Now, I'm sure there is a-- OK. In the notes, I go through a little development to show this. The development is, first of all, based on the fact that if I ask-- now in the corresponding expression here, I want to have the sum over all y and s such that

the behaviot-- I'm going to just lose myself in notation here. We only have a few minutes left.

This is basically going to be over a code that's consistent with the state having this particular variable. But when I cut this code up like this, the overall code of all configurations that are consistent with a certain state variable sk is going to divide up into a Cartesian product of the past part of the code that's consistent with sk across the future part of the code that's consistent with sk.

And this was precisely the argument we went through when we developed the cut-set bound. That given sk, we can take any past that's consistent with sk and any future that's consistent with sk and map them together. And in effect, fixing the state does disconnect the graph. And this is the basic Markov property that we used to develop the cut-set bound. Really, just depending on that again.

Plus the basic Cartesian product, lemma, which is that if I have a sum over a Cartesian product, x cross of f of x g of y, what's a fast way of computing that?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Right. So this is trivial. But in fact, that's what we do in many fast algorithms. It's just the sum over the x of the f part, the sum over y of the g part. And you multiply those two together. Just think of x and y being on some rectangular array.

Here, we compute all the products corresponding to every combination of x and y individually.

Here, we first take the sum of all those, the sum of all those, the sum all those. We multiply them times the sum of-- in this way. And we will get exactly the same terms if we multiply them out. And so this is just a fast way of computing this. And that's really what we're doing when we divide this up in this way.

I think I'm incapable of explaining that any more clearly right now And we're at the end of our time, anyway. So we do want to not do problem 8.3. You have only one problem due on Wednesday, which is 8.1. It will probably take you a while, anyway.

We didn't get to the BCJR algorithm. We'll take this up again and complete the sum product algorithm next time. See you Wednesday.