Simplified Implementation of the MAP Decoder

Final Project
ECE 259B

Shouvik Ganguly
Electrical and Computer Engineering
University of California, San Diego
La Jolla, California 92092
shgangul@eng.ucsd.edu

***Abstract*—In this report, I have studied a simplification, suggested by Viterbi [1], to the BCJR algorithm for the bitwise MAP decoder for convolutional and turbo codes. The simplifications involve reduction in both computational complexity and memory requirements, using, first, a good approximation to the metric update rules for BCJR algorithm and, second, a scheduling algorithm to reduce memory requirements.**

I. Introduction : BCJR Algorithm

We recall that the bitwise MAP decoder computes an estimate of the k^{th} input bit u_k in terms of the received vector $\underline{R}_1^N$, according to the relation

$$\widehat{u}_k = \arg \max_{i \in \{0,1\}} \Pr[u_k = i | \underline{R}_1^N]. \tag{1}$$

We define the 'log aposteriori probability ratio' (LAPPR) as

$$\Lambda_k = \log \frac{\Pr[u_k = 1 | \underline{R}_1^N]}{\Pr[u_k = 0 | \underline{R}_1^N]} = \log \frac{\Pr[u_k = 1, \underline{R}_1^N]}{\Pr[u_k = 0, \underline{R}_1^N]}. \tag{2}$$

In terms of the Λ_k's, the MAP rule (1) becomes

$$\widehat{u}_k = \begin{cases} 1, & \Lambda_k \geq 0 \\ 0, & \text{otherwise.} \end{cases} \tag{3}$$

Let $\mathcal{S}$ be the set of states of a trellis for the encoder, and let S_k be the random variable representing the state at time k. The LAPPR can be written as (Berrou et. al., [3])

$$\Lambda_k = \log \frac{\sum_{m' \in \mathcal{S}} \sum_{m \in \mathcal{S}} \alpha_{k-1}(m') \gamma_k^1(m', m) \beta_k(m)}{\sum_{m' \in \mathcal{S}} \sum_{m \in \mathcal{S}} \alpha_{k-1}(m') \gamma_k^0(m', m) \beta_k(m)}, \tag{4}$$

where

$$\begin{aligned} \alpha_k(m) &= \Pr[S_k = m, \underline{R}_1^k] \\ \beta_k(m) &= \Pr[\underline{R}_{k+1}^N | S_k = m] \\ \gamma_k^i(m', m) &= \Pr[u_k = i, S_k = m, \underline{R}_k | S_{k-1} = m'] \end{aligned} \tag{5}$$

The α_k's and β_k's can be computed through 'forward' and 'backward' recursions as

$$\begin{aligned} \alpha_k(m) &= \sum_{m' \in \mathcal{S}} \sum_{i=0}^{1} \alpha_{k-1}(m') \gamma_k^i(m', m) \\ \beta_k(m) &= \sum_{m' \in \mathcal{S}} \sum_{i=0}^{1} \beta_{k+1}(m') \gamma_{k+1}^i(m, m') \end{aligned} \tag{6}$$

If the initial state of the encoder is constrained to be s_0 and the final state constrained to be s_N, then the forward and backward state metrics can be initialized as

$$\begin{aligned} \alpha_0(m) &= \begin{cases} 1, & m = s_0, \\ 0, & \text{otherwise.} \end{cases} \\ \beta_N(m) &= \begin{cases} 1, & m = s_N, \\ 0, & \text{otherwise.} \end{cases} \end{aligned} \tag{7}$$

The 'branch transition probabilities' $\gamma_k^i(m', m)$ can be written as
$\gamma_k^i(m', m) = \Pr[u_k = i] \Pr[S_k = m \Big| S_{k-1} = m', u_k = i] \Pr[\underline{R}_k \Big| S_{k-1} = m', u_k = i, S_k = m]$.

II. Computation and Memory Requirements

From the expression (4) for the LAPPR, we see that we need $\mathcal{O}(|\mathcal{S}|^2)$ multiplications and additions for computing the LAPPR for each k. Also, from the forward and backward recursions (6), we see that we need $\mathcal{O}(|\mathcal{S}|^2)$ multiplications and additions for

computing the forward and backward state metrics for each k.

As far as storage requirements are concerned, the forward state metrics have to be stored for every k from 1 through N, before we start computing the backward state metrics starting from N and can use them to compute the LAPPR. The metric values are typically floating point numbers and need a lot of storage space for accurate results.

Both the computational complexity and the memory requirements become challenging as the block length N grows large, and much more quickly as $|\mathcal{S}|$ grows large, i.e. the code memory becomes larger.

Finally, the metric values become progressively smaller as we progress with the recursions (6), so in order to keep them stable, some intelligent normalization is required at each step. This also adds to the computation requirements.

III. A New 'Maximum' Function

Let us define the function

$$\max{}^*(x, y) \triangleq \log(\mathrm{e}^x + \mathrm{e}^y). \tag{8}$$

It is easy to see that the $\max^*$ function can also be written as

$$\max{}^*(x, y) = \max(x, y) + \log(1 + \mathrm{e}^{-|y-x|}). \tag{9}$$

We get a key insight from (9), that $\max^*(x, y)$ can be approximated very well by $\max(x, y)$ as long as x and y differ by about 5%. We can generalize the definition of $\max^*$ to include more than two variables, as follows.

$$\max{}^*(x, y, z) \triangleq \max{}^*(\max{}^*(x, y), z), \text{ or} \tag{10}$$

$$\max{}^*(x, y, z) \triangleq \log(\mathrm{e}^x + \mathrm{e}^y + \mathrm{e}^z), \tag{11}$$

and so on, for any finite set. We also see that as long as the members of the set of variables are all different, we can approximate $\max^*$ by $\max$ quite well.

IV. Simplification of Computation

Let us define

$$\begin{aligned} \widetilde{a}_k(m) &\triangleq \log \alpha_k(m). \\ \widetilde{b}_k(m) &\triangleq \log \beta_k(m). \\ \widetilde{c}_{i,k}(m', m) &\triangleq \log \gamma_k^i(m', m). \end{aligned} \tag{12}$$

Then the LAPPR can be written as

$$\Lambda_k = \max_{m,m' \in S}{}^*[\widetilde{a}_{k-1}(m') + \widetilde{c}_{1,k}(m', m) + \widetilde{b}_k(m)] - \max_{m,m' \in S}{}^*[\widetilde{a}_{k-1}(m') + \widetilde{c}_{0,k}(m', m) + \widetilde{b}_k(m)]. \tag{13}$$

The forward and backward recursions become

$$\begin{aligned} \widetilde{a}_k(m) &= \max{}^*{}_{m' \in S, i \in \{0,1\}}[\widetilde{a}_{k-1}(m') + \widetilde{c}_{i,k}(m', m)]. \\ \widetilde{b}_j(m) &= \max{}^*{}_{m' \in S, i \in \{0,1\}}[\widetilde{b}_{j+1}(m') + \widetilde{c}_{i,j+1}(m', m)]. \end{aligned} \tag{14}$$

If the initial state of the encoder is constrained to be s_0 and the final state constrained to be s_N, then these new forward and backward state metrics can be initialized, as before, as

$$\begin{aligned} \widetilde{a}_0(m) &= \begin{cases} 0, & m = s_0 \\ -\infty, & \text{otherwise.} \end{cases} \\ \widetilde{b}_N(m) &= \begin{cases} 0, & m = s_N \\ -\infty, & \text{otherwise.} \end{cases} \end{aligned} \tag{15}$$

We also have

$\widetilde{c}_{i,k}(m', m) = \log \Pr[u_k = i] +$
$\log \Pr[S_k = m | S_{k-1} = m', u_k = i] +$
$\log \Pr[\underline{R}_k | S_{k-1} = m', u_k = i, S_k = m]$.

Approximating $\max^*$ by $\max$ in (13) and (14), we see that the computations are considerably simplified. Moreover, the forward and backward metrics can now be computed using the Viterbi Algorithm. Also, since we are dealing with logarithms of the previous state metrics, the requirement of normalizing the metrics at every step vanishes, further reducing computation.

The only problem left now is that of storing all the forward metric values for every stage k, leading to potentially huge memory requirements. Moreover, the decoded bits can be computed only at the end of a long data block and leads to a large delay, which is unacceptable in some situations for convolutional codes.

V. Reduction of Memory Requirements and Delay

A crucial observation towards reducing the memory requirements for the BCJR algorithm is the fact that the behaviour of the Viterbi Algorithm is nearly indpendent of the initial conditions beyond a few constraint (memory) lengths of the code. Let us denote this 'learning period' of the algorithm by L. Let us also assume that the received symbols are delayed by $2L$ branch times. The key idea here is to use 2 backward Viterbi processors in tandem, so that both delay and memory requirements can be substantially reduced. Calling the processor that implements (13) as the 'dual-maxima' processor, the overall decoding proceeds as follows.

- The forward Viterbi processor starts at branch 0 at time $2L$ and moves forward, storing every forward state metric at each branch time.
- The first backward Viterbi processor starts at branch $2L$ at time $2L$ and moves backward, storing only the most recent state metrics at each branch time till branch L.
- At time $3L$, the first backward Viterbi processor meets the computed forward metrics at branch L, and the dual-maxima processor starts outputting the soft decisions for the first L branches.
- From time $3L$ to time $4L$, the first backward Viterbi processor moves till branch 0, and the dual-maxima processor outputs soft decisions for the first L branches.
- Meanwhile, at time $3L$, the second backward Viterbi processor starts at branch $3L$ and moves backward, storing only the most recent state metrics at each branch time till branch $2L$.
- At time $4L$, the second backward Viterbi processor meets the computed forward metrics at branch $2L$, and the dual-maxima processor starts outputting the soft decisions for the $2L^{th}$ through the L^{th} branches.
- From time $4L$ to time $5L$, the second backward Viterbi processor moves till branch L, and the dual-maxima processor outputs soft decisions for the $2L^{th}$ through the L^{th} branches.
- The two backward Viterbi processors hop forward $4L$ branches, every time $2L$ sets of backward state metrics have been generated.

We see that in the above process, the two backward Viterbi processors use the dual-maxima processor at exactly complementary times, so time-sharing of the dual-maxima processor is possible. Moreover, the forward processor only needs to store the forward metrics for $2L$ branch times at a time, so this greatly reduces memory requirements for large block lengths. Also, the soft decisions for the first $2L$ branches can be output at time $5L$, so this reduces the delay as well. Finally, we observe that we use one forward Viterbi processor, two backward Viterbi processors and one dual-maxima processor with the same complexity as a Viterbi processor, so the total complexity of the algorithm becomes only 4 times that of a conventional Viterbi decoder for the same convolutional code. Thus if we use this decoder for Turbo decoding with I iterations, the overall complexity is $8I$ times the complexity of Viterbi decoding for each of the consituent convolutional codes. This is not a huge price to pay, since Turbo codes typically produce better performance than even much more complex convolutional codes.

Figure 1 explains the steps diagrammatically.

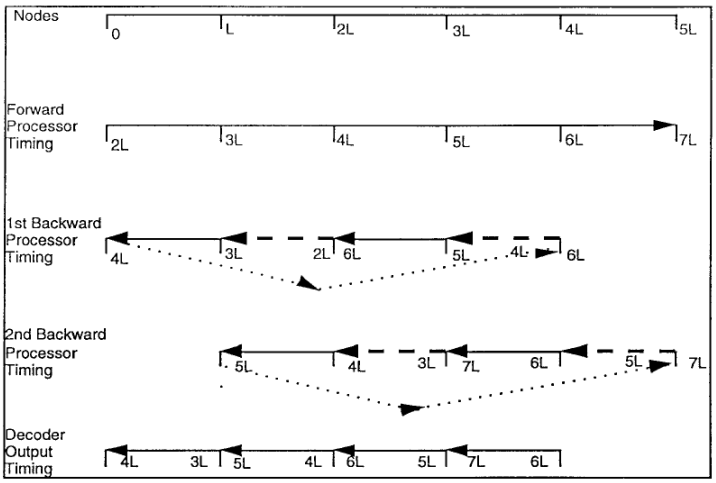

Fig. 1: Schematic Representation of Viterbi's Scheduling [1]

VI. Simulation Results

Using the approximations and the scheduling algorithm for decoding a rate $1/3$ Turbo code with the constituent RSC codes given by the generator matrix $G(D) = \left[1, \frac{1+D+D^2}{1+D^2}\right]$, we get the bit-error rate vs E_b/N_0 plot in Figure 2, which clearly shows that the good performance persists under this setting.

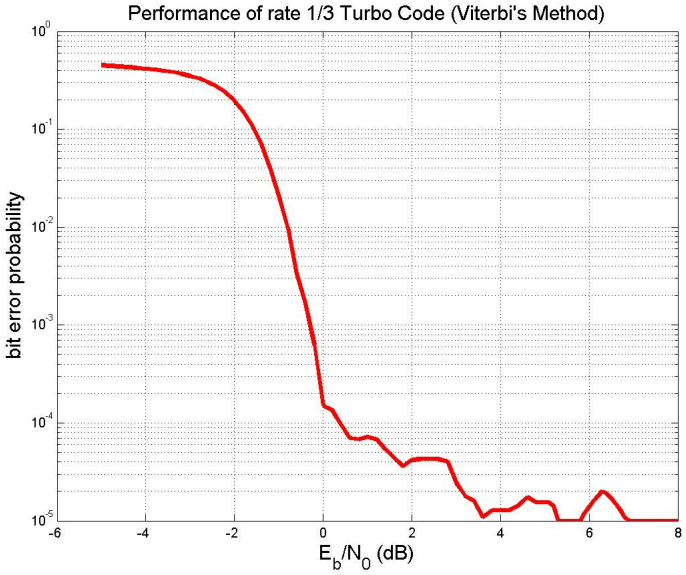

Fig. 2: Error Performance using the Scheduling Algorithm

REFERENCES

[1] A. Viterbi. *An Intuitive Justification and a simplified implementation of the MAP decoder for Convolutional Codes*. IEEE Journal on Selected Areas in Communications, 16(2), pp. 261–264, Feb 1998.

[2] L.R. Bahl, J. Cocke, F. Jelinek and J. Raviv. *Optimal Decoding of Linear Codes for Minimizing Symbol Error Rate*. IEEE Transactions on Information Theory, 20(2), pp. 284–287, 1974.

[3] C. Berrou, A. Glavieux and P. Thitimajshima. *Near Shannon limit error-correcting coding and decoding : Turbo-codes . Proc. IEEE International Conference on Communications*, 2, pp. 1064–1070, May 1993.