Codes on Graphs: Behavioral Realizations and the Sum-Product Algorithm*

Arun 'Nayagam

February 2001

1 Introduction

There are many methods of representing a code C, three of which are listed below.

- Using the Generator Matrix, $\mathbf{G}$: C is all possible linear combinations of the rows of $\mathbf{G}$
- Using the Parity Check Matrix, $\mathbf{H}$ (the generator matrix of the dual code C^D): C consists of all possible n-tuples $\mathbf{v}$ satisfying $\langle \mathbf{v}, \mathbf{H}^T \rangle = \mathbf{0}$
- Trellis (state space) representation: C consist of all distinct paths through the trellis

The above representations are special cases of a general class of representations called **Behavioral Realizations** (Originated from the behavioral approach to system theory). In this tutorial-level survey, an attempt is made to introduce a technique of modelling codes from a *behavioral* perspective. A method of graphically viewing this behavior is also described. Finally a technique to decode codes based on their behavior is given. The paper is organized as follows: Section 2 introduces the concept of behavioral modelling. Section 3 describes factor and TWL graphs, which provide a graph-based realization of the behavior of a code. Section 4 introduces the sum-product algorithm which is the decoding basic algorithm for codes on graphs. Section 5 illustrates 2 applications/instances of the sum-product algorithm : the BCJR MAP algorithm for convolutional codes and turbo decoding. The paper is concluded in section 6.

*This paper does not represent any original work by the author. This is just a compilation of material available in the literature. See section 6 for more details

2 Elementary behavioral realizations of linear block codes

A behavioral realization $\boldsymbol{B}$ defines a code C by a set of constraints that the code symbols and other auxiliary state variables must satisfy. For linear block codes, it is sufficient to consider linear behavioral realizations where the constraints are linear.

- A simple example : Variables are elements over GF(q) and the constraints are linear equations over these variables
- In Coding Theory: Variables are vector spaces over GF(q) and constraints are expressed in terms of linear codes.

There are three components to an elementary linear behavioral realization.

1. n code words,$\mathbf{x}$
2. s state variables $\mathbf{s}$,also called hidden or state variables.(s need not have any dependence on n).
3. e linear homogeneous equations over GF(q) involving code words and state variables.

The full behavior $\boldsymbol{B}$ generated by the above realization is the set of all $\mathbf{x}$ and $\mathbf{s}$ that satisfy *all* the constraint equations. The code C generated by the behavior $\boldsymbol{B}$ is the set of all n-tuples $\mathbf{x}$,such that for any $\mathbf{x} \in C$, there exist an $\mathbf{s}\ni (\mathbf{x}, \mathbf{s}) \in \boldsymbol{B}$.
In general the e homogeneous constraint equations in matrix can be represented in matrix notation as,

$$\mathbf{x}_{1\times n}\mathbf{A}_{n\times e} + \mathbf{s}_{1\times s}\mathbf{B}_{s\times e} = \mathbf{0} \tag{1}$$

Some examples:
1. A code defined by a generator matrix $\mathbf{G}_{k\times n}$. Given a message sequence $\mathbf{u}$, the code word $\mathbf{v}$ is obtained as

$$\mathbf{v} = \mathbf{u}.\mathbf{G} \tag{2}$$

or

$$\mathbf{v} - \mathbf{u}.\mathbf{G} = \mathbf{0} \tag{3}$$

Comparing (3) with (1) it is seen that the linear code C has an elementary linear behavioral realization with the state k-tuple $\mathbf{u}$ and n constraint equations defined by (3).
2. A code defined by the parity check matrix $\mathbf{H}_{n\times n-k}$. All the code words $\mathbf{v} \in C$ satisfy

$$\mathbf{v}.\mathbf{H}^T = 0 \tag{4}$$

Hence, the code C defined by $\mathbf{H}$ has an elementary linear realization with no state variables.

The characteristic function for a behavior $\boldsymbol{B}$ is defined as

$$\psi_C(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } (x_1, \ldots, x_n) \in \mathbf{B} \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

A probabilistic interpretation can be given to the characteristic function by noting that ψ_B is proportional to a probability mass function that is uniform over all valid codewords. The variables in **general linear behavioral realizations** are not just elements of GF(q) but are vector spaces over GF(q). In other words, the symbol and state variables are m-tuples over GF(q). And also, the constraint equations are replaced by constraints that a certain subset of variables must lie in a linear block code over GF(q). A trellis representation of a code is very good example of a general linear behavioral realization. At any time the 3-tuple (S_i, x_i, S_{i+1})fully characterizes all possible paths through the trellis at that time instant. It is also easy to show that the 3-tuple is a linear code.

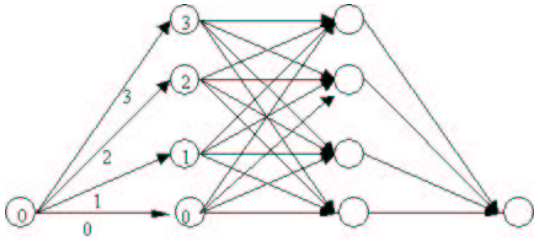

Figure 1: Trellis representation of a linear block code

For the trellis shown in figure 1, at time T=2, the trellis is fully characterized by the following 3-tuples, T2={(0,0,0)(0,1,1)(0,2,2)(0,3,3)}. Both the input and state variables are in GF(4). It can be seen the 3-tuples governing the local behavior of the second trellis section form a linear code i.e., any 3 -tuple in T2 can be expressed as a linear combination of the other 3-tuples.

3 Graphs of linear behavioral realizations

A graph representing an elementary linear behavioral realization is called a Tanner graph or a factor graph.Strictly speaking, a factor graph does not represent a code but represents a code's characteristic function. For the rest of this survey, the factor graph will be referred to as representing the code itself. A factor graph has two types of vertices: variable vertices (symbol and state/hidden variables) and constraint vertices. An edge is drawn between a constraint vertex and a variable vertex iff the variable is involved in the corresponding constraint equation. The graph so formed is bipartite because the variable vertices only

map to constrain vertices and vice-versa.A sample factor graph is shown in figure 2. The filled circles indicate state variables and the empty circles indicate symbol variables.

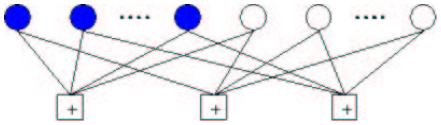

Figure 2: A Tanner/Factor graph of a generic elementary linear behavioral realization.

Degree of a variable vertex is the number of constraint equations(aka checks in coding theory) it is involved in. Degree of a constraint vertex is the number of variables that it involves. The degree of any vertex can be determined by counting the number of edges incident on it.

Figure 3 shows the factor graph for (7,4) Hamming code with generator matrix,

$$\mathbf{G} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} \tag{6}$$

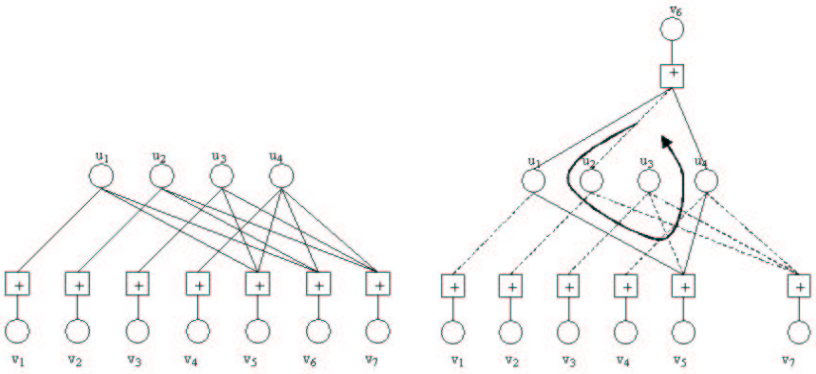

Figure 3: a.Factor graph for the (7,4) dual Hamming code. b.The factor graph rearranged to show the loop formed by u_1, v_5, u_4, v_6

It is seen that the corresponding factor graph has cycles. Figure (b) explicitly shows one loop by rearranging one constraint vertex. Cycles or loops in factor graphs are formed when the same set of two or more variables participate in more than one constraint equation (check). The cycle shown in figure(b) is formed because symbol variables u_1 and u_4 participate in two parity checks(the constraint equations calculating the values of v_5 and v_6). Cycles in factor graphs

can lead to problems when we apply the belief propagation algorithm over the graph. This is described in the next section.

The (7,4)dual-Hamming code can also be represented by a trellis.. Trellises are just Markov models for codes. Since the the behavior of the trellis at any instant is governed by the current state, input and next state, it can be represented by a factor graph as shown in figure 4. The hidden variables are represented

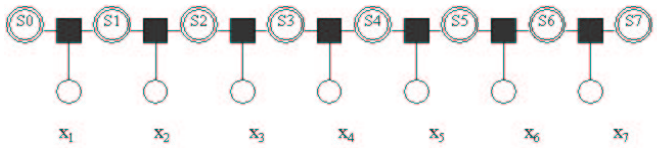

Figure 4: The TWL graph for the trellis in figure 1

by double circles. A factor graph with hidden variables, that represents a general behavioral realization is also known as a TWL (Tanner/Wiberg/Loeliger) graph. We can see from figure 4 that the TWL graph has no loops. By taking into account the trellis structure of the Hamming code, we introduced state variables into the behavioral realization. This helped to get rid of the cycles in the factor graph.

Comment: Often a behavioral realization, which is nothing but a set theoretic description of a system is simplified by introducing hidden (also referred to as auxiliary, latent or state) variables.

Since, every code has a trellis representation, every code can be represented by a cycle-free factor graph. But as the block size increases the state space becomes very big to be of any practical use.

4 The Sum-Product (Belief Propagation) algorithm

The sum-product(SP) algorithm is the basic decoding algorithm for codes on graphs. Instances and variants of the SP algorithm exist in a number of fields like communications, controls and statistical inference.

1. Instances: APP decoding, the BCJR decoding algorithm for convolutional codes, Turbo decoding, Kalman filter, belief propagation.

2. Variants: MLSD, Viterbi Algorithm, belief revision.

Looking back at the definition of factor graphs in section 2 and by looking at figure 2, factor graphs can also be thought of as a bipartite graph that expresses the structure of a factorization, i.e., how a function of several variables factors into several functions of several smaller variables.Consider the parity

check matrix of the (7,4) dual Hamming code,

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix} \tag{7}$$

The parity check matrix verifies if a given sequence is a valid codeword by performing the check given in 4. Figure 5 gives the factor graph for the behavioral realization of the (7,4) dual Hamming code as defined by the parity check matrix.

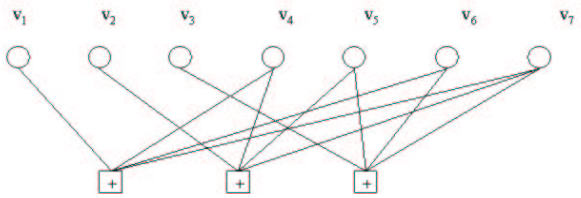

Figure 5: The factor graph of the (7,4) dual Hamming code as defined by the parity check matrix

The box plus in figure 5 represents the sum to zero operation, i.e, the nodes participating in each constraint equation should sum to zero (over GF(2)). The characteristic function for the behavioral relationship governed by $\mathbf{H}$ can be written as

$$\psi_B(v_1, v_2, v_3, v_4, v_5, v_6, v_7) = (v_1 \oplus v_4 \oplus v_6 \oplus v_7 = 0)(v_2 \oplus v_4 \oplus v_5 \oplus v_7 = 0)(v_3 \oplus v_5 \oplus v_6 \oplus x_7 = 0) \tag{8}$$

This factorization is captured by the factor graph in figure 5.Hence, if all the checks are satisfied, the characteristic function ψ_B takes on the value 1. Even if one check fails, ψ_B takes the value 0 implying $[v_1, \ldots, v_7] \notin$ code C with parity check matrix $\mathbf{H}$. A probabilistic interpretation can also be given to (8) by rewriting it as

$$Prob(v_1, v_2, v_3, v_4, v_5, v_6, v_7) = Prob(v_1 \oplus v_4 \oplus v_6 \oplus v_7 = 0) Prob(v_2 \oplus v_4 \oplus v_5 \oplus v_7 = 0) Prob(v_3 \oplus v_5 \oplus v_6 \oplus x_7 = 0) \tag{9}$$

The probability on the left in (9) is the probability that the sequence $[v_1, v_2, \ldots, v_7]$is a valid codeword. The parity check works over a smaller set of elements(4-tuples) to verify if the given 7-tuple is a codeword.

The SP algorithm is used to compute marginal functions associated with a function of several variables. The SP algorithm is finite and exact for graphs without loops. For graphs with cycles, the algorithm becomes iterative and approximate. To illustrate the operation of the SP algorithm a graph without loops will be considered. The application of SP algorithm to graphs with cycles will be discussed later.

The factor graph that will be used to illustrate the SP algorithm is shown in figure 6. The factor graph represents the factorization,

$$g(x_1, x_2, x_3, x_4, x_5) = f_A(x_1) f_B(x_2) f_C(x_1, x_2, x_3) f_D(x_3, x_4) f_E(x_3, x_5) \tag{10}$$

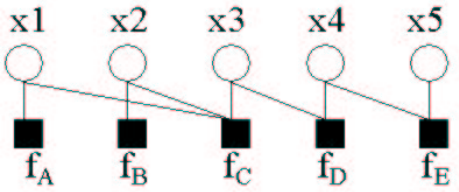

Figure 6: The factor graph of representing the factorization in (10)

The filled squares in figure 6 are called factor nodes. Factor nodes represent a function of the variables that are incident on them. The constraint vertices that were used in the previous sections are examples factor nodes where the function was linear in nature.

The following steps should be followed to compute a single marginal function:

1. Redraw the factor graph with the corresponding node as the root.(x_1 is chosen for illustration, i.e., we want to calculate $g_1(x_1)$). This is shown in figure 7.

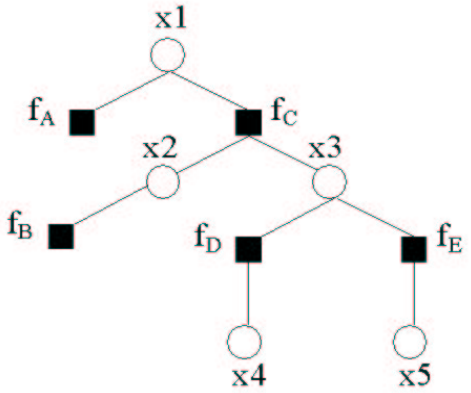

Figure 7: The factor graph in figure 6 redrawn with x1 as the root

2. Begin at the leaves of the factor graph.
3. Each leaf variable node sends a trivial identity function to its parent.
4. Each leaf factor node f sends a description of 'f' to its parent.
5. Each vertex waits for messages from all its children before computing the message to be sent to its parent. i.e., a variable node sends the product of the messages received from its children to its parents, while a factor node f with parent x forms the product of the messages received from its children and operates on the result with a $\sum_{\sim x}$ operator.

6. The algorithm terminates at node x_i where $g(x_i)$ is obtained as the product of the messages received from all its children.

Since we are computing one single marginal probability, this algorithm is also called the single-i SP algorithm. The $\sum_{\sim x}$ denotes a sum w.r.t to all variables except x and is called the summary operator .Hence, a factor node waits for messages from all its children and sums the function with respect to all the variables except its parent. Applying the single-i to the factor graph shown in figure 7:

- First nodes x_4 and x_5 send identity messages to the factor nodes f_D and f_E.
- Then the factor nodes operate on the result with a $\sum_{\sim x_3}$ operator or equivalently f_D applies the $\sum_{x_4}$ operator and f_E applies the $\sum_{x_5}$ operator.
- Node x_3 forms the product of the messages received from its children f_D and f_E and sends it to its parent f_C.
- Proceeding as described in the previous steps, f_C forms the product of the messages received from its children and operates on them with the $\sum_{\sim x_1}$ operator (which is equivalent to the $\sum_{x_2,x_3}$ operator) and sends them to x_1.
- The marginal function of x_1 is then obtained as the product of all the messages received from its children f_A and f_C as

$$g_1(x_1) = f_A(x_1) \sum_{x_2,x_3} f_B(x_2) f_C(x_1, x_2, x_3) \sum_{x_4} f_D(x_3, x_4) \sum_{x_5} f_E(x_3, x_5) \tag{11}$$

From (10), the marginal function of x_1 can be written as

$$g_1(x_1) = f_A(x_1) \sum_{x_2} f_B(x_2) \sum_{x_3} f_C(x_1, x_2, x_3) \sum_{x_4} f_D(x_3, x_4) \sum_{x_5} f_E(x_3, x_5) \tag{12}$$

It is observed that the marginal function computed using the SP algorithm (11) and the marginal function calculated directly (12) are equivalent. The single-i SP algorithm can be applied by considering each variable vertex as a root to compute all the marginal probabilities.

Comment: The SP algorithm computes exact marginal probabilities when it is applied on a cycle-free factor graph.

Computing several marginal functions by repeatedly applying the single-i SP algorithm more than once is inefficient because many of the subcomputations will be repeated over and over. An efficient approach to calculate all the marginal functions would be to not have a fixed parent-child relationship among the vertices of the factor graph. No particular vertex is considered as the root.

Instead each neighbor of a node is considered a parent at some point of time and as a child at another. The algorithm is initiated at the leaves of the factor graph. Once a node receives messages from all its neighbors except one (say i), it regards i as its parent and computes the message to be sent to i. Once a message has been sent to i, the node being considered waits for a message to return from i. Once it gets the message from i, it can compute messages to be sent to its other neighbors, each considered in turn to be a parent. The algorithm stops when two messages have been passed in opposite directions on each edge. In other words the algorithm stops when there are no messages pending. Equations (13) and (14) specify the messages that are passed around in the SP algorithm.

$$\mu_{x\to f}(x) = \prod_{y\in\aleph(x)\setminus f} \mu_{y\to x}(x) \tag{13}$$

$$\mu_{f\to x}(x) = \sum_{\sim x} f(\aleph(f)) \prod_{z\in\aleph(f)\setminus x} \mu_{z\to f}(z) \tag{14}$$

where,
$\mu_{x\to f}$ is the message passed from variable node x to the factor node f.
$\mu_{f\to x}$ is the message passed from the factor node f to the variable node x.
$\aleph(x)$ denotes the set of all neighbors of node x.
$\aleph(x) \setminus f$ denotes the set of all neighbors of x excluding f.
Note that a message flowing on an edge is always a function of the variable vertex associated with the edge irrespective of the direction of flow of the message.

5 Applications to coding theory

In section 2 and 3 factor graphs were shown to capture behavioral realizations and the structure of the factorization of a function of several variables. In coding theory, the latter is used for probabilistic modelling. Since the independence of random variables is dependent on how the joint probability mass function factors out, factor graphs can be used to model this independence. In coding, the problem is determine what the transmitted codeword ($\mathbf{x}$) was given the corrupted received codeword($\mathbf{y}$). The encoding of the codeword can be modelled by a factor graph (behavioral modelling) as shown in section 2. But, it is not possible to observer the output of the encoder directly, instead we only have the corrupted version of the encoder output as the observable quantity. Hence, we just add to the factor graph of the encoder, factor nodes that represent the quantity $Prob\{y_i \mid x_i\}$. This a mixed modelling style using both the behavioral and the probabilistic methods. Figure 8 shows a TWL graph with the probabilistic nodes added. The inputs u_i's are suppressed towards the right end of the trellis to allow trellis termination.

The a posteriori probabilities (APP) associated with the code word components x is by Baye's rule, proportional to $g(x) = p(x)p(y \mid x)$. The function

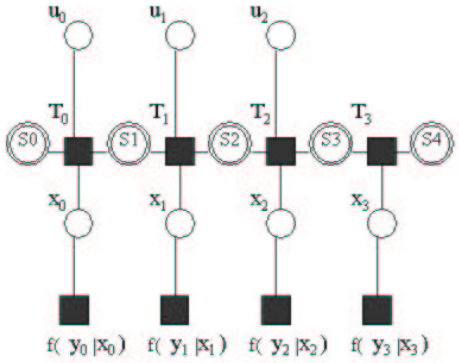

Figure 8: A TWL graph with the probabilistic nodes attached

$g(x)$ is considered as a function of x with parameter y. If all the codewords were uniformly distributed,

$$p(\mathbf{x}) = \frac{\psi_C}{|C|} \tag{15}$$

where ψ_C is the characteristic function of the code C and $|C|$ is the number of codewords in C. It follows that

$$g(x_1, \ldots, x_n) = p(\mathbf{x}) p\big\{(y_1, \ldots, y_n) \mid (x_1, \ldots, x_n))\big\} \tag{16}$$

Assuming that the channel is memoryless, the second term on the RHS in (16) factors out into product of individual conditional probabilities. Using this fact and (15)in (16) ,we get

$$g(x_1, \ldots, x_n) = \frac{\psi_C}{|C|} \prod_{i=1}^{n} p(y_i \mid x_i) \tag{17}$$

Applying (17)to the code defined by the parity check matrix of the (7,4) dual Hamming code in (7), we get

$$g(\mathbf{v}) = (v_1 \oplus v_4 \oplus v_6 \oplus v_7 = 0)(v_2 \oplus v_4 \oplus v_5 \oplus v_7 = 0)(v_3 \oplus v_5 \oplus v_6 \oplus x_7 = 0) \prod_{i=1}^{7} p(y_i \mid v_i) \tag{18}$$

The characteristic function of the (7,4) dual Hamming code as given in (8) is used to get (18). It was shown that the factor graph representation of behaviors represented by the generator or parity-check matrices have loops in them. Since all codes have a trellis representation that lead to a cycle-free TWL graph, the TWL graph would be more suitable for the application of the sum product algorithm. In TWL graphs, the behavior is checked by the Trellis nodes (the black square labelled T_i's). In other words, the constraints defining a valid behavior are local checks performed at the trellis nodes. A trellis nodes $T_i$checks

if the 4-tuple (s_i, u_i, x_i, s_{i+1}) represents a valid behavior (physically, a valid path through the trellis at time i). A behavior is valid only if it passes all the trellis checks. Modifying the characteristic function to represent the trellis constraint, the joint probability mass function of $\mathbf{u}, \mathbf{s}, \mathbf{x}$ and $\mathbf{y}$ can be written as

$$g(\mathbf{u}, \mathbf{s}, \mathbf{x}, \mathbf{y}) = \prod_{i=0}^{n-1} T_i(u_i, s_i, x_i, s_{i+1})(\prod_{i=1}^{7} p(y_i \mid x_i) \tag{19}$$

Since the APPs are proportional to the marginal functions associated with the function $g(\mathbf{u}, \mathbf{s}, \mathbf{x}, \mathbf{y})$ (i.e.,$p(u_i \mid \mathbf{y}) \propto \sum_{\sim x_i} g(\mathbf{u}, \mathbf{s}, \mathbf{x}, \mathbf{y})$), the SP algorithm can be applied over the graph in figure 8 to compute the APPs.

5.1 BCJR algorithm as an instance of the SP algorithm

The SP algorithm when applied to a TWL graph representing a trellis becomes the famous BCJR algorithm. This fact is illustrated using figures in this section. For the sake of illustration a small sample TWL graph shown in figure (9) is considered.

BCJR operation

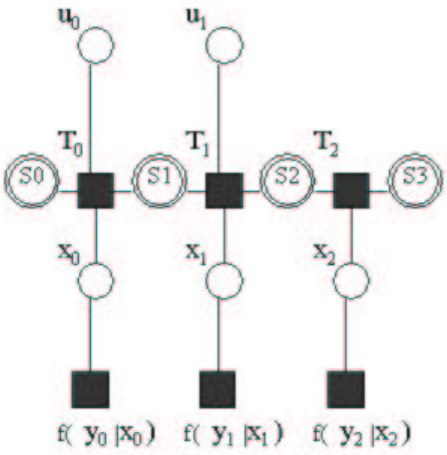

Figure 9: A TWL graph to illustrate the BCJR algorithm

- STEP 1: The leaf nodes send messages to their parents,i.e., the trellis check nodes receive messages from all its neighbors. These messages are indicated by black dots in the vicinity of the trellis checks. At the same time, the channel nodes send the likelihoods to the corresponding variable node. The arrows indicate the direction of message flow.
- STEP 2: Once the leaf nodes have sent their messages, they go into the waiting mode, waiting for a message to come from the trellis check nodes.

(Recall from section 4, the SP algorithm is completed when two messages flow in opposite directions on each edge). The black circles in the vicinity of the trellis check nodes can also be thought of as pending messages that should be sent back to the node they originated from. The variable nodes x_i are of degree 2 and hence they just have one incoming message (the likelihood) and they just pass this to the trellis check node. The message passed from the variable node to the trellis node ($\mu_{x_i \to T_i}(x_i)$) is referred to as $\gamma(x_i)$ in the the literature.

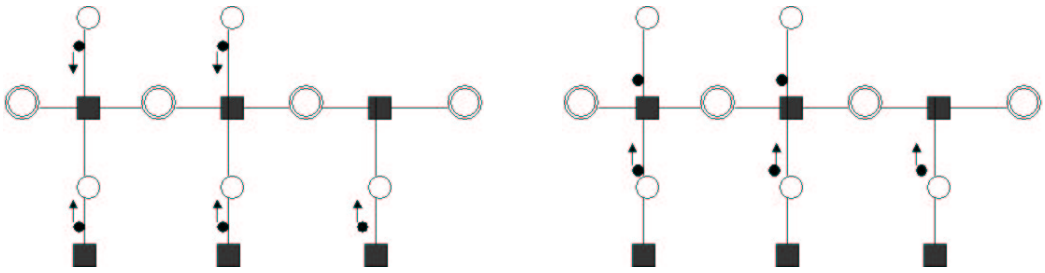

Figure 10: Step 1 and Step 2 of the BCJR algorithm

- STEP 3: Traverse the trellis from left to right. The trellis check node T_i is first considered as the parent and receives the messages are sent by s_i.The trellis nodes then compute the messages to be sent to s_{i+1} as given in (20) . Since s_{i+1} is the parent, the product of the messages received by T_i is summarized with respect s_{i+1}. In literature, the message $\mu_{s_i \to T_i}(s_i)$ is referred to as $\alpha(s_i)$.

$$\alpha(s_{i+1}) = \sum_{\sim s_{i+1}} T_i(u_i, s_i, x_i, s_{i+1})\alpha(s_i)\gamma(x_i) \tag{20}$$

  This flow of messages continues till the last trellis variable T_{n-1} is reached. It does not matter if the information from T_{n-1} is passed to s_n. (Passing or not passing this information on the last edge does not affect the APPs). Hence, the message pending to go the last state variable node s_{n-1} may be ignored.The computation of the α's, illustrated in figure 11,are therefore completed after a message flows from s_{n-1} into T_{n-1}.

- STEP 4: Now the parent-child relationship is reversed i.e., traverse the trellis from right to left. The trellis check T_i becomes the parent and sends a message to the previous state variable s_{i-1}. On receiving the message from T_i, the state variables in turn computes the message to be sent to the previous check T_{i-2}. The message $\mu_{s_i \to T_{i-1}}(s_i)$ is referred to as $\beta(s_i)$. Just like the α's were calculated in a forward recursion (20) ,the β's are computed in a backward recursion in the following manner

$$\beta(s_i) = \sum_{\sim s_i} T_i(u_i, s_i, x_i, s_{i+1})\beta(s_i + 1)\gamma(x_i) \tag{21}$$

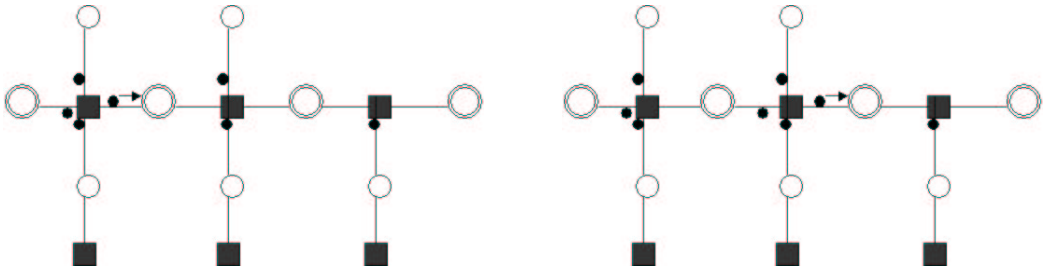

Figure 11: Step 3 of the BCJR algorithm - α calculation

Similar to the computation of the *alpha*'s, the computation of the betas are completed when the messages flowing from right to left reach the first trellis check node, T_0. Again, we can ignore the message pending for s_0 as this does not affect the computation of the APP's. Figure 12 illustrates the computation of the betas.

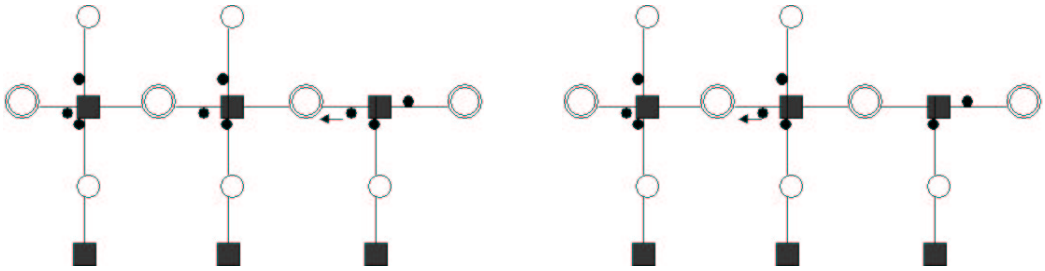

Figure 12: Step 4 of the BCJR algorithm -β calculation

- STEP 5: Once the betas have been calculated, the pending messages that matter are only the messages to be sent to the input variable nodes (the APPs). The message is computed as the product of the messages received at a trellis check node on all the other edges, summarized w.r.t the corresponding input variable node.The expression for computing the APPs is given in (22)and figure 13 illustrates this operation. The message $\mu_{T_i \to u_i}(u_i)$ is referred to as $\delta(u_i)$. These messages are also called extrinsic information in the turbo code literature.

$$\delta(u_i) = \sum_{\sim u_i} T_i(u_i, s_i, x_i, s_{i+1})\alpha(s_i)\beta(s_i+1)\gamma(x_i) \tag{22}$$

After the *delta*'s have been computed, the only messages remaining are the ones that do not affect the calculation of the APPs (sending the messages pending for the channel output nodes serves no purpose.). Hence, for all practical purposes it can be stated that the SP algorithm terminates on a factor graph having

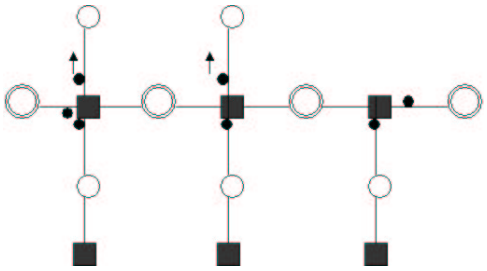

Figure 13: Step 5 of the BCJR algorithm - δ/APP calculation

no cycles when there are no messages pending on the graph. It can be seen that $\delta(u_i)$ is indeed the marginal function associated with $g(\mathbf{u}, \mathbf{s}, \mathbf{x}, \mathbf{y})$ which is proportional to the APP $p(u_i \mid \mathbf{y})$.

The way the algorithm is described above, it takes 3 passes through the graph to compute the APP's. The first pass calculates the α's, also called the forward state metrics based on how they are computed. The second pass calculates the β's, also called the reverse state metrics and the final pass calculates the δ's. This way of calculating the APPs is not efficient. It was noted in section 4 that as soon as messages arrive on all but one of the branches, a message can be computed to be sent on that branch. During the computation of the reverse state metric, once the trellis node T_i gets a message from state $s_i + 1$, it has all the information required to compute $\delta(u_i)$ (the other messages needed to compute $\delta(u_i)$,see (22), are $\alpha(s_i)$,$\beta(s_{i+1})$ and $\gamma(x_i)$, which have all been acquired). Hence, T_i can simultaneously calculate the messages $\beta(s_i)$ and $\delta(u_i)$ and send them on the corresponding edges. This eliminates one pass through the network and speeds up the computation of the APPs. This is shown in figure 14.

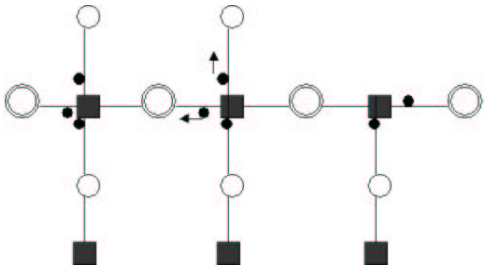

Figure 14: Single pass computation of $\beta(s_i)$ and $\delta(u_i)$

This example illustrates two facts: the SP algorithm computes exact marginal functions when the factor graph is cycle free and that the BCJR decoding method is indeed an instance of the SP algorithm.

5.2 Turbo decoding as an instance of the SP algorithm

A turbo encoder consists of two recursive systematic convolutional encoders. One of the constituent encoders receives the input directly while the other encoder receives a permuted version of the input. Since the convolutional codes have a natural trellis representation they can be represented using a TWL graph. The two encoders are identical and hence a factor graph for a typical turbo code would look like the one shown in figure 15. The same code used in the previous section is used. The only difference is that an input is also associated with the trellis termination stage. In figure 15, the sequence **x1** is a permuted version of the input sequence **x**.

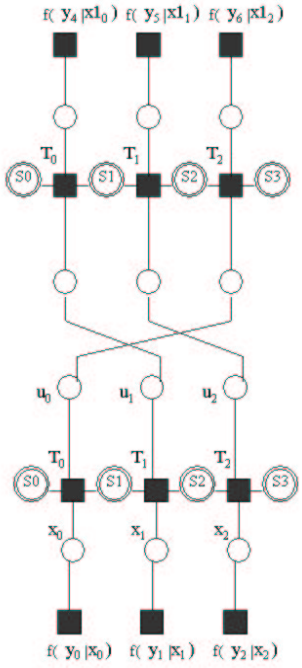

Figure 15: TWL graph of a rate $\frac{1}{3}^{rd}$Turbo code

We observer that the concatenation of two encoders creates loops in the factor graph. Thus, the SP algorithm becomes iterative and approximate. The BCJR algorithm is applied to one of the decoders first. For the sake of illustration the TWL graph at the bottom in figure 15 is taken as the first encoder. The BCJR algorithm proceeds as explained in the previous section. Since the only function of the probability nodes is to provide the likelihood value, these nodes are not drawn in some of the figures in this section. After the initializations and the channel likelihoods are sent to the variable nodes, the pending messages (only the messages that are of direct consequence are shown, the ones that can be ignored are not shown) are shown in figure 16. Similarly the hidden variable

nodes x_i's have no role to play except to forward the likelihoods to the trellis check nodes. So the hidden nodes are also suppressed in most of the figures in this section. Figures 17 and 18 shows the calculation of the forward state met-

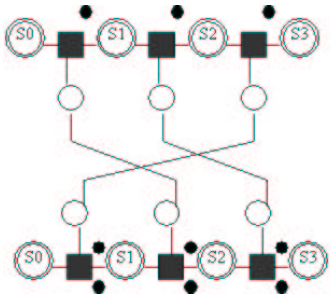

Figure 16: Pending messages at the beginning of turbo decoding

rics, the reverse state metrics and the likelihoods as described in the previous section.

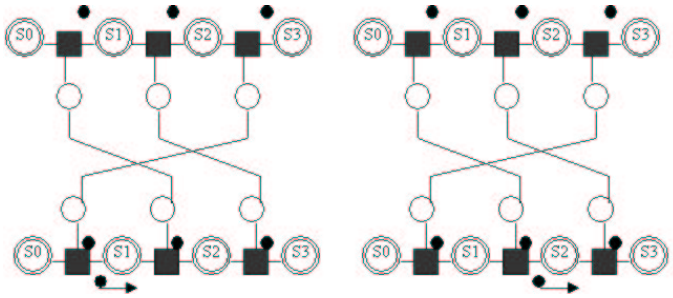

Figure 17: Forward state metric computation in encoder 1

Once the all the extrinsic information reach the second encoder the same process is repeated. Note that at this point, the edges connecting the trellis check nodes and the input variable nodes of encoder 1 have passed messages in opposite directions and so according to encoder 1, the SP algorithm has terminated. But after passing through the second encoder, the extrinsic from the second encoder again pass on the edge from the input variable node to the trellis check node on encoder one. This causes the input variable nodes of encoder 1 to consider this as a new run of the SP algorithm and it again goes into the waiting phase (waiting for a message to return in the opposite direction). Thus, the SP algorithm does not self-terminate as it did when applied on a graph with no loops because at any instant there are messages pending (nodes in the waiting phase) in a graph with cycles. This illustration shows us that for a factor graph with loops, the SP algorithm has to be iterated and that turbo decoding is indeed an instance of the SP algorithm (when applied to a graph with loops).

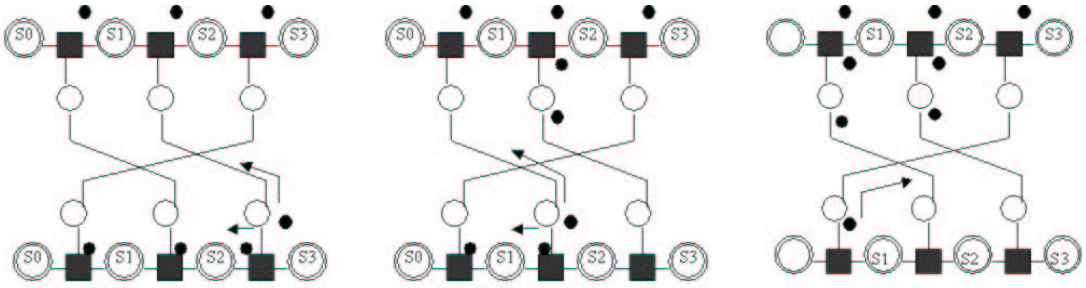

Figure 18: Reverse state metric and extrinsic computation in encoder 1

6 Conclusions, comments and references

This paper gives a tutorial level overview of modelling codes on graphs based on their behavior. Two different graphical methods of describing valid code behavior was introduced:the factor and the TWL graph. The sum product algorithm was described as the basic decoding algorithm for codes on graphs and two instances of the operation of the sum-product algorithm were studied.

This paper just scratches the surface of an area that is vast and interesting. There is a lot of material in the literature that can be baffling for a novice in this area. This paper aims to demystify and illustrate some of the concepts surrounding codes on graphs. The tutorial does not provide a complete coverage of this topic and is intended to motivate further study in this area. The best place to start looking for material on this topic is

[1] "IEEE Transactions on Information Theory-Special issue on codes on graphs and iterative algorithms, Vol. 47,Issue 2, February 2001." This issue contains a number of excellent papers on this topic.

The material covered in section 2 of this tutorial was adapted from G.D.Forney's graduate course handouts.

[2] "Lecture [5]: Codes on Graphs, 6.451 Principles of digital communications II- M.I.T,Spring 2001",G.D.Forney.

Most of the material in sections 3 and 4 (including the illustrative example of the operation of the single-i SP algorithm) borrowed heavily from the classic paper,

[3] "Factor graphs and the sum-product algorithm", F.R.Kschischang, B.J.Frey and H.A.Loeliger, IEEE Transactions on Information Theory-Special issue on codes on graphs and iterative algorithms, Vol. 47,Issue 2, February 2001.

This is a very good tutorial paper with lots of illustrations of instances of the SP algorithm. The turbo code illustration in section 5 was adapted from

[4] "Iterative decoding of compound codes by probability propagation in graphical models",F.R.Kschischang and B.J.Frey, IEEE journal on selected areas in communications, vol.16, No.1, JAN. 1998.