BIOINFORMATICS

Vol. 14 no. 5 1998
Pages 401–406

Reduced space hidden Markov model training

Christopher Tarnas and Richard Hughey

Department of Computer Engineering, Jack Baskin School of Engineering, University of California, Santa Cruz, CA 95064, USA

Received on September 5, 1997; revised and accepted on January 2, 1998

Abstract

Motivation: *Complete forward–backward (Baum–Welch) hidden Markov model training cannot take advantage of the linear space, divide-and-conquer sequence alignment algorithms because of the examination of all possible paths rather than the single best path.*

Results: *This paper discusses the implementation and performance of checkpoint-based reduced space sequence alignment in the SAM hidden Markov modeling package. Implementation of the checkpoint algorithm reduced memory usage from O(mn) to $O(m\sqrt{n})$ with only a 10% slowdown for small m and n, and vast speed-up for the larger values, such as m = n = 2000, that cause excessive paging on a 96 Mbyte workstation. The results are applicable to other types of dynamic programming.*

Availability: *A World-Wide Web server, as well as information on obtaining the Sequence Alignment and Modeling (SAM) software suite, can be found at http://www.cse.ucsc.edu/research/compbio/sam.html.*

Contact: *rph@cse.ucsc.edu*

Introduction

Dynamic programming forms the core of many sequence analysis methods, including classic methods for sequence–sequence comparison and alignment (Needleman and Wunsch, 1970; Smith and Waterman, 1981), as well as more recent methods such as profiles and linear hidden Markov models (HMMs) (Gribskov *et al.*, 1990; Krogh *et al.*, 1994).

The typical process of training an HMM consists of performing many complete dynamic programming calculations on the model and each of the training set sequences. This is repeated many times until the model converges to a statistical representation of the family of sequences. The model may then be used to create a multiple alignment or to find other related sequences.

Perhaps the simplest form of the dynamic programming equation is that of edit distance (Wagner and Fischer, 1974; Sellers, 1974):

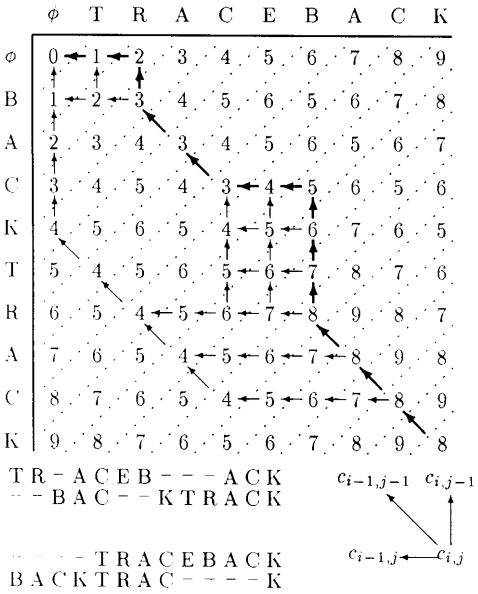

Fig. 1. Dynamic programming example to find least cost edit of 'BACKTRACK' into 'TRACEBACK' using Sellers' evolutionary distance metric (Sellers, 1974). Below the dynamic programming table are two possible alignments and an illustration of the data dependencies.

$$c_{i,j} = \min \begin{cases} c_{i-1,j-1} + dist\ (a_i b_j) & match \\ c_{i-1,j} + dist\ (a_i \phi) & insert \\ c_{i,j-1} + dist\ (\phi, b_j) & delete \end{cases} \quad (1)$$

The calculation of this recurrence can be arranged in a table of costs for matching each prefix of *a* to each prefix of *b*, and the selection information from the minimizations can be used to trace back the optimal alignment (Figure 1).

The addition of affine gap costs *g* to the recurrence (Gotoh, 1982) produces three interleaved recurrences of similar form. Here, the comparison or alignment can be thought of as being in one of three states: in the midst of a sequence of matches, deletions or insertions, where the cost for changing states provides the constant term in the affine gap cost. To emulate the simplest affine gap cost model, many of the *g*

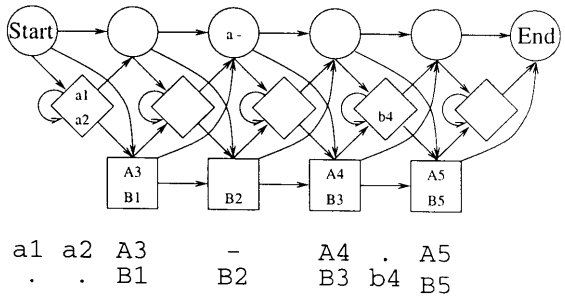

Fig. 2. An example of an HMM with two sequences whose characters are generated by the HMM, and the corresponding alignment. Positions modeled by the HMM's match states are indicated with upper case letters, while those modeled by unaligned insertion states are indicated with lower case letters.

values are zero, indicating, for example, that there is no extra cost in matching one pair of characters and then matching the next pair of characters. In the most general forms of sequence comparison, such as HMMs and generalized profiles (Bucher and Bairoch, 1994; Krogh *et al.*, 1994), all transition or gap costs (g) between the three states (match, insert or delete) and character distance costs are position dependent:

$$c^M_{i,j} = \min\,(c^M_{i-1,j-1} + g^{M\to M}_j, c^I_{i-1,j-1} + g^{I\to M}_j, c^D_{i-1,j-1} + g^{D\to M}_j) + dist\ (a_i, b_j)$$

$$c^I_{i,j} = \min\,(c^M_{i-1,j} + g^{M\to I}_j, c^I_{i-1,j} + g^{I\to I}_j, c^D_{i-1,j} + g^{D\to I}_j) + dist\ (a_i, \phi)$$

$$c^D_{i,j} = \min\,(c^M_{i,j-1} + g^{M\to D}_j, c^I_{i,j-1} + g^{I\to D}_j, c^D_{i,j-1} + g^{D\to D}_j) + dist\ (\phi, b_j)$$

The typical linear HMM (Figure 2) is a chain of match (square), insert (diamond) and delete (circle) nodes, with all transitions between nodes and all character costs in the insert and match nodes trained to specific probabilities. When performing dynamic programming between a sequence and an HMM, the HMM will index one dimension, such as the columns, of the matrix. The single best path through an HMM corresponds to a path from the Start state to the End state in which each character of the sequence is related to a successive match or insertion state along that path (delete states indicate that the sequence has no character corresponding to that position in the HMM). If two sequences are aligned to the model, a multiple alignment between those sequences can be inferred from their alignments to the model, though it must be remembered with HMMs that characters modeled by insert states are not aligned between sequences. This HMM multiple alignment algorithm requires time proportional to the number of sequences or, alternatively, the product of the total number of residues and the model length.

The forward–backward method of HMM training considers probabilities rather than scores over all possible states, replacing the minimizations with the addition of probability, and the additions with multiplication of probability (the probabilities are stored as log probabilities to avoid underflow). Thus, the forward pass becomes:

$$f^M_{i,j} = (f^M_{i-1,j-1} \cdot g^{M\to M}_j + f^I_{i-1,j-1} \cdot g^{I\to M}_j + f^D_{i-1,j-1} \cdot g^{D\to M}_j) \cdot P^M_j(a_i, b_j)$$

$$f^I_{i,j} = (f^M_{i-1,j} \cdot g^{M\to I}_j + f^I_{i-1,j} \cdot g^{I\to I}_j + f^D_{i-1,j} \cdot g^{D\to I}_j) \cdot P^I_j(a_i, \phi)$$

$$f^D_{i,j} = (f^M_{i,j-1} \cdot g^{M\to D}_j + f^I_{i,j-1} \cdot g^{I\to D}_j + f^D_{i,j-1} \cdot g^{D\to D}_j) \cdot P^D_j(\phi, b_j)$$

and, considering the various parameters as probabilities, computes

$$f_{i,j} = P(a_1 \ldots a_i, b_1 \ldots b_j|\text{HMM})$$

where in this simplified version of the HMM notation, the ranges on a and b indicate that the first i characters of a were generated by a chain of j states of the HMM ending at state b_j. A similar recurrence, starting at the end of the sequence and the end of the model, is used to compute the backward values

$$b_{i,j} = P(a_n \ldots a_i, b_m \ldots b_j|\text{HMM})$$

indicating that the last $n - i + 1$ characters of a can be generated by a chain of $m - j + 1$ states of the model. The forward and the backward values are combined to yield

$$P(a_i \text{ is generated by state } b_j|\text{HMM}) = f_{i,j} \cdot b_{i,j}$$

This value, the probability that a certain character was generated by a certain state of the HMM, is then used to update the probabilities in the HMM, in association with the values for other sequences in the training set and a regularizer or Dirichlet mixture prior (Sjölander *et al.*, 1996). The notation has been simplified; the reader is referred to the literature for a more detailed treatment (Rabiner, 1989; Krogh *et al.*, 1994) and an HMM review (Eddy, 1996).

The simplest approach to computing these $O(nm)$ dynamic programs is to create a large, $n \times m$ table in memory to store values. Unfortunately, this table will not fit entirely in a workstation's memory for large model and sequence lengths. For example, if a family of long molecules is to be modeled, say 2000 nucleotides and 2000 model nodes, the dynamic programming matrix will include 12×10^6 entries, one for each state at each index point. To provide sufficient precision, in the Sequence Alignment and Modeling (SAM) software suite each of those entries requires four bytes, so 48×10^6 bytes are required just for the dynamic programming table. When added to the memory requirements for the rest of the application, not to mention the operating system and

other users, this will consume most all the memory of a typical workstation with 64 Mbytes of main memory. If the sequence or model length is larger, or memory available for the program is lower, virtual memory will be used extensively. That is, blocks of memory (or pages—typically 2–8 kilobytes of data each) will be temporarily stored on disk, with the computer's real memory and cache storing only currently active pages. In a current machine, a cache may require one clock cycle to access, main memory 10–20 clock cycles, and disk over one million cycles (Hennessy and Patterson, 1996). Thus, the cost of paging into virtual memory is high, and can make runs of a given size effectively uncomputable.

The solution to this is to use a sequence alignment method that requires less space. In the case of finding the single best path, there is an elegant divide-and-conquer algorithm that requires only $O(n + m)$ space, where n is the sequence length and m is the model length (Hirschberg, 1975). The approach of this algorithm is to find a midpoint of the best path without saving all $O(nm)$ dynamic programming entries, and then to solve two smaller problems, each of approximate size $nm/4$ using the same algorithm. This algorithm is well known in the computational biology community (Myers and Miller, 1988), and has, for example, been implemented in the HMMer package for sequence alignment to a trained HMM (Eddy *et al.*, 1995).

The divide-and-conquer algorithm does not work with forward–backward training. The efficiency of the divide-and-conquer algorithm is in its partitioning into subproblems so that while the entire dynamic programming matrix must be evaluated once, recursive calls consider smaller and smaller segments of the matrix. With forward–backward training, all paths through the matrix are an important part of training the HMM, and thus this gain cannot be used.

One answer is a recently introduced checkpointing algorithm (Grice *et al.*, 1997). In the simplest case, diagonals, rows or columns of the dynamic programming matrix are stored at regular intervals to reduce space use to $O(m\sqrt{n})$ while increasing runtime by a small constant factor. In the experiments discussed below, the new checkpoint-based HMM training requires ~10% more time than standard dynamic programming up to the point of virtual memory paging, at which point the checkpoint algorithm is considerably faster. The reduced space requirements are particularly valuable in multiple-user environments.

System and methods

The work described herein was performed using the Sequence Alignment and Modeling (SAM) HMM software suite (http://www.cse.ucsc.edu/research/compbio/sam.html). Code development and performance analysis were performed on a DEC Alpha 255 with a 233 MHz clock, 96 Mbytes of main memory and a 1 Mbyte external cache. SAM is written in ANSI C and was compiled using the Alpha's native compiler at its highest optimization level.

Algorithm

The family of checkpoint algorithms asymptotically provides for an arbitrary integer L, a factor of cL slowdown ($c < 1$) in exchange for reducing memory use from $O(mn)$ to $O(m\sqrt[L]{n})$. A single-best-path variant of this family with $L = \log(n)$ matches the quadratic time and linear space of the divide-and-conquer algorithm. For SAM running on a workstation, we decided that the $L = 2$ algorithm, which reduces space use by the square root of the sequence length, would be sufficient. We describe this variant in more detail below, and refer the reader to the original work for information on and analysis of the complete family of algorithms (Grice *et al.*, 1997).

2-Level checkpoints

Simply stated, the idea behind 2-level checkpoints is to segment the backwards computation. For each of the segments to be computed, a single checkpoint of all values along a row, column or diagonal is saved during a global forward calculation. The backward computation now has two parts: recalculating and saving the forward values of that segment using the checkpoint for initial conditions, and then performing the backward dynamic programming on that segment. Thus, with each (i,j) index point in the dynamic programming matrix, there will be an associated global forward calculation, a local forward calculation and a backward calculation.

While the 2-level checkpoint algorithm performs more computation than the standard method (two forward calculations and one backward calculation for each dynamic programming cell), it gains greatly in memory performance. In the simplest case, it requires space for all the checkpoints and for performing the complete forward–backward calculation on one segment. The result is that the 2-level method requires $O(m\sqrt{n})$ space, where m is the model length and n is the sequence length. For the example above, this will reduce space use from 48×10^6 bytes to 1×10^6 bytes. If even longer sequences and models are required, an additional level of checkpointing can be added to the algorithm to reduce space use to $O(m\sqrt[3]{n})$, or 300×10^3 bytes in the example.

Viterbi checkpoints

The Viterbi algorithm, used to find the single best path through a dynamic programming matrix (i.e. an alignment of a sequence to a model) is, as mentioned, simpler than the forward–backward algorithm. Rather than performing costly exponentiation and logarithm extraction operations on log probabilities (in SAM, these are greatly speeded up using table look-up), addition and minimization can be used for the

forward-going part of the Viterbi algorithm, while a simple tracing back of the single best path using saved selector bits from the minimizations is all that is required to establish the model to sequence correspondence.

When applying checkpointing to the Viterbi algorithm, diagonal checkpoints can be used to reduce the amount of recalculation greatly, making use of the same principle introduced in a parallelization of the divide-and-conquer algorithm (Huang, 1989). That is, if the single best path has been traced back to a point (i, j) on the $i + j = k$ diagonal, and the next checkpoint is l diagonals away at $i + j = k - l$, then only the triangular region bounded by $(i - 1, j)$, $(i, j - l)$ and (i, j), and containing $l^2/2$ dynamic programming cells needs to be recalculated to find the best path between the $i + j = k$ and $i + j = k - l$ diagonals. For the Viterbi algorithm with row checkpoints, a larger, $l \times m$ (worst case) strip must be recalculated, depending on the model node back to which the path has currently been traced. For the full forward–backward dynamic programming, an entire $l \times m$ strip of the matrix is always required.

For these reasons, we used diagonal checkpoints rather than row or column checkpoints. To speed code development, we decided to use diagonal checkpoints for both the Viterbi algorithm and the forward–backward algorithm.

With hindsight, the added programming complexity and runtime overhead of boundary conditions with the diagonal checkpoints (especially for the local and semi-local algorithms discussed next), as well as the difficulty of maintaining the Viterbi and the forward–backward routines in parallel, made this a poor choice. Our forward–backward and Viterbi procedures would be significantly simpler had we implemented them with row checkpoints. This simplicity, combined with programmer and compiler optimizations, could reduce or eliminate the theoretical advantage of diagonal checkpoints in the case of Viterbi training.

Local and semi-local checkpointing

A second goal of SAM's inner loop rewrite was to provide local and semi-local scoring, alignment, and training. This added even more complexity to the decision to use diagonal rather than row checkpoints. Fully local alignment allows the matching of a subregion of the sequence to a subregion of the model. Because HMMs are a probabilistic model, this is more complicated than, for example, the zero-thresholding used in the Smith and Waterman algorithm (Smith and Waterman, 1981).

For fully local alignment, a SAM model is flanked by two copies of the null model, in SAM called free-insertion modules, or FIMs (Hughey and Krogh, 1996; Barrett *et al.*, 1997). The null model is a simple probabilistic model, such as the background distribution of amino acids, of the universe of sequences not modeled by the HMM. The logarithm of the ratio of the probability of the sequence being generated by the model and by the null model, the log-odds score, indicates how much better (or worse) the structured HMM models the sequence than the simple unstructured null model (Altschul, 1991).

For SAM, a sequence can make use of fully local alignment as follows. The initial segment of the sequence that does not correspond to any part of the HMM will align to the initial FIM. The net effect on the log-odds score of this is zero because the FIM is a copy of the null model. From the FIM node (thought of as the first column of the dynamic programming matrix if the model nodes are used to index the columns), we allow a jump directly into the delete state of any position within the model. Thus, in the calculation of the $c^D_{i,j}$ values, an additional term is added to the minimization: $c^I_{i,0} + g_j^{S \to D}$, where $g_j^{S \to D}$ is the gap cost of skipping over an initial segment of the model from the Start node to the jth delete node.

In fully local alignment, the correspondence between the sequence and the model can also end at any point within the sequence and the model. This corresponds to allowing jumps from the delete state of any position in the model to the final FIM. Much like the initial FIM, the final FIM will then match any remaining characters of the sequence with a net effect of zero on the final log-odds score. Thus, for the final FIM's delete state in node n (for simplicity, jumps are only allowed into and out of delete states), the term

$$\min_{0<k<n} \left(c^D_{i,k} + g_j^{D \to E}\right)$$

is combined with the standard minimization for $c^D_{i,n}$. For SAM, the costs of jumping into the End node, $g_j^{D \to E}$, and jumping out of the Start node, $g_j^{S \to D}$, are global and default to zero cost, as with standard Smith and Waterman, primarily for compatibility with existing models. In the more general form, these costs would be position dependent (Bucher and Bairoch, 1994), and could also be trained given sufficient data.

Analogous changes are made to the forward–backward calculation.

Semi-local dynamic programming, which allows a complete sequence to match a subsection of the model, is simpler than fully local alignment. Jumps into the model are only allowed for the first character of the sequence, while jumps out of the model are only allowed for the last character of the sequence. Thus, the special boundary conditions of fully local alignment only need to be evaluated for two rows of the dynamic programming matrix rather than all rows of the dynamic programming matrix.

Implementation of local and semi-local jumps in combination with the diagonal checkpoints proved time consuming. The jumps occur across values in a row of the dynamic programming matrix. Evaluation of the jumps would be simple and regular with row or column checkpoints, but required

more complicated indexing with our chosen diagonal checkpoints.

Results

After implementing and optimizing the checkpoint algorithm, we performed several experiments on the host workstation in which the dynamic programming calculation was allowed to use a variable amount of memory. Our initial thought was that we would see different levels of efficiency as the problem size grew first beyond the primary cache size and then, in the case of the original space-inefficient code, beyond the main memory size. This turned out not to be the case: the code is computation bound because of the log-probability manipulation.

Because we did not see great variation in runtime for performing partial checkpointing (e.g. checkpointing only twice to reduce the dynamic programming table's memory requirements to $mn/2$ entries and only having to recalculate half of the forward values), the code now always uses full 2-level checkpointing and $O(m\sqrt{n})$ space.

Performance is summarized in two sets of graphs. Figure 3 shows runtime versus sequence length for performing four dynamic programming calculations between a length 500 model and sequences of various lengths. The upper graph shows central processing unit (CPU) time, while the lower graph shows real, or wall clock, time. We took data for forward–backward training using both full 2-level checkpointing (top curve) and no checkpointing (middle curve), as well as checkpointed and uncheckpointed implementations of the Viterbi best path algorithm. The checkpointing forward–backward code is ~10% slower than the uncheckpointed version. The Viterbi method is approximately seven times faster than forward–backward because of its simpler computation. The checkpointed and the uncheckpointed Viterbi algorithm perform similarly until paging begins. This could be due to the Viterbi forward calculation being relatively simple, reducing the penalty of recalculation when compared to the overhead of updating an HMM. Above a sequence length of 7000 residues, the 96 Mbytes of main memory on the test machine are exhausted, and the wall clock time of the uncheckpointed code increases to a factor of five at 10 000 residues.

Data for a 2000-node RNA model show similar results (Figure 4), although in this case, because of the larger model, virtual memory degradation begins around a length 2000 sequence (4×10^6 dynamic program cells or 48 Mbytes, consistent with the protein results). We were unable to obtain uncheckpointed results beyond sequences of length 2000 because of the excessive virtual memory use.

Since including the checkpoint method in SAM, the code has been used to model 31 sequences of ~9500 bases as part of an effort to determine how well HMMs can align RNA sequences (the folding of which is not modeled by a linear HMM), as well as to form a multiple alignment of 60 16S RNA sequences. Neither of these tasks could be completed with the previous version of SAM.

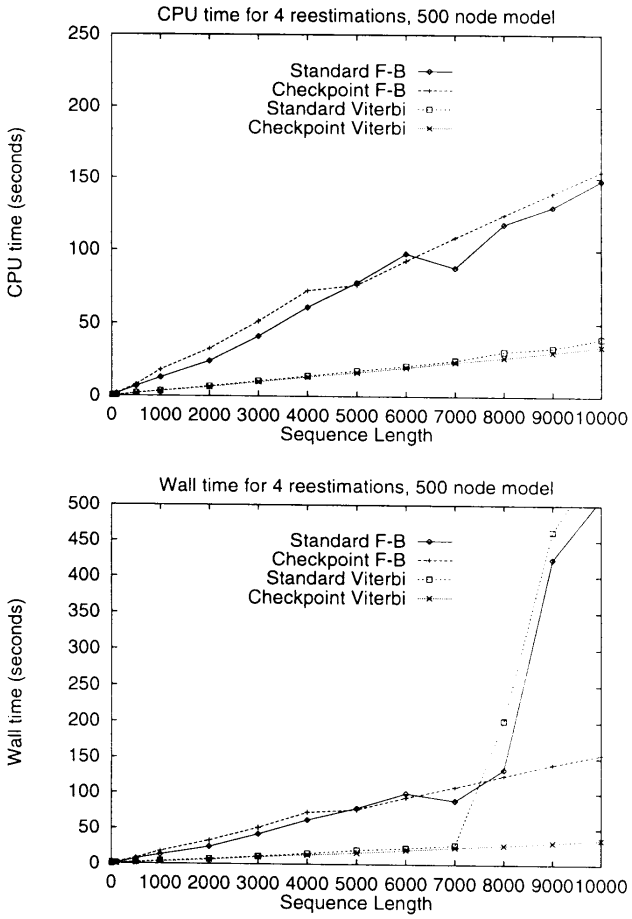

Fig. 3. CPU time and wall clock time for performing four dynamic programming calculations with a 500-node model and several protein sequence lengths using the forward–backward (F–B) and Viterbi algorithms. As can be seen comparing the CPU and wall time graphs, above 8000 amino acids, the memory requirements cause the 96 Mbyte workstation to page excessively when the standard algorithm is used.

Discussion

The most interesting result of this work is that the checkpointing method, which performs significantly more computation than the simple method, only slightly slowed down the program for problems that fit in memory even though the vast majority of SAM's computation time is the dynamic programming calculation. This may be due to the greater cache efficiency of the checkpointing algorithm.

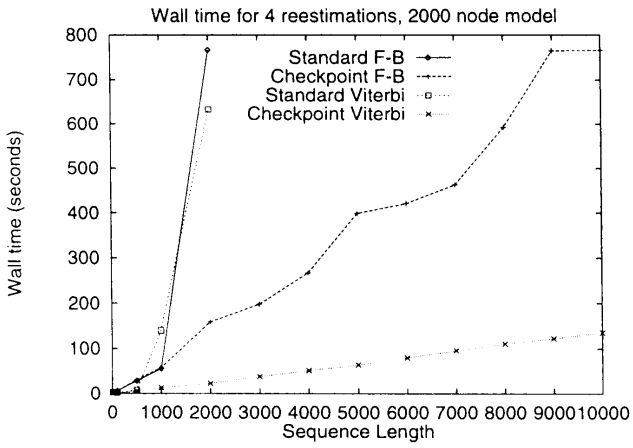

Fig. 4. Wall clock time for performing four dynamic programming calculations with a 2000-node model and several RNA sequence lengths. Virtual memory problems occur at a similar product of the model length and sequence length, in comparison to Figure 3.

Computing the dynamic programming matrix along diagonals has several advantages. Such computation removes all data dependencies from the inner loop of the dynamic programming calculation. This frees the compiler to perform more extensive code reorganization and optimization within the inner loop. Diagonal computation is required when performing dynamic programming on a vector or fine-grain parallel processor (Huang, 1989; Hughey, 1996). With hindsight, though, the added complexity of the boundary conditions when processing diagonals was not worth the potential performance gain. To enhance maintainability, we plan to re-reimplement SAM's inner loop with row checkpoints, learning from the experiences of this work.

The 2-level checkpointing algorithm is not as memory efficient as the divide-and-conquer approach, requiring $O(m\sqrt{n})$ space rather than $O(n + m)$ space. It has three advantages that make it a strong alternative to the divide-and-conquer approach. First, it can be used with the forward–backward calculation. Second, coding may be somewhat simpler than the divide-and-conquer, especially if row or column checkpoints are used. Third, the constant overhead appears to be less than that of the divide-and-conquer approach, perhaps because it is not a fully recursive algorithm.

Acknowledgements

This work was supported in part by National Science Foundation grant DBI-9408579 and its Research Experiences for Undergraduates supplement, as well as an equipment donation from Digital Equipment Corporation. The original algorithm development was supported in part by NSF grant MIP-9423985. C.T. is currently with Pangea Systems, Inc., 1999 Harrison Street, Suite 1100, Oakland, CA 94612, USA.

References

Altschul,S.F. (1991) Amino acid substitution matrices from an information theoretic perspective. *J. Mol. Biol.*, **219**, 555–565.

Barrett,C., Hughey,R. and Karplus,K. (1997) Scoring hidden Markov models. *Comput. Applic. Biosci.*, **13**, 191–199.

Bucher,P. and Bairoch,A. (1994) A generalized profile syntax for biomolecular sequence motifs and its function in automatic sequence interpretation. In Altman,R. *et al.* (eds), *Proceedings of the International Conference on Intelligent Systems for Molecular Biology.* AAAI/MIT Press, Menlo Park, CA, pp. 53–61.

Eddy,S. (1996) Hidden Markov models. *Curr. Opin. Struct. Biol.*, **6**, 361–365.

Eddy,S., Mitchison,G. and Durbin,R. (1995) Maximum discrimination hidden Markov models of sequence consensus. *J. Comput. Biol.*, **2**, 9–23.

Gotoh,O. (1982). An improved algorithm for matching biological sequences. *J. Mol. Biol.*, **162**, 705–708.

Gribskov,M., Lüthy,R. and Eisenberg,D. (1990) Profile analysis. *Methods Enzymol.*, **183**, 146–159.

Grice,J.A., Hughey,R. and Speck,D. (1997) Reduced space sequence alignment. *Comput. Applic. Biosci.*, **13**, 45–53.

Hennessy,J.L. and Patterson,D.A. (1996) *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, Los Altos, CA.

Hirschberg,D.S. (1975) A linear space algorithin for computing maximal common subsequences. *Commun. ACM*, **18**, 341–343.

Huang,X. (1989) A space-efficient parallel sequence comparison algorithm for a message-passing multiprocessor. *Int. J. Parallel Program.*, **18**, 223–239.

Hughey,R. (1996) Parallel sequence comparison and alignment. *Comput. Applic. Biosci.*, **12**, 473–479.

Hughey,R. and Krogh,A. (1996) Hidden Markov models for sequence analysis: Extension and analysis of the basic method. *Comput. Applic. Biosci.*, **12**, 95–107.

Krogh,A., Brown,M., Mian,I.S., Sjölander,K. and Haussler,D. (1994) Hidden Markov models in computational biology: Applications to protein modeling. *J. Mol. Biol.*, **235**, 1501–1531.

Myers,E.W. and Miller,W. (1988) Optimal alignments in linear space. *Comput. Applic. Biosci.*, **4**, 11–17.

Needleman,S.B. and Wunsch,C.D. (1970) A general method applicable to the search for similarities in the amino acid sequences of two proteins. *J. Mol. Biol.*, **48**, 443–453.

Rabiner,L.R. (1989) A tutorial on hidden Markov models and selected applications in speech recognition. *Proc. IEEE*, **77**, 257–286.

Sellers,P.H. (1974) On the theory and computation of evolutionary distances. *SIAM J. Appl. Math.*, **26**, 787–793.

Sjölander,K., Karplus,K., Brown,M.P., Hughey,R., Krogh,A., Mian,I.S. and Haussler,D. (1996) Dirichlet mixtures: A method for improving detection of weak but significant protein sequence homology. *Comput. Applic. Biosci.*, **12**, 327–345.

Smith,T.F. and Waterman,M.S. (1981) Identification of common molecular subsequences. *J. Mol. Biol.*, **147**, 195–197.

Wagner,R.A. and Fischer,M.J. (1974) The string-to-string correction problem. *J. ACM*, **21**, 168–173.