AN ALGORITHM FOR LOCATING NON-OVERLAPPING REGIONS OF MAXIMUM ALIGNMENT SCORE

SAMPATH K. KANNAN* AND EUGENE W. MYERS†

Abstract. In this paper we present an $O(N^2 \log^2 N)$ algorithm for finding the two non-overlapping substrings of a given string of length N which have the highest-scoring alignment between them. This significantly improves the previously best known bound of $O(N^3)$ for the worst-case complexity of this problem. One of the central ideas in the design of this algorithm is that of partitioning a matrix into pieces in such a way that all submatrices of interest for this problem can be put together as the union of very few of these pieces. Other ideas include the use of candidate-lists, an application of the ideas of Apostolico *et al.*[1] to our problem domain, and divide and conquer techniques.

1. Introduction. Let $A = a_1 a_2 ... a_N$ be a sequence of length N, and let $A[p..q]$ denote the substring $a_p a_{p+1} ... a_q$ of A. The problem we consider is that of finding the score of the best alignment between two substrings $A[p..q]$ and $A[r..s]$ under the the generalized Levenshtein model of alignment [6, 11] which permits substitutions, insertions, and deletions of arbitrary score. This problem is a formalization of the problem, encountered by molecular biologists, of automatically detecting repeated regions in DNA and protein sequences. This problem has recently been considered by Miller[8]. When there is no restriction that the regions be non-overlapping, he points out that the problem can be solved in $O(N^2)$ time by a straight-forward modification of the algorithm of Smith and Waterman[10] that finds the highest-scoring local alignment between two sequences. Miller then goes on to consider the restriction that the regions be non-overlapping and presents a worst-case $O(N^3)$ algorithm which runs in $O(N^2)$ in practice. His method involves the use of the candidate-list paradigm which we review briefly in Section 2 since we use some of these ideas in our algorithm. Miller calls non-overlapping regions *twins*, and as his result indicates, this constraint appears to make the problem much harder. For the rest of the paper we consider only the problem of finding the best scoring twins.

There is a related problem where the goal is to find non-overlapping regions which are *exact* repeats. This problem has been dealt with before and turns out to be of significantly lower complexity than the problem of finding the best scoring twins. When the exactly repeating regions are required to be adjacent or *tandem*, i.e. when the goal is to find the longest substring of A of the form ww, Main and Lorentz[7] provide an $O(N \log N)$ algorithm. If gaps are allowed between exactly repeating substrings, the problem becomes even simpler and can be handled in $O(N)$ time, by first creating a suffix tree and then computing the largest and smallest indices (of suffixes) which go through each internal node in post-order. The path to the deepest internal node whose smallest and largest index suffixes are sufficiently far apart, gives us the desired repeated substring.

In a development parallel to ours, Landau and Schmidt [5] have extended the algorithm of Main and Lorentz to find approximate *tandem* repeats with *K-or-less differences*: a thresholded variation of the problem considered here restricted to the simple Levenshtein measure of similarity (i.e., unit cost insertion, deletion, and substitution). Under these stronger conditions they were able to develop an $O(NK log N log K)$ threshold-sensitive algorithm. It is interesting to note that their algorithm provides a bridge between the $O(N log N)$ exact algorithm of Main and Lorentz (where $K = 0$) and our unthresholded $O(N^2 log^2 N)$ algorithm (where $K = N$) for generalized Levenshtein

* Supported in part by NSF grant CCR-9108969.
† Supported in part by NLM grant LM-04960 and NSF grant CCR-9002351

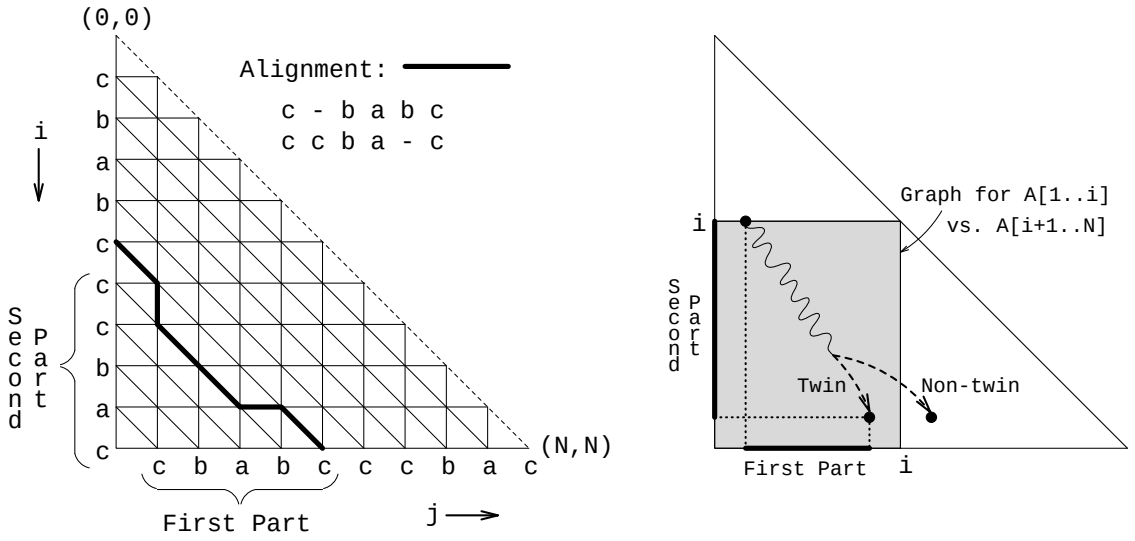

FIG. 1. *Edit Graph Illustrations*

measures. While the results for exact matching might suggest that approximate tandem repeats would be harder to find than twins, we will show that our algorithm can easily be modified to report only tandem repeats as a special case.

The rest of the paper is organized as follows. In Section 2 we define some concepts and review the results that are used in the construction of our algorithms. In Section 3 we present a relatively simple algorithm for the problem of finding twins which runs in time $O(N^{2.5}\log^{0.5} N)$. This algorithm already incorporates some of the ideas used in the more complex algorithm and so serves as a useful preliminary exercise. In Section 4 we present an algorithm which achieves a running time of $O(N^2\log^2 N)$ which is within a polylog factor of being optimal. In Section 5 we consider the problem of finding the best twins under the condition that the best twins are of size no more than $O(N^{1-\epsilon})$ for arbitrarily small values of ϵ. For this problem we present an algorithm which runs in time $O(N^2)$. In Section 6 we describe open problems mainly concerned with improving the space complexity of our algorithm.

2. Preliminaries. Throughout the paper we wish to think about the problem in terms of finding paths in a weighted *edit graph*[9] and performing the computation over the associated *dynamic programming matrix*[11]. The edit graph for sequence A versus itself consists of a lower triangular matrix of vertices (i, j) for $0 \leq j \leq i \leq N$ with up to three edges directed into (i, j): a *substitution* edge from $(i-1, j-1)$ weighted $\delta(a_i, a_j)$, an *insertion* edge from $(i-1, j)$ weighted $\delta(a_i, \epsilon)$, and a *deletion* edge from $(i, j-1)$ weighted $\delta(\epsilon, a_j)$. Edges from nonexistent vertices and substitution edges on the main diagonal are not present. The scoring scheme δ may be chosen arbitrarily but in most application contexts is such that edge weights are negatively biased and only the substitution of similar symbols are given positive score. As illustrated in Figure 1, any path from vertex (p, r) to (q, s) models an alignment between $A[p+1..q]$ and $A[r+1..s]$ and the weight of the path is the score of the alignment. The correspondence is isomorphic, and so it suffices to think in terms of finding high-scoring paths in the edit graph. Limiting the graph to the lower-triangular part simply eliminates local alignments that cannot be twins, because any path crossing the main diagonal aligns overlapping regions.

The Smith-Waterman algorithm for local alignments applied to the edit graph for A reduces to evaluating the following fundamental recurrence for $C(i, j)$, the cost of the best path to (i, j) from some predecessor in the graph, in lexicographical order of i and j:

$$\text{For } j \leq i : C(i,j) = \max \begin{cases} C(i-1, j-1) + \delta(a_i, a_j) & \text{if } i > j > 0 \\ C(i-1, j) + \delta(a_i, \epsilon) & \text{if } i > 0 \\ C(i, j-1) + \delta(\epsilon, a_j) & \text{if } j > 0 \\ 0 & \text{always} \end{cases}$$

The terms qualified by an *if*-clause are present only if the condition is true. The score of the best substring alignment is given by $\max_{j \leq i} \{C(i, j)\}$. Because the edit graph on A involves just the lower triangular part of the underlying dynamic programming matrix, the trivial answer of aligning A with itself is precluded.

However, while the above finds the best scoring path in the edit graph, it does not necessarily align non-overlapping substrings. Figure 1 illustrates that if a path starts at (i, j) and ends at (x, y) then it models a twin only if $y \leq i$, i.e. it ends in a column whose index is not greater than that of the row it starts in. Alternatively, a path is a twin if there exists an i such that the path lies entirely in the rectangular subgraph delimited by row i and column i, in which case we say the twins are *separated by* i. This observation leads to the obvious $O(N^3)$ algorithm for the twins problem: For each $i \in [1, N-1]$, run the Smith-Waterman algorithm for $A[1..i]$ versus $A[i+1..N]$, and record the best answer over all possible separators i.

Miller[8] obtained an algorithm for finding twins that is more efficient in practice by computing $C(i, j, k)$, the best path to (i, j) from row k for each value of $k \leq i$.

$$\text{For } j \leq k \leq i\text{: } C(i,j,k) = \max \begin{cases} C(i-1, j-1, k) + \delta(a_i, a_j) & \text{if } i > j > 0 \text{ and } i > k \\ C(i-1, j, k) + \delta(a_i, \epsilon) & \text{if } i > 0 \text{ and } i > k \\ C(i, j-1, k) + \delta(\epsilon, a_j) & \text{if } j > 0 \text{ and } i \geq k \\ 0 & \text{if } i = k \end{cases}$$

Given these quantities, the best twin ending at (i, j) is simply $\max_{j \leq k} C(i, j, k)$. Of course computing the above directly still takes $O(N^3)$ time, but Miller made the further observation that if $C(i, j, k) \geq C(i, j, h)$ and $k \geq h$ then there is no need to compute $C(i, j, h)$ because the other term can participate in any maximum twin that the former does. We say that k *dominates* h at (i, j). Miller levers this observation by keeping at (i, j), a list of *candidates* (k, c) in increasing order of k such that $c = C(i, j, k)$ and k is not dominated by any other row at (i, j) (which implies that the list is also in decreasing order of c). In practice, the number of rows not dominated at a vertex appears to be constant, and when this is true computing a candidate list from the candidate lists of its immediate predecessors takes constant time. Thus Miller observes $O(N^2)$ behavior in practice, although it is possible that each candidate-list could have as many as $\Omega(N)$ elements in it and so take $O(N^3)$ in the worst case.

Note that finding tandem or adjacent twins is simply a matter of examining just the entries $C(i, j, j)$ (i.e., the best tandem twin ending at (i, j)) in the formulation above. So Miller's algorithm immediately solves the tandem variation as a special case. Since our algorithms effectively perform the same computation as Miller's, it will follow that both our simple and penultimate algorithms solve the tandem variation within the same complexities as they solve the basic twins problem.

We also make extensive use of the results of Apostolico *et al.*[1]. Although their paper is concerned with the design of a parallel algorithm for string matching, some of the techniques in

it carry over to the sequential domain. Specifically, for any $m \times n$ rectangular[1] subgraph E of an edit graph where $m \leq n$, Apostolico *et al.*[1] show that the problem of computing the shortest distances between every one of the $n+m+1$ vertices on the left or top boundary of E to every one of the $n+m+1$ vertices on the right or bottom boundary of E can be done in $O((m+n)^2 \log n)$ time. We will refer to the resulting $(n+m+1) \times (n+m+1)$ table of distances between pairs of boundary vertices of E by $DIST_E$. Another result from this paper will also be relevant to us. This is an 'incremental' version of the previous result and states that given a square, $m \times m$ edit graph E that is decomposed into 4 $m/2 \times m/2$ subgraphs A, B, C, and D, by bisecting vertical and horizontal lines, $DIST_E$ can be computed from $DIST_A$, $DIST_B$, $DIST_C$, and $DIST_D$ in $O(m^2)$ time. Unfortunately, it is beyond the scope of this paper to explain the ingenious algorithms of Apostolico *et al.* that are based on a two-tiered development. However, we note that the procedure *Propagate* described later in this paper is essentially a small variation on the first tier of their design. We highly recommend that interested readers refer to this fundamental result.

3. A Simple $O(N^{2.5} \log^{0.5} N)$ Algorithm. The candidate-list technique of Miller does not give us a better than $O(N^3)$ algorithm because each list can get as large as $\Omega(N)$. In this section we present an algorithm which achieves a better worst-case running time by eliminating the need for candidate-lists with more than b elements in them, where we will choose b as a function of N later. Consider partitioning the interval $[0, N]$ into N/b *panels*, $[0, b]$, $[b, 2b]$, ..., $[N-b, N]$, with the *partition indices* b, $2b$, $3b$, ..., $N-b$. Consider the following first phase:

1. `For each partition index` $i = b, 2b, \ldots, N-b$ `run the Smith-Waterman algorithm on` $A[1..i]$ `versus` $A[i+1..N]$.

The step above detects every twin that is separated by one of the partition indices. Thus the only twins not captured are those where the first part ends and the second part begins within one of the panels. The second phase of our algorithm processes each panel in search of such *panel twins*. The outer loop of the second phase is:

2. `For each panel` $[j, i] = [0, b], [b, 2b], \ldots, [N-b, N]$ `do the following 4 steps:`

Consider a panel twin of $[j, i]$. Figure 2 shows the path of such a twin which must begin at a vertex between rows j and i, and which must end at a vertex between columns j and i. Thus it suffices to compute for all vertices (x, y) between columns j and i, the best paths originating from vertices between rows y and $min(i, x)$. That is, $max_{y \leq z \leq \min(i,x)} C(x, y, z)$ is the cost of the best panel twin ending at (x, y). Moreover, the maximum involves at most $b+1$ candidates, one from each row between j and i. To begin the processing of the panel, consider:

2.1. `For each vertex on boundaries` A `and` B `shown in Figure 2, find the best paths to them that originate in rows` j `through` i `using Miller's recurrence.`

Rigorously stated this step computes $C(x, y, z)$ for $x \in [j, i]$, $y \in [0, j]$, and $z \in [j, x]$ in lexicographical order, and retains the computed quantities at vertices (x, y) such that $x = i$ or $y = j$ in candidate *arrays* indexed by z. Now the crux of the problem is to compute the best paths originating in rows j through i to the vertices on boundary C shown in Figure 2. Using Miller's recurrence and computing all the necessary intermediate quantities in the rectangular subgraph,

[1] Here m and n refer to the length of the sides and not the number of vertices in them, which are $m+1$ and $n+1$ respectively.

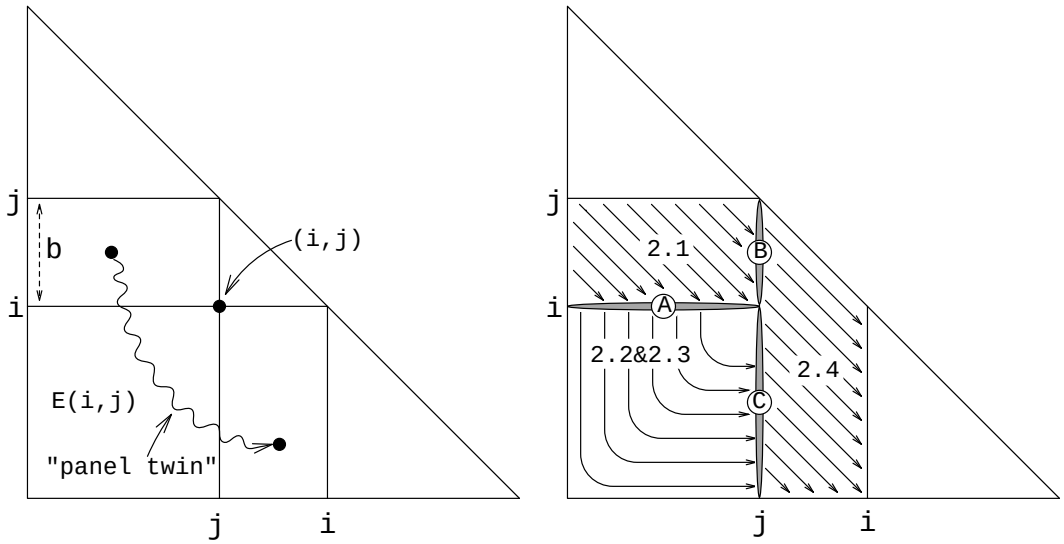

FIG. 2. *Panel Processing for the Simple Algorithm*

$E(i, j)$, whose upper right corner is vertex (i, j), would require $O(N^2 b)$ time which is too costly. This is circumvented by the next two steps as follows:

2.2. `Compute` $M = DIST_{E(i,j)}$ `with the algorithm of Apostolico et al.`

The table M of distances between pairs of points bounding $E(i, j)$ is used to efficiently "propagate" the candidate arrays on boundary A to boundary C. The basic observation is that: $C(x, j, z) = max_{0 \le y \le j}\ C(i, y, z) + M[(i, y)][(x, j)]$ for all $x \in [i, N]$ and $z \in [j, i]$. That is, a best path from row z to (x, j) must pass through row i, and so is decomposable into (1) a best path to some vertex (i, y) on this row, followed by (2) a best path from (i, y) to (x, j). With this preliminary, the next step is:

2.3. `For each vertex on boundary` C `shown in Figure 2, efficiently compute the best paths originating in rows` j `through` i `by propagating the candidate lists from boundary` A `by consulting table` M.

If the maximum embodying propagation were computed directly for each boundary C vertex, we would again take too much time. Fortunately, for a fixed originating row z, it is possible to compute $C(x, j, z)$ for all (x, j) on boundary C in $O(\log N)$ amortized time per vertex by the procedure *Propagate* given below.

Note that the pseudo-code below assumes pass-by-reference semantics and that the index ranges of *Propagate*'s formal arguments are correlated with the matrix slices passed to it as actual arguments. Further note that $C(i, 0..j, z)$ is passed to $I[1..n]$, but that $C(N..i, j, z)$ is passed to $O[1..m]$. The indices must decrease in the later actual argument because *Propagate*'s divide and conquer strategy implicitly requires that the vertices on the boundary of $E(i, j)$ be ordered so that paths from the input boundaries (upper and left) to the output boundaries (lower and right) cross when their start and finish points are inverted with respect to this order. We use as the increasing input boundary order: up the left side and then across the top to the right, and as the increasing output boundary order: across the bottom to the right and then up the right side. For example, for $E(i, j)$ the increasing input boundary order is $(N, 0)$, $(N - 1, 0)$, ..., $(i + 1, 0)$, $(i, 0)$, $(i, 1)$,

..., $(i, j-1)$, (i, j), and the increasing output boundary order is $(N, 0)$, $(N, 1)$, ..., $(N, j-1)$, (N, j), $(N-1, j)$, ..., $(i+1, j)$, (i, j). Our simple algorithm only requires propagation from the top part of the input boundary to the right part of the output boundary, so the call to *Propagate* only passes the appropriate sections of the matrices and distance table M, in the appropriate order. Later in Section 4, we will use *Propagate* for propagation across the entirity of each boundary and the issue of boundary vertex orders will again be important.

Propagate$(I[1..n], D[1..n][1..m], O[1..m])$
  if $m > 0$ **then**
    { $p \leftarrow \lfloor(\frac{m+1}{2})\rfloor$
      $O[p] \leftarrow max_{1 \le x \le n} \{I[x] + D[x][p]\}$
      Determine $\bar{x}$ maximizing the above.
      Propagate$(I[1..\bar{x}], D[1..\bar{x}][1..p-1], O[1..p-1])$
      Propagate$(I[\bar{x}..n], D[\bar{x}..n][p+1..m], O[p+1..m])$

    }

for $z \in [j, i]$ **do**
  Propagate$(C(i, 0..j, z), M[(i, 0..j)][(N..i, j)], C(N..i, j, z))$

A call to *Propagate* correctly sets $O[p] \leftarrow max_{1 \le x \le n} I[x] + D[x][p]$ for every $p \in [1, m]$ by the same observation used by Apostolico *et al.*[1]. Namely, if $O[p]$ is maximized for index $\bar{x}$, then the value of $O[q]$ for some $q < p$ must be maximized for an index between 1 and $\bar{x}$, because otherwise the shortest paths through the subgraph of D used by p and q cross and this leads to an easy contradiction of optimality. Similarly, the value of $O[q]$ for some $q > p$ must be maximized for an index between $\bar{x}$ and n. By choosing p as the bisecting index and recursively solving for the points on the left and right, we achieve an efficient divide-and-conquer procedure as proven later in Theorem 3.1.

2.4. `Given the candidate arrays for vertices on column` j`, find the candidate arrays for every vertex between column` j `and` i`, and record the best scoring panel-twin ending at each.`

This last step is easily accomplished using Miller's recurrence. Rigorously stated we compute $C(x, y, z)$ for all vertices (x, y) between columns j and i, and all $z \in [y, \min(i, x)]$. Simultaneously, we determine the cost, $max_{y \le z \le \min(i,x)} \{C(x, y, z)\}$, of the best panel twin ending at each (x, y).

Note that Steps 2.2 and 2.3 are not needed for panels $[0, b]$ and $[N-b, N]$ because the region $E(i, j)$ degenerates to a line. During the execution of Phase 1 and each execution of Step 2.4, the algorithm keeps a record of the best scoring twin so far. We state this as a final phase:

3. `Output the score of the best twin found over all the stages.`

Although the problem is to output just the score of the best twin, note that the algorithm actually computes the best scoring twin to every vertex in the edit graph of A, and so can produce more than one twin if desired. Further note, that if all that is desired is to examine just adjacent or tandem twins then it suffices in Step 2.4 to take the maximum over just the entries $C(x, y, y)$.

THEOREM 3.1. *The algorithm above computes twins in* $O(N^{2.5}\log^{0.5} N)$ *time and* $O(N^2)$ *space.*

Proof. Phase 1 of the algorithm takes $O(N^3/b)$ time since it involves solving N/b problems each taking time $O(N^2)$. In Phase 2 we perform the minor steps N/b times. Steps 2.1 and 2.4 are similar, involving the computation of candidate arrays in rectangles whose sizes are less than $N \times b$. Since the candidate arrays have up to b elements in them this takes $O(Nb^2)$ for each pair of rectangles or a total of $O(N^2 b)$ time over all panels. Computing a distance table M in Step 2.2 takes $O(N^2 \log N)$ time for a total of $O(N^3 \log N/\, b)$ time over all panels.

It remains to analyze the complexity of Step 2.3. If $T(n,\, m)$ is the time for a call to *Propagate*, then it satisfies the recurrence: $T(n,\, m) \leq T(n_1,\, (m-1)/2) + T(n_2,\, (m-1)/2) + O(n)$ for any n_1 and n_2 such that $n_1 + n_2 = n+1$ and boundary condition $T(n,\, 1) = O(n)$. An easy induction shows that $T(n,\, m)$ is $O(n \log m + m)$. Thus each invocation of *Propagate* in Step 2.3 takes $O(N \log N)$ time for a total time in the step of $O(Nb \log N)$. Over the entire algorithm the time spent is then $O(N^2 \log N)$.

The overall complexity of the algorithm is determined by choosing b to equalize the $O(N^2 b)$ time spent in Steps 2.1 and 2.4 and the $O(N^3 \log N/b)$ time spent in Step 2.2. This is achieved when $b = \sqrt{N \log N}$ and gives a running time of $O(N^{2.5} \log^{0.5} N)$. For the space complexity, note that at worst one $DIST$ table needs to be maintained at any given point in the algorithm. □

4. An Improved Algorithm. In essence the algorithm above is designed around computing panel twins where the size of panels is $N^{1/2}$. One hopes that an $O(N^{7/3}\, polylog\, N)$ algorithm is possible using panels of size $N^{1/3}$, and if so, one can get a progression of decreasing times by choosing panels of size $N^{1/K}$, ultimately yielding an $O(N^2\, polylog\, N)$ algorithm for $K = \log N$. We are indeed able to pull off such a progression but doing so requires several refinements. First, we have to abandon computing distance tables from scratch for each panel problem. Instead, in a preprocessing step we produce a mesh of carefully chosen distance tables that permit us to subsequently propagate candidate lists through critical subgraphs in $O(N \log^2 N)$ time. Second, is that for a basic block size of $N^{1/K}$, we must proceed in K phases, where in the J^{th} phase the panels are of size $N^{J/K}$. In the J^{th} phase, we find the twins that are in a given $N^{J/K}$ panel but not in any $N^{(J-1)/K}$ sub-panel.

To get a more intuitive feel and motivation for what follows, let us consider the development of an algorithm based on multiples of $N^{1/3}$. Suppose we tried our simple 2-level algorithm from the previous section with $N^{2/3}$ panels each of size $N^{1/3}$. In this case we would run into two problems. Performing $N^{2/3}$ Smith-Waterman computations at the outer level (Step 1) would take $O(N^{8/3})$ total time, and in the inner level we would have to compute $N^{2/3}$ distance tables (Step 2.2) for $O(N^{8/3} log N)$ total time. On the other hand, if we tried $N^{1/3}$ panels each of size $N^{2/3}$, the two previously problematic facets would involve only $O(N^{7/3} log N)$ total time which is an improvement over the simple algorithm, but now there would be $N^{2/3}$ candidates to propagate in each lower-level problem giving rise to a component (Steps 2.1 and 2.4) that takes $O(N^{8/3})$ total time. The solution is to go to a 3-level scheme: at the outer level one performs $N^{1/3}$ Smith-Waterman computations to capture all twins that are not in any $N^{2/3}$-panel, at the next level one iterates over the $N^{1/3}$ panels of size $N^{2/3}$ to capture all twins that are in an $N^{2/3}$-panel but not in any $N^{1/3}$-panel, and at the lowest level one iterates over the $N^{2/3}$ panels of size $N^{1/3}$ to captures all twins that lie within them. The complexity of the middle level at first seems to still be problematic as there can be $N^{2/3}$ candidates in an $N^{2/3}$-panel. However, since $N^{1/3}$-twins will be captured by the lowest level, one need not consider every possible start for a twin at this level, but simply the best one in each $N^{1/3}$

panel, reducing the number of "pseudo-candidates" for this level to $N^{1/3}$. This in effect reduces the total complexity to $O(N^{7/3})$ ignoring for the moment the time for building distance tables. Indeed this scheme generalizes to an arbitrary number of levels. In a K-level scheme the outer level captures all twins not within any $N^{(K-1)/K}$-panel, the next level captures all twins within an $N^{(K-1)/K}$-panel but not within a nested $N^{(K-2)/K}$-panel, and so on down to a bottom level that captures all twins within an $N^{1/K}$-panel. At level $J < K$, where one is capturing all twins in an $N^{J/K}$-panel, one need only process the $N^{1/K}$ pseudo-candidates that record the best path starting in a nested $N^{(J-1)/K}$-panel as twins within a panel of that size will be found by the next lower level.

The K-level scheme thus leads to an algorithm requiring $O(KN^{2+1/K})$ time ignoring the time needed to propagate the $O(N^2)$ pseudo-candidates through various regions via distance tables. This is the most difficult obstacle to overcome: we cannot afford to build even the $N^{(K-1)/K}$ distance tables needed for just the lowest level. Instead we take the $O(N)$ set of all rectangular regions of the edit graph that candidates need to be propagated across, conceptually construct a quad-tree decomposition (see [3] for a description of quad-trees) of these regions, and then build a distance table for each region corresponding to a vertex in the quad-tree decomposition. This permits us to propagate candidates across a region by propagating them through a logarithmic number of precomputed distance tables that partition the region. Moreover, the set of all necessary tables can be efficiently precomputed and stored in $O(N^2 logN)$ time and space. We thus find a good tradeoff in the time for propagation against the time for constructing distance tables.

With this preamble we will begin the description of the algorithm. Section 4.1 describes the mesh of distance tables needed to efficiently propagate candidates at every level of the algorithm. Then Section 4.2 describes the K-level decomposition of the problem and the algorithm based on it. Finally, Section 4.3 caps the treatment with an analysis of time and space complexity. Throughout we will assume that $b = N^{1/K}$ is the *basic block size* for some $K \geq 2$ that will be chosen in the final subsection when the competing complexity terms are fully understood. For simplicity we assume that b is a power of 2 and consequently that N is a power of 2 as well.

4.1. Preprocessing and Propagation Technique. The preprocessing step consists of computing distance tables for a collection of subproblems. Term an edit graph vertex (i, j) *critical* if it meets the conditions: (1) i and j are multiples of b and (2) $0 < j < i < N$. For each critical (i, j) we will compute and associate a number of distance tables. Let $D(p, i, j)$ denote the distance table $DIST_{E(p,i,j)}$ for the square subgraph, $E(p, i, j)$ consisting of vertices (x, y) such that $x \in [i, i+p]$ and $y \in [j-p, j]$. In terms of the edit graph, $E(p, i, j)$ is a $p \times p$ square whose upper-right corner is the vertex (i, j). We say $E(p, i, j)$ is *cornered at* (i, j). The distance tables to be associated with a given critical vertex correspond to a doubling progression of square subgraphs cornered at the vertex. Specifically, the list of distance tables built for critical vertex (i, j) is: $D(b, i, j)$, $D(2b, i, j)$, $D(4b, i, j)$, ..., $D(2^x b, i, j)$ where $2^x b$ is the largest power of 2 times the block size that divides both i and j. Let $maxp(i, j) = 2^x b$. Note that because vertices for which $i = j$ are not critical, the largest subgraph for which a distance table is built has $N/4 + 1$ vertices on each side, i.e., $maxp(i, j) \leq N/4$. The left half of Figure 3 illustrates the *mesh* of tables just described. Our preprocessing algorithm is simply:

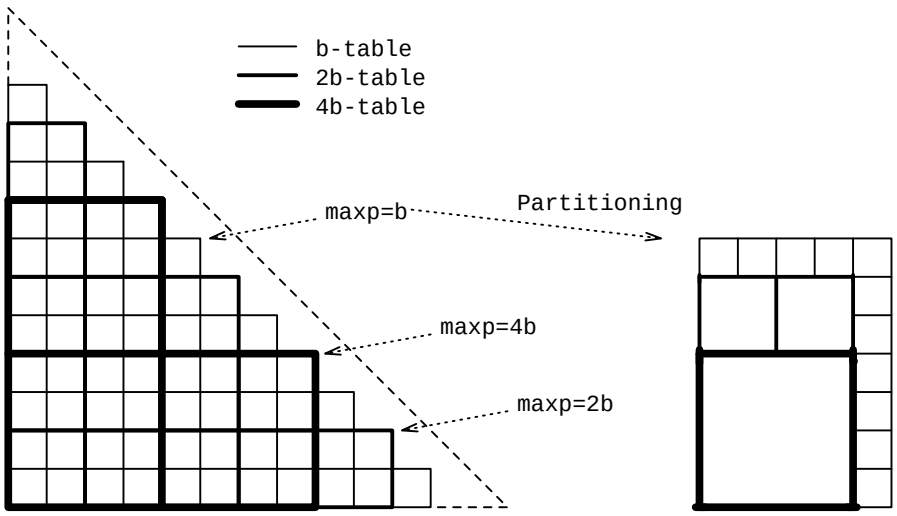

FIG. 3. *Mesh Partitioning for Preprocessing Step*

for $p \leftarrow b, 2b, 4b, ..., N/4$ **do**
  for $i \leftarrow 2p, 3p, 4p, ..., N - p$ **do**
    for $j \leftarrow p, 2p, 3p, ..., i - p$ **do**
    { $maxp(i, j) \leftarrow p$
      if $p = b$ **then**
        Compute $D(p, i, j)$ *de novo* using the algorithm in [1]
      else
        Compute $D(p, i, j)$ by fusing the 4 tables $D(p/2, i, j)$, $D(p/2, i + p/2, j)$, $D(p/2, i, j - p/2)$, and $D(p/2, i + p/2, j - p/2)$ using the algorithm in [1]
    }

In a given iteration of the outer loop, observe that $O((N/p)^2)$ tables are built, and since p doubles with each iteration, $O((N/b)^2)$ tables are built over the entire algorithm. For the iteration where $p = b$, each distance table takes $O(b^2 \log b)$ time to build and occupies $O(b^2)$ space, for upper bounds of $O(N^2 \log N)$ time and $O(N^2)$ space for the iteration. Since fusing 4 $p/2 \times p/2$ tables only takes time $O(p^2)$, the time spent in every iteration other than the first is bounded by $O(N^2)$ time and space. There are $O(\log N)$ iterations of the outer loop, for a grand total of $O(N^2 \log N)$ time and space for the preprocessing step.

The *mesh* of $O((N/b)^2)$ distance tables computed in the preprocessing step, have been chosen so that rectangular subgraphs of the edit graph relevant to our algorithm can be partitioned into $O(N)$ *mesh squares* for which tables have already been computed. For a critical vertex (i, j), as in Section 3 let $E(i, j)$, the *critical subgraph cornered at* (i, j), be the rectangular subgraph consisting of all vertices (x, y) such that $x \in [i, N]$ and $y \in [0, j]$. Suppose 2^x is the largest power of 2 dividing i/b, and 2^z is the largest power of 2 dividing j/b. Let $mp = maxp(i, j) = 2^{\min(x, z)}b$. The subgraph $E(i, j)$ can be partitioned into mesh squares in the following greedy way:

> *Case 1:* $x < z$. In this case $E(i, j)$ is partitioned into (a) the row of $mp \times mp$ mesh squares $E(mp, i, mp)$, $E(mp, i, 2mp)$, $E(mp, i, 3mp)$, ..., $E(mp, i, j)$ along the top boundary of $E(i, j)$, and (b) the partition of $E(i+mp, j)$ obtained by recursively applying this procedure

(unless $i + mp = N$ in which case one is done). Note that $maxp(i + mp, j)$ must be $2mp$ or greater in this case, and so the recursive partitioning of $E(i+mp, j)$ will involve squares of at least twice the size.

Case 2: $x > z$. The situation is symmetric and $E(i, j)$ is partitioned into (a) the recursive partitioning of $E(i, j - mp)$ if $j > mp$, and (b) the column of $mp \times mp$ mesh squares $E(mp, i, j)$, $E(mp, i + mp, j)$, $E(mp, i + 2mp, j)$, ..., $E(mp, N - mp, j)$ along the right boundary of $E(i, j)$. Again note that $maxp(i, j - mp)$ must be $2mp$ or greater in this case, and so the recursive partitioning of $E(i, j - mp)$ will involve squares of at least twice the size.

Case 3: $x = z$. Here one "peels" off both a column and row of $mp \times mp$ squares along the top and right boundaries of $E(i, j)$, and then recursively partitions $E(i + mp, j - mp)$ if it is non-empty. Once again $maxp(i + mp, j - mp)$ must be $2mp$ or greater.

At right, Figure 3 illustrates the partitioning for a particular critical corner. Note that in all cases the partition is into subgraphs cornered at critical vertices and so the distance tables for all the squares constituting the partition of $E(i, j)$ have been computed in the preprocessing step as part of the mesh. Note that the non-recursive part of the partitioning process employs at most $O(N/mp)$ $mp \times mp$ squares, with a total perimeter of $O(N)$ over all squares. Since mp at least doubles with each level of recursion, it follows that the $E(i, j)$ is partitioned into at most $O(N/mp)$ $mp \times mp$ squares, $O(N/2mp)$ $2mp \times 2mp$ squares, $O(N/4mp)$ $4mp \times 4mp$ squares, ..., and $O(1)$ $N/4 \times N/4$ squares. Therefore, $E(i, j)$ is in general partitioned into at most $O(N/maxp(i, j))$ mesh squares whose total perimeter sum is at most $O(N \log N)$. This characteristic of the partitioning is important since the running time of a step in the ensuing algorithm is directly proportional to it.

Having now described how to partition the subgraphs $E(i, j)$, we next show how to propagate candidate list information from the left and top boundaries of $E(i, j)$ to its right and bottom boundaries. For the ensuing algorithm it will only be necessary, as in the simple algorithm of Section 3, to propagate candidates from the top boundary to the right boundary, but in the recursive process below we need to solve the more general left and top boundary to right and bottom boundary propagation problem. Recall from Section 3, the subprocedure *Propagate*$(I[1..n], D[1..n][1..m], O[1..m])$ that propagates a vector I through a distance table D to produce the vector O. Propagation through $E(i, j)$ is accomplished by propagating candidates through the mesh partition using *Propagate* to propagate candidates through each mesh square. Recall also that in general the procedure *Propagate* takes input values along the left and top boundaries and produces output values along the bottom and right boundaries of a rectangular subgraph. In the *Mesh_propagate* procedure described below, we will sometimes have separate vectors representing the values at the left boundary and the values at the top boundary. When we want to make a call to *Propagate* we will combine these vectors using a "concatenate" operator denoted by "$\cdot$". Similarly when we want to separately name the values at the bottom and right boundaries we will also use the concatenate operator and specify two separate output arguments.

Mesh_propagate$(L[i..N], T[0..j], R[i..N], B[0..j])$
  { **vector** $X[mp + 1], Y[N + 1]$
    $mp \leftarrow maxp(i, j)$

```
if j ≠ mp and mp = maxp(i, j − mp) then
  { R[i..i + mp] ← L[i..i + mp]
    for k ← mp, 2mp, 3mp, ..., j do
      { X[0..mp] ← R[i..i + mp]
        Propagate(X[mp..1] · T[k − mp..k], D(mp, i, k), Y[k − mp..k − 1] · R[i + mp..i])
        Y[k] ← R[i + mp]
      }
    if i < N − mp then Mesh_propagate(L[i + mp..N], Y[0..j], R[i + mp..N], B)
  }

else
  { if j > mp then Mesh_propagate(L, T[0..j − mp], Y[i..N], B[0..j − mp])
    B[j − mp..j] ← T[j − mp..j]
    for k ← i, i + mp, i + 2mp, ..., N − mp do
      { X[0..mp] ← B[j − mp..j]
        Propagate(Y[k + mp..k] · X[1..mp], D(mp, k, j), B[j − mp..j − 1] · R[k + mp..k])
        B[j] ← R[k + mp]
      }
  }
}
```

The procedure *Mesh_propagate* above, propagates the values across $E(i, j)$ where the formal parameters are as follows. Vector $L[i..N]$ contains the input values on the left boundary vertices $(i..N, 0)$ (denoting the sequence: $(i, 0)$, $(i + 1, 0)$, ..., $(N, 0)$). Vector $T[0..j]$ contains the input values on the top boundary vertices $(i, 0..j)$, $R[i..N]$ receives the propagated output values on the right boundary vertices $(i..N, j)$, and $B[0..j]$ receives the output values along the bottom boundary $(N, 0..j)$. Note that the input and output vectors redundantly cover the upper-left and lower-right corner vertices, i.e., $L[i] \equiv T[0]$ and $R[N] \equiv B[j]$. *Mesh_propagate* propagates the input vectors through the mesh using the recursive decomposition above. For example, if case 1 is true for (i, j) then the procedure begins by propagating T and $L[i..i + mp]$ through the row of $mp \times mp$ mesh squares to a temporary vector $Y[0..j]$ along the lower output boundary and $R[i..i + mp]$ along the right boundary. Another temporary vector X is used to chain together the propagations through each mesh square via calls to *Propagate*. Note that as in Section 3, great care is taken to feed to *Propagate* the relevant portions of input and output vectors in the relevant order, so as to observe the ordering requirement for distance table boundary vertices. Once propagation through the row of mesh squares is accomplished, the task is completed by recursively mesh propagating $L[i + mp..N]$ and Y to B and $R[i + mp..N]$. Case 2 is also similarly reflected in the pseudo-code of *Mesh_propagate* and Case 3 is effectively handled by first applying Case 1 and then noting that the recursive call will immediately apply Case 2.

Recall that a call to *Propagate* with an $n \times n$ problem takes $O(n \log n)$ time. Because the mesh partition for $E(i, j)$ contains at most $O(N/mp)$ $mp \times mp$ mesh squares, it then follows that at most $O(N \log mp) = O(N \log N)$ time is spent propagating through mesh squares of this size. Moreover, square size doubles with each recursion, so there are a maximum of $O(\log (N/maxp(i, j))) = O(\log N)$ square sizes in the partition of $E(i, j)$. Thus the total time spent in *Mesh_propagate* is bounded above by $O(N \log^2 N)$.

4.2. The K-phase Algorithm. Recall that the basic block size b is $N^{1/K}$ where K is yet to be chosen. For $J \in [1, K]$. Let the N/b^J intervals $[0, b^J]$, $[b^J, 2b^J]$, $[2b^J, 3b^J]$, ..., $[N - b^J, N]$ constitute the set of *J-panels*. A *J-panel twin* is a panel twin for some J-panel but not for any of the $(J-1)$-panels within it. Our K-phase algorithm proceeds through phases $J = K$, $K-1$, ..., 1, where in phase J all J-panel twins are found. Note that the simple algorithm of Section 3 is a specialization of this approach with $K = 2$: Step 1 found the 2-panel twins of the sole 2-panel $[0, N]$, and Step 2 found the 1-panel twins. Note that the general algorithm succeeds in finding every twin as a twin must be a J-panel twin for some $J \in [1, K]$.

Finding the K-panel twins is an easy generalization of Step 1 of the algorithm of Section 3. Namely, for each partition index $i = b^{K-1}, 2b^{K-1}, ..., N - b^{K-1}$ run the Smith-Waterman algorithm on $A[1..i]$ versus $A[i+1..N]$. For phase $J < K$, the algorithm mimics steps 2.1 through 2.4 of the simple algorithm with two significant modifications. First, step 2.2 is no longer required as the mesh is computed earlier, and step 2.3 uses *Mesh_propagate* instead of the simpler *Propagate*. The second modification is in the nature of the C-values used in the phase. Since the goal of phase J is to find J-panel twins and no $(J-1)$-panel twins, one need only keep track of the best path that starts in a given $(J-1)$-panel. That is to know if a path is a J-panel twin of a given J-panel, it suffices to only know which of its b $(J-1)$-subpanels the path starts in. To this end, we introduce the quantity $C^J(i, j, k)$, the best path to (i, j) that begins in the k^{th} $(J-1)$-panel of $[0, N]$, i.e., starts at some vertex between rows kb^{J-1} and $(k+1)b^{J-1}$. Note that for $J = 1$, the definition of C^J coincides with that of C. The recurrence of Section 2 is easily modified to describe the computation of C^J below.

For $j \leq i$ and $k \in [\lceil j/b^{J-1} \rceil, \lfloor i/b^{J-1} \rfloor]$:

$$C^J(i, j, k) = \max \begin{cases} C^J(i-1, j-1, k) + \delta(a_i, a_j) & \text{if } i > j > 0 \text{ and } i > kb^{J-1} \\ C^J(i-1, j, k) + \delta(a_i, \epsilon) & \text{if } i > 0 \text{ and } i > kb^{J-1} \\ C^J(i, j-1, k) + \delta(\epsilon, a_j) & \text{if } j > 0 \text{ and } i > kb^{J-1} \\ 0 & \text{if } kb^{J-1} \leq i \leq (k+1)b^{J-1} \end{cases}$$

With the introduction of C^J the entire computation of phase $J < K$ can now be described succinctly as follows.

Phase J:

For each J-panel $[j, i] \in [0, b^J], [b^J, 2b^J], [2b^J, 3b^J], \ldots, [N - b^J, N]$:

Step $J.1$: `For` $x \in [j, i], y \in [0, j]$, `and` $z \in [j/b^{J-1},\ \lfloor x/b^{J-1} \rfloor]$ `compute` $C^J(x, y, z)$.

Step $J.2$: `For` $z \in [\, j/b^{J-1},\ i/b^{J-1}\,]$
Mesh_propagate($\langle -\infty, -\infty, \ldots, \rangle$, $C^J(i, 0..j, z)$, $C^J(i..N, j, z)$, $\langle -\infty, -\infty, \ldots, \rangle$)

Step $J.3$: `For` $y \in [j, i]$, $x \in [y, N]$, `and` $z \in [\, \lceil y/b^{J-1} \rceil,\ \lfloor \min(i, x)/b^{J-1} \rfloor\,]$ `compute` $C^J(x, y, z)$ `and keep track of the maximum` J`-panel twin found.`

Each quantity $C^J(x, y, z)$ is called a panel-twin candidate. Note that no more than $b+1$ such candidates are computed at any vertex (x, y) in any phase $J < K$. During the execution of the phases the best twin of any type is recorded and this twin is reported upon completion of all the phases.

4.3. Running Time Analysis and Space Refinement. We can now state and prove the following running time bound on our algorithm.

THEOREM 4.1. *The above algorithm computes the optimal pair of twins in time* $O(N^2 \log^2 N)$

Proof. We will consider the total running time for each activity of the algorithm. Firstly, we have already argued that the preprocessing stage takes time $O(N^2 \log N)$. Phase K of the algorithm invokes the $O(N^2)$ Smith-Waterman algorithm b times for a total of $O(bN^2)$ time. Now consider any other phase $J < K$. Steps $J.1$ and $J.3$ compute $O(b)$ panel-twin candidates at each vertex in rectangles of size less than $N \times b^J$, for a total of $O(b^{J+1}N)$ time. Each mesh propagation in Step $J.2$ takes $O(N \log^2 N)$ as analyzed in Section 4.1 and is repeated for each of the $O(b)$ panel-twin candidates at the top boundary for a total of $O(bN \log^2 N)$ for the step. The total number of panels processed in phase J is N/b^J, giving a total time for phase J of $O(N^2(b + \log^2 N/b^{J-1}))$. Summing over all phases, gives a grand total for the algorithm of $O(KbN^2 + N^2 \log^2 N)$ time as the $\log^2 N/b^{J-1}$ term telescopes. Choosing $K = \log N$, makes $b = N^{1/K} = 2$ and gives us the bound $O(N^2 \log^2 N)$. Note that the dominant term comes from the cost of propagation through the mesh. □

In a conference version of this paper [4] the authors asked if the space complexity of the algorithm could be reduced from $O(N^2 logN)$. This was subsequently answered affirmatively by Benson [2] who gave an $O(N^2)$ space algorithm with the same time complexity as ours. While his result uses a distinctive line of attack, we thought that the real essence of the improvement involved abandoning precomputing the entire mesh of distance matrices, and instead computing components of the mesh "on-the-fly" as necessary.

Indeed we now see that the following simple modification to our algorithm gives us $O(N^2)$ space. Instead of proceeding *recursively* phase-by-phase, proceed instead *iteratively* in increasing order of 1-panels $[j, i]$, also processing any J-panels, for $J > 1$, that begin at j.

Recall that the basic block size, b is 2. Formally, consider the following outermost structure:

for $j \leftarrow 0, b, 2b, \ldots, N - b$ **do**
  for $p \leftarrow 2^J b = 2^{J+1}$ s.t. p divides j in order of J **do**
    Perform Steps $J.1$, $J.2$, and $J.3$ on panel $[j, j + p]$

where *no* preprocessing is undertaken. For a given j, we will need the set of distances matrices that partition $E(j + b, j)$. Observe that within this partition we have the partitions for $E(j + p, j)$ for each of the p that will be processed for a given choice of j. So in order to do mesh-propagation in the step $J.2$ of the outline above it suffices to deliver the mesh partition of $E(j + b, j)$ as we iterate through progressive values of j in the outer loop. Suppose we have the partition for $E(j, j - b)$. To obtain the one for $E(j + b, j)$ discard every distance matrix in the former partition and not the latter, and build every distance matrix in the latter but not in the former. We can afford to build a new $p \times p$ matrix "on-the-fly" using the less-efficient $O(p^2 logp)$ *de novo* algorithm of Apostolico *et al.* [1]. At any moment the space required during this process is easily seen to be $O(N^2)$. The final fact required is to observe that a given distance table $DIST(p, k, h)$ is a part of the partition of $E(j + b, j)$ *exactly* when either (a) $j + b \leq k$ and $h \leq j < h + p$, or (b) $h \leq j$ and $k - p < j + b \leq k$. Thus a given distance table of the mesh will be "demanded" and subsequently discarded at most twice. There are $O((N/p)^2)$ tables of size $p \times p$ all together and $O(logN)$ choices of p for a total time to build the tables "on-the-fly" of $O(N^2 log^2 N)$. Note that whereas before we took $O(N^2 logN)$ time and space in the preprocessing, we now take $O(N^2 log^2 N)$ time to build tables in order to use

only $O(N^2)$ space.

5. Finding Short Twins Efficiently. In this section we describe a very simple algorithm to find twins efficiently if we know that the length of the twins is no more than $N^{1-\epsilon}$ for arbitrarily small ϵ. Let $L = N^{1-\epsilon}$ and consider the following recursive approach. Explicitly solve the problem of finding twins between $A[1..N/2]$ and $A[N/2+1..N]$. Recursively solve the subproblems of finding twins within $A[1..N/2+L]$ and $A[N/2-L..N]$ until the length of the string in the subproblem is no more than $3L$. For subproblems of this size solve them by using the algorithm of the previous section in time $O(L^{2-\epsilon})$. Letting $T(n)$ be the time to find optimal twins in a string of length n, we get the following recurrence:

$$T(n) = O(n^2) + 2T(L + n/2)$$

with initial condition, $T(3L) = O(L^{2-\epsilon})$. Solving this recurrence reveals that $T(N) = O(N^2)$.

This algorithm can be simplified somewhat if it is known that the twins sought are of size no more than $L = N^{2/3-\epsilon}$ in which case the algorithm of Section 3 is sufficient to handle the base case problems while still giving a time bound of $O(N^2)$. Finally if it is known that the twins are not of size more than $O(\sqrt{N})$ then the base cases can simply be handled by the brute-force cubic algorithm (or by Miller's improvement of this algorithm) while still giving an overall running time of $O(N^2)$.

6. Conclusions and Open Problems. A practical method for detecting repeated regions in DNA and protein sequences needs to find *multiple* (possibly more than 2) non-overlapping regions of maximum alignment score. Since the algorithm presented in the paper finds the best twins ending at each point in the edit graph it is a relatively simple bookkeeping matter to identify and report more than 2 regions that have high pairwise alignment scores.

The algorithm presented in this paper is perhaps close to optimal in time complexity, but there is the vexing factor of $\log^2 N$ as opposed to $\log N$. Perhaps some modification of the algorithm would make this improvement. One important concern is also the space complexity of the algorithm. Can this be brought down from $O(N^2 \log N)$ to $O(N^2)$ or even less?

The property exploited by the procedures *Propagate* and *Mesh_Propagate* is that optimal scoring paths do not cross. This property no longer holds when the scoring scheme is generalized further to allow affine or concave gap costs. Thus for these generalized scoring schemes it is still open whether there is an algorithm that achieves a running time close to the running time of the algorithm in the paper.

REFERENCES

[1] A. Apostolico, M.J. Atallah, L.L. Larmore, and S. McFaddin, "Efficient Parallel Algorithms for String Editing and Related Problems," *SIAM J. Comput.* **19** (1990), 968–988.

[2] G. Benson, "A space efficient algorithm for finding the best non-overlapping alignment score," *Proc. 5th Symp. on Combinatorial Pattern Matching (Asilomar, CA),* Lecture Notes in Computer Science Vol. 807, (Springer Verlag, 1994), 1–14.

[3] R.A. Finkel and J.L. Bentley, "Quad-trees: a data structure for retrieval on composite key," *Acta Inform.* 4 (1974) 1–9.

[4] S. Kannan and E. Myers, "An algorithm for locating non-overlapping regions of maximum alignment score," *Proc. 4th Symp. Combinatorial Pattern Matching*, Springer-Verlag Lecture Notes in Computer Science, Vol. 648 (1993), 74–86.

[5] G.M. Landau and J.P. Schmidt, "An algorithm for approximate tandem repeats," *Proc. 4th Symp. Combinatorial Pattern Matching* Springer-Verlag Lecture Notes in Computer Science, Vol. 648 (1993), 120–133.

[6] V.I. Levenshtein, "Binary codes of correcting deletions, insertions and reversals," *Soviet Phys. Dokl.* **10** (1966), 707.

[7] M.G. Main and R.J. Lorentz, "An $O(n \log n)$ algorithm for finding all repetitions in a string," *J. of Algorithms* **5** (1984), 422–432.

[8] W. Miller, "An algorithm for locating a repeated region," *manuscript.*

[9] E.W. Myers, "An $O(ND)$ difference algorithm and its variants," *Algorithmica* **1** (1986), 251–266.

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

[11] R.A. Wagner and M.J. Fischer, "The String-to-String Correction Problem," *J. of ACM* **21** (1974), 168–173.