# A Fast Maximum-Likelihood Decoder for Convolutional Codes

Jon Feldman Ibrahim Abou-Faycal Matteo Frigo

***Abstract*—The *lazy Viterbi decoder* is a maximum-likelihood decoder for block and stream convolutional codes. For many codes of practical interest, under reasonable noise conditions, the lazy decoder is much faster than the original Viterbi decoder. For a code of constraint length $6$, the lazy algorithm is about 50% faster than an optimized implementation of the Viterbi decoder whenever SNR $> 6$ dB. Moreover, while the running time of the Viterbi decoder grows exponentially with the constraint length, under reasonable noise conditions, the running time of the lazy algorithm is essentially independent of the constraint length. This paper introduces the lazy Viterbi decoder and shows how to make it efficient in practice.**

## I. Introduction

Maximum-likelihood (ML) decoding of convolutional codes is often implemented by means of the Viterbi algorithm [12], [5], [4]. The main drawback of the Viterbi decoder is execution time: To decode a single binary information symbol, the decoder performs $O(2^k)$ operations, where $k$ is the size of the internal memory of the encoder ($k+1$ is often referred to as the *constraint length* of the code). This exponential dependence on $k$ makes a software implementation of the algorithm inefficient for many codes of interest, such as the one used in the IS-95 CDMA standard for which $k = 8$. To overcome this problem, other decoder structures, namely sequential decoders [3] and $A^*$ search [2], [7], have been investigated in the literature. Under good Signal-to-Noise Ratio (SNR) conditions, sequential decoders are more efficient than the Viterbi algorithm, but, in addition to being suboptimal, they become prohibitively slow at low SNR [3]. The $A^*$ decoder combines the reliability and performance of the Viterbi algorithm while running as efficiently as a sequential decoder when the SNR is high. However, previous descriptions of $A^*$ decoders do not apply to continuous streams of data, and they do not address certain implementation problems that are critical to the practicality of the algorithm. Specifically, under high noise conditions the implementations detailed in the literature lead to a running time asymptotically even worse than Viterbi's.

In this paper, we extend the $A^*$ approach to apply to continuous streams of data, and we solve the implementation problems. Specifically, we introduce the *lazy Viterbi decoder*, which offers (i) maximum-likelihood decoding, (ii) best-case running time much better than the Viterbi algorithm, (iii) worst-case asymptotic running time no worse than the Viterbi algorithm, and (iv) simple data structures that allow for an efficient software implementation. Figure 1 summarizes the asymptotic complexity of the best and worst cases of the three algorithms.

| Algorithm | Best case | Worst case |
|---|---|---|
| Viterbi | $\Theta(2^k)$ | $\Theta(2^k)$ |
| $A^*$ | $\Theta(\log L)$ | $\Theta(2^k \log(L2^k))$ |
| Lazy Viterbi | $\Theta(1)$ | $\Theta(2^k)$ |

**Figure 1.** Asymptotic running time of three decoders in the best and worst cases. In the formulas, $k+1$ is the constraint length of the code, and $L$ is the length of the block.

Jon Feldman, NE43-309, Laboratory for Computer Science, M.I.T., Cambridge, MA, 02139. Email:jonfeld@theory.lcs.mit.edu. This work was done while the author was visiting Vanu Inc.

Ibrahim Abou-Faycal, Vanu Inc., 1 Porter Square, Suite 18, Cambridge, MA 02140. Email: ibrahim@vanu.com.

Matteo Frigo, Vanu Inc. Email: athena@vanu.com.

### A. Background

Maximum-likelihood decoding of convolutional codes is equivalent to the computation of a shortest path on a particular directed graph called a *trellis*. A trellis node is labeled with a pair $(s,t)$, where $s$ represents the state of the encoder at time $t$. An edge $(s,t) \to (s', t+1)$ in the trellis represents the transition of the encoder at time $t$ from state $(s,t)$ to state $(s', t+1)$. Each edge $(s,t) \to (s', t+1)$ in the trellis is labeled with a nonnegative *branch metric* $d$, which measures the likelihood that the encoder moves into state $s'$ at time $t+1$ given that the encoder is in state $s$ at time $t$ and given the received symbol at time $t$. The branch metrics can be defined in such a way that the sum of the branch metrics on a path is a measure of the likelihood of that path.

A trellis contains a distinguished *start node* at time 0. The *accumulated metric* of a node is the distance of the node from the start node. The goal of the decoder is to identify, for each time step $t$, the node at time $t$ with the smallest accumulated metric.

Both the Viterbi and the $A^*$ algorithm maintain an upper bound to the accumulated metric of all nodes. The basic operation is the *expansion* of a node: Once the accumulated metric of a node $u$ is known, the upper bound of all its successors is updated. The Viterbi algorithm expands nodes breadth-first, and it expands the whole trellis no matter what the noise conditions are. The $A^*$ algorithm always greedily expands the node with the lowest accumulated metric.

Figure 2 shows the number of expansions performed by both strategies as a function of the SNR. (See also [7] for a more detailed comparison of the two strategies.) At high SNR, the $A^*$ algorithm performs far fewer expansions than the Viterbi algorithm. However, it is wrong to conclude that $A^*$ is unconditionally better than Viterbi, because in practice, expansion is much cheaper computationally for the Viterbi algorithm than

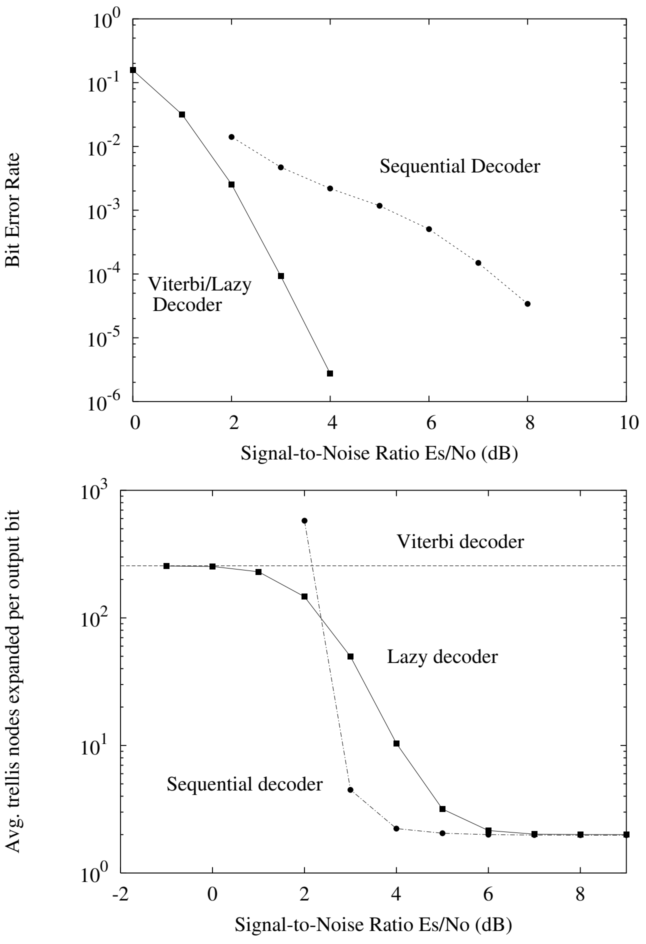

**Figure 2**. **(Top)** *Bit error rate;* **(Bottom)** *Average number of expanded trellis nodes per information symbol. Both are given as a function of the SNR, for the Lazy, Viterbi and Sequential decoders, under AWGN. The code is a rate-$1/2$, constraint length $9$ code used in CDMA, generator polynomials (753,541) (octal). For the sequential decoder, experiments were performed on blocks of 100 encoded information bits, as this decoder does not work for stream codes.*

it is for the $A^*$ algorithm. The Viterbi algorithm expands every node of the trellis, and consequently it does not incur the overhead of keeping track of which nodes to expand. Moreover, for Viterbi the order of expansion is known at compile time, which allows for optimizations such as constant folding of memory addresses, efficient pipelining, and elimination of most conditional branches. In contrast, the $A^*$ algorithm maintains a priority queue of nodes, keyed by accumulated metric. Such a priority queue slows down practical implementations of the $A^*$ algorithm because of two reasons. First, for a trellis with $n$ nodes, insertion, deletion and update in a general priority queue requires $\Theta(\log n)$ operations, which is asymptotically worse than the $\Theta(1)$ time per expansion of Viterbi. Second, a general priority queue using heaps or some kind of pointer-based data structure is not amenable to the compile-time optimizations that apply to Viterbi.

Our goal with the lazy Viterbi decoder is to make the $A^*$ approach useful in practice. By exploiting the structural properties of the trellis, we can perform all priority-queue operations in constant time, thereby eliminating the $\Theta(\log n)$ slowdown. A careful design of the data structures maintained by the lazy Viterbi decoder allows us to implement the whole expansion operation in constant time, and furthermore, as a short sequence of straight-line code, which is important for efficient pipelining on present-day processors.

## B. Outline

In Section II, we present the lazy Viterbi decoder, and give the details necessary to achieve an implementation that is useful in practice. We compare the speed of the lazy Viterbi decoder with other implementations of the Viterbi algorithm in Section III. We conclude in Section IV.

## II. The lazy Viterbi decoder

In this section, we present the lazy Viterbi decoder. We first discuss the operation of the algorithm for block convolutional codes, and then show how the algorithm can be extended to handle stream convolutional codes. Familiarity with the standard Viterbi algorithm, for example in the form presented in [13], is assumed.

The lazy decoder maintains two main data structures, called the trellis and the priority queue. The trellis data structure contains the nodes of the trellis graph whose shortest path from the start node has been computed. Each node $u$ in the trellis data structure holds a pointer $Prev(u)$ to its predecessor on the shortest path. We maintain the invariant that every node in the trellis has been expanded.

The priority queue contains a set of *shadow nodes*. A shadow node $\hat{u}$ is a proposal to extend a path in the trellis data structure by one step to a new node $u$. Each shadow node $\hat{u}$ in the priority queue holds an accumulated metric $acc(\hat{u})$ equal to the length of the proposed path extension, and a pointer $Prev(\hat{u})$ to the predecessor of $u$ on that path. Nodes $\hat{u}$ in the queue are keyed by $acc(\hat{u})$.

We note that $acc(\hat{u})$ is not stored explicitly at $\hat{u}$, but rather is implicitly stored by the data structure, a detail we will cover later. The predecessor $Prev(\hat{u})$ of a shadow node is always a "real" node in the trellis data structure. All nodes in both the priority queue and the trellis also hold their time and state value.

Initially, the trellis is empty and the queue consists of a shadow $\hat{s}$ of the start node $s$ with $acc(\hat{s}) = 0$. After initialization, the algorithm repeatedly extracts a shadow node $\hat{u}$ of minimum metric $m$ from the priority queue. Such a shadow node thus represents the best proposed extension of the trellis. If $u$, the "real" version of $\hat{u}$ with the same time and state, is already in the trellis, then $\hat{u}$ is discarded, since a better proposal for $u$ was already accepted. Otherwise, the algorithm inserts a new node $u$ into the trellis with $Prev(u) = Prev(\hat{u})$, and, for each successor $v$ of $u$, $\hat{v}$ is inserted in the priority queue with metric $acc(\hat{v}) = m + d(u, v)$. This process is repeated until the trellis contains a node at time $T$.

Unlike the $A^*$ algorithm, in our decoder a node can be both in the trellis and as a shadow in the priority queue; in fact, more than one shadow of the same node can be in the priority queue at the same time. This is one of the "lazy" features of the algorithm: instead of demanding that all nodes be uniquely stored in the system, we trade a test for priority-queue membership for a delayed test for trellis membership. This choice is advantageous because the check can be avoided altogether if a shadow node is still in the priority queue when the algorithm terminates. Moreover, trellis membership is easier to test than priority-queue membership, as will be clear after we detail the implementation of both data structures below.

The trellis data structure is a sparse matrix. It is sparse because in practice only a small fraction of the trellis nodes are actually expanded (see Figure 2). It is a matrix because the two indices $s$ and $t$ belong to an interval of integers. Many sparse-matrix representations (including a dense matrix) could be used to represent the trellis. We found it convenient to implement the trellis as a hash table, where the pair $(s, t)$ is the hash key. Using standard techniques [1], trellis lookup and insertion can be implemented in expected constant time. In alternative, the "sparse array trick" [11, Section 2.2.6, Problem 24] could be employed for a deterministic $O(1)$ implementation of the trellis.

## *A. Implementation of the priority queue*

The priority queue supports two main operations: insertion of a node, and extraction of a node of minimum metric. In this section, we give a careful examination of the range of accumulated metric values taken on by shadow nodes in the priority queue. Our insights lead to an implementation that allows both insertion and extraction in constant time.

We begin by making the following assumption: *Branch metrics are integers in the range* $[0..M]$, for some integer $M$ independent of the constraint length. This assumption holds for hard-decision decoders, where the branch metric is the Hamming distance between the received symbol and the symbol that should have been transmitted. For soft-decision decoding, this assumption requires quantization of the branch metrics. It is known [8] that quantization to 8 levels is usually sufficient to achieve most of the coding gains, and therefore this assumption is not restrictive.

This assumption allows us to show the following property:

*Claim 1:* At any time during the execution of the lazy decoder, all metrics in the priority queue are in the range $[m..(m+M)]$, where $m$ is the minimum metric in the queue.

*Proof:* The property trivially holds initially, when the priority queue consists of only one node. Whenever a node $u$ gets inserted into the queue, it is the successor of a node with metric $m$, and so it is inserted with metric $acc(u) \leq m + M$. Since the minimum metric in the queue never decreases, the property always holds. ■

Based on Claim 1, we implement the priority queue as an array $[m..m + M]$ of linked lists of nodes. The metric of a node is not stored in the node, but it is implicitly given by which list the node belongs to. The array can be implemented as a circular buffer of $M + 1$ pointers. Alternatively, one can maintain the invariant that $m = 0$ by periodically adjusting all metrics when the invariant is violated. (This is a simple $O(M)$ operation that only involves a shift of the array.) In either implementation, insertion of a new node and extraction of a minimal-metric node are constant-time operations.

## *B. Computation of the branch metrics*

Let $u = (s, t)$ be a trellis node and let $u' = (s', t + 1)$ be a successor of $u$. In the Viterbi decoder, the branch metric $d(u, u')$ is the conditional log-likelihood of the encoder moving to state $s'$, given a received symbol and given the initial state $s$ of the encoder. The same metric could be employed in the lazy Viterbi decoder, but it turns out that a slight modification of the metric reduces the number of expanded nodes without affecting the performance of the algorithm.

Because all edges span consecutive time steps, adding a constant to all branch metrics at time $t$ does not change which path is shortest. In the lazy Viterbi algorithm, for each time $t$ we add a constant $h(t)$ to the metric on all branches $(s, t) \rightarrow (s', t+1)$. The constant $h(t)$ is chosen such that for all $t$, the minimum branch metric at time $t$ is 0. This metric is equivalent to the one used by Han, Chen, and Wu [7], and can also be seen as a form of "heuristic function" used in $A^*$, similar to the ones used by Ekroot and Dolinar[2], and Han and Hartmann [6].

To see why such an adjustment of metrics may be desirable, consider the case when all branch metrics at a certain time $t$ are high, possibly because of some noise spike. Without adjustment, the decoder would expand most of the trellis at time earlier than $t$, without "realizing" that every path going through time $t$ must incur a large penalty eventually. The metric adjustment step avoids this situation by reinterpreting edge metrics as penalties with respect to the best edge at the same time step.

Computing the branch metrics at time $t$ can be done in constant time, since for all edges $e$ at time $t$, the metric $d(e)$ belongs to a set $M_t$, where $|M_t|$ is a constant that does not depend on the constraint length. Specifically, $|M_t| = 2^R$ where $R$ is the number of bits output by the encoder at time $t$, which depends only on the rate of the convolutional code.

## *C. Stream decoding*

We now discuss how to modify the lazy decoder to process an infinite stream of data.

Fix a *traceback length* $L \approx 5.8k$ as in [4]. At time $T$, the decoder processes a new symbol, and expands until the trellis data structure contains a node $u$ at time $T$. It then outputs its best estimate of the information bit at time $T - L$ by means of a traceback process [13, Section 12.4.6]. The traceback starts at the node $u$, and follows a path back (using the $Pred()$ pointers) until it reaches a node at time $T - L$. It then outputs the information bit(s) associated with the first transition on the path.

After this procedure, all nodes at time $T - L$ are no longer needed, and the memory that they occupy must be reclaimed. Specifically, we must delete all the nodes from the trellis, and all the shadow nodes from the priority queue, whose time is equal to $T - L$. To this extent, we maintain a linked list of all nodes and shadow nodes at time $t$. We maintain an array of pointers into such *time lists* indexed by time. Since only lists in the range $t \in [T - L, T]$ are nonempty, this array can be managed as a circular buffer of length $L + 1$.

After the traceback, we walk down the time list for $T - L$, deleting every node (or shadow node) along the way. Because the trellis is maintained as a hash table, deletion of a node takes

```
PROCESS-SYMBOL($x$):
1  $T \leftarrow T + 1 \pmod{L}$;
2  delete nodes at time $T - L$;
3  COMPUTE-BRANCH-METRICS(X);
4  repeat
5      repeat
6          $(\hat{u}, m) \leftarrow$ PQ-EXTRACT-MIN;
7      until not TRELLIS-LOOKUP($\hat{u}$);
8      Shadow node $\hat{u}$ is now considered a "real" node $u$.
9      EXPAND($u, m$);
10 until $time(u) = T$;
11 perform traceback, output bit $T - L$ from best path.

COMPUTE-BRANCH-METRICS($x$):
1  $S \leftarrow$ set of all branches at time $T$;
2  for all $(u, u') \in S$, do
3      compute $d(u, u')$ as in Viterbi given symbol $x$;
4  $m \leftarrow \min_{(u,u') \in S} d(u, u')$;
5  for all $(u, u') \in S$, do
6      $d(u, u') \leftarrow d(u, u') - m$.

EXPAND($u, m$):
1  $(s, t) \leftarrow u$;
2  TRELLIS-INSERT($u$);
3  for all successors $u'$ of $u$, do
4      Create a shadow node $\hat{u}'$.
5      $Prev(\hat{u}') \leftarrow u$;
6      $acc \leftarrow m + d(u, u')$;
7      PQ-INSERT($(u'), acc$);
8      insert $u'$ into $Time\ List(t + 1)$.
```

**Figure 3**. Pseudo code for the lazy Viterbi decoder. The main entry point is the procedure PROCESS-SYMBOL. The code for the priority-queue operations PQ-INSERT and PQ-EXTRACT-MIN, and for the trellis operations TRELLIS-LOOKUP and TRELLIS-INSERT is not shown.

constant time or expected constant time, depending on the hashing technique used [1]. Deletion of a shadow node from the priority queue takes constant time if each priority bucket is maintained as a doubly-linked list, a well-known technique [11].

The running time of the lazy Viterbi decoder applied to stream decoding is not affected asymptotically by the need to maintain the time lists; the time needed to walk down the list can be amortized against the time already taken to create the node (or shadow node). The running time can be affected, however, by the need to perform a traceback. This affect can be minimized by performing a traceback only every $B$ time steps, for some number $B$, where each traceback outputs the information bits associated with the first $B$ transitions of its path.

### *D. Pseudo code for the lazy decoder*

A pseudo code for the lazy Viterbi decoder appears in Figure 3. The entry point of the decoder is the procedure PROCESS-SYMBOL($x$), which is called on each symbol $x$ received from the channel. Line 1 advances the current time $T$, but, since the decoder only keeps nodes from the last $L$ time steps, time is advanced $(\bmod L)$. Because of this time periodicity, in line 2 we delete all the nodes at time $T - L$ from the trellis and from the priority queue. This step uses the time lists described in Section II-C. After these deletions, PROCESS-SYMBOL iteratively adds nodes to the trellis until it adds one with time $t = T$. At each iteration, the procedure repeatedly calls PQ-EXTRACT-MIN to extract the lowest-metric node $\hat{u}$ in the queue, until the "real" version $u$ of $\hat{u}$ is not already in the trellis (this test is performed by the procedure TRELLIS-LOOKUP($\hat{u}$)). Once such a shadow node $\hat{u}$ is found, it is considered a "real" node $u$, retaining its place in a time list, and maintaining its $Prev$ pointer. This node, along with its metric $m$ (returned by PQ-EXTRACT-MIN), is passed on to the EXPAND($u, m$) procedure.

The procedure COMPUTE-BRANCH-METRICS($x$) operates as described in Section II-B. Each branch metric $d(u, u')$ for edges at time $t$ is computed using the symbol $x$ received from the channel at time $t$.

Finally, the EXPAND($u, m$) procedure inserts the node $u$ into the trellis data structure. For all successors $u'$ of $u$, a new shadow node $\hat{u}'$ is created with an appropriate accumulated metric, and with $u$ as their predecessor. These new shadow nodes are then inserted into the priority queue and the time list.

## III. SPEED OF THE LAZY VITERBI DECODER

In this section, we report on the running time of the lazy decoder on four different processors, and we compare our decoder with optimized implementations of the Viterbi algorithm.

Figure 4 supports our claim that the lazy Viterbi decoder is a practical algorithm. We compared the lazy decoder with the Viterbi decoder written by Phil Karn [9] and with our own optimized implementation of Viterbi. The "unoptimized Karn" decoder works for all constraint lengths and for all polynomials. Karn also provides an optimized decoder which is specialized for constraint length 7 and for the "NASA" polynomials `0x6d`, `0x4f`. This optimized code unrolls the inner loop completely, and precomputes most memory addresses at compile time. Because Karn's optimized decoder only works for constraint length 7, we programmed our own optimized Viterbi decoder that works for constraint lengths up to 6. This program is labeled "optimized Viterbi" in the figure.

Karn [10] also has an implementation that uses SSE instructions on the IA32 architecture. These instructions operate on eight array elements at the same time. Karn's SSE implementation is a great hack, as it expands one node in slightly more than one machine cycle, but it only works for constraint lengths 7 and 9. As can be seen in the table, even the eight-fold gain in processing power is not sufficient to beat the lazy decoder for constraint length 9. Moreover, SSE instructions do not apply to the PowerPC processor or the StrongARM. (The PowerPC 7400 processor implements instructions similar to SSE, but no implementation was available that exploits them.)

The running times in the figure refer to the case of high SNR, where the lazy decoder performs a minimum number of node expansions. This is the most favorable case for the lazy decoder. Our focus on the best case is legitimate because, as can be seen in Figure 2, the lazy decoder operates in the best-case scenario

| Decoder | Constraint length | Athlon XP cycles/bit | Pentium III cycles/bit | PowerPC 7400 cycles/bit | StrongARM cycles/bit |
|---|---|---|---|---|---|
| Lazy | 6 | 193 | 201 | 200 | 226 |
| Viterbi Optimized | 6 | 275 | 316 | 239 | 310 |
| Karn Unoptimized | 6 | 1041 | 1143 | 626 | 892 |
| Lazy | 7 | 198 | 205 | 203 | 232 |
| Karn Optimized | 7 | 530 | 558 | 486 | 641 |
| Karn Unoptimized | 7 | 1806 | 2108 | 1094 | 1535 |
| Karn SSE | 7 | 107 | 108 | N/A | N/A |
| Lazy | 9 | 217 | 235 | 225 | 343 |
| Karn Unoptimized | 9 | 6300 | 8026 | 3930 | 5561 |
| Karn SSE | 9 | 307 | 310 | N/A | N/A |

**Figure 4**. Running time of various convolutional stream decoders under high SNR conditions. Times are expressed in cycles per decoded bit. Code for constraint length 6: TIA/EIA-136 code, polynomials `0x2b`, `0x3d`. Constraint length 7: "NASA" code `0x6d`, `0x4f`. Constraint length 9: IS-95 code `0x1af`, `0x11d`. Processors: 1466 MHz Athlon XP 1700+, 600 MHz Intel Pentium III, 533 MHz PowerPC 7400, 200 MHz StrongARM 110. All programs compiled with `gcc-2.95 -O2 -fomit-frame-pointer` and the most appropriate CPU flags.

as long as the SNR is at least 5–6 dB, which is a reasonable assumption in practice. (See [7] for further evidence that the number of expansions is small for a variety of codes.)

## IV. Conclusion

We discussed the lazy Viterbi decoder, a practical maximum-likelihood convolutional decoder. While the Viterbi algorithm lends itself to efficient hardware implementations, because it is has a simple and highly parallel structure, the lazy decoder is meant for use in software communication systems such as software radios. We built upon the $A^*$ algorithm of [2] and the experiments of [7] to design an algorithm that is efficient both asymptotically and practically. For sufficiently large constraint lengths, our algorithm outperforms optimized implementations of the Viterbi decoder, even if they use special processor instructions.

Some questions remain open. We assumed in this paper that sufficient memory is available to expand the whole trellis if necessary. In practice, so much memory may not be available. It is not clear how to modify the algorithm for this case. One alternative is to reduce the traceback length when running out of memory, but this choice reduces the error-correcting performance of the decoder. Another possibility is to discard trellis nodes with high accumulated metric, but this operation is not efficiently supported by the data structures of the lazy decoder. Yet another possibility is to switch to the original Viterbi algorithm when running out of memory. Since the lazy algorithm is expanding a significant fraction of the trellis, we may as well expand it all and avoid the overhead. We plan to investigate these possibilities in future work.

Finally, in this paper we focused on general-purpose processors. It is not clear how the lazy decoder would compare to the Viterbi algorithm on DSP chips with special hardware instructions that execute many Viterbi "butterflies" in one machine cycle.

## References

[1] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms*. M.I.T. Press, Cambridge, Massachusetts, U.S.A., 2001.

[2] L. Ekroot and S. Dolinar. $A^*$ decoding of block codes. *IEEE Trans. Comm.*, 44(9):1052–1056, 1996.

[3] R. M. Fano. A heuristic discussion of probabilistic decoding. *IEEE Trans. on Inform. Theory*, IT-9:64–73, 1963.

[4] G. Forney. Convolutional codes II: Maximum likelihood decoding. *Inform. Control*, 25:222–266, 1974.

[5] G. D. Forney. The Viterbi algorithm. *Proceedings of the IEEE*, 61:268–278, 1973.

[6] Y. Han, C. Hartmann, and C. Chen. Efficient priority-first search maximum-likelihood soft-decision decoding of linear block codes. *IEEE Transactions on Information Theory*, 39:1514–1523, 1993.

[7] Yunghsiang S. Han, Po-Ning Chen, and Hong-Bin Wu. A maximum-likelihood soft-decision sequential decoding algorithm for binary convolutional codes. *IEEE Transactions on Communications*, 50(2):173–178, February 2002.

[8] J. A. Heller and I. M. Jacobs. Viterbi decoding for satellite and space communication. *IEEE Transactions on Communications Technology*, pages 835–848, October 1971.

[9] Phil Karn. KA9Q Viterbi decoder V3.0.2, `viterbi-3.0.2.tar.gz`. `http://people.qualcomm.com/karn/code/fec/`, October 1999.

[10] Phil Karn. SIMD-assisted convolutional (Viterbi) decoders, `simd-viterbi-2.0.3.tar.gz`. `http://people.qualcomm.com/karn/code/fec/`, February 2002.

[11] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, 3rd edition, 1997.

[12] A. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Trans. Inform. Theory*, IT-13:260–269, Apr. 1967.

[13] S. Wicker. *Error Control Systems for Digital Communication and Storage*. Prentice-Hall, Englewood Cliffs, NJ, 1995.