An Overview of Sequence Comparison Algorithms in Molecular Biology*

Eugene W. Myers

TR 91-29

ABSTRACT

Molecular biologists frequently compare biosequences to see if any similarities can be found in the hope that what is true of one sequence either physically or functionally is true of its analogue. Such comparisons are made in a variety of ways, some via rigorous algorithms, others by manual means, and others by a combination of these two extremes. The topic of sequence comparison now has a rich history dating back over two decades. In this survey we review the now classic and most established technique: dynamic programming. Then a number of interesting variations of this basic problem are examined that are specifically motivated by applications in molecular biology. Finally, we close with a discussion of some of the most recent and future trends.

Key words: Approximate Match, Block Indel, Dynamic Programming, Local Similarity, Multiple Sequence Alignment, Similarity Search, Sequence Comparison

December 20, 1991

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported in part by the National Institutes of Health under Grant R01 LM04960-01 and the Aspen Center for Physics.

An Overview of Sequence Comparison Algorithms in Molecular Biology

1. The Basic Problem and Dynamic Programming Algorithm.

Molecular biologists frequently compare biosequences to see if any similarities can be found in the hope that what is true of one sequence either physically or functionally is true of its analogue. Generally, such comparisons involve aligning sections of the two sequences in a way that exposes the similarities between them. For example, consider the following **alignment** between a portion of the monkey somatotropin protein and a rainbow trout somatotropin precurser protein.

```
Sequence A:   LEPVQFLRSVFANSL-VYGTSYS
Sequence B:   EYPSQTL--IISNSLMV-RNA-N
```

In such alignments, **aligned pairs** of symbols are written one above the other. Symbols in one sequence that are not aligned with symbols in the other are written opposing the symbol ’-’. For example, the substring ’RS’ in sequence *A* constitutes a **gap** of two symbols. From the view of **maximizing similarity** one is intuitively seeking alignments that have few gaps and align symbols that are identical or whose denoted function is highly similar. The dual perspective is to consider sequence *A* as evolving into sequence *B* and here one seeks alignments that **minimize distance** or intuitively require a minimal number of ‘‘evolutionary’’ operations that effect the conversion. In this view, an aligned symbol of *B* is **substituted** for its counterpart in *A*, an unaligned symbol in *A* is **deleted**, and an unaligned symbol in *B* is **inserted**. For example ’E’ is substituted for ’L’ in the leftmost position, and the substring ’RS’ constitutes a deletion gap. In this view *A* is the source and *B* the result as contrasts the symmetry of the similarity perspective.

The notion of a best alignment requires some scoring or optimization criterion. Traditionally, a user supplies a scoring function δ that assigns a real-valued cost to each possible aligned pair. That is, if the sequences are over alphabet Σ, then $\delta(a, b)$ specifies the cost of substituting *a* for *b*, $\delta(a, -)$ specifies the cost of deleting *a*, and $\delta(-, b)$ specifies the cost of inserting *b*. If Σ is a small finite alphabet, as is true for proteins and nucleic acids, then typically δ is given by a $|\Sigma| + 1$ by $|\Sigma| + 1$ table or matrix of numbers. The score of an alignment between sequences *A* and *B* is the sum of the costs of the aligned pairs in it. In the minimal distance version of the problem one seeks the alignment with minimal score; in the similarity version one seeks the alignment with maximal score. As we shall see later, these two views are formally dual problems for the basic models and so for the moment we proceed with the evolutionary view. Our problem is: given sequences $A = a_1 a_2 \cdots a_M$ and $B = b_1 b_2 \cdots b_N$ and scoring function δ, find an alignment denoting a set of ‘‘evolutionary operations’’ that converts *A* to *B* and minimizes the sum of the operations’ costs.

There is an intuitive graph-theoretic formulation of the problem that is particularly illuminating. The **edit graph** $G_{A,B}$ for comparing sequences *A* and *B* is an edge-labeled directed graph. The vertices of $G_{A,B}$ are the pairs (i, j) where $i \in [0, M]$ and $j \in [0, N]$. Imagine these vertices arranged in an $M+1$ by $N+1$ grid. Distinguish $\Theta = (0, 0)$ as $G_{A,B}$’s source vertex, and $\Phi = (M, N)$ as its sink vertex. The following edges, and only these edges, are in $G_{A,B}$.

1. If $i \in [1, M]$ and $j \in [0, N]$, then there is a *deletion* edge $(i-1, j) \rightarrow (i, j)$ labeled $\begin{bmatrix} a_i \\ - \end{bmatrix}$.
2. If $i \in [0, M]$ and $j \in [1, N]$, then there is an *insertion* edge $(i, j-1) \rightarrow (i, j)$ labeled $\begin{bmatrix} - \\ b_j \end{bmatrix}$.
3. If $i \in [1, M]$ and $j \in [1, N]$, then there is a *substitution* edge $(i-1, j-1) \rightarrow (i, j)$ labeled $\begin{bmatrix} a_i \\ b_j \end{bmatrix}$.

Figure 1 provides an example of the construction.

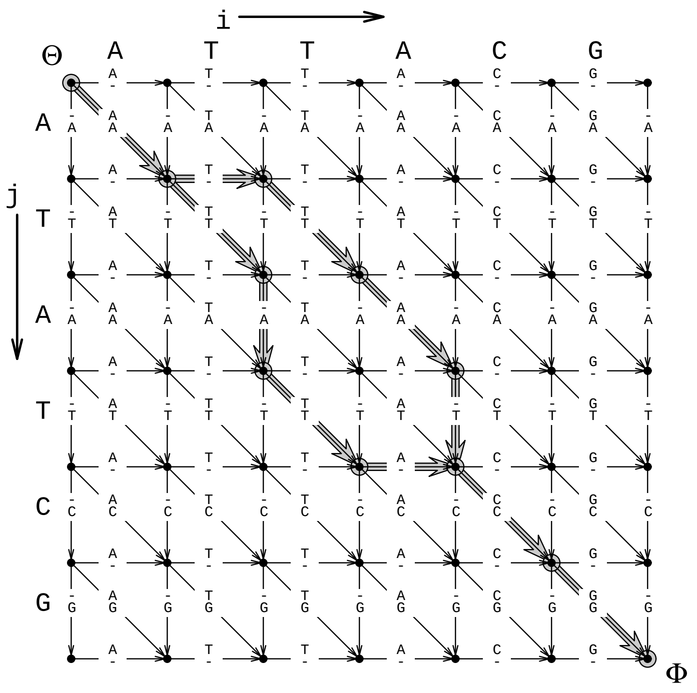

Figure 1: $G_{A,B}$ for $A = ab$ and $B = baa$.

Let A_i denote the i-symbol prefix of A, i.e., $A_0 = \varepsilon$ and $A_i = a_1 a_2 \cdots a_i$ for $i \in [1, M]$. Similarly, B_j is the j-symbol prefix of B. A path in $G_{A,B}$ is said to **spell** the alignment obtained by concatenating its edge labels. The central feature of the construction of the edit graph is that each path from Θ to (i, j) spells an alignment of A_i and B_j, and a different path spells a different alignment. To see this note that an alignment between A_i and B_j must end with one of the aligned pairs, $\begin{bmatrix} a_i \\ - \end{bmatrix}$, $\begin{bmatrix} - \\ b_j \end{bmatrix}$, or $\begin{bmatrix} a_i \\ b_j \end{bmatrix}$. Thus an alignment between A_i and B_j must be either (1) an alignment between A_{i-1} and B_j concatenated with $\begin{bmatrix} a_i \\ - \end{bmatrix}$, (2) an alignment between A_i and B_{j-1} concatenated with $\begin{bmatrix} - \\ b_j \end{bmatrix}$, or (3) an alignment between A_{i-1} and B_{j-1} concatenated with $\begin{bmatrix} a_i \\ b_j \end{bmatrix}$. This property of the construction thus implies that there is a one-to-one correspondence between paths in $G_{A,B}$ from Θ to Φ and alignments of A and B. In the parlance of computer science, $G_{A,B}$ is a **finite automaton** [HoU79] that accepts the set of all alignments between A and B.

Our goal is an algorithm that computes an optimal alignment of A and B. Consider weighting $G_{A,B}$ by assigning cost $\delta(a, b)$ to each edge labeled $\begin{bmatrix} a \\ b \end{bmatrix}$. Notice that the sum of the weights of a path's edges is exactly the sum of the scores of the aligned pairs in its corresponding alignment. Thus the problem reduces to computing a minimum-weight path from Θ to Φ in $G_{A,B}$ as weighted by δ. In the parlance of computer science, sequence comparison has been reduced a shortest paths problem [Joh77] over a graph with a highly regular structure.

Because $G_{A,B}$ is acyclic, a minimum-weight path can be determined in a single pass over its vertices, so long as they are taken in a **topological order**, i.e., an ordering of the vertices with the property that every edge is directed from a vertex to a successor in the ordering. One topological order for $G_{A,B}$'s vertices is to treat the rows in order, sweeping left to right within a row. Using this vertex ordering, the following procedure computes the score, $C(i,j)$, of a minimum-weight path from Θ to each vertex (i,j) where $i \in [0,M]$ and $j \in [0,N]$.

```
$C(0,0) \leftarrow 0$
for $j \leftarrow 1$ to $N$ do
    $C(0,j) \leftarrow C(0,j-1) + \delta(-, b_j)$
for $i \leftarrow 1$ to $M$ do
  {  $C(i,0) \leftarrow C(i-1,0) + \delta(a_i, -)$
     for $j \leftarrow 1$ to $N$ do
          $C(i,j) \leftarrow \min\{C(i-1,j) + \delta(a_i,-),\ C(i,j-1) + \delta(-,b_j),\ C(i-1,j-1) + \delta(a_i,b_j)\}$
  }
write "Difference score is" $C(M,N)$
```

Figure 2: The traditional dynamic programming algorithm.

Note that $C(i,j)$ is the cost of the best alignment between A_i and B_j. From yet another perspective, the matrix of C-values are defined by the recurrence:

$$C(i,j) = \begin{cases} \min\left\{\begin{array}{l} C(i-1,j-1) + \delta(a_i,b_j) \\ C(i-1,j) + \delta(a_i,-) \\ C(i,j-1) + \delta(-,b_j) \end{array}\right\} & \text{if } i > 0 \text{ and } j > 0 \\ C(i,j-1) + \delta(-,b_j) & \text{if } i = 0 \text{ and } j > 0 \\ C(i-1,j) + \delta(a_i,-) & \text{if } i > 0 \text{ and } j = 0 \\ 0 & \text{if } i = 0 \text{ and } j = 0 \end{cases}$$

The algorithm of Figure 2 can be viewed as simply evaluating C-values in an order consistent with the data-dependencies of the recurrence. This is the central **dynamic programming** recurrence for the basic sequence comparison problem. Frequently dynamic programming has been confused in this areas' literature as referring to this particular recurrence or one of its myriad minor variations. Here we emphasize that dynamic programming is a general computational paradigm of wide applicability [HoS78]. A problem is solved by dynamic programming if the answer can be efficiently determined by computing a tableau of optimal answers to progressively larger and larger subproblems. The **principle of optimality** requires that the optimal answer to a given subproblem be expressible in terms of optimal answers to ''smaller'' subproblems. Our basic sequence comparison problem does yield to this principle: the optimal answer for the problem of A_i versus B_j can be expressed in terms of optimal answers to the subproblems A_{i-1} vs. B_{j-1}, A_i vs. B_{j-1}, and A_{i-1} vs. B_j. The recurrence describes the relationship and gives rise to the procedure above for filling in the ''tableau'' C of answers to progressively larger problems.

The algorithm of Figure 2 only computes the score of a minimal cost alignment between A and B. Note that the use of the article 'a' is important as there can be more than one alignment achieving the minimal score. One or all of these optimal alignments can be recovered by tracing back from the final value $C(M, N)$ to a term in its 3-way minimum that yielded it, determining which term gave rise to that term, and so on back to the source. Unfortunately this requires saving the entire matrix C, giving an algorithm that takes $O(MN)$ space as well as time. Note that if one is interested only in the cost of an optimal alignment then $O(min(M, N))$ space suffices as one need keep only the previous row of the matrix C to compute the next row, thanks to the nature of the data-dependencies of the recurrence. Using such a cost-only algorithm as a sub-procedure, Hirschberg [Hir75] gave a divide and conquer algorithm that can determine an optimal alignment in $O(M+N)$ space. The divide step consists of finding the midpoint of an optimal source-to-sink path by running the cost-only algorithm on the first half of A and the reverse of its second half. The conquer step consists of determining the two halves of this path by recursively reapplying the divide step to the two halves. Hirschberg gave this procedure in the context of the simpler longest common subsequence problem, but Myers and Miller [MyM88,MiM88] have shown it to apply to most comparison algorithms that have linear space cost-only algorithms. This refinement is very important, since space, not time, is often the limiting factor when computing optimal alignments between large sequences. One can always run a program longer, but when you're out of memory you're out of business.

Let $D(A, B)$ be the score of a minimal cost alignment between sequences A and B. This score is termed the **generalized-Levenshtein distance** between sequence A and B and indeed forms a metric space over sequences if the underlying scoring function δ forms a metric space over the underlying alphabet [Sel74]. Thus calling this optimal score a distance is formally correct for a wide class of scoring schemes.

Another interesting and important property of all alignments between sequences A and B is the relation $2S+I = M+N$, where S is the number of substitutions and I is the number of **indels** (insertions and deletions, or unaligned symbols) in the alignment. This is easily seen in terms of the edit graph. Any source to sink path needs to move M grid spaces to the right, and N spaces down, for a total displacement of $M+N$ grid units. But each substitution counts for 2 units (one left and one down) and each indel counts for 1 unit (either left or down). Thus a path with S substitutions and I indels, moves $2S+I$ units and the identity follows.

This identity leads to a simple scaling result for scoring schemas. Given function δ and real-valued parameters X and Y, let δ' be defined as follows:

$$\delta'(a, b) = \begin{cases} X\delta(a, b) + 2Y & \text{if } a, b \in \Sigma \\ X\delta(a, b) + Y & \text{if } a = - \text{ or } b = - \end{cases}$$

If $C(\pi)$ is the score of alignment π under δ and $C'(\pi)$ is the score under δ' then we see that:

$$\begin{aligned} C'(\pi) &= \Sigma\{\ \delta'(a, b) : \begin{bmatrix} a \\ b \end{bmatrix} \in \pi\ \} \\ &= X\Sigma\{\ \delta(a, b) : \begin{bmatrix} a \\ b \end{bmatrix} \in \pi\ \} + (2YS + YI) \\ &= XC(\pi) + Y(M+N). \end{aligned}$$

Thus the transformation scales the score of every alignment by X and translates it by $Y(M+N)$. So it follows that the minimum scoring alignments are still minimal after the transform and that $D'(A, B) = XD(A, B) + Y(M+N)$ as long as X is positive. Thus one can scale and translate scoring schemes to values that are convenient or aesthetic without changing which alignments are optimal. For example, a floating point δ can be scaled so that its embedding into the integers is ''reasonably snug'', the goal being an integer program that is much faster than its floating point analogue on typical computers.

Reconsider now the similarity concept of comparison and suppose $S(A, B)$ is the score of the *maximum* scoring alignment. Observe that if X is negative, then $D'(A, B) = XS(A, B) + Y(M+N)$. Thus, one can map a similarity problem into a distance problem just by negatively scaling the underlying scoring scheme. For example in the simple case where $X = -1$ and $Y = 0$, one negates the scores of the similarity scheme, computes the minimum cost alignment, and reports that its negation is the score of the maximum cost alignment under the original scheme. If one wants the δ' to be positive for every symbol pair (as required by some specialized algorithms) then just pick an appropriately large translation factor Y. One may thus conclude that the similarity and distance problems are duals [SWF81].

Within the field of computer science, comparison problems arose somewhat later than in molecular biology and in a somewhat simpler incarnation. The **longest common subsequence** problem is to find an alignment that maximizes the number of identical aligned pairs, i.e., $\delta(a, b) = 1$ if $a = b$ and 0 otherwise. Its traditional dual, the **shortest edit script** problem seeks an alignment with the fewest number of indels and non-identical substitutions, i.e. $\delta(a, b) = 0$ if $a = b$ and 1 otherwise. These problems arose in the context of applications such as comparing the contents of computer files and correcting the spelling of words. What is of particular interest here is that the unit-cost and discrete integer nature of the underlying cost functions offers algorithmic leverage not obtainable in the case of generalized Levenshtein measures. In the next few paragraphs we review the history of progress by computer scientists on this special version of the comparison problem. Describing the complexity of the algorithms is simplified by assuming that $M \le N$ without loss of generality.

In 1977 Hunt and Szymanski [HuS77] devised a comparison algorithm for the lcs problem that required $O(R \log\log N)$ time where R is the number of **matchpoints** or pairs of matching symbols from each sequence, i.e., $|\{(i, j) : a_i = b_j\}|$. While the parameter R is potentially $\Omega(MN)$, it is significantly smaller for many applications. For example, when comparing computer files each is viewed as a sequence of lines and R is the pairs of matching lines. Apart from blank lines, most are unique and R is usually on the order of $O(M+N)$. On the other hand when comparing DNA sequences where there are only four distinct symbols, R is at least $MN/4$. Thus while the algorithm does not improve on the complexity of the problem in terms of the input parameters M and N it is quite effective in those applications that are sparse in terms of the parameter R.

A few years later, in 1980, Masek and Paterson [MaP80] devised an algorithm using what is called the **Four-Russian's paradigm** that compares two sequences over a finite alphabet in $O(MN/\log^2 N)$ time.†
This result is very interesting for it takes less than quadratic worst-case time in terms of the input parameters M and N, and to date it is the only algorithm that has this property. Somewhat earlier, Aho, Hirschberg, and Ullman [AHU76] had shown that any algorithm that can only compare symbols to see if they are equal or unequal must take at least $O(MN)$ time. On the other hand, when one assumes that symbols are ordered and can be compared in this order, the best lower bound is only $O(N\log N)$ [Hir78]. Thus better-than-quadratic techniques using arithmetic on the symbols have not been ruled out and this result confirmed this was possible. No one has done better to date.

More recently, Ukkonen and Myers [Ukk85a,Mye86a] independently arrived at an $O(DN)$ algorithm for the shortest edit script problem where D is the number of indels and substitutions in the optimal alignment (i.e., $D = D(A, B)$). The parameter D characterizes the size of the output and so this algorithm is termed **output sensitive**. The more similar the sequences, the more efficient the algorithm becomes. On

† The knowledgeable reader may recognize that the result as stated by the authors was a log-factor slower. They assumed a model of computation where operations take time proportional to their bit-length. To be consistent with the rest of the results discussed here, we assume the uniform cost model where word-sized operations are assumed to take constant time.

the other hand, D may be as large as $O(M+N)$ and so this algorithm also does not improve on quadratic complexity in terms of the input parameters M and N. Nonetheless, it is very simple and Myers showed that it performs in $O(N+D^2)$ time in expectation. Moreover, using complex suffix tree and constant time least common ancestor algorithms as subprocedures, Myers produced a variation that takes $O(NlgN + D^2)$ worst-case time.

Some very recent results return to the sparse problem concept embodied by the parameter R of the Hunt and Szymanski algorithm. Wilber and Lipman [WiL83] first posed the following variation but only gave an efficient heuristic for the problem. For very small k consider the set of all k-substrings pairs between the two sequences, i.e., $\{ (i, j) : a_i a_{i+1} \cdots a_{i+k-1} = b_j b_{j+1} \cdots b_{j+k-1} \}$. Let R be the size of this set; we seek the best alignments among those whose aligned, equal symbols come from one of these substring pairs. As k is increased, the optimal restricted alignment becomes progressively less optimal than the unrestricted optimum, but the size of R rapidly becomes smaller. Moreover, the substring pairs can rapidly be computed in $O(R)$ time. In 1990, Eppstein, et. al. [EGG90], devised an algorithm for this **sparse variation** of the problem in $O(R \log \log N)$ time. For k = 2, 3, or 4 this algorithm produces alignments very near the unrestricted optimum very efficiently as R is quite small, i.e., about $MN/4^k$ in expectation for DNA sequences. Note that their work generalizes that of Hunt and Szymanski (i.e., the case k = 1). In another direction of generalization, Manber and Wu [MaW90], produced an algorithm with the same complexity that can accommodate indel scores that are arbitrary real-values.

While the algorithms above cannot be extended to handle general Levenshtein scoring schemes, they do present a panoply of algorithmic lines of attack that are worthy of study. Some of these algorithms do extend to the case of small integer scores with and without additional restrictions. But for the general problem, there has been only one potentially useful algorithmic result. In the case where one wants to know if there is an alignment with score less than some threshold T, Fickett [Fik84] designed what we call a **zone algorithm** that computes a portion of the matrix that is guaranteed to contain all entries that are less than T. This region or zone of the matrix may contain internal pockets of values greater than T but in general is not the entire matrix. Unfortunately, for problem such as comparing proteins under the popular PAM scoring scheme of Margaret Dayhoff [DBH83], the zone constitutes about 80-90% of the matrix and so represents little savings.

The only other speedup for the problem of arbitrary weights has been the exploitation of parallel computers and customized hardware. As stated earlier the C-matrix can be computed in any order consistent with data-dependencies of the recurrences. One naturally thinks of a row-by-row or column-by-column evaluation, but other orders are possible, in particular, proceeding by **anti-diagonals**. Let anti-diagonal k be the set of entries $\{ (i, j) : i+j = k \}$. Note that to compute anti-diagonal k, one only needs anti-diagonals $k-1$ and $k-2$. But the truly critical observation is that each entry in this anti-diagonal may be computed independently of the other entries in the anti-diagonal, a fact not true of other topological orders [LiL85]. Thus for large SIMD machines a processor may be assigned to each entry in a fixed anti-diagonal and compute its result independently of the others [LMT88]. Thus with $O(M)$ processors, each anti-diagonal can be computed in constant time, for a total of $O(N)$ total elapsed time.

This observation about anti-diagonals has also been used to design custom VLSI chips configured in what is called a **systolic array** [LiL85]. The ‘‘array’’ consists of a vector of processors each of which is identical, performs a dedicated computation, and communicates only with its left and right neighbors, making it easy to physically layout on a silicon wafer. For sequence comparisons, processor i computes the entries for row i and contains three register we will call $L(i)$, $V(i)$, and $S(i)$. At the completion of the k^{th} step, the processors contain anti-diagonals k and $k-1$ in their L and C registers respectively, and the characters of B flow through their S registers. Formally, let $X(i)_k$ denote the value of register X at the end of the k^{th} step. At ‘‘time’’ k we have: $L(i)_k = C(i,k-i-1)$, $V(i)_k = C(i,k-i)$, and $S(i) = b_{k-i}$. In order to advance the array, observe the recurrences:

$$S(i)_{k+1} = S(i-1)_k$$
$$L(i)_{k+1} = V(i)_k$$
$$V(i)_{k+1} = \min\{\ V(i)_k + \delta(\text{-}, S(i-1)_k),\ L(i-1)_k + \delta(a_i, S(i-1)_k),\ V(i-1)_k + \delta(a_i, \text{-})\ \}$$

Thus to accomplish step $k+1$, processor i must pass its register values to processor $i+1$ and each processor must have just enough hardware to perform the additions and minimum computations of the recurrence. Figure 3 shows the basic hardware configuration of an element of the chip and its interconnections with its neighbors. It directly reflects the recurrence above: registers are shown as rectangles, logic elements as triangles, and the ovals are Σ-element memories indexed by the inputs coming from the symbol register *S*. These ‘‘scoring’’ memories are loaded once before the computation proper begins. The beauty of the systolic array is that it can perform comparisons of *A* against a stream of *B* sequences, processing each symbol of the target sequences in constant time per symbol. With current technology, chips of this kind operate at rates of 10 million symbols per second. A systolic array of 1000 of these processors computes an aggregate of 10 billion dynamic programming entries per second.

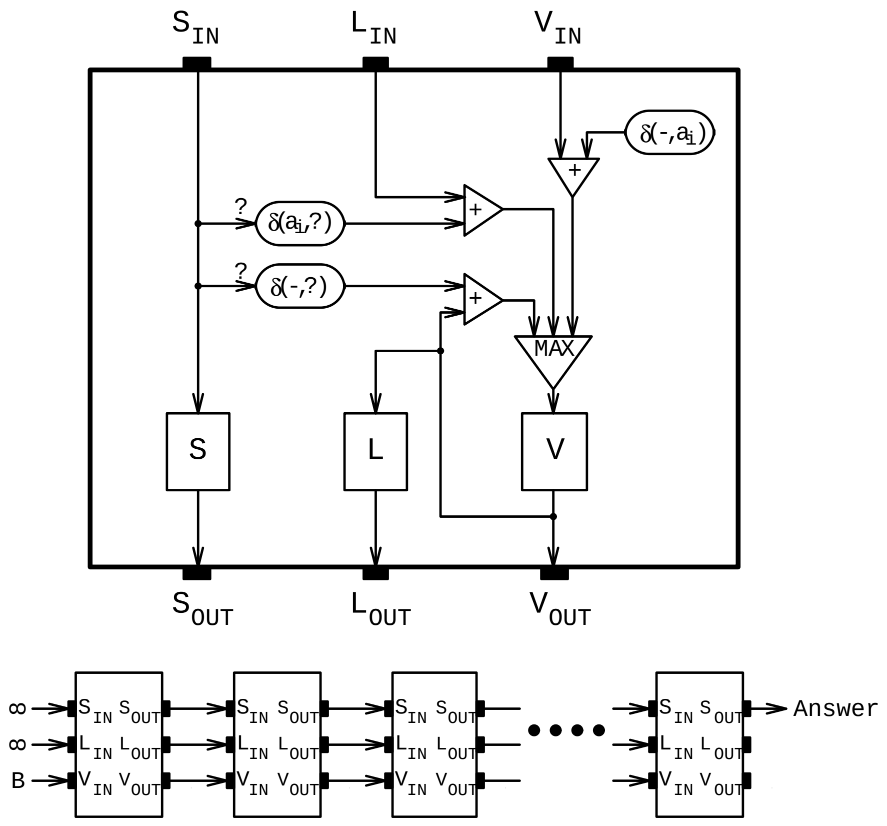

Figure 3: Component Diagram for a Systolic Array

One final result about parallel computation is theoretical in nature. Given a CREW PRAM† machine with $O(MN/\log M)$ processors, a sequence comparison can be performed in $O(\log N \log M)$ time [AAL90]. Note that the total work (time × processors) is $O(MN \log N)$ as opposed to $O(MN)$ for systolic arrays, and thus further improvement is still possible. This result is theoretically interesting but the PRAM model assumes an architecture where all processors may access memory simultaneously in constant time for each processor. Such a device has not been built and the design for one not been demonstrated. Nonetheless, as hardware technology advances we are arriving at multiprocessor machines which at least approximate this idealistic model.

2. Variations on the Basic Problem.

Applications of sequence comparisons in molecular biology have motivated a number of interesting variations to the basic problem not arising elsewhere. For example, from an evolutionary point of view, nature frequently deletes or inserts entire substrings as a unit, as opposed to individual polymer elements. It is thus natural to think of cost models in which a gap's score is a function of its length as opposed to the sum of scores that depend on its composition. We call this the **block indel variation**. As another example, one is frequently interested not only in the optimal alignment, but in the 100 best scoring alignments in order of score as a means to understand alternative yet plausible evolutionary pathways. We call this the **K-best variation**. Another variation of great interest to biologists is the **multiple sequence variation** where one wishes to simultaneously align a set of sequences. This difficult problem is essential to testing phylogenetic hypotheses and detecting weak signals. Finally, it is frequently the case that sequences share only specific substrings of their overall sequence in common and one seeks algorithms that will detect these **local similarities**. We briefly review these four variations in this subsection, and reserve another very important variation, **similarity database searches**, as the focus of the final subsection.

2.A. The Block Indel Variation.

Consider an alignment model where the cost of a gap, $gap(x)$, is an arbitrary function of the length x of the gap. The original dynamic programming algorithm of Needleman and Wunsch [NeW70], latter cleanly exposited by Waterman, Smith, and Byers [WSB76], accommodated such general gap scores and required the following minor variation on our earlier recurrence:

$$\begin{aligned}
D(i,j) &= \min\{\ C(k,j) + gap(i-k)\ :\ k \in [0, i-1]\ \}.\\
I(i,j) &= \min\{\ C(i,k) + gap(j-k)\ :\ k \in [0, j-1]\ \}.\\
C(i,j) &= \min\{\ C(i-1,j-1) + \delta(a_i, b_j),\ D(i,j),\ I(i,j)\ \}.
\end{aligned}$$

For each subproblem, A_i versus B_j, one has recurrences for (1) the best alignment *that ends with a deletion gap*, $D(i,j)$, (2) the best alignment *that ends with an insertion gap*, $I(i,j)$, and (3) the best overall alignment, $C(i,j)$. We only give the recurrence for $i, j > 0$. The boundary cases are left as an exercise to the reader both here and in all forthcoming equations. While the recurrence is easily derived for this more general problem, note that the algorithm directly realizing these recurrences takes $O(MN(M+N))$ time as the I and D entries take $O(M+N)$ time to compute.

A special case of the block indel problem that is of particular interest is where $gap(x)$ is an **affine function** $g+hx$ for non-negative constants g and h. Intuitively, in the underlying model, starting a gap costs g and each additional symbol in the gap costs h. This model seems to be essential to obtaining biologically relevant alignments [FiS83], and using the decomposition of the problem above one easily arrives at the recurrences:

$$\begin{aligned}
D(i,j) &= \min\{\ D(i-1,j)+h,\ C(i-1,j)+g+h\ \}.\\
I(i,j) &= \min\{\ I(i,j-1)+h,\ C(i,j-1)+g+h\ \}.\\
C(i,j) &= \min\{\ C(i-1,j-1)+\delta(a_i, b_j),\ D(i,j),\ I(i,j)\ \}.
\end{aligned}$$

———————

† CREW abbreviates *C*oncurrent *R*ead, *E*xclusive *W*rite and PRAM abbreviates *P*arallel *R*andom *A*ccess *M*achine.

Gotoh [Got82] first presented this recurrence in 1982 and the algorithm that directly follows is easily seen to take $O(MN)$ time. The division of the recurrence into three cases was arbitrary for the general problem, but here is essential. C terms contributing to a D or I value are charged $g + h$ because a gap is being initiated from that term. $D(I)$ terms contributing to $D(I)$ values are charged only h because the gap is just being extended. It has been common knowledge that by continuing such a case-wise decomposition, one may handle gap costs that are P-piece affine curves in $O(PMN)$ time.

Lying between the affine and arbitrarily scored block indel problems, is the case where $gap(x)$ is a **concave function**, i.e., $gap(x+1) - gap(x) \le gap(x) - gap(x-1)$. In words, the first forward differences are non-increasing, or intuitively, if one imagines *gap* as a continuous curve then its second derivative is non-positive. For example, affine functions are concave and so is $gap(x) = a \log x + b$ for positive a and b. Such a gap model arises naturally in the dynamic programming formulation for RNA secondary structure prediction, and it has been conjectured that such a model may be useful for biological sequence comparisons. Waterman [Wat84]introduced the concave-gap comparison problem and in 1988 two teams independently arrived at an $O(MN \log N)$ algorithm [MiM88,EGG88].

To understand the basic idea of this algorithm it suffices to focus on the subproblem of determining the D-values in a given row, say j, in a sweep of increasing i. When at column i in this sweep, a proceeding **candidate** from column $k < i$ will contribute $Cont_k(x) = C(k, j) + gap(x-k)$ to the minimum for $D(x, j)$ at future column $x > i$. Observe that $Cont_k(x)$ is simply a copy of the concave curve $gap(x)$ translated k units to the right and $C(i, j)$ units upward. As the algorithm sweeps to column i (in row j), it keeps a representation of the **minimum envelope** $Env_i(x) = \min\{ Cont_k(x) : k \in [0, i-1] \}$ of the $Cont_k$ curves for all $k < i$. In essence Env_i models the future contribution of all candidates $k < i$. Observing that two copies of the same but differently displaced concave curves intersect at most once, it is not difficult to see that the envelope can be modeled by a list of candidates each of which gives the value of the envelope over a specific interval. Figure 4 gives an example. The essence of the algorithm is a method for updating the **candidate list** representing the minimum envelope in $O(\log N)$ time per entry swept.

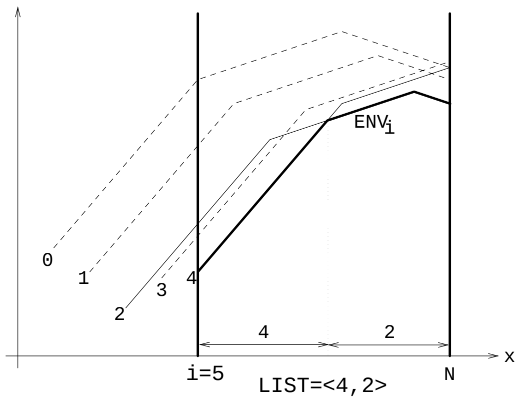

Figure 4: A Sample Minimum Envelope.

For the case were the intersection between two differently displaced copies of *gap* can be computed in constant time, the complexity of the algorithm above improves to $O(MN)$. For example, this property is true of $gap(x) = a \log x + b$. Furthermore, for P-piece affine curves Miller and Myers showed this algorithm to admit to an $O(MN \log P)$ variation. Recently Larmore and Schieber [LaS90] have devised an improved algorithm that runs in $O(MN \log \log N)$ time.

2.B. The K-best Variation.

The basic algorithm reports an optimal alignment, and the choice of scoring scheme affects which alignments are optimal. Unfortunately, biologists have not yet ascertained which scoring schemes are ‘‘correct’’ for a given comparison domain. This uncertainty suggests listing all alignments near the optimum in the hope of generating the biologically correct alignment. The ability to list alternate, near-optimal solutions has also been advocated for predicting RNA secondary structure by Dumas and Ninio [DuN82].

From the view of the edit graph formulation, the K-best problem is to deliver the K-best shortest source-to-sink paths, a problem much studied in the operations research literature. The algorithm of Hoffman-Pavley-Dreyfus [HoP59,Dre69] when implemented as suggested by Fox or Lawler [Law72,Fox73], delivers the K-best paths over an edit graph in $O(MN + KN)$ time and space. The algorithm delivers these paths/alignments in order of score and K does not need to be known *a priori*, the next best alignment is available in $O(N)$ time. The essential idea of the algorithm is to keep, at each vertex v, a heap of the next best path to the sink through each edge out of v. The next best path is traceable using these heaps, extracted, and the heaps appropriately updated.

In 1985, Waterman and Byers [WaB85] devised a method which requires only the matrix C of the dynamic programming computation, and so reduces the space requirement to $O(MN)$. It requires an *a priori* threshold ε and reports all alignment of score less than $D(A, B) + \varepsilon$ but not necessarily in order. One can imagine tracing back all paths from the sink to the source in a recursive fashion. The essential idea of the Waterman and Byers algorithm is to limit the traceback to only those paths of score not greater than $D(A, B) + \varepsilon$. Supposing one reaches vertex (i, j) and the score of the path from the sink back to this vertex is $T(i, j)$. Then one traces back to predecessor vertices $(i-1, j)$, $(i-1, j-1)$, and $(i, j-1)$ if and only if

$$\begin{aligned}
&C(i-1, j) + \delta(a_i, \text{-}) + T(i, j) \le D(A, B) + \varepsilon,\\
&C(i-1, j-1) + \delta(a_i, b_j) + T(i, j) \le D(A, B) + \varepsilon,\\
&C(i, j-1) + \delta(\text{-}, b_j) + T(i, j) \le D(A, B) + \varepsilon,
\end{aligned}$$

respectively. This procedure is very simple, space economical and quite fast. However, one should note that the operations research shortest paths algorithm can easily be modified to produce the S best *scores* (and the number of alignments having that score) in $O(SMN)$ time and space. For example, Waterman reported that there are 1,317 alignments within 5% of optimal for the α and β chicken hemoglobin chains under a weighting function from the study of Fitch and Smith [FiS83]. But using the $O(SMN)$ algorithm one can determine that there are 17 scores and 20,137,655 alignments within 20% of optimal in less than a minute on a conventional workstation, a computation not possible with the straightforward enumerative algorithms.

2.C. The Multiple Sequence Variation.

In an attempt to distinguish more distant relationships, molecular biologists are frequently interested in comparing more than two sequences simultaneously. For instance, given 10 sequences of the same functionality, it is much more likely that the similarity that gives this common function will be more evident among the group than among two sequences from the group. In a related problem, a collection of sequences labels the leaves of a **phylogenetic tree** and the goal is to label the interior vertices in a fashion that minimizes the sum of the pairwise distances between the sequences labeling the tail and head of an edge. As we shall see later these problems are basically identical.

Suppose we wish to align K sequences $A^1, A^2, \cdots, A^K$ where $A^i = a^i_1 a^i_2 \cdots a^i_{N^i}$ is of length N^i. As for the basic problem we wish to arrange the sequences into a tableau using dashes to force the alignment of certain characters in given columns. For example,

```
ctcgcg-cacat-agggcg-gtc-cgaga-ga-taggcaagcc
ctcgcggcacattcgggcg-gtctcgagatgactaggc-agcc
ctcgcggca-attcgggcg-gtctcgaga-gactaggcaagcc
ctcgcgtcacattcgggcgtgtctcgaga-gactaggcaagcc
ctcgcgg-acattcgggcg-gt-tcg-ga-gactaggcaagcc
-------------------------------------------
ctcgcggcacattcgggcg gtctcgaga gactaggcaagcc
```

is an alignment of five electrophoresis gel readings from a DNA sequencing experiment, and the sequence under the row of dashes is the purported ‘‘consensus’’. Ignoring dashes, each row contains one of the K sequences. Each column gives a set of aligned symbols and is scored according to a user supplied function δ. δ gives the score of each possible column of K symbols except for the one containing all dashes that is not allowed. For example, if δ is the number of dashes in the column then the cost of the alignment above is 19, the sum of the column costs. Note that it is no longer sensible to talk of deletion and insertion gaps.

The problem of finding a minimum scoring alignment among K sequences can be solved by simply extending the dynamic programming recurrence for the basic problem [WSB76]. Let $\vec{i}$ = $< i_1, i_2, \cdots i_K >$ be a K-vector in the K-dimensional Cartesian space $\prod_{i=1}^{K} [0, N^i]$. Now we compute a K-dimensional array C, where $C(\vec{i})$ is the score of the best alignment among the sequences $A^1_{i_1}, A^2_{i_2}, \cdots A^K_{i_K}$. The central recurrence is by simple extension:

$$C(\vec{i}) = \min\{\ C(\vec{i} - \vec{e}) + \delta(e_1{:}a^1_{i_1}, \cdots, e_K{:}a^K_{i_K}) \ :\ \vec{e} \in \{0,1\}^K - <0,0,\cdots,0>\ \}$$

$$\text{where } e{:}a \ \equiv\ \text{if } e = 1 \text{ then } a \text{ else `-'}.$$

In terms of an edit graph model, imagine a grid of vertices in K-dimensional space where each vertex $\vec{i}$ has $2^K - 1$ edges directed into it, each corresponding to a column that when appended to the alignment for the edge’s tail gives rise to the alignment for the prefix sequences represented by $\vec{i}$. Computing the C values in some topological ordering requires a total of $O(\prod_{i=1}^{K} N^i) = O(N^K)$ time where we let N = $\max\{\ N^i : i \in [1, K]\ \}$. Thus while the extension to multiple sequences is conceptually quite straightforward, the obvious algorithm takes an amount of time exponential in K and so is generally impractical in practice for $K > 3$.

Multiple sequence comparison has been shown to be NP-complete [GaJ79] implying that any algorithm will ultimately exhibit exponential behavior in K. Thus many authors have sought heuristic approximations, the most popular of which is to take $O(K^2 N^2)$ time to compute all pairwise optimal alignments between the K sequences, and then produce a multiple sequence alignment by merging these pairwise alignments. Note that any multiple sequence alignment induces an alignment between a given pair of sequences (i.e., take the two rows of the tableau and remove any columns consisting of just dashes). However, given $K(K-1)/2$ pairwise alignments it is almost always impossible to arrange a multi-alignment consistent with them all (e.g., try merging the best pairwise alignments among *acg*, *cga*, and *gac*). But given any $K-1$ alignments relating all the sequences (i.e., a spanning tree of the complete graph of sequence pairs), it is always possible to do so. Feng, Johnson, and Doolittle [FJD85] compare a number of methods based on this approach and the most recent algorithms utilize the natural choice of the $K-1$ alignments whose score sum is minimal [FeD87]. Regardless, such merges do not always lead to optimal alignments as illustrated by the example below.

```
g-caca                               g--caca     g-caca
ggca-a   and   gg-caa   yields   gg-ca-a   but   ggca-a   is better.
               ggaca-            ggaca--         gg-aca
```

There are a number of multi-alignment scoring schemes δ that are defined in terms of a pairwise scoring function δ'. For example, the **sum-of-pairs score** is defined as $\delta(a_1, a_2, \cdots a_K) = \sum_{i<j} \delta'(a_i, a_j)$ where one must let $\delta'(-,-) = 0$. In essence, the multi-alignment score is the sum of the scores of the $K(K-1)/2$ pairwise alignments it induces. Another common scheme, is the **consensus score** which defines $\delta(a_1, a_2, \cdots, a_K)$ as $\min \{ \sum_i \delta'(c, a_i) : c \in \Sigma \}$. The symbol c that gives the minimum is said to be the consensus symbol for the column, and the concatenation of these symbols is the consensus sequence. In effect, the multi-alignment score is the sum of the scores of the K pairwise alignments of the sequences versus the consensus. As our final example, consider an unoriented tree T with K leaves each assigned to one of the sequences to be aligned. To define δ for a column, orient T by arbitrarily picking a *Root* vertex and imagine placing each symbol of the column at its corresponding leaf. The following recurrent definition is computable in $O(K|\Sigma|^2)$ time.

$$\delta(\vec{a}) = \min_{a \in \Sigma} \{ \delta_{Root}(a) \}$$

$$\text{where } \delta_v(a) = \begin{cases} \sum_{w \in Son(v)} \min_{b \in \Sigma} \{ \delta_w(b) + \delta'(a, b) \} & \text{if } v \text{ is internal} \\ 0 & \text{if } v \text{ is a leaf labeled } a \\ \infty & \text{otherwise} \end{cases}$$

By recording the symbols chosen at each level of the recurrence in order to achieve the minimum defining δ, one effectively assigns symbols to the interior vertices such that the sum of the pairwise scores along edges of T is minimal. Thus an alignment based on this scoring scheme, provides a labeling of the interior vertices of T with sequences such that the score of the pairwise alignments along edges of the tree equals the score of the multi-alignment [San75]. Thus the phylogeny problem mentioned at the start of this subsection is indeed an instance of multiple sequence alignment. Further note, that this **tree-based score** generalizes the consensus score whose underlying tree is a single interior root connected to all the leaves. The more general phylogeny problem requires that one also determine the tree that provides the minimal score, and in this case one must simply try each tree and see which gives the best score.

A recent and promising advance, is a simple bounding result by Carillo and Lipman [CaL88] that significantly reduces the portion of the K-dimensional dynamic programming matrix that must be evaluated for the case of sum-of-pairs scores. Let $P(A^i, A^j)$ be the cost of the pairwise alignment induced between A^i and A^j by an optimal sum-of-pairs multi-alignment. Suppose that U is an upper bound on the cost of the best multi-alignment. For example, it might be obtained by the $O(K^2 N^2)$ heuristic algorithm. Note the $L = \sum_{i<j} D(A^i, A^j)$ is a lower bound on the cost of the best multi-alignment. The simple observation is:

$$\text{For all } p \text{ and } q\text{:} \quad U - L \geq \sum_{i<j} P(A^i, A^j) - D(A^i, A^j) \geq P(A^p, A^q) - D(A^p, A^q).$$

Thus for any pair, A^p and A^q, one knows that the path represented by an optimal alignment in the K-dimensional edit graph, when projected onto the 2-dimensional face for A^p and A^q, lies within $U-L$ of the optimal pairwise path on that face. The Carillo and Lipman algorithm exploits this fact by computing the region of each 2-dimensional face in which the projected optimal path may lie, and then observing that the optimal multi-path must lie in the intersection of the K-dimensional zones delineated by these $K(K-1)/2$ regions. In most cases, this intersection is a very small portion of the overall matrix. The algorithm can align up to 5 or 6 sequences of length 100 in time on the order of minutes on a typical 20MHz workstation.

2.D. Local Similarity.

Proteins and DNA sequences are the result of evolutionary processes that tend to preserve the portions central to function and to vary the rest. For example, the chemically active site of a protein tends to be strongly conserved since any change ruins the protein's function. Thus when comparing two related biosequences one frequently finds that the alignment between the entire sequences is very poor, but that there are regions (presumably the functional parts) that are very similar. In other words, while there may be little **global similarity** between related sequences, there are usually one or more strong **local similarities**. For this reason, the problem of finding ''interesting'' similarities between substrings of two sequences is an important problem. A variety of approaches have been taken, most varying on the definition of what is considered to be ''interesting''.

From the view of the edit graph of sequences A and B, one is looking for paths in the graph that are unusual. If a path begins at vertex (g, h) and ends at (i, j) then it aligns the substrings $a_{g+1} a_{g+2} \cdots a_i$ and $b_{h+1} b_{h+2} \cdots b_j$. One can define the length of a path as either the number of edges in it or as the number of units displaced, $(i-g)+(j-h)$. In the first case, the length is the number of columns in the alignment, and in the second, it is the sum of the lengths of the substrings being aligned.

A simple approach by Smith and Waterman [SmW81], assumes the maximizing similarity perspective, and that δ is such that gaps are scored negatively and the mean substitution score is also less than zero. The central recurrence is modified to contain a zero-term as follows:

$$C(i, j) = \max\{0,\ C(i-1, j-1) + \delta(a_i, b_j),\ C(i-1, j) + \delta(a_i, \text{-}),\ C(i, j-1) + \delta(\text{-}, b_j)\ \}.$$

With this modification, $C(i, j)$ is the score of the best path from some preceding vertex (g, h) and not necessarily the source. Notice that because δ is negatively biased, it is only in regions of high similarity (lots of positive scores) that the maximum evaluates to other than zero. In the Waterman and Smith formulation, the path ending at the maximum C-value in the matrix is reported as the best local similarity. The path is easily found by the standard traceback procedure. In a more recent refinement [WaE87], the algorithm reports the second best path disjoint from the first, the third best, and so on. This procedure is exposited first because it is the most simple. In this authors opinion it is also the best, but in order to stimulate further ideas we present the other major approaches to the problem.

A number of investigators [GoK82,Sel84] considered a standard distance-based scoring scheme δ and decided that what was interesting was those paths whose **mismatch density** was below a certain user-settable threshold R. The mismatch density of a path P, is the ratio of its score to its length, and we thus seek paths such that $\delta(P)/Len(P) \le R$. Interestingly, what the authors did next was to translate this requirement into a similarity problem involving a scoring scheme δ' defined as follows:

$$\delta'(a, b) = \begin{cases} 2R - \delta(a, b) & \text{if } a, b \in \Sigma \\ R - \delta(a, b) & \text{if } a = \text{-} \text{ or } b = \text{-} \end{cases}$$

Assuming length is the sum of the substrings, observe that $\delta'(P) = RLen(P) - \delta(P)$ and thus one is seeking paths for which $\delta'(P) \ge 0$. (If length were defined as the number of edges than one need simply remove the 2 in the definition above.) So using the recurrence presented for the previous algorithm, one seeks all positive matrix entries.

Unfortunately, even for reasonable choices of R this leaves too many paths to examine, and so the authors added additional constraints. Let F be the set of edit graph edges which give rise to a positive C-value, i.e., the C-value at the edge's tail plus its cost equals the positive C-value at its head. Let B be identically defined except for a computation that proceeds over the *reverse* of the edit graph (i.e., conceptually reverse the direction of every edge and perform the computation in a topological order of this reversed graph). In the first such formulation by Goad and Kanehesia [GoK82], the paths reported were those in $F \cap B$. This limited the number of paths, but the condition defining these paths defies definitions other than the procedural one just given.

In a more rigorous attempt, Sellers [Sel84] described a multi-sweep algorithm potentially of $O(N^3)$ complexity, that produces all paths P such that (1) all prefixes of P have mismatch ratio less than R, (2) all suffixes of P have mismatch ratio less than R, and (3) the path is locally maximal (i.e., of the paths meeting (1) and (2) and sharing a common vertex, P is the highest scoring). The algorithm involves alternating forward and reverse computations in which edges are culled at each stage. The edges that remain constitute the paths satisfying the conditions above. Initially one starts with the complete edit graph and in a forward sweep removes all edges not in F. Then in a reverse computation over the remaining subgraph, all edges not meeting the defining condition for B are removed. Then another forward pass over the resulting subgraph removes edges not meeting the defining condition for F. And so on until a pass is completed in which no edges are removed. Examples can be constructed in which the number of sweeps required is $\Omega(N^{1/2})$ although certainly no more than $O(N)$ sweeps are ever required. A tight upper bound on the number of sweeps is not known.

The edges left by Sellers' 1984 algorithm are a subset of those left by the Goad and Kanehesia algorithm. In 1987 Sellers [Sel87] further refined the condition to report those paths in $F \cap B$ that in addition do not contain a vertex that is the head of an edge in $F - B$ or the tail of an edge in $B - F$. Interestingly, this condition is stronger than the 1984 condition yet is $O(N^2)$ time computable as it only requires a single forward and reverse sweep. Despite the potential appeal of these approaches, they are complex and do not produce any better results than the simple Waterman and Smith algorithm.

All the preceding methods assume that the notion of similarity can be captured in an underlying scoring scheme δ and is expressible as the sum of the parts of the alignment. While this model has mathematical appeal it is not clearly apropos for the comparison of biopolymers such as proteins. As a consequence, other, more ad hoc measures have been tried in order to obtain the desired results. For example Patrick Argos [Arg87] has had considerable success using a scoring scheme based on physical characteristics. We give a procedural description since the method does not lend itself to a terse analytical characterization. The user supplies a parameter L and all paths consisting of L substitution edges are scored in two ways: the first is the standard Dayhoff PAM scoring scheme [DBH83] and the second is an average of five correlation coefficients reflecting physical characteristics of the amino acids denoted by the symbols. The coefficients are based on the L symbols correlated by the path where each symbol is assigned a physical characterization index with respect to the properties of hydrophobicity, size, shape, and ability to participate in turns and β-strands. The distribution of the physical scores over the $O(MN)$ paths examined is scaled so that the mean and deviation match those of the distribution of PAM scores. After this scaling the two scores are averaged and all scores are converted to standard deviations with respect to the distribution of the $O(MN)$ scores. From this point the best scoring segments are coalesced and the resulting local similarities presented to the user. As can be seen from this instance, there is still much debate over what the appropriate measure of similarity should be.

3. Similarity Searches Over Large Databases.

Sequence data is rapidly being accumulated by the molecular biology community. The current GENBANK database of DNA sequences contains approximately 40 million bases of sequence in about 10,000 sequence entries [FiB88]. The PIR database of protein sequences contains about 4 million residues of data in about 10,000 protein entries [GBH86]. Whenever a new DNA or protein sequence is produced in a laboratory, it is now routine practice to search these databases to see if the new sequence shares any similarities with existing entries. In the event that the new sequence is of unknown function an interesting global or local similarity to an already studied sequence may suggest possible functions.

For a given **query** sequence one may consider performing any of the global or local pairwise comparison algorithms discussed in the preceding sections against every **entry** sequence in the **database**. However, performing 10,000 such comparisons for the current databases can be very time consuming. For example, comparing a 300 residue protein against the current PIR database with the basic dynamic programming algorithm requires the computation of 1.2 billion matrix entries. This is a task that takes on the order of an hour on a current workstation. However, current vectorizing supercomputers, such as the CRAY II, or massively parallel SIMD computers, such as the Connection Machine, can perform these computations within seconds [LMT88]. So the problem is not out of the current technological range, in fact, a soon to be released systolic array processor [Wat90] implements the Waterman-Smith local similarity search and will take about a half a second to search the current protein database. Nonetheless, the databases are growing exponentially. For example, with the advent of the Human Genome Initiative [CMS88], the DNA sequence data available will be in the trillions of bases within the decade. For this reason, and one of economy, designing algorithmically faster techniques for similarity searches is a very important problem.

3.A. Heuristic Algorithms.

The problem of searching for protein similarities efficiently has lead many investigators to consider designing very fast **heuristic** procedures: simple, often ad-hoc, computations that produce answers that are ‘‘nearly’’ correct with respect to a formally stated optimization criterion. One of the most popular database searching tools of this genre is FASTA developed by Lipman and Pearson in 1985 [LiP85]. FASTA is heuristic: it reports most of the alignments that would be produced by an equivalent dynamic programming calculation, but it misses some matches and also reports some spurious ones with respect to the more robust computation. On the other hand it is very fast and reports results of interest to investigators. The underlying idea is very simple and based on the sparse variation of Wilbur and Lipman discussed in Section 1.

For a small integer *k*, recall that the set of *k*-substring pairs is $P_k = \{ (i, j) : a_i a_{i+1} \cdots a_{i+k-1} = b_j b_{j+1} \cdots b_{j+k-1} \}$ and assume its size is *R*. For each entry, *B*, of the database and the query *A*, FASTA counts the number of pairs in each **diagonal** of the *C*-matrix. Formally, diagonal *d* is the set, $\{ (i, j) : i - j = d \}$, and the algorithm computes $Count(d) = | \{ (i, j) : j - i = d$ and $(i, j) \in P_k \} |$ for each diagonal $d \in [-M, N]$. FASTA computes these counts efficiently in $O(R)$ time by encoding each *k*-substring as an integer. Intuitively the code for a substring over alphabet Σ is obtained by viewing it as a *k*-digit $|\Sigma|$-ary number, i.e., $Code(c_{k-1} c_{k-2} \cdots c_0) = \sum_{i=0}^{k-1} c_i |\Sigma|^i$. The code for each *k*-substring of the query and entry can be computed in linear time using the simple relationship $Code(a_i a_{i+1} \cdots a_{i+k-1}) = a_i |\Sigma|^{k-1} + Code(a_{i+1} a_{i+2} \cdots a_{i+k}) / |\Sigma|$. The substring codes for the query are computed once and each substring's position is stored in a $|\Sigma|^k$-element array according to its code. As each database entry is processed, its substring codes are computed, used to index the positions of the matching substrings in the query, and the diagonal counts incremented accordingly. Thus if *k* is large enough so that *R* is small relative to *MN*, then this procedure is quite fast. FASTA examines the diagonal counts and if there are ‘‘high enough’’ counts in one or more diagonals, reports the entry as being similar to the query. In a postpass, it produces an alignment between the query and entry as in the heuristic procedure of Wilbur and Lipman [WiL83]. This procedure essentially builds a source to sink path by greedily selecting, in order of length, substring pairs compatible with those already chosen to be a part of the path.

There are a number of other heuristic algorithms of the same ilk as FASTA. We focus on one more such tool, BLASTA, that has been developed very recently and is notable for being faster than FASTA, yet capable of detecting biologically meaningful similarities with comparable accuracy [AGM90]. Given a query and an entry, BLASTA looks for **segment pairs** of high score. A segment pair is a substring from *A* and a substring from *B* of equal length and the score of the pair is that of the no-gap alignment between them. One can argue that the presence of a high-scoring segment pair or pairs is evidence of functional similarity between proteins, since indel events tend to significantly change the shape of a protein and hence its function. Note that segment pairs embody a local similarity concept. What is particularly useful, is that a recent result by Karlin and Altschul gives a formula for the probability that two sequences have a segment pair above a certain score. Thus BLASTA gives an assessment of the statistical significance of any match that it reports. For a given threshold, *S*, BLASTA returns to the user all database entries that have a segment pair with the query of score greater than *S* ranked according to probability. BLASTA is heuristic, for it may miss some such matches although in practice it misses very few.

The central idea used in BLASTA is the notion of a **neighborhood**. The *T*-neighborhood of a sequence *W* is the set of all sequences that align with *W* with score better than *T*. Underlying and critical to the precise definition is the particular alignment and scoring model. For BLASTA, the model is no-gap alignments and the scoring scheme is the PAM mutation scores of Dayhoff. So for BLASTA, the *T*-neighborhood of *W* is exactly those sequences of equal length which form a segment pair of score higher than *T*. This concept suggests a simple strategy for finding all entries that have segment pairs of length *k* and score greater than *T* with the query: generate the set of all sequences that are in the *T*-neighborhood of some *k*-substring of the query and see if an entry contains one of these strings. The particular appeal here is that one can build a deterministic finite automaton (DFA) for the sequence set and use it to scan an entry in linear time for a match. The scanning speed of such an algorithm is exceedingly fast, e.g., on the order of half a million characters a second on a 20MHz computer. Unfortunately, for the general problem, the length of the segment pair is not known in advance and more devastating is that fact that the number of sequences in the neighborhoods grows exponentially in both *k* and *T* rendering it impractical for reasonable values of *T*.

To circumvent this difficulty, BLASTA uses the fast scanning strategy above to find short segment pairs above a certain score and then checks each of these to see if they are a portion of a segment pair of score *S* or greater. We outline the algorithm without, as yet, specifying what the parameters *k* and *T* should be, save that they be ‘‘sufficiently small’’.

(1) For each of the $M-k+1$, *k*-substrings of the query *A*, the *T*-neighborhood is generated and a DFA accepting this set of strings is built in $O(kMZ)$ time where *Z* is the average cardinality of a *T*-neighborhood.

(2) The DFA scans the database and each time a string in the neighborhood set is found Step 3 is executed. A total of $O(L)$ time is taken in this step where *L* is the sum of the lengths of all the entries in the database.

(3) For a given ‘‘hit’’ the DFA delivers the substring(s) from whence the matching sequence was generated. Thus Step 2 delivers a segment pair of length k and score T or greater between the query and an entry. BLASTA considers extending this segment pair both by appending and prepending symbols to the given pair in order to see if it is a subsegment of a pair of score S of greater. If so it reports the entry and the segment pair. In practice the process of extending in a given direction is quit once the score drops beyond a settable threshold, so that on average a constant amount of time is spent checking a given hit. The probability of a hit occuring at a given position is roughly $O(MZ/|\Sigma|^k)$, so on average $O(LMZ/|\Sigma|^k)$ time is spent in this step.

The procedure above is heuristic in that there is some chance that a segment pair of score S or greater does not contain a subsegment of length k and score T or greater. This probability increases with increasing T, and decreases with increasing k. The problem is too choose k and T such that (1) the probability of missing a segment pair of score S or greater is not too great, (2) the size of the neighborhoods is not too large, and (3) the frequence of hits for Step 3 is not too great. These criterion are mutually incompatible, e.g. increasing k improves (1) and (3) but worsens (2), and increasing T improves (2) and (3) but worsens (1). For the case of protein comparisons under an integerized version of the PAM scores, a reasonable compromise for the default choice was $k = 4$ and $T = 17$ (for which $Z = 25$). In practice BLASTA is about an order of magnitude faster than FASTA and can process a database at a rate of about one-third of a million characters per second on a 20MHz general purpose machine.

3.B. Approximate Pattern Matching.

We have thus far considered comparing a query against similar-sized entries in a database. But in cases such as the DNA database, individual entries will eventually become very long, e.g. the sequence of a human chromosome is of length a hundred million. So let us now consider the database to be a single very long sequence B of length L as opposed to a set of sequence entries. If one wishes, one can consider concatenating all the entries in a database together to form B. Typically the length of the query sequence A is on the order of 100 to 1000, and the database sequence B has length on the order of 10^6 to 10^{10}. Given this perspective of the database, the similarity searching problem becomes one of finding *substrings* of B that align particularly well with A. Frequently one gives a **threshold**, say D, for what is considered to be an interesting alignment. For this **thresholded similarity search** problem one seeks all substrings that align with score better than D [Sel80].

Solving the thresholded similarity search problem for the basic alignment model is simply a matter of modifying the boundary of the dynamic programming recurrence. Specifically modify the standard recurrence for $C(i, j)$ so that it is 0 whenever $j = 0$. With this modification, $C(i, j)$ becomes the score of the best alignment between A_i and *some substring* of B ending at its j^{th} symbol. Imagine the C matrix as a very long skinny rectangle, the short side being of length proportional to the query, the long side being proportional to the database, and the long upper boundary being preset to 0. Compute each column (the short lengths) according to the modified recurrence as symbols of the database are read from disk. Each time $C(M, j)$, the last entry in the column, is not greater than D report that an approximate match to A ending at position j of the database has been found. Finding the other end(s) of the match and an alignment can easily be done with the traceback and reverse computation ideas described earlier.

The thresholded similarity search problem is actually a special instance of a broader class of approximate pattern matching problems. First consider **exact pattern matching** problems where one is given a pattern and a database, and one seeks substrings of the database, called **occurrences**, that exactly match the pattern. Such problems have been much studied by computer scientists for a variety of pattern types such as a simple sequence, a regular expression, and a context free language. Such patterns are notations that denote a possibly infinite set of sequences each of which is said to exactly match the pattern, e.g., the regular expression $a(b|c)d*$ denotes the set $\{ ab, ac, abd, acd, abdd, acdd, abddd, \cdots \}$. For these exact matching problems there are $O(L)$ algorithms [ACo75,KMP77,BoM77] when the pattern is a given sequence or finite set of sequences, an $O(LM)$ algorithm [Tho68] for regular expressions where M is the length of the pattern, and an $O(L^3 M)$ algorithm [Ear70] for context free languages where M is the ‘‘size’’ of the grammar of the language. The concept of exact pattern matching and sequence comparison can be fused to introduce a class of problems we call **approximate pattern matching**. Given a pattern, a database, a scoring scheme, and a threshold, one seeks all substrings of the database that align to some sequence denoted by the pattern with score better than the threshold. In essence, we are looking for substrings that are within a given similarity neighborhood of an exact match to the pattern. Within this framework, the thresholded similarity search problem is an approximate pattern matching problem where the pattern is a given query sequence and as seen above this problem can be solved in $O(LM)$ time. For the case of regular expressions the approximate match problem can also be solved in $O(LM)$ time [MyM89] and for context free languages an $O(L^3 M^2)$ algorithm is known [AhP72].

We close this section by examining algorithmic progress that has been made on the **approximate string matching** problem much studied by computer scientists. Given threshold D and string A we seek all occurences of approximate matches to A where the scoring scheme is that of the simple shortest edit script model: each indel and unequal substitution is charged 1, and identically aligned symbols are charged 0. This is a special instance of the thresholded approximate match problem so we immediately know it can be solved in $O(LM)$ time. But the idea for the zone algorithm of Fickett mentioned earlier can be applied here. Essentially only the initial portion of each column of the C-matrix whose values are D or less need be computed. It can be shown that against a database that is the result of equi-probable Bernouilli trials, the number of entries computed in each column is $O(D)$ in expectation. In roughly the same time frame a number of investigators [Ukk85b,MyM86] reported this $O(LD)$ expected-time variation.

Not long thereafter Landau and Vishkin [LaV86] arrived at an $O(LD)$ worst-case algorithm based on the $O(ND)$ greedy comparison algorithm mentioned in Subsection 1. The algorithm required $O(L)$ space and the use of some rather complex subprocedures that rendered it less desirable than the $O(LD)$ expected-time algorithm. At about the same time Myers [Mye86b] also reported an $O(LD)$ algorithm based on an incremental variation of the greedy algorithm that required only $O(D^2)$ space and gave not only the ending position of a match but each possible starting position. Thus, to date the best result for approximate string matching is $O(LD)$ worst-case time. Note that even heuristic procedures such as FASTA and BLASTA require $O(LM)$ time (but with very small coefficients of proportionality). So all current methods require time linear in L, even systolic array processors. Given that L, the size of the databases is growing exponentially, what we really desire is methods that are sublinear in L. We close the section with very recent results by Chang and Lawler and by Myers that move in this direction.

3.C. Sublinear Approximate String Matching.

Chang and Lawler [ChL90] have devised an algorithm that takes $O((L/M)D\log M)$ expected-time when the threshold D is less than $M/(\log M + O(1))$. It does so by quickly eliminating stretches of the database where a match cannot occur. If there is an approximate match beginning at position j of the database then there must be a series of $k+1$ or less common substrings between A and the aligned substring of B and this stretch of B must be of length at least $M-D$. Let M_i be the length of the longest common substring between a prefix of $b_i b_{i+1} \cdots b_L$ and some substring of A. Further, let S_{j+1} = S_j + M_{S_j} + 1. Starting at position S_j in the database, the best possible span of disjoint substrings is $S_{j+k+2} - S_j$ and if this is less than $M-D$ then we know there cannot be an approximate match to A whose first aligned symbols is at S_j. This central observation is used to rapidly move over such stretches which are very frequent. The algorithm spends most of its time jumping from position S_j to S_{j+1} and only occasionally applying an $O(MD)$ alignment computation in regions that might contain a match. A suffix tree data structure delivers the quantities M_i efficiently and a detailed probabilistic analysis delivers the stated expected-time complexity. The algorithm is still linear in L but note that under the range of D for which the result holds, the algorithm takes no more than $O(L)$ expected-time, a definite improvement. The authors' claim of sublinearity is in the sense originally posed by Boyer and Moore whose algorithm for exact string matching examined only a fraction of the L symbols in the database. The one potential drawback of the algorithm is that as M becomes larger the threshold D cannot be proportionally increased, i.e., D/M must be less than $1/\log M$.

The quantity $\varepsilon = D/M$ is the maximum fraction of errors permitted per unit length of the query and we call it the **mismatch ratio** (akin to the mismatch density of the local similarity algorithms in Section 2.D). We close with an as yet unpublished result by the author of this material that solves the approximate string matching problem in $O(DL^{pow(\varepsilon)} \log L)$ expected-time where $pow(\varepsilon)$ is an increasing and concave function that is 0 when $\varepsilon = 0$ and further depends on the size, $|\Sigma|$, of the, underlying alphabet. Thus the algorithm is superior to the $O(LD)$ algorithms and truly sublinear in L when ε is small enough to guarantee that $pow(\varepsilon) < 1$. For example, $pow(\varepsilon)$ is less than one when $\varepsilon < 33\%$ for $|\Sigma| = 4$ (DNA alphabet), and when $\varepsilon < 56\%$ for $|\Sigma| = 20$ (Protein alphabet). More specifically, $pow(\varepsilon) \leq .22 + 2.3\varepsilon$ when $|\Sigma| = 4$ and $pow(\varepsilon) \leq .17 + 1.4\varepsilon$ when $|\Sigma| = 20$. The bounding argument used in proving the expected complexity is rather crude. Consequently, performance in practice is much superior. In preliminary experiments, the approach appears to represent a 100- to 500-fold improvement over the $O(DN)$ search algorithms for problems of interest in molecular biology.

The algorithm assumes that an **inverted index** for the database sequence has already been constructed [McC76,Apo85,MaM90]. An inverted index for a large text is a data structure that allows one to rapidly find all occurrences of a given query string in the text. For our algorithm, queries will all be of length $T = \log_{|\Sigma|} L$. Thus, each query can be uniquely encoded as an $O(L)$ integer, and we can store the results of all possible queries (which are lists of indices where the corresponding strings of size T appear in B) in an $O(L)$ table. This simple structure can be built in $O(L)$ time and $2L$ words of space.

The other major ingredient of the algorithm is the neighborhood concept exploited in BLAST. Let the **D-neighborhood** of a sequence W be the set of all sequences distance less than or equal to D from W, i.e., $N_D(W) = \{ V : D(V, W) \leq D \}$. The underlying model here allows indels and recall that $D(V, W)$ is the distance between the sequences under the shortest edit script model. Let the **condensed D-neighborhood** of W be the set of all strings in the D-neighborhood of W that do not have a prefix in the neighborhood, i.e., $\overline{N_D}(W) = \{ V : V \text{ in } N_D(W) \text{ and no prefix of } V \text{ is in } N_D(W) \}$. One way to find all approximate matches to W is to generate every string in the condensed D-neighborhood of W, then, for each such string, to find the locations at which the string occurs in the database using an inverted index. Each such location is the rightmost position of an approximate match to W. The obvious problem is that as W or D become large the number of strings in $\overline{N_D}(W)$ explodes exponentially, making the standard $O(LD)$ algorithm superior. However, for strings W whose length is $T = \log_{|\Sigma|} L$, the following are true:

- $|\overline{N_D}(W)| \leq L^{pow(\varepsilon)}$ where $pow(\varepsilon) = \log_{|\Sigma|} (c+1)/(c-1) + \varepsilon \log_{|\Sigma|} c + \varepsilon$ and $c = \varepsilon^{-1} + \sqrt{1+\varepsilon^{-2}}$.
- There exists an algorithm to generate the strings in $\overline{N_D}(W)$ in lexicographical order in $O(DL^{pow(\varepsilon)})$ worst-case time. The algorithm involves computing rows of a dynamic programming matrix in response to a backtracking search that essentially traces a trie of the strings in $\overline{N_D}(W)$.

- Using the simple inverted index suggested above, the algorithm above can look up the locations of these strings at no additional overhead, and under the assumption that the database B is the result of Bernouilli trials, finds $O(L^{pow(\varepsilon)})$ matches in expectation.

Therefore, for small strings the strategy of generating all strings in the neighborhood of the query is effective for sufficiently small thresholds. To extend this strategy to larger queries requires the following observation. Consider dividing the query W in a binary fashion until all pieces are of size $\log_{|\Sigma|} L$. To model the various pieces, let W_α for $\alpha \in \{0, 1\}^*$ be recursively defined by the equations: $W_\varepsilon = W$, $W_{\alpha 0}$ = first_half_of(W_α), and $W_{\alpha 1}$ = second_half_of(W_α). The following lemma follows from a simple application of the Pigeon-Hole Principle [StM77].

Lemma: If W aligns to a string V with not more than D errors, then there exists a string α such that for every prefix β of α, W_β aligns to a substring V_β of V with not more than $\lfloor D/2^{|\beta|} \rfloor$ errors. Moreover, V_β is a prefix or a suffix of $V_{\beta a}$ according to whether a is 0 or 1.

This suggests that we can efficiently find approximate matches to A by first finding approximate matches to the strings A_α of length T and then verifying that A_β approximately matches for progressively shorter prefixes β of α. At each stage, the string in question is twice as large and twice as much distance is allowed in the match, but the number of matches found drops hyperexponentially except where the search will reveal a distance D-or-less match to A.

Suppose for simplicity that, M, the length of A equals $2^K T$ for some value of K. For $d = \lfloor D/2^K \rfloor$, we begin by generating the strings in $\overline{N_d}(A_\alpha)$ for each of the 2^K strings A_α of length T. From the above this takes $O(DL^{pow(\varepsilon)})$ total time and delivers this many d-matches. We then see if these d-matches can be extended to $\lfloor D/2^{K-1} \rfloor$-matches to the strings A_β of length $2T$. This step requires envisioning the result of the string generation lookups as delivering parallelogram shaped regions of the dynamic programming matrix for A versus B. All d-matches to a T-string of A are guaranteed to lie in one of these parallelograms. To find matches to $2T$-strings it suffices to do a dynamic programming calculation over a $2T$-by-$2d$ parallelogram about each parallelogram of a T-string ‘‘hit’’. At the k^{th} stage of this process, the number of matches is reduced by a factor of $1/L^{2^k(1-pow(\varepsilon))}$. Thus while the time to extend matches grows by a factor of 4 at each stage, this is overwhelmed by the reduction in the number of surviving matches. In the end, the total time consumed in expectation is $O(DL^{pow(\varepsilon)} \log L + HDM)$ where H is the number of approximate matches to A found.

Note that our algorithm's expected time complexity is based on the assumption that the database is the result of random Bernouilli trials. In this case its expected complexity is $O(DL^{pow(\varepsilon)} \log L)$. However, if the database is not random but preconditioned to have H matches to A, then $O(HDM)$ additional time will be spent on each of these matches. Consequently, our algorithm does not improve upon the $O(ND)$ algorithm for the sequence-vs.-sequence problem. Nonetheless, for searching problems where the database is large and ‘‘sufficiently’’ random, this algorithm can find approximate string matches in an amount of time that is sublinear in the dominant parameter L.

4. Open Problems and Summary.

While sequence comparison is a much studied problem area, new problems continue to arise and new progress is made on existing problems. A fundamental issue is the definition of similarity. We have really focused only on the indel/substitution model and some small variations. The work of Argos [Arg87] mentioned in Section 2.D suggests other possibilities. Other authors have looked at scoring schemes that are intended to reflect the probability of finding a given alignment by chance [BiT86,AlE86]. A fundamental change in the optimization criterion for alignment creates a new set of algorithmic problems.

What about fundamentally speeding up sequence comparisons? As mentioned in Section 1 the best known lower bound is $O(N \log N)$ yet the best worst-case algorithm takes $O(MN/\log^2 N)$ time. Can this gap be narrowed, either from above or below? Can we perform faster database searches for the case of generalized-Levenshtein scores as is suggested by the results of Section 3.C for the approximate string matching problem. Speeding up database searches is very important. Are there other effective ways to parallelize such searches and/or to exploit preprocessing of the databases such as an inverted index [Apo85]?

Biologists are interested in searching databases for patterns other than given strings or regular expressions. Are there fast algorithms for finding approximate repeats, e.g., $x < 5,\ 10 > x\%5$ is my notation for a pattern that matches some string *x* and then 5 to 10 symbols to the right matches the same string modulo 5% differences. Many DNA structures are induced by base-pairing that can be viewed as finding approximate palindromes separated by a given range of spacing. More intricate patterns for protein motifs and secondary structure are suggested by the systems QUEST [AWM84] and ARIADNE [LSW87], both of which pose problems that could use algorithmic refinement.

Finally, biologists compare objects other than sequences. For example, the partial sequence information of a *physical map* can be viewed as a string on which one has placed a large number of beads of one of, say eight colors, at various positions along the string. Given two such maps, are they similar? This problem has been examined by several authors [WSK84,MOR90,MyH90,RMW90]. There are still fundamental questions as to what the measure of similarity should be and for each, designing efficient algorithms. There has also been work on comparing phylogenetic trees and chromosome staining patterns. Indubitably the list will continue to grow.

References.

[AAL90] Apostolico, A., M.J. Atallah, L.L. Larmore, and S. McFaddin, ‘‘Efficient parallel algorithms for string editing and related problems’’, *SIAM J. on Computing* **19**(5) (1990), 968-988.

[ACo75] Aho, A.V. and M. Corasick, ‘‘Efficient string matching: and aid to bibliographic search’’, *Comm. ACM* **18**(6) (1975), 333-340.

[AGM90] Altschul, S.F., W. Gish, W. Miller, E.W. Myers, and D.J. Lipman, ‘‘A basic local alignment search tool’’, *J. Mol. Biol.* **215** (1990), 403-410.

[AhP72] Aho, A.V. and T.G. Peterson, ‘‘A minimum distance error-correcting parser for context-free languages’’, *SIAM J. on Computing* **1**(4) (1972), 305-312.

[AHU76] Aho, A.V., D. Hirschberg and J. Ullman, ‘‘Bounds on the complexity of the longest common subsequence problem’’, *J. ACM* **23**(1) (1976), 1-12.

[AlE86] Altschul, S.F. and B.W. Erikson, ‘‘Locally optimal subalignments using nonlinear similarity functions’’, *Bull. of Math. Biol.* **48**(5/6) (1986), 633-660.

[Apo85] Apostolico, A., ‘‘The myriad uses of subword trees’’, in *Combinatorial Algorithms on Words*, A. Apostilico and Z. Galil, Eds. (Springer-Verlag, 1985), 85-96.

[Arg87] Argos, P., ‘‘A sensitive procedure to compare amino acid sequences’’, *J. Mol. Biol.* **193** (1987), 385-396.

[AWM84] Arbarbanel, R.M., P.R. Wieneke, E. Mansfield, D.A. Jaffe, and D.L. Brutlag, ‘‘Rapid searches for complex patterns in biological molecules’’, *Nucleic Acids Research* **12**, 1 (1984), 263-280.

[BiT86] Bishop, M.J. and E.A. Thompson, ‘‘Maximum likelihood alignment of DNA seqeunces’’, *J. Mol. Biol.* **190** (1986), 159-165.

[BoM77] Boyer, R. and J. Moore, ‘‘A fast string searching algorithm’’, *Comm. ACM* **20**(10) (1977), 262-272.

[CaL88] Carillo, H. and D. Lipman, ‘‘The multiple sequence alignment problem in biology’’, *SIAM J. Appl. Math.* **48** (1988), 1073-1082.

[ChL90] Chang, W.I. and E.L. Lawler, ‘‘Approximate matching in sublinear expected time’’, *Proc. 31st IEEE Symp. on Foundation of Computer Science* (1990), 116-124.

[CMS88] Committee on Mapping and Sequencing the Human Genome, *Mapping and Sequencing the Human Genome* (National Academy Press, 1988), ISBN# 0-309-03840-5.

[DBH83] Dayhoff, M.O., W.C. Barker, and L.T. Hunt, ‘‘Establishing homologies in protein sequences’’, *Methods in Enzymology* **91** (1983), 524-545.

[Dre69] Dreyfus, S., ‘‘Appraisal of some shortest path algorithms’’, *Oper. Res.* **17** (1969), 395-412.

[DuN82] Dumas, J.P. and Ninio, J. ‘‘Efficient algorithms for folding and comparing nucleic acid sequences,’’ *Nucleic Acids Research* **10** (1982), 197-206.

[Ear70] Earley, J., ‘‘An efficient context-free parsing algorithm’’, *Comm. ACM* **13**(2) (1970), 94-102.

[EGG88] Eppstein, D., Z. Galil, and R. Giancarlo, ‘‘Speeding up dynamic programming’’, *IEEE Symp. on Foundations of Computer Science* (1988), 488-496. (Also *Theoretical Computer Science* **64** (1989), 107-118.)

[EGG90] Eppstein, D., Z. Galil, R. Giancarlo, and G.F. Italiano, ‘‘Sparse dynamic programming’’, *Proc. Symp. on Discrete Algorithms* (1990), 513-522.

[FeD87] Feng, D.F. and R.F. Doolittle, ‘‘Progressive sequence alignment as a prerequisite to correct phylogenetic trees’’, *J. Mol. Evol.* **25** (1987), 351-360.

[FiB88] Fickett, J.W. and C. Burks, ‘‘Developement of a database for nucleic acid sequences’’, in *Mathematical Methods for DNA Sequences*, M.S. Waterman, Ed. (CRC Press, 1988), 1-34.

[Fic84] Fickett, J.W., ‘‘Fast opitimal alignment’’, *Nucleic Acids Research* **12**(1) (1984), 175-179.

[FiS83] Fitch, W.S. and Smith, T.F. ‘‘Optimal sequence alignments,’’ *Proc. Natl. Acad. Sci. USA* **80** (1983), 1382-1386.

[FJD85] Feng, D.F., Johnson, M.S., and Doolittle, R.F. ‘‘Aligning amino acid sequences: comparision of commonly used methods,’’ *J. Molecular Evolution* **21** (1985), 112-125.

[Fox73] Fox, B., ‘‘Calculating the Kth shortest paths’’, *INFOR — Canad. J. Oper. Res. Inform. Process* **11** (1973),66-70.

[GaJ79] Garey, M.R. and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Complete Problems*, W.H. Freeman, 1979.

[GBH86] George, D.G., W.C. Barker, and L.T. Hunt, ‘‘The protein identification resource (PIR)’’, *Nucleic Acids Research* **14** (1986), 11-15.

[Got82] Gotoh, O., ‘‘An improved algorithm for matching biological sequences’’, *J. Molec. Biol.* **162** (1982), 705-708.

[GoK82] Goad, W.B. and M.I. Kanehisa, ‘‘Pattern recognition in nucleic acid sequences I: a general method for finding local homologies and symmetries’’, *Nucleic Acids Research* **10**(1) (1982), 247-263.

[Hir75] Hirschberg, D. S., ‘‘A linear space algorithm for computing longest common subsequences’’, *Comm. ACM* **18**, 341-343.

[Hir78] Hirschberg, D.S., ‘‘An information-theoretic lower bound for the longest common subsequence problem’’, *Information Processing Letters* **7**(1) (1978), 40-41.

[HoP59] Hoffman, W., and R. Pavley, ‘‘Method of solution ot Nth best path problem’’, *J. of ACM* **6** (1959), 506-514.

[HoS78] Horowitz, E. and S. Sahni, *Fundamentals of Computer Algorithms* (Computer Science Press, 1978), 198-247.

[HoU79] Hopcroft, J.E. and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation* (Addison-Wesley, 1979), 13-76.

[HuS77] Hunt, J. W. and T. G. Szymanski, ‘‘A fast algorithm for computing longest common subsequences’’, *Comm. ACM* **20**(5) (1977), 350-353.

[Jho77] Johnson, D.B., ‘‘Efficient algorithms for shortest paths in sparse networks’’, *J. ACM* **24**(1) (1977), 1-13.

[KMP77] Knuth, D.E., J.H. Morris, and V.R. Pratt, ‘‘Fast pattern matching in strings’’, *SIAM J. on Computing* **6**(2) (1977), 323-350.

[LaV86] Landau, G.M. and U. Vishkin, ‘‘Introducing efficient parallelism into approximate string matching and a new serial algorithm’’, *Symp. on Theory of Computing* (1986), 220-230.

[LaS90] Larmore, L.L. and B. Schieber, ‘‘On-line dynamic programming with applications to the prediction of RNA secondary structure’’, *Proc. Symp. on Discrete Algorithms* (1990), 503-512.

[Law72] Lawler, E.L. ‘‘A procedure for computing the K best solutions to discrete optimization problems and its application to the shortest paths problem,’’ *Management Science* **18** (1972), 401-405.

[LiL85] Lipton, R.J. and D. Lopresti, ‘‘A systolic array for rapid string comparison’’, *Chapel Hill Conference on Very Large Scale Integration*, Henry Fuchs, Ed. (Computer Science Press, 1985), 363-376.

[LiP85] Lipman, D.J. and W.R. Pearson, ‘‘Rapid and sensitive protein similarity searches’’, *Science* **227** (1985), 1435-1441.

[LMT88] Lander, E., J.P. Mesirov, and W. Taylor, ‘‘Protein sequence comparison on a data parallel computer’’, *Proc. International Conference on Parallel Processing* (1988), 257-263.

[LWS87] Lathrop, R.H., T.A. Webster, and T.F. Smith, ‘‘ARIADNE: A flexible framework for protein structure recognition’’, *Comm. of the ACM* **30** (1987), 909-921.

[MaP80] Masek, W.J. and M.S. Paterson, ‘‘A faster algorithm for computing string-edit distances’’, *J. Computing Systems Science* **20**(1) (1980), 18-31.

[MaM90] Manber, U. and E. Myers, ‘‘Suffix Arrays: A New Method for On-Line String Searches’’, *Proc. Symp. on Discrete Algorithms* (1990), 319-327.

[MaW90] Manber, U. and S. Wu, ‘‘An algorithm for approximate string matching with non-uniform costs’’, Tech. Rep. 89-19, Dept. of Computer Science, U. of Arizona, Tucson, AZ 85721.

[McC76] McCreight, E.M., ‘‘A space-economical suffix tree construction algorithm’’, *J. of the ACM* **23**(2) (1976), 262-272.

[MiM88] Miller, W. and E.W. Myers, ‘‘Sequence comparison with concave weighting functions,’’ *Bull. Math. Biology* **50**(2) (1988), 97-120.

[MOR90] Miller W., J. Ostell and K.E. Rudd, ‘‘An algorithm for searching restriction maps’’, *CABIOS* **6** (1990), 247-252.

[Mye86a] Myers, E.W., ‘‘An O(ND) difference algorithm and its variants’’, *Algorithmica* **1** (1986), 251-266.

[Mye86b] Myers, E.W., ‘‘Incremental alignment algorithms and their applications’’, Tech. Rep. 86-22, Dept of Computer Science, U. of Arizona, Tucson, AZ 85721.

[MyH90] Myers, E.W. and X. Huang, ‘‘An $O(N^2 \log N)$ Restriction Map Comparison and Search Algorithm,’’ *Bulletin of Mathematical Biology*, to appear.

[MyM86] Myers, E.W. and D. Mount, ‘‘Computer program for the IBM personal computer that searches for approximate matches to short oligonucleotide sequences in long target DNA sequences’’, *Nucleic Acids Research* **14**(1) (1986), 1025-1041.

[MyM88] Myers, E.W. and W. Miller, ‘‘Optimal alignments in linear space’’, *CABIOS* **4**(1) (1988), 11-17.

[MyM89] Myers, E.W. and W. Miller, ‘‘Approximate matching of regular expressions’’, *Bull. of Math. Biol.* **51**(1), 5-37.

[NeW70] Needleman, S.B. and Wunsch, C.D. ‘‘A general method applicable to the search for similarities in the amino-acid sequence of two proteins,’’ *J. Molecular Biology* **48** (1970), 443-453.

[RMW90] Rudd K.E., W. Miller, C. Werner, J. Ostell, C. Tolstoshev and S.G. Satterfield, ‘‘Mapping sequenced E. coli genes by computer: Software, strategies, and examples’’, *Nucleic Acids Research* **19** (1991), 637-647.

[San75] Sankoff, D. ‘‘Minimal Mutation Trees of Sequences’’, *SIAM J. appl. Math* **28**(1) (1975), 35-42.

[Sel74] Sellers, P.H., ‘‘On the theory and computation of evolutionary distances’’, *SIAM J. app. Math.* **26** (1974), 787-793.

[Sel80] Sellers, P.H., ‘‘The theory and computation of evolutionary distances: pattern recognition’’, *J. Algorithms* **1** (1980), 359-373.

[Sel84] Sellers, P.H., ‘‘Pattern recognition in genetic sequences by mismatch density’’, *Bull. Math. Biol.* **46**(4) (1984), 501-514.

[Sel87] Sellers, P.H., Personal communication.

[SmW81] Smith, T.F. and M.S. Waterman, ‘‘Identification of common molecular sequences’’, *J. Mol. Biol* **147** (1981), 195-197.

[StM77] Stanat, D.F. and D.F. McAllister, *Discrete Mathematics in Computer Science* (Prentice-Hall, 1977), 219.

[SWF81] Smith, T.F., M.S. Waterman, and W.M. Fitch, ‘‘Comparative biosequence metrics’’, *J. Molec. Evol.* **18** (1981), 38-46.

[Tho68] Thompson, K. ‘‘Regular expression search algorithm.’’ *Comm. ACM* **11**, 6 (1968), 419-422.

[Ukk85a] Ukkonen, E., ‘‘Algorithms for approximate string matching’’, *Information and Control* **64** (1985), 100-118.

[Ukk85b] Ukkonen, E., ‘‘Finding approximate patterns in strings’’, *J. of Algorithms* **6** (1985), 132-137.

[WaB85] Waterman, M.S. and T.H. Byers, ‘‘A dynamic programming algorithm to find all solutions in a neighborhood of the optimum’’, *Math. Biosciences* **77** (1985), 179-188.

[WaE87] Waterman, M.S. and M. Eggert, ‘‘A new algorithm for best subsequence alignments with application to tRNA-rRNA comparisons’’, *J. Mol. Biol.* **197** (1987), 723-728.

[Wat84] Waterman, M.S., ‘‘Efficient sequence alignment algorithms’’, *J. Theor. Biol.* **108** (1984), 333-337.

[Wat90] Waterman, M.S., personal communication.

[WiL83] Wilbur, W.J. and D.J. Lipman, ‘‘Rapid similarity searches of nucleic acid and protein data banks’’, *Natl. Acad. Sci. USA* **80** (1983), 726-730.

[WSB76] Waterman, M.S., T.F. Smith, and W.A. Beyer, ‘‘Some biological sequence metrics’’, *Advances in Math.* **2O** (1976), 367-387.

[WSK84] Waterman, M.S., T.F. Smith and H.L. Katcher, ‘‘Algorithms for restriction map comparisons’’, *Nucl. Acids Res.***12** (1984), 237-242.