

Sequence analysis

The Treeterbi and Parallel Treeterbi algorithms: efficient, optimal decoding for ordinary, generalized and pair HMMs

Evan Keibler¹, Manimozhiyan Arumugam^{1,2} and Michael R. Brent^{1,*}¹Laboratory for Computational Genomics, Campus Box 1045, Washington University, St. Louis, MO 63130, USA and ²Present address: The European Molecular Biology Laboratory (EMBL), 69117 Heidelberg, Germany

Received on September 15, 2006; revised on December 18, 2006; accepted on December 19, 2006

Advance Access publication January 18, 2007

Associate Editor: Alex Bateman

ABSTRACT

Motivation: Hidden Markov models (HMMs) and generalized HMMs have been successfully applied to many problems, but the standard Viterbi algorithm for computing the most probable interpretation of an input sequence (known as decoding) requires memory proportional to the length of the sequence, which can be prohibitive. Existing approaches to reducing memory usage either sacrifice optimality or trade increased running time for reduced memory.

Results: We developed two novel decoding algorithms, Treeterbi and Parallel Treeterbi, and implemented them in the TWINSCAN/N-SCAN gene-prediction system. The worst case asymptotic space and time are the same as for standard Viterbi, but in practice, Treeterbi optimally decodes arbitrarily long sequences with generalized HMMs in bounded memory without increasing running time. Parallel Treeterbi uses the same ideas to split optimal decoding across processors, dividing latency to completion by approximately the number of available processors with constant average overhead per processor. Using these algorithms, we were able to optimally decode all human chromosomes with N-SCAN, which increased its accuracy relative to heuristic solutions. We also implemented Treeterbi for Pairagon, our pair HMM based cDNA-to-genome aligner.

Availability: The TWINSCAN/N-SCAN/PAIRAGON open source software package is available from <http://genes.cse.wustl.edu>.

Contact: brent@cse.wustl.edu

1 INTRODUCTION

The hidden Markov model (HMM) and generalizations of it have been successfully applied to many problems in biological sequence analysis (Eddy, 1998; Durbin *et al.*, 1998; Krogh *et al.*, 1994, 2001). For example, generalized HMMs (GHMMs) and pair HMMs are among the best models for *de novo* gene prediction and sequence alignment, respectively (Arumugam *et al.*, 2006; Burge and Karlin, 1997; Eddy, 1995; Korf *et al.*, 2001; Pachter *et al.*, 2002). However, computing the most probable interpretation using these models (known as *decoding*)

can be impractical for large input sequences due to the memory requirements of standard decoding algorithms. Standard algorithms for ordinary and generalized HMMs require memory proportional to the length of the sequence times the number of states in the model, which, for gene prediction on full mammalian chromosomes, can reach hundreds of gigabytes (GB). The fundamental problem is that one can construct HMMs and arbitrarily long input sequences in which the end of the sequence determines the most probable analysis at the beginning. Such models do not seem natural for biological analysis or other common applications—one expects that a change to the sequence will not affect the analysis at very distant locations. However, standard decoding algorithms do not take advantage of the locality that occurs in practice—they wait until the end of the input sequence to finalize the analysis of the beginning.

1.1 The Viterbi algorithm for ordinary HMMs

The standard dynamic programming algorithm for decoding HMMs is called the Viterbi algorithm (see Durbin *et al.*, 1998; Forney, 1973; Viterbi 1967 for detailed descriptions). Given an input sequence $o_1 \dots o_T$, it works by calculating, for each model state S_i and each symbol o_t of the input sequence, the Viterbi probability

$$v[i, t] = b_i(o_t) \max_{j=1}^s a_{ji} v[j, t-1],$$

where $b_i(o_t)$ is the probability that the model emits the symbol o_t when it is in state S_i and a_{ji} is the probability that the model transitions to state S_i when it is in S_j . These probabilities are calculated from left ($t=0$) to right ($t=T$). $v[i, 0]=1$ if $S_i = \text{'begin'}$, the start state of the model; 0 otherwise. After column T , column $T+1$ is calculated with $b_i(o_{T+1})$ defined to be 1 if $S_i = \text{'end'}$, the end state of the model; 0 otherwise. After each maximum is taken, the argmax is stored in a matrix:

$$B[i, t] = \arg \max_{j=1}^s a_{ji} v[j, t-1]$$

The entries in the B matrix are called traceback pointers (or simply 'pointers' when no ambiguity results). Once the B matrix has been completely filled, the traceback pointers are followed from the 'end' state in column $T+1$ back to the 'begin' state. The sequence of states traversed by this traceback path is the

*To whom correspondence should be addressed.

most likely state sequence for the HMM as it emitted $o_1 \dots o_T$, i.e. the *optimal path*.

1.2 Generalized HMMs and pair HMMs

The probability that an ordinary HMM in state S_i will emit exactly d characters before changing states is governed by a geometric length distribution, $(1-a_{ii})a_{ii}^{d-1}$, where a_{ii} is the self-transition probability of state i . For many modeling tasks this is undesirable. In gene finding, for example, real exon-length distributions are not geometric. In a generalized HMM, on the other hand, each state has its own, arbitrary length distribution, and states do not have transitions back to themselves (Burge and Karlin 1997; Rabiner, 1989). Decoding is done with a simple generalization of the Viterbi Algorithm in which

$$v[i, t] = \max_{k=0}^{t-1} \max_{j=1}^s \left[\prod_{r=0}^k b_i(o_{t-r}) \right] a_{ji} \ell_i(k) v[j, t-k-1] \quad (1)$$

where $\ell_i(k)$ is the probability that the GHMM will emit exactly $k+1$ symbols while in state S_i . Traceback pointers can go to cells in any earlier column of the matrix, not just the previous column.

Although the GHMM formalism allows states to have unbounded, arbitrary length distributions, decoding with such models takes time proportional to the square of the input sequence length. This is prohibitive for long sequences, so typical applications use states that either have a maximum length or become geometric beyond some length. In some cases there may be application-specific, sequence-based limitations on traceback pointers. In gene finding, for example, states representing internal coding exons are often required to follow an intron ending in AG, precede an intron beginning with GT, and be free of in-frame stop codons. Our GHMM decoding algorithm exploits such limitations on potential traceback pointers (whether length- or sequence-based) to reduce memory usage and parallelize decoding.

Pair HMMs, which emit two symbols at each time step, are useful for sequence alignment. Because the standard optimal decoding algorithms for pair HMMs require memory and time proportional to the product of the sequence lengths, heuristics such as the Stepping Stone algorithm (Meyer and Durbin, 2002) are typically used. Even with heuristics, memory requirements can be excessive for applications like cDNA-to-genome alignment in mammals (Arumugam et al., 2006). Our method significantly reduces the memory requirements of both optimal pair HMM decoding and heuristic decoding with Stepping Stone.

1.3 Previous approaches to decoding in reduced memory

One approach to reducing the memory requirements of Viterbi for standard HMMs (and potentially GHMMs) is checkpointing (Grice et al., 1997; Tarnas and Hughey, 1998; Wheeler and Hughey, 2000). In this approach, Viterbi probabilities are stored for $O(n^{1/2})$ regularly spaced columns, known as checkpoints, for sequences of length n . In the intervening segments, which also have $O(n^{1/2})$ columns, the matrix can be recomputed starting from the probabilities stored in the preceding checkpoint. This is done in reverse order,

starting at the final segment. Once the final segment has been recomputed, it is decoded and the associated memory freed. Thus, the memory requirement is reduced to $O(n^{1/2})$ while the running time is doubled. Using L levels of embedded checkpoints, space can be reduced to $O(n^{1/L})$ in exchange for a factor of L slowdown.

Filtering and smoothing are heuristic methods for online decoding of large sequences. Since they are primarily used in fields other than biological sequence analysis, such as digital signal processing, and since they do not give the optimal solution, we will not deal with them here (but see Anderson and Moore, 1979; Elliot et al., 1994; Forney, 1973; Khasminskii and Zeitouni, 1996; Rabiner, 1989).

The best gene prediction programs use GHMMs. Because of the memory demands of standard Viterbi and the large input sequences, gene prediction is commonly done by splitting large sequences into many non-overlapping windows each of which is decoded independently. The results are then concatenated to form an approximate decoding of the full sequence. The drawback of this method is that the decoding near the ends of the subsequences will likely be non-optimal (i.e. different from the most probable decoding for the full sequence) and the decodings of adjacent windows are not guaranteed to be compatible with one another. For example, a gene finder could predict that one window ends in an intergenic region and the next starts in an intron. Overlapping windows can be used to mitigate these problems, but decoding the overlap region twice increases the computation time, and the results are still not guaranteed to be either compatible or optimal. There is no method for determining whether compatibility or optimality can be achieved in a particular case, and if so, how much overlap is needed.

The straightforward extension of the Viterbi algorithm for pair HMMs requires memory proportional to the product of the lengths of the two sequences to be aligned. The divide-and-conquer algorithm (Hirschberg, 1975; Myers and Miller, 1988) decodes pair HMMs (but not ordinary or generalized HMMs) in space proportional to the shorter of the two input sequences, but it doubles the running time. Checkpointing columns for pair HMMs reduces the space from $O(sm)$ to $O(sm^{1/L})$, where s is the number of states, m is the length of the shorter sequence, and n is the length of the longer sequence, in exchange for a factor of L slowdown. Theoretically, checkpointing diagonals with $L=O(\log n)$ can reduce space to $O(s(m+n))$ and time to $O(s^2nm)$, asymptotically matching the divide-and-conquer algorithm (Grice et al., 1997), although it is not clear that this is useful in practice (Tarnas and Hughey, 1998).

In this article, we present a practical algorithm that reduces the memory requirements for optimal decoding of ordinary and generalized HMMs to near constant. The two key ideas behind our algorithm are *garbage collection*—deallocating memory for matrix cells that will never be referenced again, and *early decoding*—identifying situations in which decoding can proceed up to a certain point even before the entire input has been read (see Section 2 for details). The usefulness of garbage collecting cells for ordinary HMMs was independently discovered and reported in (Henderson et al., 1997). The fundamental observation behind early decoding—that all traceback paths may coalesce and thereby identify a state on the optimal

path—was also made independently (e.g. Forney, 1973). However, we are not aware of any previous work suggesting that this could be detected dynamically and used to free memory in an optimal decoding algorithm. The combination of garbage collection and early decoding is what gives our algorithm its unique space efficiency. The modifications needed to make these ideas work for GHMMs have not been described previously.

We extend this approach to allow parallelization of optimal HMM decoding. Versions for GHMMs are presented and extensions for pair HMMs are briefly described. We report experiments showing that these algorithms reduce both memory requirements and latency for optimal decoding of large sequences using a GHMM gene predictor on a computing cluster. Experiments with pair HMMs show that this approach is also effective at reducing the memory requirements of a cDNA-to-genome aligner.

1.4 Application to gene prediction and cDNA alignment

The algorithms presented here have been tested on three programs: TWINSKAN, N-SCAN, and Pairagon. TWINSKAN is a state-of-the-art gene prediction system that takes advantage of alignments between two genomes (Flicek *et al.*, 2003; Korf *et al.*, 2001; Wu *et al.*, 2004). TWINSKAN's mammalian genome model is a GHMM with 49 states. For computational efficiency, 17 of the states have geometric length distributions and are decoded as in an ordinary HMM. Of the remainder, 4 have fixed lengths and 28 are fully generalized states with smoothed empirical length distributions. These fully generalized states represent coding exons and are constrained by the locations of in-frame stop codons, splice sites, and start and stop codons. N-SCAN is a variant of TWINSKAN that uses multi-genome alignments and phylogenetic tree models (Gross and Brent, 2005, 2006). The N-SCAN model includes all TWINSKAN states along with non-coding exon states for modeling spliced 5'UTRs (Brown *et al.*, 2005) (58 states total). Pairagon is a pair HMM designed for cDNA-to-genome alignment (Arumugam *et al.*, 2006). Pairagon uses nine ordinary HMM states and four states that emit fixed-length sequences.

With the standard Viterbi algorithm, TWINSKAN uses nearly 2 GB of memory per megabase (MB) of input sequence, and mammalian chromosomes can be hundreds of MB long. Using non-overlapping windows, information such as reading frame cannot be maintained across boundaries, so genes that cross the boundaries are almost always predicted incorrectly. For 1 MB windows, ~5% of known human genes cross boundaries, limiting accuracy to 95%. As more states are added to improve the accuracy of a model, memory requirements grow, window size must be decreased to fit in memory and the number of genes that cross window boundaries grows, reducing accuracy. Overlapping windows can improve accuracy, but only at a large cost in running time. For example, using 0.5 MB overlaps doubles computing time but is not sufficient to find optimal or even compatible analyses at every boundary.

Without the Stepping Stone heuristic (Meyer and Durbin, 2002), Pairagon fails to align 90% of human cDNAs within

a 4 GB limit. Stepping Stone greatly reduces Pairagon's memory requirements, but it still fails to align 5% of human cDNAs within 4 GB.

We have developed two novel decoding algorithms, Treeterbi and Parallel Treeterbi, and implemented them in TWINSKAN and N-SCAN. Treeterbi reduces TWINSKAN's memory requirement from 2 GB per million bases to <0.1 GB for sequences of any length, with no increase in running time. The memory requirements for N-SCAN go from ~3 GB/MB to ~1 GB for sequences of any length. We also implemented Treeterbi for pair HMMs. This reduced Pairagon's memory requirements for aligning cDNA sequence to the genome using the Stepping Stone heuristic from a worst case of 1000 + GB (avg. 2 GB) to 0.23 GB (avg. 0.1 GB).

For ordinary and generalized HMMs, Treeterbi makes it possible to decode large sequences in a single process without increasing total CPU time, but the latency to completion of that process may be significant. The Parallel Treeterbi algorithm, which we implemented for TWINSKAN/N-SCAN, allows optimal decoding to be run in parallel at a small overhead cost.

2 RESULTS

2.1 Memory optimization: ordinary HMMs

At any point in the execution of the standard Viterbi algorithm, the set of traceback pointers can be viewed as a tree, with the cells of the matrix as vertices and the begin-state cell of the first column as root. They form a tree because each cell contains exactly one traceback pointer, and all traceback paths lead to the 'begin' state (Fig. 1A). In the literature on Viterbi

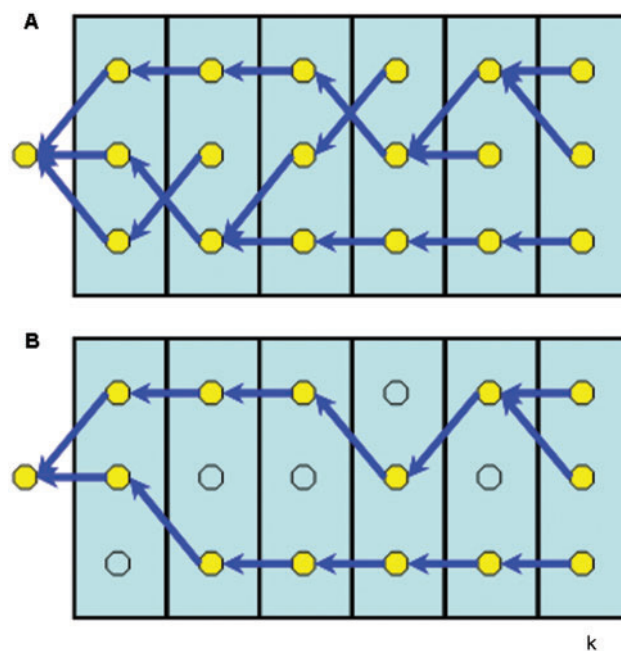


Fig. 1. (A) The trellis for a standard HMM after scoring the first k columns. The root is the 'begin' state in column 0. (B) The trellis in Panel A after freeing cells that cannot be reached from column k (open circles).

(e.g. Forney, 1973) this graph is called a *trellis*. The traceback pointers that can be reached from the most recently constructed column of the matrix constitute the *survivor tree* (Forney, 1973, closed circles in Fig. 1B).

Whereas the standard Viterbi algorithm stores probabilities and traceback pointers in a matrix whose size increases linearly with the length of the input, our algorithm stores them in a tree data-structure mirroring the survivor tree. We call this data-structure the Treeterbi tree. Cells that cannot be reached by any traceback path are garbage collected (Fig. 1B, open circles). Although cells are allocated individually, we continue to refer to the cells representing the potential model states at a given time as a column.

Each cell in the Treeterbi tree includes a reference count indicating how many cells have traceback pointers to it. When the reference count reaches zero, the cell is not reachable and can be freed. When a cell is freed, its traceback pointer is removed and the reference count of the cell it pointed to is decremented, possibly to zero.

In our algorithm, garbage collection occurs after each column of traceback pointers is completed. After column $k + 1$ is completed, all cells in column k that cannot be reached from $k + 1$ are freed. Their traceback pointers are followed up the tree, freeing cells whose reference counts are zero and stopping at the first cell whose reference count remains positive (Fig. 2).

Garbage collection saves a great deal of memory, but asymptotically, memory usage still grows in proportion to the length of the input. In an ordinary HMM, for example, at least one cell in each column is always reachable—the one on the optimal path. Treeterbi can free most of this memory by incrementally identifying initial segments of the optimal path and outputting them. In practice, this reduces asymptotic memory usage from linear to constant.

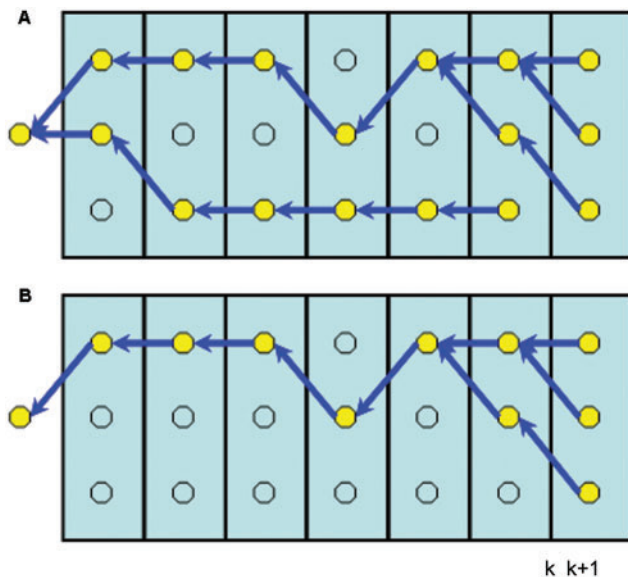


Fig. 2. (A) Treeterbi tree after constructing column $k + 1$, but before freeing cells that are not reachable from $k + 1$. (B) The same tree after cells that are not reachable from column $k + 1$ have been freed.

If the root of the Treeterbi tree has only one incoming edge, the one cell that can trace back to it must be on the optimal path to the root. Each cell therefore has a list of cells whose edges point to it (its children). Whenever the reference count of the root is one, the state and column of its child are output as part of the optimal path, the child becomes the new root, and the old root is freed. This early decoding process continues until the root has at least two incoming traceback pointers. We call such roots coalescence points, since all possible traceback paths coalesce at them, when it is necessary to distinguish them from roots that exist only transiently during early decoding.

The worst case asymptotic running times for both standard Viterbi and Treeterbi are proportional to the number of Viterbi probabilities evaluated. For ordinary HMMs this is $O(mn)$, where t is the number of state transitions with non-zero probability and n is the input length. This dominates the number of traceback pointers created, which is at most the number of states s times n . Each pointer is created and deleted at most once, and the corresponding cell is allocated and deallocated once. Reference counts change only when a pointer is set or removed, and they are checked for being 0 or 1 only when they change.

For standard Viterbi, both the best and worst case memory usage are $\Theta(sn)$. For Treeterbi, the worst case is also $\Theta(sn)$. A worst case example is an HMM that can transition from ‘begin’ to either S_1 or S_2 . From S_1 it deterministically changes to S_2 and from S_2 it can change to either S_1 or ‘end’. It always emits the same character, so the input provides no information about the state until the end, when the final state must be S_2 . Traceback pointers from S_1 always go to S_2 and vice-versa, so there can be no coalescence of paths except at the ‘begin’ state. However, this kind of long distance, deterministic effect of reaching the end (or any other state) is rare in practical HMM applications. The optimal state at any time is typically determined by the nearby input characters, not by distant ones. Typically, the state transitions are not all deterministic, and the emissions tend to give substantial information about the state. This leads to rapid coalescence of paths, which limits the amount of memory used by Treeterbi. In communication theoretic terms, where the state sequence is viewed as a pure signal and emission as a noise generating process, we are typically interested in systems whose original signal contains information (hence non-deterministic transitions) that is not completely masked by noise (hence the emissions provide information about the state). See (Shue, 1999) for further discussion of convergence rates.

2.2 Memory optimization: generalized HMMs

In GHMMs, traceback pointers can go to any preceding column, skipping intermediate columns (Fig. 3A). However, every path from the final column to the root must cross over any given column, k . That is, it must include a traceback pointer originating in column $k + 1$ or a later column and pointing to column k or an earlier column. The set of all such pointers forms a cut set that separates the vertices of the trellis in columns k and earlier (k^-) from the vertices in columns $k + 1$ and later (k^+) (Fig. 3B).

Our Treeterbi algorithm works from left to right, constructing all pointers *into* each column in turn. This is the opposite of the traditional order, in which all pointers *out of* each column are constructed in the outer loop. Switching the order of the loops guarantees that, once the outer loop has passed column k , the reference counts of cells in column k never increase. They decrease when an incoming pointer is moved to a cell in a later column that yields a higher Viterbi probability. Thus, these pointers are only tentative until the outer loop reaches the column from which they originate. Once the outer loop passes a column, pointers originating in it are final.

Pseudocode for this algorithm is shown in Figure 4, where *tree* initially consists of a root cell with column zero, state ‘begin’, and Viterbi probability one. *GHMM* is the model, and *sequence* is the input. LOOKUP-CELL looks up the cell for a given state and column in the index carried by the tree, and GET-CELL does the same thing, but if the cell is not found it allocates a new cell with score 0, enters it in the index, and returns it. SCORE(*to_cell*, *from_state*, *from_column*) computes the Viterbi probability as defined by the quantity inside the maxima of equation (1). The corresponding variable names are *from_state* = i , *from_column* = t , and *to_cell* = (state j , column $t - k - 1$). SET-TRACEBACK(*from_cell*, *to_cell*, *score*) sets the pointer of *from_cell* to *to_cell*, sets the Viterbi probability of *from_cell* to *score* and adds *from_cell* to the list of children of *to_cell*. COLLECT

recursively checks the reference counts of cells for zeros until it hits a cell with a non-zero count. If the root of the tree has reference count 1, it performs early decoding up to the first node with a reference count of two or greater. Pseudocode for COLLECT is shown in Figure 5, where FREE(*cell*) removes a cell from the tree index and then deallocates it, and FIRST(*list*) returns the first element of a list.

When TREETERBI is called, the only cell in the tree is the root. Lines 3 and 4 guarantee that no other cell is allocated or checked for incoming pointers unless it already has an outgoing pointer to an existing cell. Outgoing pointers are only removed when the cell they originate from has reference count zero or becomes the root. Cells with reference count zero are removed immediately. Thus, by induction, every non-root cell in the tree has a path back to the root. Since reference counts are non-increasing, cells with a reference count of zero can never be reached on any path, so they can be safely deallocated without changing the traceback path or the score of any accessible node.

Treeterbi provides memory savings to the extent that cells in some columns cannot be reached directly by traceback pointers from other columns. For example, consider what happens when *to_column* is 1. If every future cell can trace back to column 1 in a single step, then memory must be allocated to store all those tentative traceback pointers. However, if only a small fraction

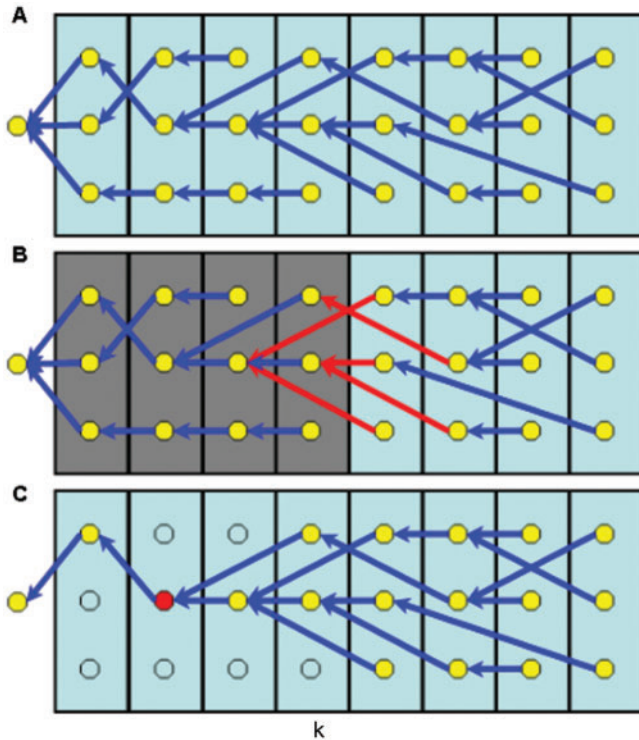


Fig. 3. (A) The complete trellis for the first 8 columns of a GHMM traceback matrix. (B) The cut set of edges crossing column k . k^- columns are gray and k^+ columns are blue. (C) The Treeterbi tree after unreachable nodes in k^- are removed. After the root is updated, the red cell will be the root.

```

TREETERBI(tree, GHMM, sequence)
1 for to_column ← 0 to length[sequence]
2   for each state to_state ∈ states[GHMM]
3     to_cell ← LOOKUP-CELL(index[tree], to_state, to_column)
4     if to_cell ≠ NIL ▷ NIL indicates no outgoing pointers
5       for each (from_state, from_column) that can traceback to to_cell
6         SCORE ← SCORE(to_cell, from_state, from_column)
7         if score ≠ 0
8           from_cell ← GET-CELL(tree, from_state, from_column)
9           if score > score[from_cell]
10            if parent[from_cell] ≠ NIL
11              decrement reference_count[parent[from_cell]]
12              COLLECT(tree, parent[from_cell])
13              SET-TRACEBACK(from_cell, to_cell, score)
14            COLLECT(tree, to_cell)

```

Fig. 4. Pseudocode for the Treeterbi algorithm. The tree in the initial call consists of only one cell, which contains state ‘begin,’ column 0 and Viterbi probability 1, and serves as the root.

```

COLLECT(tree, cell)
1 while reference_count[cell] = 0
2   parent ← parent[cell]
3   decrement reference_count[parent]
4   FREE(cell)
5   cell ← parent
6 while reference_count[root[tree]] = 1
7   old_root ← root[tree]
8   print column[old_root], state[old_root]
9   root[tree] ← FIRST(children[old_root])
10  FREE(old_root)

```

Fig. 5. Pseudocode for garbage collection and early decoding.

of future cells can trace back to column 1 with non-zero probability, then relatively little memory is needed to store pointers into column 1, and the reference counts in column 1 are correspondingly smaller. As pointers into later columns are constructed, some pointers that originally pointed to column 1 may be changed to point to later columns, decreasing the reference counts in column 1 and potentially allowing garbage collection and early decoding.

The number of cells that can potentially traceback to a given column may be limited in an input-independent way, as when a state only emits strings of bounded length or has a geometric length distribution, which requires only pointers of length 1. It can also happen in an input-dependent way. For example, N-SCAN states emitting transcript-internal, protein-coding exons must start after ‘AG’, end before ‘GT’, and contain no in-frame stop codons. Construction of traceback pointers from such a state can halt as soon as an in-frame stop codon is encountered. For applications with very long input strings, practical models will always have some limitations on traceback pointers; without them the running times of both standard Viterbi and Treeterbi grow as the square of the input length, which is not feasible for very long inputs.

The running time for both Viterbi and Treeterbi is determined by the number of Viterbi probabilities evaluated—i.e. the number of possible traceback pointers considered. Viterbi probabilities are never evaluated for emissions that exceed length limits or contain unallowable sequences such as in-frame stop codons—the iteration in line 5 of Figure 4 is implemented so as to exclude them. Keeping reference counts, garbage collecting and early decoding do not change the asymptotic running time for the reasons described above. If the index used by LOOKUP-CELL and GET-CELL is implemented as a hash table, and hash table operations are considered constant time, then these lookups do not change the asymptotic running time.

For GHMMs as for ordinary HMMs, both the worst case and the average case asymptotic memory usage of standard Viterbi are $\Theta(sn)$. For Treeterbi, the worst case is also $\Theta(sn)$. If there exists some column such that the number of traceback pointers into it grows linearly with sequence length, then memory usage also grows linearly with sequence length. By the same token, the time needed to evaluate all traceback pointers into that column grows linearly. If the number of such columns also grows linearly with n , running time grows quadratically while memory usage remains linear as existing pointers are updated. In that case, running time is likely to limit input length, not memory usage. In situations where the number of possible traceback pointers is small enough that evaluating them is feasible, Treeterbi typically makes storing them feasible, too. Models in which all states have bounded length emissions behave well on all input sequences, but a much broader range of models may behave well on nearly all input sequences.

For models in which traceback pointers cannot exceed some relatively small maximum length d , the index can be implemented as a matrix of length d that holds cells with indices between to_column and $to_column + d$. In TWINSCAN/N-SCAN,

some states have geometric length distributions (hence traceback pointers of length one) and some have lengths that are bounded by a small constant. Cells for these states are indexed by a matrix. Other states have traceback pointers that are sparse, due to limitations on the sequences they can emit, but potentially very long (e.g., exon states). In our current implementation, cells for these states are indexed by a list, although a hash table provides better worst-case running time guarantees.

2.3 Memory optimization: pair HMMs

Garbage collection and early decoding work for pair HMMs, too. In this case the Viterbi matrix has three dimensions: one for each sequence and one for the states of the model. The recursion for filling the matrix with Viterbi probabilities is:

$$v[k, l, i] = \max \left\{ \begin{array}{l} b_i(x_k, y_l) \max_{j=1}^s a_{ji} v[k-1, l-1, j] \\ b_i(x_k, _) \max_{j=1}^s a_{ji} v[k-1, l, j] \\ b_i(_, y_l) \max_{j=1}^s a_{ji} v[k, l-1, j] \end{array} \right\} \quad (2)$$

where x_k is the k th character of input string x , y_l is the l th character of input string y , and b_i gives the probability that when the HMM is in state S_i it emits the pairs (x_k, y_l) , $(x_k, _)$, $(_, y_l)$, respectively. The underscore represents the absence of any character.

The simplest way to apply the methods described above is to construct all pointers into successive planes defined by an index k in the longer of the two sequences. Thus, the loop in line 1 of Figure 4 is replaced by a pair of nested loops, first over the index into the longer string, then over the index into the shorter string. For a generalized pair HMM the remainder of the code is the same, except that *to_column* and *from_column* are replaced by pairs of indices into the strings, (*to_column*, *to_plane*) and (*from_column*, *from_plane*).

Ordinary pair HMMs can be treated like ordinary non-pair HMMs, where the outer loops are over *from_column* and *from_plane*. Garbage collection is done after each plane is completed.

It is also possible to iterate over the pairs of string indices in a different order, such as enumerating diagonals.

2.4 Parallelization

With the Treeterbi algorithm, memory limitations no longer prevent optimal decoding of very large sequences within a single process, but the time required still grows at least linearly with the length of the input (quadratically for worst-case GHMMs). It can take hours to decode one large mammalian chromosome using the current N-SCAN model on current computers. Although the total amount of computation cannot be reduced, splitting it across multiple processors can reduce the latency to completion.

The Parallel Treeterbi algorithm divides the input sequence into windows that are decoded in independent processes without sacrificing optimality. Each process attempts to find a cell in its window through which the optimal path must pass,

regardless of the results of the other process. Such cells are called *pins*. At the beginning of the sequence we know that the HMM starts in the ‘begin’ state, but for a window in the middle, there is no way to know which state the HMM is in without finding a pin. Once a pin is found, it plays a role identical to the ‘begin’ state, in that it is the root of the Treeterbi tree.

By using interprocess communication and load balancing it would be possible to have each process dynamically adjust the boundaries of its decoding window to start and end at a pin. However, we have taken a different tack that allows the windows to be fixed at the start and eliminates all interprocess communication. In our approach, each process identifies a pin and then divides its window in two at the pin. It decodes from the pin forward until early decoding has reached the right boundary of the window—i.e. until the root has advanced beyond the end of the window. It then decodes backward from the pin to the start of the window. During the backward decoding phase, the pin is equivalent to the ‘end’ state, and the Viterbi probability for cell (q, r) is the probability of emitting the input sequence from column r to the pin while traversing the most likely state sequence beginning with state q . The traceback pointers point forward to the next state in that most likely state sequence.

To find a pin in a window, Parallel Treeterbi considers all cells from which there is a potential traceback pointer that crosses the left boundary of the window. The optimal path must use one of these cells to cross the window boundary, so the algorithm considers all of them. Each of these cells serves as the root of a new, forward-decoding tree. A slightly modified TREETERBI is then run on each of these trees until the optimal paths passing through all of their roots have some cell in common. That cell is a pin, because regardless of how the optimal path crosses the initial boundary of the window, it must pass through that cell.

During the pin search, if the updated roots of two trees with different initial roots are the same, then we can discard one of the trees. All their subsequent traceback pointers would be identical because the trees are constructed according to identical rules, so all differences are due to different initial roots. Once early decoding has identified an updated root, we know that all paths must pass through that root, so the initial root is no longer relevant in determining the traceback path. (The Viterbi probabilities at a common updated root may still depend on the initial root, but the difference will be a constant factor that applies to all paths in that tree, so it will not affect the choice of path.) When only one tree remains, all paths must pass through its updated root, regardless of the initial root. Thus, the last remaining root is a pin.

To make this efficient, the TREETERBI decodings for all roots are synchronized so they all have the same value for *to_column*. This allows us to easily check whether two trees that started out with different roots have the same updated root at the same time. If the root updates were not synchronized, we would have to store and compare updated roots for multiple time points for each tree, defeating the point. The initial value of *to_column* for each TREETERBI run is the column of its root. To synchronize

the values, a global variable *min_column* is iterated. Each TREETERBI decoding starts when *min_column* is the same as its root column, and continues by using *min_column* in place of *to_column*.

In principle, pin search could be done with the standard Viterbi algorithm as well. However, finding a pin may require running a large number of simultaneous decodings on a single processor. For our applications, that would not be feasible using standard Viterbi because it requires so much memory per decoding. With Treeterbi, however, memory has not been a problem.

2.5 Memory optimization experiments

We compared the Treeterbi Algorithm to the standard Viterbi algorithm using our gene prediction program N-SCAN and our cDNA-to-genome alignment program Pairagon.

2.5.1 N-SCAN memory usage and running time Using the Treeterbi algorithm, we ran N-SCAN on the human genome without splitting the chromosomes. One measure of the advantage that can be gained by using Treeterbi is how much of the sequence gets decoded at a time during early decoding—i.e. the distance between adjacent coalescence points (CPs). If they are very far apart, we will have to store a large tree of traceback pointers while waiting for the next one to appear. Likewise, Parallel Treeterbi may have to search further to find a pin. In our experiment, the average distance between CPs was <2000 bases, and 99% of CPs were followed by another CP within 150 Kb.

A more direct measure of memory usage is the tree size in number of cells. Figure 6 shows the distribution of the number of cells in the tree during our experiment (for readability, the figure does not show the distribution beyond 10 000 cells). The median size is less than 1700 cells and 99% of the time the tree has <20 000 cells. The maximum number observed in our tests was ~45 000 cells. For comparison, decoding human chromosome 1 using standard Viterbi with our model would require more than 10 billion cells, while Treeterbi never needs more than 45 000, a reduction of more than 200 000 fold.

Although using the tree structure rather than the standard matrix requires some bookkeeping, our Treeterbi implementation actually runs slightly faster than our standard Viterbi implementation.

2.5.2 N-SCAN accuracy Using non-overlapping windows of 1 MB, 759 of the aligned human RefSeq mRNAs (Pruitt *et al.*, 2005) cross window boundaries (~5%) and N-SCAN cannot predict these correctly. When we run N-SCAN on full chromosome sequences, which is only possible using the Treeterbi algorithm, we correctly identify a transcript in 177 (~23%) of these. This increases the fraction of RefSeq genes which we correctly predict (gene sensitivity) from 37.5% to 38.6%. N-SCAN also predicts 1306 fewer genes when run on whole chromosomes as compared to 1 MB windows, increasing the fraction of predictions that match aligned RefSeqs (gene specificity) from 23.9% to 26.2%. The total number of human genes predicted declined from 21 444 to 20 138, which is very similar to estimates made by other

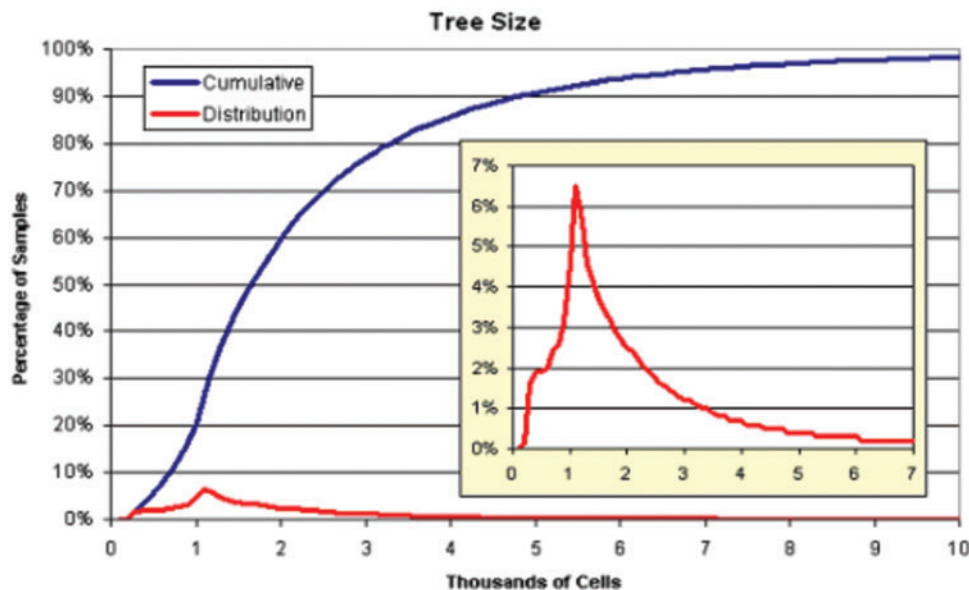


Fig. 6. A distribution of the size of the tree when running NSCAN on the human genome. Tree size was sampled every 1000 bases and binned at 100-cell resolution.

methods (Lindblad-Toh *et al.*, 2005). When N-SCAN was run on 1 MB windows, 2201 predicted genes at the boundaries were incomplete, meaning that the boundary fell within a predicted exon or intron. When it was run on whole chromosomes, 1267 of these were extended or merged into 888 complete gene predictions. The remainder were changed or eliminated as a result of considering the sequence outside the original 1 MB window.

N-SCAN's sensitivity for boundary-crossing genes (~23%) is less than its overall sensitivity (~39%) because long genes are more difficult to predict exactly right and more likely to cross window boundaries. On average, genes that crossed window boundaries were 4.3 times longer than those in the full set.

2.5.3 Pairagon memory usage Using the Treeterbi algorithm with the Stepping Stone heuristic and the Pairagon model, we aligned 23 302 human cDNAs to the human genome. The average number of cells required was 225 000, as compared to ~200 million for Viterbi with Stepping Stone and ~3 trillion for Viterbi without Stepping Stone. Treeterbi with Stepping Stone requires 0.1% as many cells as Viterbi with Stepping Stone on average and 1.5% as many in the worst example.

Using Treeterbi and Stepping Stone, all 23 302 alignments completed without using more than 0.25 GB at any time. Using standard Viterbi with Stepping Stone, 5% of these cDNAs cannot be aligned in 4 gigabytes and 1.5% cannot be aligned in 8 GB. Using Treeterbi for optimal alignment (no heuristics), we aligned 1351 human cDNAs (the subset of our 23 302 that maps to chromosomes 20, 21 and 22). None of these required more than 0.35 GB. Using standard Viterbi, only 1 of the 1351 could be aligned optimally within 4 GB.

2.6 Parallelization experiments

Running Treeterbi in parallel can reduce latency to completion greatly, but it incurs some overhead for finding pins and finding coalescence points on both sides of each window. To assess the overhead cost in a practical application, we implemented Parallel Treeterbi for N-SCAN and ran it on the full human genome three times using 1 MB, 5 MB and 10 MB non-overlapping windows. The median number of bases searched before finding a pin was <0.1 MB and 99% of pins were found in <1.75 MB. The longest pin search was 3.4 MB. This does not necessarily determine the running time though, as the number of trees being maintained may have a greater effect. Therefore, we calculated the percentage of the time spent on overhead when decoding windows of 1, 5 and 10 MB. Regardless of windows size, the average overhead takes about the same time as decoding 1 MB. If a single n -megabase sequence were divided into w non-overlapping windows to be run on w separate processors, the total running time would be expected to increase by a factor of $(n+w)/n$ while the latency to completion would decrease by a factor of $(n+w)/nw$.

3 DISCUSSION

In practical applications of generalized HMMs, the Treeterbi algorithm makes it possible for the first time to carry out optimal decoding of arbitrarily long sequences within bounded memory. The Parallel Treeterbi algorithm makes it possible to split the work of optimal decoding across arbitrarily many processes, dividing latency to completion by approximately the number of available processors with constant average overhead per processor. Using these algorithms, we were able to carry out optimal decoding of all human chromosomes with the N-SCAN gene predictor, which increased its accuracy.

We were also able to optimally align 1351 cDNA sequences to the human genome using Pairagon, a relatively complex pair HMM, within 0.35 GB of memory, whereas standard Viterbi was able to align only one of these within 4 GB.

The divide-and-conquer and checkpointing algorithms are viable alternatives for pair HMMs. For ordinary and generalized HMMs, however, Treeterbi is the only method that can, in practical applications, optimally decode arbitrarily long sequences with no increase in running time. Several heuristic schemes have been used for decoding long sequences with ordinary HMMs, including filtering and smoothing, which are methods for approximate decoding with only a fixed number of columns available at any one time. The one that most resembles Treeterbi stores only the n most recent columns of the standard Viterbi matrix, for some fixed n (Forney, 1972, 1973). If all traceback paths coalesce within these columns, states on the unique path back from the coalescence point are output, and the memory in which those columns were stored is recycled for new columns. If more than one traceback path is possible in the first of the n stored columns, one of them is chosen heuristically. This keeps the decoding within fixed memory, but at the cost of optimality. This approach does not appear to have been applied to GHMMs or pair HMMs.

Ordinary HMMs can be viewed as a simple special case of graphical models. The Treeterbi approach can also be applied to more general graphical models, such as Bayesian networks, in which each node represents a variable and the edges represent the dependency structure of the joint distribution on the variables (Frey, 1998; Jordan, 1998; Pearl, 1988). Such models have been widely applied in artificial intelligence, medical informatics and data mining. For singly connected graphical models, a direct generalization of the Viterbi algorithm can be used to determine the most likely instantiation of the hidden variables given the values of the observed variables. However, the standard implementation is designed for the worst case, in which an observed variable determines the most likely instantiation of another variable even when their interaction is mediated by thousands of hidden variables on the path connecting them.

The Treeterbi approach can be used to exploit whatever locality exists in any particular graphical model instantiation problem. When Treeterbi runs on an ordinary HMM, each column t represents all possible values of a variable q_t for the state at time t . Each state variable separates the HMM into two sub-processes, one before t and one after, that are independent, given the state at time t . In the terminology of Bayesian networks, each state variable *dependency-separates* the network into two sets of state variables. In more general Bayesian networks, some variables may dependency-separate the network into two or more subgraphs while others may not. Each variable that dependency-separates the network can be treated analogously to a state variable for an ordinary HMM. Let K and V be variables that dependency-separate the model into subgraphs V^-/V^+ and K^-/K^+ , respectively, where $V^- \subset K^-$. Suppose Treeterbi has constructed a tree of traceback pointers throughout V^- and K^- . If all possible traceback paths from K to V converge on a single value $V = v$, then V is a coalescence point. All variables in V^- can be instantiated and the corresponding memory freed. Thus, when a variable

dependency-separates the model into sub-networks, the order in which the sub-networks are processed corresponds to the temporal order of state variables in an ordinary HMM. As in an ordinary HMM, either end can be processed first without affecting the outcome, so from the point of view of decoding, it is the connectivity that matters, not the temporal order. The same logic applies when K and V are sets of variables that jointly dependency-separate the network and $V = v$ is a value of their joint distribution.

Parallel Treeterbi generalizes in a similar way.

In summary, the algorithms and approaches described above have wide applicability for finding the most likely values of the joint distributions on large sets of related variables. Although their worst-case asymptotic behavior is no better than that of previous algorithms, they allow significant memory reduction and parallelization in a range of practical applications.

The most important open problem with regard to early decoding and parallelization is developing a formal theory of convergence of traceback paths. For any given ordinary HMM, we would like to be able to bound the depth of the Treeterbi tree—i.e. the number of characters over which state ambiguity can persist (see Shue, 1999 for an early effort in this direction). Such a theory might provide an absolute bound for all input strings or it might provide a probabilistic bound under certain assumptions about the distribution of input strings, such as the assumption that they are generated from the HMM that is used for decoding. A weaker, but still very useful result would be the ability to prove that the traceback paths for a given HMM will converge after a finite number of inputs with asymptotic probability one. The ability to efficiently identify any ‘locking sequence’ that guarantees convergence, regardless of the context in which it appears, would represent significant progress—currently, the only general way to identify a locking sequence is by enumerating input strings and running Treeterbi on them.

ACKNOWLEDGEMENTS

This article benefited greatly from the comments of an anonymous reviewer. We thank Sean Eddy and William Smart for helpful conversations on alternative approaches, and Randall Brown for comments on an earlier draft. E.M.K. was supported by NIH training grant T32-GM008802. M.A. and M.R.B. were supported in part by NIH R01 HG002278, in part by U01 HG003150, and in part by the National Cancer Institute, NIH, under Contract No. N01-CO-12400. The content of this publication does not necessarily reflect the views or policies of the Department of Health and Human Service. Funding to pay the Open Access publication charges was provided by Washington University.

Conflict of Interest: none declared.

REFERENCES

- Anderson, B.D.O. and Moore, J.B. (1979) Forwards and backwards models for finite-state Markov processes. *Adv. Appl. Probab.*, **11**, 118–133.
- Arumugam, M. *et al.* (2006) Pairagon + N-SCAN_EST: a model-based gene annotation pipeline. *Genome Biol.*, **7** (Suppl 1), S5 1–10.

- Brown,R.H. et al. (2005) Begin at the beginning: predicting genes with 5' UTRs. *Genome Res.*, **15**, 742–747.
- Burge,C. and Karlin,S. (1997) Prediction of complete gene structures in human genomic DNA. *J. Mol. Biol.*, **268**, 78–94.
- Durbin,R. et al. (1998) *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, Cambridge, UK.
- Eddy,S.R. (1995) Multiple alignment using hidden Markov models. *Proc. Int. Conf. Intell. Syst. Mol. Biol.*, **3**, 114–120.
- Eddy,S.R. (1998) Profile hidden Markov models. *Bioinformatics*, **14**, 755–763.
- Elliot,R.J. et al. (1994) Hidden Markov models: estimation and control. In *Applications of Mathematics*. Vol. 29, 2nd edn. Springer-Verlag.
- Flicek,P. et al. (2003) Leveraging the mouse genome for gene prediction in human: from whole-genome shotgun reads to a global synteny map. *Genome Res.*, **13**, 46–54.
- Forney,G., Jr. (1972) Maximum-likelihood sequence estimation of digital sequences in the presence of intersymbol interference. *Information Theory, IEEE T. on*, **18**, 363.
- Forney,G.D., Jr. (1973) The viterbi algorithm. *Proc. IEEE*, **61**, 268.
- Frey,B.J. (1998) *Graphical Models for Machine Learning and Digital Communication*. The MIT Press, Cambridge, Mass.
- Grice,J.A. et al. (1997) Reduced space sequence alignment. *Comput. Appl. Biosci.*, **13**, 45–53.
- Gross,S.S. and Brent,M.R. (2005) Using multiple alignments to improve gene prediction. In Miyano, S., Kasif, S., Pevzner, P., Mesirov, J., Istrail, S. and Waterman, M. (eds.) In *9th Annual International Conference, RECOMB 2005*. Springer, Boston, pp. 374–388.
- Gross,S.S. and Brent,M.R. (2006) Using multiple alignments to improve gene prediction. *J. Comput. Biol.*, **13**, 379–393.
- Henderson,J. et al. (1997) Finding genes in DNA with a hidden Markov model. *J. Comput. Biol.*, **4**, 127–141.
- Hirschberg,D.S. (1975) A linear space algorithm for computing maximal common subsequences. *Commun. ACM.*, **18**, 341–343.
- Jordan,M.I. (1998) North Atlantic Treaty Organization. Scientific Affairs Division. In *Learning in Graphical Models*. Kluwer Academic Publishers, Dordrecht, Boston.
- Khasminskii,R. and Zeitouni,O. (1996) Asymptotic filtering for finite state Markov chains. *Stoch. Proc. Appl.*, **63**, 1–10.
- Korf,I. et al. (2001) Integrating genomic homology into gene structure prediction. *Bioinformatics*, **17** (Suppl 1), S140–S148.
- Krogh,A. et al. (1994) Hidden Markov models in computational biology. Applications to protein modeling. *J. Mol. Biol.*, **235**, 1501–1531.
- Krogh,A. et al. (2001) Predicting transmembrane protein topology with a hidden Markov model: application to complete genomes. *J. Mol. Biol.*, **305**, 567–580.
- Lindblad-Toh,K. et al. (2005) Genome sequence, comparative analysis and haplotype structure of the domestic dog. *Nature*, **438**, 803–819.
- Meyer,I.M. and Durbin,R. (2002) Comparative ab initio prediction of gene structures using pair HMMs. *Bioinformatics*, **18**, 1309–1318.
- Myers,E.W. and Miller,W. (1988) Optimal alignments in linear space. *Comput. Appl. Biosci.*, **4**, 11–17.
- Pachter,L. et al. (2002) Applications of generalized pair hidden markov models to alignment and gene finding problems. *J. Comput. Biol.*, **9**, 389–399.
- Pearl,J. (1988) *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufman, CA. San Mateo.
- Pruitt,K.D. et al. (2005) NCBI Reference Sequence (RefSeq): a curated non-redundant sequence database of genomes, transcripts and proteins. *Nucleic Acids Res.*, **33**, D501–D504.
- Rabiner,L.R. (1989) A tutorial on hidden Markov models and selected applications in speech recognition. *Proc IEEE*, **77**, 257–286.
- Shue,L. (1999) On performance analysis of state estimators for hidden Markov models. In *Doctoral Dissertation*. The Australian National University, Canberra.
- Tarnas,C. and Hughey,R. (1998) Reduced space hidden Markov model training. *Bioinformatics*, **14**, 401–406.
- Viterbi,A.J. (1967) Error bounds for convolution codes and an asymptotically optimum decoding algorithm. *IEEE T. Inform. Theory*, **13**, 260–267.
- Wheeler,R. and Hughey,R. (2000) Optimizing reduced-space sequence analysis. *Bioinformatics*, **16**, 1082–1090.
- Wu,J.Q. et al. (2004) Identification of rat genes by TWINSKAN gene prediction, RT-PCR, and direct sequencing. *Genome Res.*, **14**, 665–671.