Dynamic Programming Alignment of Sequences Representing Cyclic Patterns

Jens Gregor and Michael G. Thomason, *Senior Member, IEEE*

***Abstract*—String alignment by dynamic programming is generalized to include cyclic shift and corresponding optimal alignment cost for strings representing cyclic patterns. A guided search algorithm uses bounds on actual alignment costs to find all optimal cyclic shifts. The bounds are derived from submatrices of an initial dynamic programming matrix. Algorithmic complexity is analyzed for major stages in the search. Applicability of the method is illustrated with satellite DNA sequences and circularly permuted protein sequences.**

***Index Terms*—Cyclic patterns, DNA and protein sequences, dynamic programming, guided search, string matching, structural pattern analysis.**

I. Introduction

COMPARISON OF one finite-length string with another is a fundamental method of structural pattern analysis employed in many applications [1], [8], [9], [11], [12]. The standard computation consists of aligning one string with the other according to a cost function computed optimally by dynamic programming. An alignment of strings $\boldsymbol{a}$ and $\boldsymbol{b}$ defines a sequence of edit operations that transforms $\boldsymbol{b}$ into $\boldsymbol{a}$ by matching, substituting, inserting, or deleting symbols in $\boldsymbol{a}$ and $\boldsymbol{b}$ on a symbol-by-symbol basis. Each individual edit is assigned a real-valued cost by the cost function, and the costs in a complete edit sequence are added. An optimal alignment of $\boldsymbol{a}$ and $\boldsymbol{b}$ is one for which the edit sequence has the minimum total cost. The computation is organized by systematically filling in a dynamic programming matrix with time and space complexity $\mathcal{O}(mn)$, where the length of $\boldsymbol{a}$ is m, and the length of $\boldsymbol{b}$ is n.

A typical application of this technique uses $\boldsymbol{a}$ as a prototype, possibly representing a larger class of strings, to which candidate string $\boldsymbol{b}$ must be compared. It may be the case, however, that a circular shift of $\boldsymbol{b}$ has a superior (lower cost) alignment with $\boldsymbol{a}$ than does $\boldsymbol{b}$ unshifted. This may occur, for example, in digital image analysis in which strings represent closed contours of objects without unique starting points for scanning [10]. It may also occur when biological strings are compared for evidence of evolutionary similarities [4].

To address these cases, we generalize the definition of an optimal alignment to include all cyclical shifts of $\boldsymbol{b}$ as candidates and require that the amount of the shift be part of an optimal solution. This paper presents an algorithm to find *all* shifts yielding optimal alignments. The algorithm is based on computing the full dynamic programming matrix M for $\boldsymbol{a}$ and $\boldsymbol{b}$ unshifted and then using information from M to process candidate circular shifts of $\boldsymbol{b}$. The goal is to avoid additional, time-consuming dynamic programming computation if possible.

Section II describes cyclic alignments in more detail, and Section III discusses partitioning $\boldsymbol{a}$ and $\boldsymbol{b}$ into substrings aligned in a candidate solution. Upper and lower bounds on the costs of these substring alignments are established. Numerical values of bounds for actual strings are computed from matrix M. Section IV discusses the algorithm using these bounds in its search for optimal solutions. As the algorithm investigates candidate solutions, it (i) updates the values of the tightest bounds on costs known to be achievable and (ii) uses these tightest bounds as references to eliminate any candidate alignment as soon as it is found to be suboptimal. Thus, the algorithm's search is guided by actively updating the bounds on optimal costs. Section IV also gives a complexity analysis of major stages in the search algorithm.

Section V describes results in two applications involving biomolecular strings. The first deals with two nucleotide sequences from satellite DNA sequences in which there are repetitive patterns. The second deals with two protein sequences evidently arising from a common ancestor but with evolutionarily induced cyclical shifts.

II. Cyclic Alignments

Let e denote the empty symbol. For all symbols a_i and b_j in the alphabet $\mathcal{A}$, the function d assigns nonnegative, real-valued costs to the string-edit operations *match*, *substitute*, *delete*, and *insert*. To make notation simpler in places, we assume that d satisfies the following requirements commonly imposed in practice [8]:

$$\begin{array}{lcll} \mathrm{d}(a_i, b_j) & = & 0 & \textit{if} \quad a_i = b_j \\ \mathrm{d}(a_i, b_j) & > & 0 & \textit{if} \quad a_i \neq b_j \\ \mathrm{d}(a_i, b_j) & = & \mathrm{d}(b_j, a_i) & \\ \mathrm{d}(a_i, e) & = & \mathrm{d}(e, a_i) & > \quad 0. \end{array}$$

The cost of a series of edits is the sum of their individual costs. Function d is extended to finite-length strings $\boldsymbol{a}$ and $\boldsymbol{b}$ in alphabet $\mathcal{A}$ by defining $\mathrm{d}(\boldsymbol{a}, \boldsymbol{b})$ to be the minimal cost of transforming $\boldsymbol{b}$ into $\boldsymbol{a}$ by a series of edits. Zero cost for match and nonzero cost for other edits ensures that $\mathrm{d}(\boldsymbol{a}, \boldsymbol{b}) = 0$ iff

Manuscript received December 24, 1990; revised November 4, 1991. This work was supported by the Danish Technical Research Council, Grant 16-4406.E, and the Danish Research Academy. Recommended for acceptance by Associate Editor H. Baird.

The authors are with the Department of Computer Science, University of Tennessee, Knoxville, TN 37996.

IEEE Log Number 9206557.

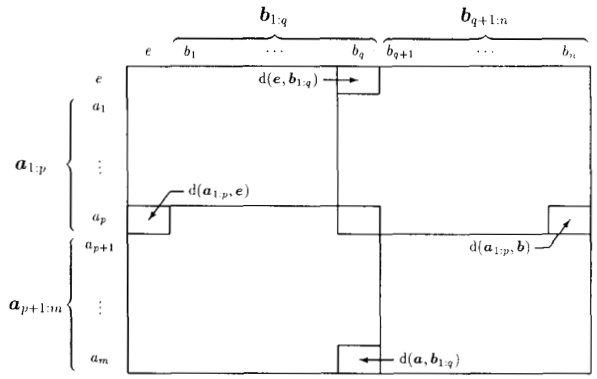


Fig. 1. Matrix M showing $\boldsymbol{a}$ partitioned at p and $\boldsymbol{b}$ partitioned at q.

$\boldsymbol{a} = \boldsymbol{b}$. Letting deletion and insertion cost be identical and substitution cost be symmetric ensures that $\mathrm{d}(\boldsymbol{a}, \boldsymbol{b}) = \mathrm{d}(\boldsymbol{b}, \boldsymbol{a})$.

The *cyclic alignment problem* is to find optimal alignment of two strings $\boldsymbol{a} = a_1 a_2 \cdots a_m$ of length m and $\boldsymbol{b} = b_1 b_2 \cdots b_n$ of length n for $m, n \geq 2$, where $\boldsymbol{a}$ is a prototype string, and $\boldsymbol{b}$ is a test string that is allowed to be cyclically shifted q symbols $0 \leq q \leq n-1$ before edit cost is computed. To refer to $\boldsymbol{a}$ aligned with a specific shift of $\boldsymbol{b}$, let

$$\mathrm{d}_q(\boldsymbol{a}, \boldsymbol{b}) = \mathrm{d}(\boldsymbol{a}, b_{q+1} b_{q+2} \cdots b_n b_1 \cdots b_q).$$

Let $\mathcal{D}_{cyc}(\boldsymbol{a}, \boldsymbol{b})$ denote the optimal cost of cyclic alignment of $\boldsymbol{a}$ and $\boldsymbol{b}$, i.e.

$$\mathcal{D}_{cyc}(\boldsymbol{a}, \boldsymbol{b}) = \min(\mathrm{d}_0(\boldsymbol{a}, \boldsymbol{b}), \mathrm{d}_1(\boldsymbol{a}, \boldsymbol{b}), \cdots, \mathrm{d}_{n-1}(\boldsymbol{a}, \boldsymbol{b})).$$

The simplest approach to find $\mathcal{D}_{cyc}(\boldsymbol{a}, \boldsymbol{b})$ is to rotate $\boldsymbol{b}$ one symbol at a time and realign each rotated string with $\boldsymbol{a}$ (cf. [5]). This brute-force approach has time complexity $\mathcal{O}(mn^2)$ as measured by the number of entries computed in dynamic programming matrices. The complexity becomes $\mathcal{O}(mn \log n)$ for all data by recursively partitioning a matrix "channel" established with initial width n by optimal alignment of $\boldsymbol{a}$ and $\boldsymbol{b}$ [7].

The technique for optimal solutions in this paper has data-dependent time complexity that can approach $\mathcal{O}(mn)$. The method is based on computing bounds on costs of aligning substrings of $\boldsymbol{a}$ and $\boldsymbol{b}$ using d values from $(m+1) \times (n+1)$ matrix M. Many applications of cyclic alignment involve strings with lengths on the order of several hundred symbols. For these applications, using values in M to reduce the number of matrix entries computed can substantially reduce relative run-time in practice. For example, Section V reports actual timing for biomolecular sequences ranging from 1.8 to 44.5% of the time for brute-force computations.

III. Substring Alignment Bounds

We use indices p and q to partition strings $\boldsymbol{a}$ and $\boldsymbol{b}$, respectively, into "head" and "tail" substrings. Let $\boldsymbol{a}_{1:p} = a_1 a_2 \cdots a_p$ and $\boldsymbol{a}_{p+1:m} = a_{p+1} a_{p+2} \cdots a_m$ denote head and tail substrings for $\boldsymbol{a}$ partitioned at p. Let $\boldsymbol{b}_{1:q} = b_1 b_2 \cdots b_q$ and $\boldsymbol{b}_{q+1:n} = b_{q+1} b_{q+2} \cdots b_n$ denote head and tail substrings for $\boldsymbol{b}$ partitioned at q. Let

$$\mathrm{d}_{pq}(\boldsymbol{a}, \boldsymbol{b}) = \mathrm{d}(\boldsymbol{a}_{1:p}, \boldsymbol{b}_{q+1:n}) + \mathrm{d}(\boldsymbol{a}_{p+1:m}, \boldsymbol{b}_{1:q})$$

i.e., $\mathrm{d}_{pq}(\boldsymbol{a}, \boldsymbol{b})$ is the optimal cost of aligning $\boldsymbol{a}$ partitioned at p with $\boldsymbol{b}$ partitioned and rotated at q. Note that an optimal cyclic alignment of $\boldsymbol{a}$ with $\boldsymbol{b}$ determines a specific value of q and associated values of p such that $\mathrm{d}_{pq}(\boldsymbol{a}, \boldsymbol{b}) = \mathcal{D}_{cyc}(\boldsymbol{a}, \boldsymbol{b})$. Essentially, the search for optimal cyclic alignment computes bounds on $\mathrm{d}(\boldsymbol{a}_{1:p}, \boldsymbol{b}_{q+1:n})$ and $\mathrm{d}(\boldsymbol{a}_{p+1:m}, \boldsymbol{b}_{1:q})$ while indexing through candidate values of (p, q).

Reference lower bounds on $\mathrm{d}(\boldsymbol{a}_{1:p}, \boldsymbol{b}_{q+1:n})$ and $\mathrm{d}(\boldsymbol{a}_{p+1:m}, \boldsymbol{b}_{1:q})$ are first established by comparing lengths of strings and frequencies of symbols without taking locations of symbols into account. Let SUB-MIN denote the minimum cost of any substitution. Let INDEL-MIN and INDEL-MAX denote the minimum and maximum costs, respectively, of any deletion or insertion. Aligning two strings, say $\boldsymbol{u}$ and $\boldsymbol{v}$, must incur cost of insertions or deletions to equalize string lengths. Minimal additional cost is obtained iff all symbols happen to be located in $\boldsymbol{u}$ and $\boldsymbol{v}$ for maximum matching. Let μ and ν represent strings $\boldsymbol{u}$ and $\boldsymbol{v}$ from which symbols that match one for one without regard for location are removed, and let $|\,|$ denote the length of a string; then, this lower bound on $\mathrm{d}(\boldsymbol{u}, \boldsymbol{v})$ is

$$\begin{aligned}\sigma(\boldsymbol{u}, \boldsymbol{v}) = {} & \min(|\mu|, |\nu|)\ \min(\text{SUB-MIN}, 2 \cdot \text{INDEL-MIN}) \\ & + \mathrm{abs}(|\mu| - |\nu|)\ \text{INDEL-MIN}.\end{aligned}$$

Applying lower bound σ to substrings for $\boldsymbol{a}$ partitioned at p and $\boldsymbol{b}$ partitioned and rotated at q gives

$$\begin{aligned}\sigma(\boldsymbol{a}_{1:p}, \boldsymbol{b}_{q+1:n}) &\leq \mathrm{d}(\boldsymbol{a}_{1:p}, \boldsymbol{b}_{q+1:n}) \\ \sigma(\boldsymbol{a}_{p+1:m}, \boldsymbol{b}_{1:q}) &\leq \mathrm{d}(\boldsymbol{a}_{p+1:m}, \boldsymbol{b}_{1:q})\end{aligned}$$

and establishes a lower bound on $\mathrm{d}_{pq}(\boldsymbol{a}, \boldsymbol{b})$:

$$\sigma(\boldsymbol{a}_{1:p}, \boldsymbol{b}_{q+1:n}) + \sigma(\boldsymbol{a}_{p+1:m}, \boldsymbol{b}_{1:q}) \leq \mathrm{d}_{pq}(\boldsymbol{a}, \boldsymbol{b}).$$

Additional bounds are based on actual alignment of $\boldsymbol{a}$ and $\boldsymbol{b}$. *Differential cost* is used in several places; therefore, we introduce operator Δ to refer to it. For any strings $\boldsymbol{u}$, $\boldsymbol{v}$, $\boldsymbol{w}$, and $\boldsymbol{x}$, let

$$\Delta(\boldsymbol{u}, \boldsymbol{v}; \boldsymbol{w}, \boldsymbol{x}) = \mathrm{d}(\boldsymbol{u}, \boldsymbol{v}) - \mathrm{d}(\boldsymbol{w}, \boldsymbol{x}).$$

For example, $\Delta(\boldsymbol{a}_{1:p}, \boldsymbol{b}; \boldsymbol{e}, \boldsymbol{b}_{1:q}) = \mathrm{d}(\boldsymbol{a}_{1:p}, \boldsymbol{b}) - \mathrm{d}(\boldsymbol{e}, \boldsymbol{b}_{1:q})$, where $\boldsymbol{e}$ is the empty string.

Proposition 3.1: $\Delta(\boldsymbol{a}_{1:p}, \boldsymbol{b}; \boldsymbol{e}, \boldsymbol{b}_{1:q})$ is a lower bound on $\mathrm{d}(\boldsymbol{a}_{1:p}, \boldsymbol{b}_{q+1:n})$ and $\Delta(\boldsymbol{a}, \boldsymbol{b}_{1:q}; \boldsymbol{a}_{1:p}, \boldsymbol{e})$ is a lower bound on $\mathrm{d}(\boldsymbol{a}_{p+1:m}, \boldsymbol{b}_{1:q})$.

Proof: Refer to Fig. 2. Note that $\mathrm{d}(\boldsymbol{a}_{1:p}, \boldsymbol{b})$ cannot exceed optimal cost of reaching location $(0, q)$ in matrix M, plus optimal cost of continuing from $(0, q)$ to (p, n). Optimal cost to reach $(0, q)$ is $\mathrm{d}(\boldsymbol{e}, \boldsymbol{b}_{1:q})$; optimal cost to continue to (p, n) is $\mathrm{d}(\boldsymbol{a}_{1:p}, \boldsymbol{b}_{q+1:n})$; therefore

$$\mathrm{d}(\boldsymbol{a}_{1:p}, \boldsymbol{b}) \quad \leq \quad \mathrm{d}(\boldsymbol{e}, \boldsymbol{b}_{1:q}) + \mathrm{d}(\boldsymbol{a}_{1:p}, \boldsymbol{b}_{q+1:n}).$$

Hence

$$\mathrm{d}(\boldsymbol{a}_{1:p}, \boldsymbol{b}) - \mathrm{d}(\boldsymbol{e}, \boldsymbol{b}_{1:q}) \quad \leq \quad \mathrm{d}(\boldsymbol{a}_{1:p}, \boldsymbol{b}_{q+1:n}).$$

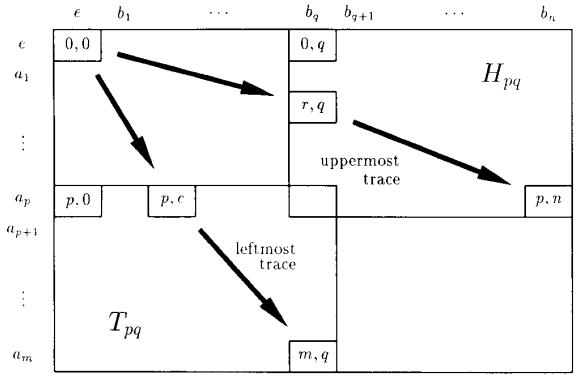


Fig. 2. Beginning and end points of uppermost trace in H_{pq} and leftmost trace in T_{pq}; the former trace has a cost of $\Delta(\boldsymbol{a}_{1:p}, \boldsymbol{b}; \boldsymbol{a}_{1:r}, \boldsymbol{b}_{1:q}) = \mathrm{d}(\boldsymbol{a}_{1:p}, \boldsymbol{b}) - \mathrm{d}(\boldsymbol{a}_{1:r}, \boldsymbol{b}_{1:q})$ and correspondingly for the latter $\Delta(\boldsymbol{a}, \boldsymbol{b}_{1:q}; \boldsymbol{a}_{1:p}, \boldsymbol{b}_{1:c}) = \mathrm{d}(\boldsymbol{a}, \boldsymbol{b}_{1:q}) - \mathrm{d}(\boldsymbol{a}_{1:p}, \boldsymbol{b}_{1:c})$.

Similarly, $\mathrm{d}(\boldsymbol{a}, \boldsymbol{b}_{1:q})$ cannot exceed the optimal cost of reaching $(p, 0)$ plus optimal cost of continuing from $(p, 0)$ to (m, q); therefore, we obtain

$$\mathrm{d}(\boldsymbol{a}, \boldsymbol{b}_{1:q}) - \mathrm{d}(\boldsymbol{a}_{1:p}, \boldsymbol{e}) \quad \leq \quad \mathrm{d}(\boldsymbol{a}_{p+1:m}, \boldsymbol{b}_{1:q}).$$

□

The proof above is a direct application of the general property that $\mathrm{d}(\boldsymbol{uv}, \boldsymbol{wx}) \leq \mathrm{d}(\boldsymbol{u}, \boldsymbol{w}) + \mathrm{d}(\boldsymbol{v}, \boldsymbol{x})$ for cost functions of the kind specified here. Since σ and Δ both establish lower bounds on the two terms of $\mathrm{d}_{pq}(\boldsymbol{a}, \boldsymbol{b})$, they can be combined into

$$\begin{aligned}\lambda_{pq}(\boldsymbol{a}, \boldsymbol{b}) \quad = \quad & \max(\sigma(\boldsymbol{a}_{1:p}, \boldsymbol{b}_{q+1:n}), \Delta(\boldsymbol{a}_{1:p}, \boldsymbol{b}; \boldsymbol{e}, \boldsymbol{b}_{1:q})) \\ & + \max(\sigma(\boldsymbol{a}_{p+1:m}, \boldsymbol{b}_{1:q}), \Delta(\boldsymbol{a}, \boldsymbol{b}_{1:q}; \boldsymbol{a}_{1:p}, \boldsymbol{e})).\end{aligned}$$

Corollary 3.1: $\lambda_{pq}(\boldsymbol{a}, \boldsymbol{b})$ is a lower bound on $\mathrm{d}_{pq}(\boldsymbol{a}, \boldsymbol{b})$. □

It is convenient to describe upper bounds by refering to two submatrices of M, which are known as H_{pq} and T_{pq}, that are associated with $\boldsymbol{a}$ partitioned at p and $\boldsymbol{b}$ partitioned at q, as shown in Fig. 2. Locations (p, n) and (m, q) in matrix M are the lower-right corners of submatrices H_{pq} and T_{pq}, which hold values $\mathrm{d}(\boldsymbol{a}_{1:p}, \boldsymbol{b})$ and $\mathrm{d}(\boldsymbol{a}, \boldsymbol{b}_{1:q})$, respectively. There is at least one optimal trace from $(0, 0)$ to (p, n) in M. Select one such trace that intersects column q at the smallest possible row number, say, row r, as in Fig. 2. The portion of this trace within H_{pq} is an *uppermost trace in* H_{pq}; it runs from (r, q) to (p, n), and its cost is $\Delta(\boldsymbol{a}_{1:p}, \boldsymbol{b}; \boldsymbol{a}_{1:r}, \boldsymbol{b}_{1:q})$. Similarly, there is at least one optimal trace from (0,0) to (m, q) in M. The portion of such a trace within T_{pq} intersecting row p at the smallest possible column number, say, column c, is a *leftmost trace in* T_{pq}; it runs from (p, c) to (m, q), and its cost is $\Delta(\boldsymbol{a}, \boldsymbol{b}_{1:q}; \boldsymbol{a}_{1:p}, \boldsymbol{b}_{1:c})$.

Let $\mathcal{R}_{pq}(\boldsymbol{a}, \boldsymbol{b}) = \Delta(\boldsymbol{a}_{1:p}, \boldsymbol{b}; \boldsymbol{a}_{1:r}, \boldsymbol{b}_{1:q}) + r \cdot$ INDEL-MAX and $\mathcal{C}_{pq}(\boldsymbol{a}, \boldsymbol{b}) = \Delta(\boldsymbol{a}, \boldsymbol{b}_{1:q}; \boldsymbol{a}_{1:p}, \boldsymbol{b}_{1:c}) + c \cdot$ INDEL-MAX..

Proposition 3.2: $\mathcal{R}_{pq}(\boldsymbol{a}, \boldsymbol{b})$ is an upper bound on $\mathrm{d}(\boldsymbol{a}_{1:p}, \boldsymbol{b}_{q+1:n})$, and $\mathcal{C}_{pq}(\boldsymbol{a}, \boldsymbol{b})$ is an upper bound on $\mathrm{d}(\boldsymbol{a}_{p+1:m}, \boldsymbol{b}_{1:q})$.

Proof: Refer to Fig. 2. An uppermost trace from (r, q) to (p, n) in H_{pq} is part of an (not necessarily optimal) alignment of $\boldsymbol{a}_{1:p}$ and $\boldsymbol{b}_{q+1:n}$ when extended by r deletions to reach $(0, q)$; hence, the optimal cost for aligning the two substrings cannot exceed the cost of an uppermost trace, which is $\Delta(\boldsymbol{a}_{1:p}, \boldsymbol{b}; \boldsymbol{a}_{1:r}, \boldsymbol{b}_{1:q})$, plus the cost of the deletions, which is at most $r \cdot$ INDEL-MAX:

$$\begin{aligned}\mathrm{d}(\boldsymbol{a}_{1:p}, \boldsymbol{b}_{q+1:n}) \quad \leq \quad & \Delta(\boldsymbol{a}_{1:p}, \boldsymbol{b}; \boldsymbol{a}_{1:r}, \boldsymbol{b}_{1:q}) \\ & + r \cdot \text{INDEL-MAX}.\end{aligned}$$

Analogously, for alignment of $\boldsymbol{a}_{p+1:m}$ and $\boldsymbol{b}_{1:q}$

$$\begin{aligned}\mathrm{d}(\boldsymbol{a}_{p+1:m}, \boldsymbol{b}_{1:q}) \quad \leq \quad & \Delta(\boldsymbol{a}, \boldsymbol{b}_{1:q}; \boldsymbol{a}_{1:p}, \boldsymbol{b}_{1:c}) \\ & + c \cdot \text{INDEL-MAX}.\end{aligned}$$

□

Note that if $r = 0$ or $c = 0$, then an upper bound coincides with the corresponding lower bound, and at least one of the two terms of $\mathrm{d}_{pq}(\boldsymbol{a}, \boldsymbol{b})$ is known exactly. Let $v_{pq}(\boldsymbol{a}, \boldsymbol{b}) = \mathcal{R}_{pq}(\boldsymbol{a}, \boldsymbol{b}) + \mathcal{C}_{pq}(\boldsymbol{a}, \boldsymbol{b})$.

Corollary 3.2: $v_{pq}(\boldsymbol{a}, \boldsymbol{b})$ is an upper bound on $\mathrm{d}_{pq}(\boldsymbol{a}, \boldsymbol{b})$. □

In an implementation, upper and lower bounds are used in the following way. For a given q, bounds are computed to establish interval $[p_{\min}, p_{\max}]$ as the smallest range on p such that $\mathrm{d}_{pq}(\boldsymbol{a}, \boldsymbol{b})$ could be optimal, i.e., $p \notin [p_{\min}, p_{\max}]$ implies that $\mathrm{d}_{pq}(\boldsymbol{a}, \boldsymbol{b})$ exceeds the lowest cost currently known to be achievable for cyclic alignment of $\boldsymbol{a}$ and $\boldsymbol{b}$. One possible outcome is that for every p, $\mathrm{d}_{pq}(\boldsymbol{a}, \boldsymbol{b})$ exceeds this lowest cost, in which case, $[p_{\min}, p_{\max}]$ is null, and q is eliminated as a candidate for optimal shift. If q remains viable, its $p_{\min}$ and $p_{\max}$ reduce the number of matrix entries needed to compute $\mathrm{d}_q(\boldsymbol{a}, \boldsymbol{b})$. Specifically, matrix entries indicated in Fig. 3 need not be computed because the various optimal traces noted in the figure may intersect, i.e., may share matrix entries, but need never cross. Thus, $\mathrm{d}_q(\boldsymbol{a}, \boldsymbol{b})$ can be computed in worst-case time $\mathcal{O}(p_{\max}(n - q) + (m - p_{\min})q)$.

IV. Guided Search Algorithm

The bounds developed in the previous section are used in a guided search algorithm to find optimal cyclic alignment of $\boldsymbol{a}$ and $\boldsymbol{b}$. These bounds are used to maintain a search list Q[ ] that holds information about candidate alignment traces, i.e., it holds specific (p, q) pairs that refer to the alignment of specific head-and-tail substrings of $\boldsymbol{a}$ and $\boldsymbol{b}$. A trace is a candidate iff its lower bound is less than or equal to the minimum upper bound for any candidate trace. New candidate traces are added to the list, and old traces that can no longer be optimal are removed. When the set of candidate traces is exhausted, optimal costs for aligning $\boldsymbol{a}$ and the shifted versions of $\boldsymbol{b}$ represented by (p, q) pairs in Q[ ] are computed by dynamic programming.

Fig. 4 lists a pseudo-code description of the algorithm. The following description includes comments on tradeoffs between computation time complexity and memory requirements.

A. Building Search List Q[ ]

Building search list Q[ ] takes places as follows:

- (Fig. 4, Steps 1 and 2) Q[ ] is initialized with $\mathrm{d}(\boldsymbol{a}, \boldsymbol{b})$, which also serves as an (initial) minimum upper bound on any $\mathrm{d}_{pq}(\boldsymbol{a}, \boldsymbol{b})$. The algorithm then loops through all combinations of p and q for $1 \leq p \leq m - 1$ and $1 \leq q \leq n - 1$.
- (Fig. 4, Steps 3 and 4) For a specific (p, q) pair, if the lower bound on $\mathrm{d}_{pq}(\boldsymbol{a}, \boldsymbol{b})$ is strictly greater than

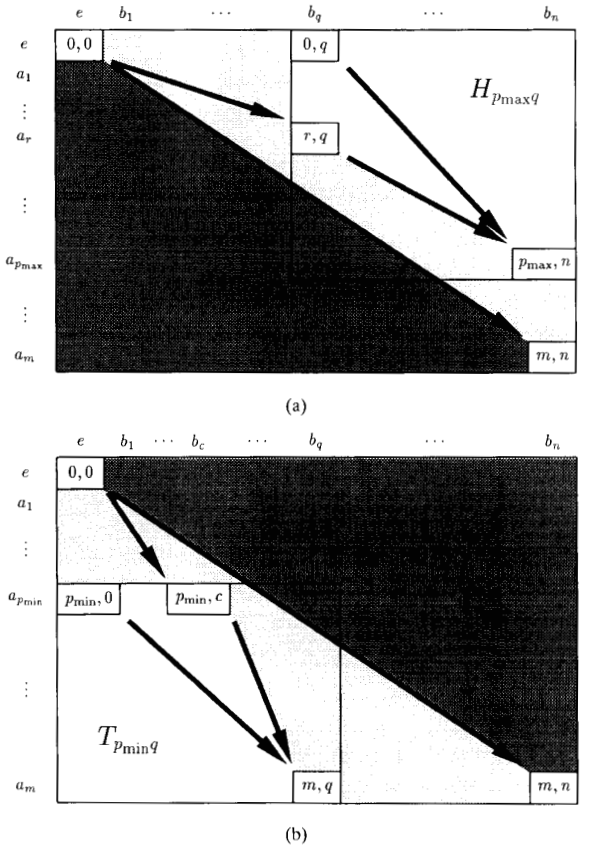


Fig. 3. Stylized outline of computational savings obtainable for realignment of $\boldsymbol{a}$ with $\boldsymbol{b}$ cyclically shifted to start at b_{q+1}. Bold arcs represent optimal traces. Matrix locations in dark gray are eliminated by optimal trace for $\boldsymbol{a}$ aligned with $\boldsymbol{b}$ unshifted. Additional locations in light gray are eliminated for a given q by its $[p_{\min}, p_{\max}]$ range.

Algorithm Guided Search

```
/* Build search list Q[ ] */
1  Initialize Q[ ] with λ00(a,b) = υ00(a,b) = d(a,b),
   and set υmin = υ00(a,b).
2  for (q = 1 ; q ≤ n − 1 ; q = q + 1) {
       for (p = 1 ; p ≤ m − 1 ; p = p + 1) {
3          Compute λpq(a,b) =
               max(σ(a1:p, bq+1:n), Δ(a1:p, b; e, b1:q)) +
               max(σ(ap+1:m, b1:q), Δ(a, b1:q; a1:p, e)).
4          if (λpq(a,b) > υmin)
               Break loop and continue with p = p + 1.
5          Add element (p,q) to Q[ ] including λpq(a,b)
           and υpq(a,b) = Rpq(a,b) + Cpq(a,b).
6          if (υpq(a,b) < υmin)
               Remove elements (s,t) from Q[ ] for which
               λst(a,b) > υpq(a,b) and set υmin = υpq(a,b).
       }
   }
   /* Find optimal cost and rotation */
7  while (Q[ ] is not empty) {
8      Extract P[ ] = {(p,q) ∈ Q[ ] | q = Qt → q}
       where Qt is a ptr to the top element in Q[ ].
9      Set pmin = min p and pmax = max p for p ∈ P[ ].
10     Compute πq(a,b) = Σp(n − q) + Σ(m − p)q for
       unknown cost substring alignments in P[ ].
11     if (πq(a,b) ≥ pmax(n − q) + (m − pmin)q)
           Align a and b using pmin, pmax and q.
12     else if (πq(a,b) ≠ 0)
           Align substrings with unknown cost only.
13     Set dq(a,b) = min dpq(a,b) for (p,q) ∈ P[ ].
14     if (dq(a,b) < υmin)
           Remove elements (s,t) from Q[ ] for which
           λpq(a,b) > dq(a,b) and set υmin = dq(a,b).
   }
15 Solution: Dcyc(a,b) = υmin with b cyclically shifted
   to start at {q + 1 | dq(a,b) = υmin}.
```

Fig. 4. Pseudocode description of guided search algorithm.

the current minimum upper bound, the trace in question cannot be optimal and is discarded.

- (Fig. 4, Steps 5 and 6) If not discarded, the (p,q) pair is added to Q[ ] in terms of the lower and upper bounds on $\mathrm{d}_{pq}(\boldsymbol{a},\boldsymbol{b})$; it is then tested to see if the minimum upper bound can be lowered with (p,q) pairs removed from Q[ ] accordingly.

The most complex part of the lower bound computations are the σ costs, all of which takes total time $\mathcal{O}((m+n)mn)$. Tabulating character counts for all possible head-and-tail substrings of $\boldsymbol{a}$ and $\boldsymbol{b}$ reduces the computational complexity to $\mathcal{O}(mn)$. The tradeoff is extra memory usage in the order of $\mathcal{O}(m+n)$.

For the upper bound computations, repetitive backtracing of both upper-most and left-most traces will take time $\mathcal{O}((m+n)mn)$ in the worst case. By tabulating row and column intersections for all upper-most traces prior to forming Q[ ] at worst-case complexity $\mathcal{O}(\sum_{p=1}^{m-1}(p+n))$ and sequentially tabulating row and column intersections for the (unique) left-most trace whenever q changes at worst-case complexity $\mathcal{O}(\sum_{q=1}^{n-1}(q+m))$, the upper bounds compute in, at most, time $\mathcal{O}(m^2+n^2)$ using $\mathcal{O}(mn)$ extra memory. When comparing complexities in practice, it should be noted that an elementary operation in this backtracing is itself simpler than, e.g., computing a value in a dynamic programming matrix.

Adding and removing trace information to and from search list Q[ ] might also be very time consuming. By building Q[ ] as a linked list and inserting new (p,q) pairs according to increasing values of the lower bounds, Q[ ] is updated in linear time. For an integer-valued cost function, the finite number of lower bounds $(\sigma(\boldsymbol{a},\boldsymbol{b}),\ldots,\mathrm{d}(\boldsymbol{a},\boldsymbol{b}))$ allows Q[ ] to be partitioned into sublists. Using a pointer to the last element in each sublist reduces maintenance of Q[ ] to constant time at very low memory cost.

Finally, in an actual implementation, new information about bounds on $\mathrm{d}_{pq}(\boldsymbol{a},\boldsymbol{b})$ for a (p,q) pair is taken into consideration immediately. The σ cost, for example, is two independent computations, one of which is the difference in length between two substrings; it often occurs that a (p,q) pair can be discarded on that basis alone, in which case, further gathering of evidence of suboptimality is unnecessary.

B. Computing Optimal Costs

When (p,q) combinations are exhausted, search list Q[ ] holds information on the set of shifts of $\boldsymbol{b}$ for which there might be an optimal alignment; each q value may be represented by one or more (p,q) pairs.

- (Fig. 4, Steps 7 and 8) Although Q[ ] is not empty, all (p,q) pairs with the same q value as the top element in Q[ ] are extracted by superimposing a new linked list P[ ] on Q[ ].

Extracting a (p, q) pair from Q[ ] involves removing it therefrom and relinking its neighbor elements in both lists. If the optimal alignment cost is known for a trace corresponding to a (p, q) pair, then (p, q) pairs still to be extracted from Q[ ] are removed but not added to P[ ]. If the cost function is integer valued, the above-mentioned list of pointers to each lower bound's last element in Q[ ] allow fastforward search for elements to be extracted. For a specific lower bound sublist of Q[ ], when a q encountered is greater than the one in question, there are no more (p, q) pairs of interest until the next lower bound sublist starts.

- (Fig. 4, Step 9) Minimum and maximum p values $p_{\min}$ and $p_{\max}$ are determined for (p, q) pairs in P[ ].
- (Fig. 4, Steps 10–13) Before computing $\mathrm{d}_q(\boldsymbol{a}, \boldsymbol{b})$, the computational complexity for aligning substring pairs for which optimal alignment is not known is compared with that of full alignment using $p_{\min}$ and $p_{\max}$. The less costly method is chosen.
- (Fig. 4, Step 14) If the minimum upper bound can be lowered on the basis of $\mathrm{d}_q(\boldsymbol{a}, \boldsymbol{b})$, then (p, q) pairs are removed from Q[ ] accordingly.
- (Fig. 4, Step 15) When Q[ ] is empty, $\mathcal{D}_{cyc}(\boldsymbol{a}, \boldsymbol{b})$ is equal to the final minimum upper bound, and the corresponding start positions are those shifts for which that cost is obtained.

In summary, as partitions of $\boldsymbol{a}$ and shifts of $\boldsymbol{b}$ are examined, the results given earlier ensure that a candidate solution is rejected iff that solution proves to be worse than the best solutions that are currently known. It follows that the guided search algorithm finds all shifts yielding optimal cyclic alignments of $\boldsymbol{a}$ and $\boldsymbol{b}$.

V. Examples of Biological Sequences

In molecular biology, sequence comparison is applied to nucleic acids and to proteins to study the evolution, structure, and function of different molecules [8]. The two examples given here illustrate the performance of the guided search algorithm. For comparison, the search is fully evaluated for each shift of the test sequences as the initial ("unshifted") string to indicate the impact of the initial alignment on performance.

The software is implemented in C and run under Unix on a Sparc Station 2.

A. Repetitive DNA Sequences

Two nucleotide sequences of satellite DNA molecules from the fruitfly constitute the first example. One sequence contains a repeated sequence that is 359 base pairs (bp) in length [6], and the other contains a 254-bp-long repeated sequence [2]. Fig. 5 shows one strand of each of the two sequences written in the four-letter alphabet {A, C, G, T} used for DNA, with X added to denote an unknown (uncertain) nucleic acid [4]. The 254-bp unit is homologous to the 359-bp repeat, except for a long sequence in the middle of the latter. Let the 359-bp sequence $\boldsymbol{a}$ be written as $\boldsymbol{a} = \boldsymbol{a}_1\boldsymbol{a}_2\boldsymbol{a}_3$ with meta substrings $\boldsymbol{a}_1$, $\boldsymbol{a}_2$, and $\boldsymbol{a}_3$, and let the 254-bp sequence $\boldsymbol{b}$ be written as

```
CCACATTTTGCAAATTTTGATGACCCCCCTCCTTACAAAAAATGCGAAAATTGATCCAAAAATT
AATTTCCCTAAATCCTTCAAAAAGTAATAGGGATCGTTAGCACTGGTAATTAGCTGCTCAAAAC
AGATATTCGTACATCTATGTGACCATTTTTAGCCAAGTTATAACGAAAATTTCGTTTGTAAATA
TCCACTTTTTTGCAGAGTCTGTTTTTCCAAATTTCGGTCATCAAATAATCATTTATTTTGCCAC
AACATAAAAAATAATTGTCTGAATATGGAATGTCATATCTCACTGAGCTCGTAATAAAATTTCC
AATCAAACTGTGTTCAAAAATGGAAATTAAATTTTTTGG
```

(a)

```
CAXATTTGCAAATTTAATGAACCCCCCTTCAAAAAATGCGAAAATTAACGCAAAAATTGATTTC
CCTAAATCCTTCAAAAAGTAAATAACAACTTTTTGGCAAAATCTGATTCCCTAATTTCGGTCAT
TAAATAATCAGTTTTTTTGCCACAACTTTAAAAATAATTGTCTGAATATGGAATGTCATACCTC
GCXXAGCTXGTAATTAAATTTCCAATGAAACTGTGTTCAACAATGAAAATTACATTTTTCGG
```

(b)

Fig. 5. Satellite DNA repeats: (a) 359-base pair sequence, and (b) the 254-base pair sequence.

TABLE I
Five String Combinations Tested Using the DNA Sequences

	Meta-strings		String Lengths	
Label	Prototype	Test	Prototype	Test
S_1	$\boldsymbol{a}_1\boldsymbol{a}_2\boldsymbol{a}_3$	$\boldsymbol{b}_1\boldsymbol{b}_2$	359	254
S_2	$\boldsymbol{a}_1\boldsymbol{a}_3$	$\boldsymbol{b}_1\boldsymbol{b}_2$	261	254
S_3	$\boldsymbol{a}_3\boldsymbol{a}_1$	$\boldsymbol{b}_2\boldsymbol{b}_1$	261	254
S_4	$\boldsymbol{a}_1\boldsymbol{a}_2\boldsymbol{a}_3$	$\boldsymbol{a}_1\boldsymbol{a}_2\boldsymbol{a}_3$	359	359
S_5	$\boldsymbol{b}_1\boldsymbol{b}_2$	$\boldsymbol{b}_1\boldsymbol{b}_2$	254	254

$\boldsymbol{b} = \boldsymbol{b}_1\boldsymbol{b}_2$ with meta substrings $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$. Alignment of $\boldsymbol{a}$ and $\boldsymbol{b}$ then corresponds to aligning $\boldsymbol{a}_1$ and $\boldsymbol{b}_1$, $\boldsymbol{a}_3$ and $\boldsymbol{b}_2$, and deleting $\boldsymbol{a}_2$. The position of the deletion sequence is due only to the cutting point of the enzyme used to isolate one period of both repeats. In fact, sequence analysis using edit costs of 0 for a match, 1 for a substitution, and 2 for both deletion and insertion (the same d as used here) suggests that a more intuitive sequencing would be $\boldsymbol{a} = \boldsymbol{a}_3\boldsymbol{a}_1\boldsymbol{a}_2$ and $\boldsymbol{b} = \boldsymbol{b}_2\boldsymbol{b}_1$ such that $\boldsymbol{a}$ and $\boldsymbol{b}$ have one large interval rather than two separate intervals in common [4].

Performance of the guided search algorithm is evaluated for five different string combinations S_1-S_5, whose characteristics are listed in Table I.

A summary of timing and computation statistics is given in Table II; timing results are relative to simple brute-force shift and realignment, and $r(q)$ and $r(p)$ are the relative number of rejected shifts and (p, q) pairs, respectively; p/q is the relative number of (p, q) pairs left per shift not rejected; mn/q is the relative number of matrix locations actually computed per realignment (cf. Fig. 3). Note that the average computation statistics listed for S_4 and S_5 all refer to a single (p, q) pair left after building the search list.

On average, the optimal alignment is found for S_1 in about one third the time of the brute-force method. This ratio is halved for S_2 and S_3 and reduced more for S_4 and S_5, as expected. More detailed timing is given in Fig. 6, which shows relative timing results as a function of the relative position of the correct start symbol in the test string; for every plot, each data point is obtained from a full guided search. It is interesting that for S_1-S_3, best performance is not obtained when the initial alignment itself is optimal. Further, even though S_2 and S_3 both align with the same cost and correspond to exactly the same symbol sequences (only one set of strings is cyclically shifted with respect to the other), their timing profiles are quite different due to the impact of the initial alignment.

TABLE II
DNA TIMING AND AVERAGE COMPUTATION STATISTICS

	Opt. Align.		Timing (%)			Avrg Comp Statistics (%)			
Label	Cost	q	μ	min	max	$r(q)$	$r(p)$	p/q	mn/q
S_1	233	1	32.6	24.9	49.2	33.1	79.6	29.9	40.1
S_2	45	1	17.1	8.8	32.3	60.6	95.9	9.0	31.0
S_3	45	1	15.8	9.0	25.9	62.6	95.9	10.1	29.4
S_4	0	1	2.6	1.8	3.8	99.7	100.0	0.3	0.0
S_5	0	1	1.8	1.3	2.7	99.6	100.0	0.4	0.0

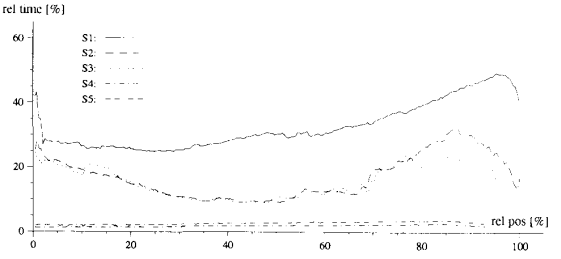


Fig. 6. DNA timing results measured relative to simple rotation and realignment.

Speed-up is a result of several factors. Many candidate q's, and even more (p, q) pairs, are discarded as suboptimal without computing their true alignment costs. For example, for S_1, approximately one third of the rejected traces are found to be suboptimal by the simple length test σ applied to $\boldsymbol{a}_{1:p}$ and $\boldsymbol{b}_{q+1:n}$, whereas another third are rejected when adding Δ costs for $\boldsymbol{a}_{p+1:m}$ and $\boldsymbol{b}_{1:q}$ to the lower bound computed for the other pair of substrings. The rejection pattern is different for S_2 and S_3 for which more than half of the traces are discarded merely by testing Δ costs and substring lengths for $\boldsymbol{a}_{1:p}$ and $\boldsymbol{b}_{q+1:n}$. The small p/q values show that the number of candidate traces is greatly reduced, and therefore, only part of the full matrix is computed for a realignment, as illustrated by the mn/q numbers. For S_4 and S_5, only a small fraction of the traces persist beyond the tests for $\boldsymbol{a}_{1:p}$ and $\boldsymbol{b}_{q+1:n}$, and no realignments take place.

B. Circularly Permuted Protein Sequences

The second example involves alignment of two amino acid sequences from the plant proteins concanavalin A (Con A) and favin. Con A is composed of a single chain that is 237 residues long; favin contains an alpha chain of 51 amino acid residues and a beta chain of 185 residues [3]. The two proteins are unique in the sense that they are evolutionarily related, but an unusual genetic event has resulted in cyclic permutation of the homologous sequences. Fig. 7 schematically illustrates the optimal alignment of favin alpha and beta chains with Con A; the sequences are similar with the end of favin beta chain resembling the beginning of Con A and vice versa [3].

Protein is described by a 20-letter alphabet of which each letter represents an amino acid corresponding to a specific set of transcribed DNA base triplets [8]. Therefore, for metric analysis of amino acid sequences, a highly specialized cost function d_1 may be defined to correlate individual edit costs with the number of base changes needed to interconvert two amino acids: the cost is 0 for match, varies between 1 and 3 for substitution, and is 3 for both deletion and insertion [4].

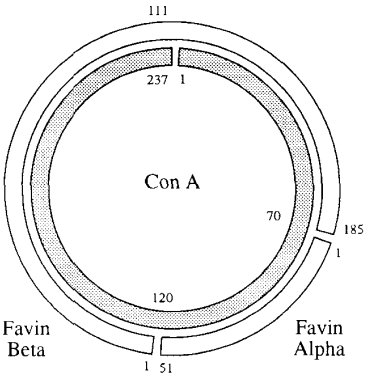


Fig. 7. Schematic drawing of the alignment of favin alpha and beta chains (open bars) with Con A (solid bar) showing the circular shift that is globally optimal.

TABLE III
PROTEIN TIMING AND AVERAGE COMPUTATION STATISTICS

	Opt. Align.		Timing (%)			Avrg Comp Statistics (%)			
Label	Cost	q	μ	min	max	$r(q)$	$r(p)$	p/q	mn/q
d_1	226	163	40.5	32.2	54.1	25.1	82.8	22.3	41.2
d_2	167	163-64	40.6	32.5	56.7	24.8	83.0	21.9	39.5
d_3	149	163-64, 166-68	44.5	32.9	65.0	31.0	74.5	36.1	41.0

The guided search algorithm is evaluated using three different cost functions. d_1 uses the above edit costs. d_2 and d_3 are both based on fixed edit costs. d_2 assigns cost 0 to a match, 1 to a substitution, and 2 to both deletion and insertion, and d_3 assigns cost 0 to a match and 1 to all other edits.

Table III summarizes timing and computation statistics similarly to Table II. d_2 and d_3 result in alignment costs much lower than d_1 because approximately half the edit operations are substitutions; yet biochemically, the sequences are said to be alike. The three cost functions yield a variety of optimal solutions, but all agree that a shift of the test string of 163 symbols is optimal. If the edit costs are more alike, the confusion about the true amount of shift is greater. The functions show approximately the same relative average timing performance but differ somewhat on the maximum time spent in computation.

Cost functions d_1 and d_2 have similar rejection patterns of candidate traces. Overall, fewer traces are rejected for d_3, although more shifts are ruled out compared with both d_1 and d_2. In all three cases, about 40% of the full matrix is computed for realignments.

Fig. 8 gives relative timing results as a function of the relative position of the correct start symbol in the test string. Compared with the DNA timing profiles in Fig. 6, the protein timing profiles are more intuitive; the worst case is when the test string is shifted to have the start symbol located in its middle.

VI. CONCLUSION

This paper develops a guided search algorithm for the optimal string-to-string alignment problem extended to include all

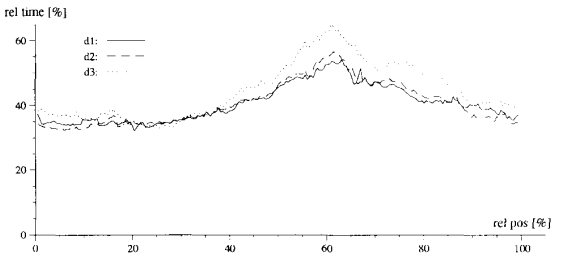


Fig. 8. Protein timing results measured relative to simple rotation and realignment.

cyclic shifts. Applied to two sets of biological sequences, the algorithm is substantially faster than brute-force computation.

ACKNOWLEDGMENT

The authors thank the anonymous referees for valuable comments that improved the paper significantly.

REFERENCES

[1] H. Bunke and A. Sanfelieu, Eds. *Syntactic and Structural Pattern Recognition Theory and Applications.* Singapore: World Scientific, 1990.

[2] M. Carlson and D. Brutlag, "Different regions of a complex satellite DNA vary in size and sequence of the repeating unit," *J. Molecular Biol.*, vol. 135, pp. 483–500, 1979.

[3] B. A. Cunningham, J. F. Hemperly, T. P. Hopp, and G. M. Edelman, "Favin versus concanavalin A: Circularly permuted amino acid sequences," in *Proc. Nat. Acad. Sci.,*, 1979, pp. 3218–3222, vol. 76.

[4] B. W. Erickson and P. H. Sellers, "Recognition of patterns in genetic sequences," in D. Sankoff and J. B. Kruskal, *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparisons.* Reading, MA: Addison-Wesley, 1983, pp. 55–90.

[5] K. S. Fu and S. Y. Lu, "Size normalization and pattern orientation problems in syntactic clustering," *IEEE Trans. Syst. Man Cybern.*, vol. 9, pp. 55–58, 1979.

[6] T. Hsieh and D. Brutlag, "Sequence and sequence variation within the 1·688 g/cm^3 satellite DNA of *drosophila melanogaster*," *J. Molecular Biol.*, vol. 135, pp. 465–481, 1979.

[7] M. Maes, "On a cyclic string-to-string correction problem," *Inform. Processing Lett.*, vol. 35, pp. 73–78, 1990.

[8] D. Sankoff and J. B. Kruskal, Eds. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparisons.* Reading, MA: Addison-Wesley, 1983.

[9] P. H. Sellers, "An algorithm for the distance between two finite sequences," *J. Combinatoric Theory*, vol. A16, pp. 253–258, 1974.

[10] W. -H. Tsai and S. -S. Yu, "Attributed string matching with merging for shape recognition,"*IEEE Trans. Patt. Anal. Machine Intell.*, vol. PAMI-7, pp. 453–462, 1985.

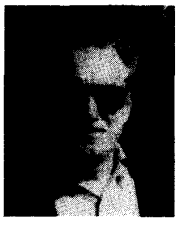
[11] R. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *J. Assoc. Comput. Machinery*, vol. 21, pp. 168–173, 1974.

[12] Y. P. Wang and T. Pavlidis, "Optimal correspondence of string subsequences," *IEEE Trans. Patt. Anal. Machine Intell.*, vol. 12, pp. 1080–1087, 1990.

Jens Gregor was born in Copenhagen, Denmark, on September 22, 1963. He received the M.S. degree in electrical engineering and the Ph.D. degree in technical science from the University of Aalborg, Denmark, in 1988 and 1991, respectively.

He is currently Assistant Professor of Computer Science at the University of Tennessee, Knoxville.

Dr. Gregor is a member of the Danish Pattern Recognition Society.



Michael G. Thomason (S'63-M'65-SM'83) received the B.S. degree from Clemson University, Clemson, SC, in 1965, the M.S. degree from Johns Hopkins University, Baltimore, MD, in 1970, and the Ph.D. degree from Duke University, Durham, NC, in 1973.

He worked at the Westinghouse Defense and Space Center, Baltimore, MD, has been a consultant for Perceptics Corp., Knoxville, TN, as well as other companies, and is currently Professor of Computer Science at the University of Tennessee, Knoxville. His research interests include structural pattern analysis and stochastic processes in computer science.

Dr. Thomason is a member of Sigma Xi, Tau Beta Pi, and the ACM.