A Time-Efficient, Linear-Space Local Similarity Algorithm*

XIANG-HUANG

Department of Computer Science, Michigan Technological University, Houghton, Michigan 49931

AND

WEIS MILLER

Department of Computer Science, The Pennsylvania State University, University Park, Pennsylvania 16802

Dynamical programming algorithms to determine similar regions of two sequences are useful for analyzing heterogeneous data. This paper presents a time-efficient algorithm that produces $k$ best non-overlapping local alignments for any chosen $k$. The algorithm's worst time complexity is $O(K^2 + N^2)$ when $K$ is the total length of the given sequences and $N$ is the length of the computed alignments.

1. Introduction

Local alignment algorithms locate matching segments within two sequences (Sellers [10]); this contrasts with global alignment methods, which align entire sequences including unconserved regions. A number of dynamic programming algorithms have been designed to produce local alignments (Goad and Kanehisa [2], Hall and Myers [4], Sellers [11], Smith [9]).

*This publication was supported in part by Grant ROI LM08110 from the National Library of Medicine.

337

0190-8269/91 $7.50

Copyright © 1991 by Academic Press, Inc.
All rights of reproduction in any form reserved.
and Waterman [13]. Waterman and Eggert [15]. Waterman [14] offers the opinion that such an algorithm is "probably the most useful dynamic programming algorithm for current problems in biology."

In many contexts, a major drawback of current dynamic programming algorithms for local sequence alignment is the need for space proportional to the product of the two sequence lengths. For global alignments, there exists a dynamic programming algorithm that requires only space proportional to the sum of the sequence lengths (Fischerberg [5], Myers and Miller [7]).

Very recently, Huang et al. [6] used the Myers-Miller global alignment algorithm as part of a local alignment algorithm. The resulting method is a "linear-space algorithm" in the sense that it needs only space proportional to the sum of the input size and the output size. Huang et al. illustrate the method's utility by computing 180 best non-intersecting local alignments of a 73,360-nucleotide sequence containing the human β-like globin cluster and a corresponding 44,394-nucleotide sequence from rabbit. Such a problem is completely outside the scope of quadratic-space algorithms, i.e., those requiring space proportional to the product of the sequence lengths, since 73,360 × 44,394 is over three billion. Huang et al. also point out certain advantages of their method over the widely-used and faster, but less accurate, LEASTA program (Pearson and Lipman [9]). Unfortunately, the simple method that they present is very slow. For each of the k alignments it makes an O(kMN) time sweep of the entire dynamic-programming matrix, so the total time is O(kMN). For example, each of the 180 local alignments for the β-globin sequences required 5 or 6 hours on a Sun 4 workstation.

Here we improve the algorithm of Huang et al. by lowering its time requirement, while retaining its space efficiency. Under reasonable assumptions, the new algorithm's running time is O(kMN + Σ^k n_i / L_i), where L_i is the length of the i-th computed alignment. This represents a considerable savings when both k is large and most of the computed alignments are short compared to the original sequences. For instance, the new algorithm finds the 180 local alignments for the β-globin sequences mentioned above in about 15 h on a Sun 4, a definite improvement over the three weeks required by the simpler algorithm.

The remainder of the paper is organized as follows. In Section 2 we review the local similarity algorithm of Smith and Waterman [13] as extended by Waterman and Eggert [15]. To prepare for later development, this method is explained in terms of an algorithm for finding optimal paths in a certain graph, following Myers and Miller [8]. Section 3 presents the new linear-space algorithm. An example is given in Section 4 to verify the need for particular care in the algorithm's formulation. Section 5 closes by discussing a program that implements the algorithm.
An implementation in the C language of the program described in this paper is freely available from the authors. The simplest way to obtain the program is by electronic mail from huang@cs.msu.edu or webb@cse.psu.edu.

2. WATERMAN'S APPROACH

The sequences A and B consist of symbols chosen from an alphabet Σ. Let s, a unique symbol not in Σ, denote the sequence of zero symbols. Then e is the identity element with respect to the concatenation of sequences, i.e., if e and w are sequences, then ew = w.

An aligned pair has the form \([a, b]\), where \(a, b \in Σ \cup \{e\}\). An alignment is a finite sequence of aligned pairs. An alignment \(S\) aligns \(A\) and \(B\) if \(A\) is the concatenation of upper elements of \(S\) and \(B\) is the concatenation of lower elements. For example, the sequence of four aligned pairs \([e, e; e, e]\) aligns \(par\) and \(xyz\). The null pair, \(\eta = [e, e]\), acts as the identity element for the concatenation of alignments, i.e. if \(a\) and \(b\) are alignments, then \(a\eta b = a\eta b\) and \(a\eta b\) are considered equal. Without loss of generality, assume that alignments consist of non-null aligned pairs.

An aligned pair \((a, b)\) is either an insertion pair, or one with lower entry \(e\) is a deletion pair. Within a given alignment, an insertion gap is a contiguous subsequence of insertion pairs delimited by non-insertion pairs or an end of the alignment. A deletion gap is similarly defined, and collectively such blocks are called gaps. For example, the alignment \([e, e; e, e]\) has an insertion gap of length two and a deletion gap of length one.

A score is assigned to an alignment based on a user-specified scoring function \(σ\) that assigns a real-valued cost to each possible non-null aligned pair, and on a gap penalty \(g > 0\). The score of an alignment \(S\) is simply the sum of the costs of each aligned pair \(σ\) minus a penalty \(g\) for each gap, i.e., score \((S) = Σσ(σ, ν)\), where \(S\) is an aligned pair of \(S\) – \(g \times (\text{number of gaps in } S)\). For local similarity problems, the “average” weight of an aligned pair should be negative so that a random extension of an alignment will lower the score. For instance, with DNA sequences, which have a four-letter alphabet, we might set \(σ([e, e]) = 1\) for all \(e \in Σ\) and of \([e, e]\) = \(-1\) if \(b ≠ e\). In addition, assume that all insertion and deletion pairs have negative weight.

The above definition is slightly more general than the traditional approach to affine gap penalties (Gotoh 33). Beside charging \(g\) for opening a gap, we assess potentially different penalties for each kind of insertion or
deletion pair, traditionally, each pair in the gap is charged a fixed "gaps-extension" penalty.

For sequences $A = a_1 a_2 \cdots a_m$ and $B = b_1 b_2 \cdots b_n$, the alignment graph, $G_{a,b}$, is an edge-labeled directed graph, defined as follows. $G$ has $3(M + 1)(N + 1)$ vertices, denoted $(i, j)_c$, $(i, j)_p$, and $(i, j)_h$, where $i \in [0, M]$ and $j \in [0, N]$. (We use $[x, y]$ to denote the set of integers $i$ such that $x \leq i \leq y$.) 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 $(i, 0)_h \rightarrow (i, j)_h$, labeled $\times$.
2. If $i \in [1, M]$ and $j \in [0, N]$, then there is a deletion initiation edge $(i, 0)_c \rightarrow (i, j)_h$, labeled $\times$.
3. If $i \in [0, M]$ and $j \in [1, N]$, then there is an insertion edge $(i, j)_c \rightarrow (i, j)_p$, labeled $\times$.
4. If $i \in [0, M]$ and $j \in [1, N]$, then there is an insertion initiation edge $(i, j)_c \rightarrow (i, 1)_p$, labeled $\times$.
5. If $i \in [1, M]$ and $j \in [1, N]$, then there is a substitution edge $(i, j)_c \rightarrow (i, j)_h$, labeled $\times$.
6. If $i \in [0, M]$ and $j \in [0, N]$, then there is a null edge $(i, j)_c \rightarrow (i, j)_c$, labeled $\eta$.
7. If $i \in [0, M]$ and $j \in [0, N]$, then there is a null edge $(i, j)_c \rightarrow (i, j)_p$, labeled $\eta$.

Note that the $\times$-vertices in row 0 and the vertices in column 0 are not reachable from other vertices.

Figure 1 illustrates $G_{a,b}$. Rows of the graph after the top-most row are labeled with successively entries of the first sequence; columns after the left-most column are labeled with entries of the second sequence. All edges are annotated with null edges labeled $\eta$.

Let $A_{m,j}$ denote the sequence $a_1 a_2 \cdots a_m$, and let $B_{m,j}$ denote $b_1 b_2 \cdots b_n$. A path in $G_{a,b}$ from $(0, 0)$ to $(i, j)_h$ is said to spell the alignment obtained by concatenating its edge labels. Each path from $(0, 0)$ to $(i, j)_h$ spells an alignment between $A_{m,j}$ and $B_{m,j}$, and every such alignment is spelled by some path. However, there may be many paths from $(m, n)_c$ to $(i, j)_h$, spelling the same alignment. Thus the correspondence between paths and alignments is not one-to-one unless one restricts attention to a canonical subset of the paths. A path is normal if and only if it does not contain subpaths of the form $(i - 1, j)_h \rightarrow (i, j)_h \rightarrow (i, j)_p$ or $(i, j - 1)_c \rightarrow (i, j - 1)_p \rightarrow (i, j)_c$. An exercise, not proven here, shows that alignments between $A_{m,j}$ and $B_{m,j}$ are in one-to-one correspondence with normal paths from $(m, n)_c$ to $(i, j)_c$. 


Our first goal is an algorithm that computes the score and final vertex \((i,j)_c\) of a highest-scoring alignment among all alignments between a substring \(A[m..i]\) of \(A\) and a substring \(B[n..j]\) of \(B\). Assign weights to edges of \(G_{x,y}\) as follows. Deletion initiation and insertion initiation edges are weighted \(o(\pi) - g\), where \(\pi\) is the aligned pair labeling the edge. Null edges are weighted 0 and all other edges are weighted \(o(\pi)\). The weight or score of a path, i.e., the sum of its edge weights, thus equals the sum of the scores of its non-null edges minus \(g\) times the number of initiation edges. For normal paths this is exactly the score of the corresponding alignment since each initiation edge corresponds to the leftmost aligned pair in a gap. Hence the problem is to locate a maximum-weight normal path between any two nodes of \(G\). However, for every non-normal path there is a normal path of greater weight spelling the same alignment, since a subpath such as \((i-1,j)_b \rightarrow (i-1,j)_b \rightarrow (i,j)_b\) can be replaced by \((i-1,j)_b \rightarrow (i,j)_b\) for a net weight gain of \(g > 0\). Thus the problem is to locate a maximum-weight path (it must be normal) in \(G\). Since insertion and deletion edges have negative weights, there is a maximum-weight path that begins and ends with a substitution edge, and hence consists of \(C\)-vertices. A path connecting \(C\)-vertices will be called a \(C\)-path.

Because \(G_{x,y}\) is acyclic, a maximum-weight path can be detemined in a single pass over its vertices, so long as they are taken in a topological order.
\[ D(0, 0) - R(0, 0) = -a \]
\[ C(0, 0) = 0 \]
\[ c_{\text{row max}} = 1, c_{\text{col max}} = 2, c_{\text{best}} = 0 \]

for \( i = 1 \) to \( n \) do
\[ R(i, 0) = -a \]
\[ D(i, 0) = \begin{bmatrix} 1 \\ \vdots \end{bmatrix} - b \]
\[ C(0, j) = 0 \]

for \( j = 1 \) to \( m \) do
\[ (R(i, 0) = -a \]
\[ D(i, 0) = \begin{bmatrix} 1 \\ \vdots \end{bmatrix} - b \]
\[ C(0, j) = 0 \]

for \( j = 1 \) to \( n \) do
\[ U(i, j) = \max (D(i - 1, j), C(i - 1, j)) + \alpha \begin{bmatrix} z_i \\ \vdots \end{bmatrix} \]
\[ U(i, j) = \max (U(i - 1, j), U(i, j - 1)) + \alpha \begin{bmatrix} z_i \\ \vdots \end{bmatrix} \]
\[ C(U, j) = \max \left( 0, (U(i, j)), (U(i, j)), C(i - 1, j) - 1 \right) + \alpha \begin{bmatrix} z_i \\ \vdots \end{bmatrix} \]

if \( C(U, j) > c_{\text{row max}} \) then
\[ c_{\text{row max}} = C(U, j) \]

end for

end for

end for

write "A best local assignment with score: c_{\text{row max}} - c_{\text{best}}" (\( i_{\text{best}}, j_{\text{best}} \))

Fig. 2. The Smith-Waterman local alignment algorithm.

In Fig. 2, the values \( D(i, j) \), \( U(i, j) \), and \( C(i, j) \) are, respectively, the minimum weight of the path from any \( C \)-vertex to \( i_{\text{best}}, j_{\text{best}} \), to \( U(i, j) \), and to \( (i, j) \). Each of these values is found by considering the edges entering the vertex where \( C(U, j) \) we also consider the zero-edge path from \( i_{\text{best}}, j_{\text{best}} \) to itself. For instance, when \( i > 0 \), there are two edges into \( (i, j) \) and one from \( (i - 1, j) \), with weight \( \alpha \begin{bmatrix} z_i \\ \vdots \end{bmatrix} \) and one from \( (i, j) \). Since a maximum weight path to \( (i, j) \), must end with one of these edges and since the portion of the path preceding that last edge must be optimal, \( D(i, j) \) is the larger of
\[ D(i - 1, j) + \alpha(c_{i,j}) \] or \[ C(i - 1, j) + \alpha(c_{i,j}) - \epsilon \] Values at unreachable vertices, i.e., \( D(0, j) \) and \( H(i, 0) \), are defined to be \(-\infty\), a suitably large negative constant.

Waterman and Eggert [15] extend the Smith-Waterman algorithm to compute \( k \) best "non-intersecting" local alignments, where \( k > 0 \) can be chosen arbitrarily. More formally, let \( P_1, P_2, \ldots, P_k \) be \( k \) paths of \( G \) in non-increasing order of scores. \( P_1, P_2, \ldots, P_k \) are \( k \) best non-intersecting paths if for each \( n \in \{1, k\} \), \( P_n \) is a maximum-score path in \( G_n \), where \( G_n \) is obtained by removing the substitution (i.e., diagonal) edges of \( P_1, P_2, \ldots, P_{n-1} \) from \( G \). This definition has the following "non-uniqueness" property. It is possible to have two sequences of \( k \) best non-intersecting paths \( (P_j) \) and \( (Q_j) \) such that for some \( n \), \( P_n \) and \( Q_n \) have different scores. An example of this phenomenon is given in Section 4.

The basic idea of the Waterman–Eggert algorithm is that once all entries \( D(x, y) \), \( I(x, y) \), and \( C(x, y) \) for paths in \( G_a \) are found, the corresponding values in \( G_{a+1} \) can be determined quite efficiently. Indeed, \( D_a(x, y) = D_a(x, y) \) for "most" positions \((x, y)\) (and similarly for \( I \) and \( C \)). Only values at positions "near" \( P_a \) need be recomputed. See the Waterman–Eggert paper for more details. Waterman [14] gives a complete implementation in the C language.

3. A Linear-Space Algorithm

A serious drawback of the Waterman–Eggert algorithm is the need to save all values \( D(x, y) \), \( I(x, y) \), and \( C(x, y) \) from the current graph \( G_a \). (A subscript \( a \) will always indicate that the value is relative to \( G_a \).) Huang et al. [6] give a local alignment algorithm that needs only "linear space," i.e., space proportional to \( M + N + K \) instead of \( MN + K \), where \( K \) is the total length of all computed alignments. Unfortunately, the technique is quite slow: for each \( n \in \{1, k\} \) it computes all entries of \( D_n \), \( I_n \), and \( C_n \) from scratch. While \( O(MN) \) space is needed to store \( k \) alignments. Our goal is to obtain a speedup comparable to that from the Waterman–Eggert approach, while maintaining the linear space bound and guaranteeing optimality of the computed solution.

In conclusion, our approach is as follows. We perform the Smith-Waterman algorithm (Fig. 2), saving \( k \) best scores instead of just one. Care is required since typically the \( k \) largest values \( C(x, y) \) will occur at vertices \((x, y)\) that are packed tightly together. To avoid this problem, we save values \( C(x, y) \) and \( C(x, y) \) only if there are disjoint optimal paths ending at \((x, y)\) and \((x, y)\). However, this approach introduces another problem. It may be that the terminal vertex of, say, the \( k \)th overall best
non-intersecting path is also the endpoint of a higher scoring path that shares its startpoint with local alignment \#1. It is only after we remove the substitution edges of the best path that the Smith-Waterman method can discern the 2nd best path. In other words, for \( k \) best non-intersecting paths that can be found in one Smith-Waterman pass over the C-matrix may not include the \( k \) overall best paths. Huang et al. show that this phenomenon is important in practice.

Once we have determined the best path, the problem is to discover any high-scoring paths that were hidden from the first Smith-Waterman pass. This can be done by making a "backward" Smith-Waterman pass over a limited portion of the graph to discover the "region of influence" of the path being removed, followed by a forward Smith-Waterman pass over this region to find the newly-exposed paths. It is not immediately apparent how one can determine a region that is guaranteed to be sufficiently large; see Lemma 3, below, for our solution. In general, we need one complete pass over the matrix, followed by limited forward and backward passes for each removed path (except for the final path). Our record of the \( k \) best non-intersecting paths is updated whenever we find a path that surpasses our current \( k \)th best path. With this outline, we are ready to consider the details.

The first observation we need is that the Smith-Waterman algorithm (Fig. 3) can be modified to compute the first vertex of an optimal path. As will be shown in Section 4, it is critical how we select this first vertex in the case of ties, i.e., when there are several optimal paths to \((i,j)\), with different initial vertices, In what follows, let \( <_s \) denote any topological order on the \( C \)-vertices of \( G \). This ordering can be the same as the order in which vertices are treated in Fig. 2, or it can be different. First \((i,j)_0 \) is defined to be the last \( C \)-vertex in this ordering such that there is a path in \( G \), from that vertex to \((i,j)\), of score \( D(i,j) \). First \((i,j)_0 \) and First \((i,j)_0 \) are defined similarly.

Figure 2 can be augmented so that vertices First \((i,j)\) are computed. The basic idea is simply to inherit First from whichever incoming path is found to be optimal. (For \( b \)-edge optimal paths set First \((i,j)_{b} = (i,j)_{b} \).) For example, the line

\[
D(i,j) = \max\{D(i-1,j), C(i-1,j) - g\} + \delta(i,j)
\]

becomes

\[
\text{if } D(i,j) > C(i-1,j) - g \text{ then } \quad D(i,j) \leftarrow D(i-1,j) + \delta(i,j)
\]

First \((i,j)_0 \) = First \((i-1,j)_0 \)
point of a higher scoring path that is not a leaf. It is only after we remove the graph the Smith–Waterman method can
words, the best non-intersecting Waterman pass over the C-matrix
paths. Huang et al. show that this 4th, the problem is to discover any path on the first Smith–Waterman pass, and
Smith–Waterman pass over a or the "region of influence" of the weighted Smith–Waterman pass over paths. It is not immediately apparent
in general, we need one complete forward and backward passes for a single path. Our record of the k best
we find a path that outscores any other. We are ready to consider the
the Smith–Waterman algorithm
of an optimal path. As we select this first vertex in the
We next select the first vertex to (i,j), with
s, let < denote any topological
ring can be the same as the other
field, First(1,1), and First(1,2),
ties First(i,j), are computed. The
which is the set First(i,j) = {i,j,1.2} For
\S_i + (i, j) - g \right] + \sigma [\begin{bmatrix} a \\ b \end{bmatrix}]

\left\{ \begin{array}{l}
\text{else if } D(i-1, j) < C(i-1, j) - g \text{ then }
\quad D(i, j) = C(i-1, j) + \sigma [\begin{bmatrix} a \\ b \end{bmatrix}]
\quad \text{First}(i, j) = \text{First}(i-1, j)
\end{array} \right.

\text{else /* tiebreaker */}
\quad D(i, j) = D(i-1, j) + \sigma [\begin{bmatrix} a \\ b \end{bmatrix}]
\quad \text{First}(i, j) = \max \{\text{First}(i-1, j), \text{First}(i-1, j)\}

\text{Lemma 1. Fix } n \in [1, k] \text{ and let } u \text{ be a C-vertex such that } G_{s+n}
s\text{ is formed from } G_{s+n} \text{ by removing the substitution edges of an optimal path from First}(u) \text{ to } u. \text{ If } v \text{ is a C-vertex with First}(v) < \text{First}(u), \text{ then } G_{s+n}(v) = C(v) \text{ and First}(v) = \text{First}(v).
\text{Proof. Let } P \text{ be the path from First}(u) \text{ to } u \text{ whose substitution edges are removed, and let } Q \text{ be an optimal path from First}(v) \text{ to } v. \text{ We first show that } P \text{ and } Q \text{ have no vertex in common. Suppose otherwise and let } s \text{ be a vertex lying on } P \text{ and } Q. \text{ The prefix of } P \text{ ending at } s \text{ and the prefix of } Q \text{ ending at } s \text{ must have the same score (sum of edge weights). To see this, suppose that the prefix of } P \text{ ending at } s \text{ had lower score. Form a path } P' \text{ from First}(v) \text{ to } u \text{ by appending the suffix of } P \text{ beginning at } s \text{ to the prefix of } Q \text{ ending at } s. \text{ Then score } P' > \text{score } P, \text{ contradicting the optimality of } P. \text{ Without loss of generality, suppose } \text{First}(u) < v, \text{ First}(v). \text{ Then the path } P' \text{ constructed as above satisfies score } P' > \text{score } P \text{ and } P' \text{ starts at a vertex that follows } \text{First}(u) \text{ in the ordering } <. \text{ This contradicts the definition of First}(u) \text{ and proves that } P \text{ and } Q \text{ are disjoint. Since } P \text{ and } Q \text{ have no vertex in common, they have no edge in common. Thus } Q \text{ is also a path in } G_{s+n}, \text{ so } C(v) \leq C(v) \text{. But since } C(w) \leq C(v) \text{ for all vertices } w, \text{ we have } C(v) = C(v) \text{. There cannot exist an optimal path in } G_{s+n}, \text{ a } < - \text{successor of First}(v) \text{ to } v, \text{ since such a path would also be a path in } G_{s+n}, \text{ so } \text{First}(v) = \text{First}(v) \text{.}
\text{As described in the previous section, let } G_{s} \text{ be the original graph } G_{s} \text{ and for each } n \in [1, k - 1], \text{ let } G_{s+n} \text{ be determined by locating a C-vertex } u \text{ that maximizes } C(u) \text{ and removing from } G_{s}, \text{ the substitution edges of an optimal path from First}(u) \text{ to } u. \text{ Define a relation } E_{n} \text{ over the C-vertices of } G \text{ by } u E_{n} v \text{ if and only if } \text{First}(u) = \text{First}(v). \text{ } E_{n} \text{ is an equivalence relation and hence } E_{n} \text{ partitions the vertices of } G \text{ into equivalence classes. For each equivalence class } S \text{ of } E_{n} \text{ define score}_{S} = \max(C(u)): u \in S \text{. Our local alignment algorithm begins by determining the } k \text{ best (i.e., highest-scoring) equivalence classes for } E_{n}. \text{ Each of these } k \text{ equiva-
ence classes \( S \) is represented by a 7-tuple:

\[
(C, F, u, T, B, L, R),
\]

where

\[
C = \text{score}(S),
\]

\[
F = \text{First}(c) \text{ for all } c \in S,
\]

\[
\text{First}(u) = \emptyset \text{ and } C(u) = \text{score}(S), \text{ and}
\]

\[
(T, B) \times [L, R] \text{ contains all } i \in S \text{ with } C(i) > W.
\]

In this definition, \( W \) is the minimum score of the retained equivalence classes, and we say that \((T, B) \times [L, R]\) contains \((t, f)\), if \(T \leq t \leq B\) and \(L \leq f \leq R\). Initially, \( C \) values and First vertices are relative to \( G_0 \); as the algorithm proceeds, the tuples are updated so as to be valid for \( G_1, G_2, \ldots, G_{k-1} \). Therefore, we use tuple to designate such a 7-tuple, and refer to the entries of tuple \( S \) by \( C.S, F.S, \ldots, S.R \).

Once \( k \) best covers of \( G_0 \) are determined, we select and delete a best class \( S \), then remove the diagonal edges of an optimal path from \( S.F \) to \( S.u \) to get \( G_2 \). Lemma 1 implies that if vertex \( i \) satisfies First(\( i \)) \( \neq \emptyset \), then \( C_i(\emptyset) = C_i(\emptyset) \) and First(\( i \)) = First(\( i \)). Thus, to find the \( k \)-1 best classes \( G_k \), we need only re-evaluate \( C \) and First for vertices in \( S \) with score \( > W \), i.e., vertices contained in \((S.T, S.B) \times [S.L, S.R]\). This is because vertices in other equivalence classes are already assigned to the proper First vertex, and vertices with score \( \leq W \) are no longer interesting, since we already have enough non-intersecting paths with score at least \( W \). However, each vertex in \((S.T, S.B) \times [S.L, S.R]\) needs to be re-scored in \( G_k \), because we may find a new path to \( i \) of score \( > W \).

The \( k \)-1 best tuples for \( c_i \) are kept in a data structure, called LIST, that supports the following operations:

- **find**: Return the tuple whose First vertex is \( i \), or \( \emptyset \) if no such tuple exists.
- **insert**: Add tuple \( S \) to LIST.
- **mattuple**: Remove and return a maximum-score tuple in LIST.
- **minscore**: Return the minimum score of a tuple in LIST.
- **replace**: Replace a minimum-score tuple in LIST by \( S \).
- **size**: Return the number of tuples in LIST.

Candidates for implementing LIST include a binary list, a balanced search tree (Aho et al. [1]) or a splay tree (Sleator and Tarjan [12]). The data structure used in our alignment software is described near the end of this section.

To maintain LIST, we use the function **enter** of Fig. 3. Let \( \pi(u) \) and \( f(a) \) denote the c-component and j-component of vertex \( a \). Enter \((c, f, u, w, i)\) performs the following tasks. If \( t \) is already a tuple \( S \) in LIST whose
a maximum-score tuple in LIST. 

score tuple in LIST by S. 

clade a linear list, a balanced search (Sleator and Tarjan [12]). The data 

ax is described near the end of this 

ion enter of Fig. 3. Let $a(u)$ and $f(u)$ 

mment of vertex $u$. Enter($c, f, u, w, l$) 

is already a tuple $S$ in LIST whose 

First vertex is $f$, then the other attributes of $S$ are adjusted if necessary. 
Otherwise, the tuple $(c, f, u, (0), (u), (a), (a), (a), (a), (a))$ either replaces a mini-

mum-score tuple in LIST or is inserted into LIST, depending on whether 
or not LIST is full (i.e., contains 1 tuple). Finally, the minimum tuple 
score is returned if LIST is full. 

Notice that enter guarantees that any two tuples in LIST have distinct 
First values. That is, if $E$ and $S$ are in LIST and $S = E$, then $S^F = E^F$. 
The rectangle $[T, B] \times [L, R]$ is properly maintained so long as enter is 
called whenever $C(u) > W$. 

Figure 4 outlines our algorithm for computing $k$ best non-intersecting 
paths. The first part of the algorithm (two nested for loops) computes 
tuples representing $k$ highest-scoring classes in $G_1$, while the second part 
reports $k$ best non-intersecting paths one at a time. The nth iteration of 
the for loop in the second part selects a highest-scoring tuple, reports its 
optimal path, removes the diagonal edges of the path to form $G_{n+1}$, and 
computes the $k - n$ best tuples in $G_{n+1}$. The algorithm mentions only 
computation of $C(i, j)$ and abbreviates $First(i, j)$ by $First(i, j)$, computa-
tion of $D(i, j)$, $First(i, j)$, $R(i, j)$, and $First(i, j)$ is implicit. In another 
effort to simplify the presentation, we assume in what follows that there 
are at least $k$ distinct values of First in $G_1$ with paths of positive score.
\[ W = 0 \]

for \( i = 0 \) to \( M \) do

for \( j = 0 \) to \( N \) do

[Compute \( C(i, j) \) and \( \text{First}(i, j) \)]

if \( \text{First}(i, j) = \text{Null} \) then

\[ W \leftarrow \text{entire}(C(i, j), \text{First}(i, j), i, j, W, k) \]

end if

end for

end for

[Set \( S = \) matching ]

algorithm(S)

if \( n = 4 \) then

Determine \( T \subseteq T \) and \( L \subseteq S \) so that no \( C \)-path starting outside \( [T, S, B] \times [L, S, R] \) and ending outside \( [T, S, B] \times [L, L, R] \) has score greater than \( W \).

for \( i = 7 \) to \( 5 \) do

for \( j = 5 \) to \( 3 \) do

[Compute \( C(i, j) \) and \( \text{First}(i, j) \) relative to \( [T, S, B] \times [L, S, R] \)]

if \( \text{First}(i, j) = \text{Null} \) then

\[ W = \text{entire}(C(i, j), \text{First}(i, j), i, j, W, k) = -\text{Null} \]

end if

end for

end for

end if

\[ F = 4 \]  Outlined of the linear-space algorithm

The procedure outlined(S) employs the global alignment of Myers and Miller to find an optimal path from \( S.F \) to \( S.H \) and then removes the diagonal edges of the path from \( G_s \). In practice, these "removed" edges are recorded in a data structure so that they will not be used again in determining \( T \) and \( L \), in the final steps for loops, or in subsequent calls to alignment. Specifically, when evaluating

\[ C(i, j) = \max \left\{ 0, D(i, j), H(i, j), C(i \cdot 1, j - 1) + \sigma \left( \begin{array}{c} c(i) \\ b(j) \end{array} \right) \right\} \]

after entering the main loop of Fig. 4, we need to check whether the edge from \((i \cdot 1, j - 1) \) to \((i, j)\) \( _c \) has been removed; if it has, then we maximize over just the first three terms. We now prove correctness of the local alignment algorithm, postponing a description of our method for computing \( T \) and \( L \) until the end of this section.

**Lemma 2.** At the start of the \( n \)th iteration of the main for loop of Fig. 4, \( \text{LIST} \) contains \( k = n + 1 \) tuples that satisfy the following properties:

- \((\min \{S.C\cdot S \in \text{LIST}\} = \text{W}_n\)

where \( \text{W}_n \) is the value of \( W \) at the start of the \( n \)th iteration,
for each $C$-vertex $u$ that satisfies $C(u) > W$, there is a $S$ in $\text{LST}$ such that $\text{First}_S(u) = S.F$, $C_S(u) \leq S.C.$, and $u \in \{S.T, S.B\} \times \{S.L, S.R\}$.

(3) for each tuple $S$ in $\text{LST}$, $\text{First}_S(u) = S.F$ and $C_S(S.u) = S.C.$ and

(4) a tuple delivers a tuple $S$ such that an optimal path from $S.F$ to $S.u$ is an optimal path in $G_S$.

**Proof.** (Induction on $n$.) For $n = 1$, the claim follows readily from the definition of $\text{enter}$ in Fig. 3. (Recall our assumption that there are at least $k$ distinct values of $\text{First}_1$ in $G_1$ with paths of positive score.) Assuming that the claim is true for $n$, we will show that it is true for $n + 1$. Consider the $n$th iteration of the loop. The assignment $S \leftarrow \text{maxtuple } (S' - \text{enter})$ reduces the size of $\text{LIST}$ from $k - n + 1$ to $k - n$. With every call to $\text{enter}$ during the current iteration, the list argument in the call is $k - n$, so the condition “size $j + i$ is true in enter. It follows that $\text{enter}$ returns the correct number of tuples, and likewise follows that $\text{enter}$ returns the minimum tuple score, which is assigned to $W$. This verifies condition 1.

Consider condition 2. and let $u$ satisfy $C_\nu(u) > W_\nu$. We have already noted that $\text{enter}$ is not called when execution reaches the main for loop; instead minimum-score tuples are replaced by ones of higher score. It follows that successive values of $W$ are non-decreasing throughout the $k$ iterations of the main loop. Thus, $C_\nu(u) \geq C_\nu(f_\nu) > W_\nu$. The induction hypothesis guarantees that at the start of the $n$th iteration there is a $S'$ in $\text{LIST}$ such that $\text{First}_S(u) = S.F$, $C(u) \leq S.C.$, and $u \in \{S.T, S.B\} \times \{S.L, S.R\}$. Let $S$ be the tuple selected and removed from $\text{LIST}$ by $\text{maxtuple}$.

First suppose that $S' = S$. Then $S.C. \geq C(u) > W_\nu$, at the start of the $n$th iteration, so $S'$ is not replaced during the $n$th iteration. Enter is written so that $S.C.$ can only increase in value and the rectangle associated with $S'$ can only increase in extent. Also, $\text{First}_S(u) = \text{First}_S(u)$ and $C(u) = C(u)$ by Lemma 1, so $u$ satisfies condition 2 with $n$ replaced by $n + 1$.

Next suppose that $S' = S$. Since $C_\nu(u) > W_\nu$, the definition of $T$ and $L$ in Fig. 4 guarantees that all paths in $G_\nu$ to cost greater than $W_\nu$ are entirely contained in $\{T, S.B\} \times \{L, S.R\}$. The tested for loop at the end of Fig. 4 discards all those paths and updates $\text{LIST}$ to contain the desired tuple. This completes the verification of condition 2.

Now consider condition 3. Let $S$ be in $\text{LIST}$ at the start of the $(n + 1)$th iteration. If $S$ is placed in $\text{LIST}$ during the $n$th iteration, or if $S.C.$ and $S.u$ are assigned new values during that iteration, then $c, f$, and $u$ for that call
to enter satisfy \( c = C_{\ast}(a) \) and \( f = \text{First}_{\ast}(a) \), verifying condition 3. Otherwise, Lemma 1 and the induction hypothesis imply that \( \text{First}_{\ast}(a) = \text{First}(S.a) = S.F \) and \( C_{\ast}(a) = C(a) = S.C \).

It remains to verify condition 4. Let \( S \) be the tuple selected by \text{maxtuple} in the \( n \)th iteration of the main \text{for} loop of Fig. 4. It follows from assertions 2 and 3 of Lemma 2 that \( C(S.a) = S.C = \max(C(v) : v \in \text{vertex of } G.I) \) and \( \text{First}(S.a) = S.F \). Thus any optimal path from \( S.a \) to \( S.u \) is a highest-scoring path in \( G_{\ast} \).

**Theorem 1.** The algorithm of Fig. 4 computes \( k \) best non-intersecting local alignments.

**Proof.** Correctness follows immediately from condition 4 of Lemma 2.

To compute values \( T \) and \( L \) needed in the main \text{for} loop of Fig. 4, we use the procedures \text{locate} and \text{disjoint} of Fig. 5. Namely, the algorithm of Fig. 4 performs the call \text{locate}(S.T., S.B., S.L., S.R.) to find \( T \) and \( L \), such that no \( C \)-path starting outside \([T, S.B.] \times [L, S.R.] \) and ending inside \([S.T., S.B.] \times [S.L., S.R.] \) has score greater than \( W \). For the most part, the computation is merely a "reverse" local similarity method that proceeds from lower right to upper left. (The differences are that no analog of \text{LIST} is maintained and that any tie-breaking rule can be used.) Values \( C', D', I' \) and \( \text{vertices Last} \) is the reverse computation corresponding to the \( C, D, I \) and \( \text{First} \) in the forward computation. For the remainder of the section, \( x \) represents any of \( C', D', \text{or} \Gamma \). The notation \( \text{Last}(x) = (T', L') \) means that both \( \text{Last}(r) \) \( \geq \) \( T' \) and \( \text{Last}(r) \leq L' \). The \text{while} loop in \text{locate} stops when either no optimal path beginning on row \( T \) and column \( L \) ends inside \([T', S.B.] \times [L, r] \) or \( T' = L = 0 \). The \text{repeat} loop in \text{locate} terminates when either \( \text{disjoint}(x, b, r) \) returns \text{true} or \( T = L = 0 \). Lastly, \( \text{disjoint}(x, b, r) \) \( \text{var} (x', r') \) returns \text{false} if there is no \( S \) in \text{LIST} such that \([S.T., S.B.] \times [S.L., S.R.] \) shares any vertex with either \([b, r] \times [b, r - 1] \) or \([b, r - 1] \times [b, r] \); it returns \text{true} with \( x' \) and \( r' \) adjusted if there is such an \( S \).

**Lemma 3.** \text{Located}(x, b, l, r) determines \( T \) and \( L \), so that any path starting outside \([T, b] \times [L, r] \) and ending inside \([l, b] \times [l, r] \) has score at most \( W \), where \( W \) is the minimum tuple score in \text{LIST}.

**Proof.** To smooth the way for a more formal proof, we first develop an intuitive understanding based on Fig. 6. If \text{locate} terminates with \( T = L = 0 \), then the result is trivial, so suppose otherwise. Indeed, suppose that the final values of \( T \) and \( L \) are both positive: we leave the case that exactly one of \( T \) or \( L \) is 0 to the reader. At termination, \( \text{Last}(x) = \text{false} \), so for each vertex \( v \) on the top and right boundaries of \([T, b] \times [L, r] \) the


**Function locate(i, b, j, r)**

Perform an exact local alignment computation on \( [i, b] \times [j, r] \) and save \( C, D, T, F \) and

Last for vertices of the forms \((i, j)_k\) for \(j \leq i \leq r\) or \((i, j)_k\) for \(i \leq j \leq r\).

1. \( T = T + 1 \)
2. \( L = L + 1 \)
3. \( \text{If flag or } T > 0 \text{ or } \text{flag and } T > 0 \text{ do} \)
4. \( \text{If flag and } T > 0 \text{ then} \)
5. \( \text{If flag and } L > 0 \text{ then} \)
6. \( \text{If flag and } L > 0 \text{ then} \)
7. \( \text{Determine } C, D, T, F, \text{and Last at } (T, L) \) for \( i > T \) down to \( L \)
8. \( \text{If Last}(T, L) \geq (T, L) \) for some \( j \leq L \) and \( X \) then
9. \( \text{If flag and } L > 0 \text{ then} \)
10. \( \text{If flag and } L > 0 \text{ then} \)
11. \( \text{If } L = 0 \text{ then} \)
12. \( \text{If Last}(T, L) \geq (T, L) \) for some \( j \) then
13. \( \text{Evaluate } T, (T, L) \text{ for some } X \text{ then} \)

14. \( \text{While } \text{ encounters } \text{ terminates} \)
15. \( \text{End loop for } \text{lo} \text{c} \text{ate} \text{ terms} \text{ with } \text{out}_{\text{es}} \text{ at most } W, \text{LIST} \)
16. \( \text{end} \text{ formal } \text{proof} \text{ we } \text{first } \text{develop} \text{ an} \)
17. \( \text{If locate terminates with } T = L \text{ otherwise. Indeed, } \text{suppose that the} \)
18. \( \text{time; we } \text{leave the case that exactly} \text{min} \text{ation, flag is } \text{false} \text{, so} \)
19. \( \text{boundaries of } [T, B] \times [L, R] \) the

**Fig. 5. The locate and disjoint functions.**
condition, "Last(s) ≥ (T', L')" is false. Pictorially, Last(s) is in the shaded region in Fig. 6. Also disjoint(T, b, L, r, T', L') is true, so no S in LIST has an associated rectangle that intersects the shaded region in Fig. 6. Part 2 of Lemma 2 then guarantees that all vertices v in the shaded region satisfy C_s(v) ≤ W.

More formally, let P be a C-path that starts outside [T, b] × [L, r] at vertex v and ends in [r', b] × [l', r] at vertex v'. Let r be the first vertex on P that has the form either (T, j_1) or (v, l_1) for j_1 ∈ L for b ∈ V, b ∈ V. Also, let r' = Last(s). Let P_1 be the prefix of P ending at v and let P_2 be the suffix of P starting at v'. Then r' must be in [l, b] × [l, r'] or [r, r' - 1] × [l, r] so r' is not in the rectangle (S, S, B) × (S, S, R) for any S in LIST. Then C_s(r') ≤ W by part 2 of Lemma 2. Let Q be the path from v to r' and form P' by appending Q to P_1. Since P' ends at r', score(P') = C_s(r'), so score(P) = score(P_1) + score(P_2) ≤ score(P') + score(Q) = score(P') ≤ C_s(r') ≤ W. □

Notice that if no path in G_s, 1 that ends in [l, b] × [l, r') has score greater than W, then no value of C_s will exceed W in the reverse pass. In that case, the forward pass at the end of Fig. 4 can be omitted.

Our preferred implementation of LIST uses an array of k records, one per tuple. Special care is taken to accelerate the frequent operations: find, minscore, and replace. The number of tuples in LIST is kept in a variable so that size takes only constant time. Since vertices in the same equivalence class form a connected region, find(f) tends to be invoked
with the same \( f \) for a period of time. Maintaining a pointer to the most recently accessed record allows \( \text{find}(f) \) to usually be served in constant time. To speed up \text{minscore}, a pointer to a minimum score record is kept, which also makes \text{repsize} efficient. Whenever this minimum-score record is replaced or modified, the pointer needs be recomputed by searching the entire array. Experimental results quoted in Section 5 show that this simple implementation strategy is entirely adequate.

The algorithm is designed to require only linear space. In the complete forward sweep of \( G_1 \), it suffices to save only the most recently computed row of each matrix. (For implementation details, see Huang et al. [6].) In the reverse pass \( r \) determines a rectangle, the most recently computed column of each matrix is saved as well. Recomputation of rectangles is handled like the complete forward sweep. Thus, \( O(M + N) \) space is sufficient for values \( C_{x,y}(i,j) \), \( F_{x,y}(i,j) \), etc. The \text{alignment} procedure of Myers and Miller [7] uses space \( O(M + N + L) \), where \( L_x \) is the length of the computed alignment. \( O(K) \) space is needed to save the aligned pairs of computed alignments, where \( K = \sum_{x=1}^{n} L_x \). Lastly, the data structure for \text{LSS} takes \( O(k) \) space, where \( k = K \). Thus, the algorithm requires \( O(M + N + K) \) space.

A completely rigorous and realistic analysis of the algorithm’s time complexity would be difficult, but the general trend is easy to grasp. The complete forward sweep clearly takes \( O(MN) \) time in the worst case, exclusive of calls to manipulate \text{LSS}, and as just noted, manipulation of \text{LIST} takes negligible time in practice. The remainder of the algorithm may take \( O(MNK) \) time in the worst case that all computed alignments are nearly as long as the original sequences. However, in practice, at most a few alignments are large, and the performance of the algorithm is much better than the worst case. Execution time is strongly affected by the choice of scoring function \( \alpha \) and gap penalty \( \delta \). When mismatches and gaps are penalized very lightly, the rectangles requiring recomputation are large and the algorithms’ efficiency suffers accordingly. When reasonable weights are used, sides of the recomputation rectangle are only a few times longer than the length \( L_x \) of the local alignment, so the recomputations take \( O(C_{x,y} + L_x^2) \) time in expectation.

One final implementation note deserves mention. Some users may prefer a cutoff score to \( k \). That is, instead of computing \( k \) best alignments, all non-intersecting alignments with scores greater than the cutoff score are computed. Our algorithm can easily be modified to do this. In Fig. 4, replace \( W \leftarrow 0 \) by \( W \leftarrow \text{cutoff} \), and replace

```plaintext
for \( n \leftarrow 1 \) to \( k \) do
```

by:

```plaintext
while \text{LIST} is not empty do
```
In addition, the function enter $b$ modified accordingly no bound is placed on the size of LFT, LIE is always called, and no minimum core is returned. The difficulty in working with cutoffs is that a setting just slightly below the optimal cutoff value may bring forth a deluge of insignificant local alignments.

4. An Example

Lemma 1 is central to our approach. It implies that we need only save endpoints of $k$ paths from a single complete sweep that computes all entries of $B$, $I$, and $C$.Lemma 1 guarantees that we will never "break" any of the retrieved paths by removing diagonal edges of a higher-scoring path. This section contains an example showing that a less meticulous definition of First may well not work properly for this purpose.

For simplicity of this example, suppose that the gap open penalty $g$ is 6. Then for each $i$ and $j$ the vertices $i, j$, for $X = D, I, and C$ can be thought of a coalescing into a single node. (Details can be found in Myers and Miller[5]). Consider the tie-breaking rule that is applied when the diagonal edge joining a node gives a path-weight identical to a horizontal or vertical edge. In particular, suppose that such a tie is broken in favor of the diagonal edge, i.e., First($i, j$) is set equal to First($i - 1, j - 1$). Assume the following matrix, $a$, of substitution costs.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

Values of $a$ not shown (including deletions and insertions) equal 1.

Let $A = \text{dced}$ and $B = \text{ared}$. A best local alignment of $A$ and $B$ is $[\text{a}][\text{r}][\text{e}][\text{d}][\text{c}]$, which scores 6. The second-best local alignment that scores no aligned pairs with the first alignment is $[\text{a}][\text{r}][\text{e}][\text{d}]$, which scores 5.

Figure 7 depicts $G_{2,4}$. Edges with positive weights are shown; edges weighted -1 are either not shown or dotted. First($a, e$) and First($d$) are shown for the rule that prefers diagonal edges. There is an alternative optimal path from First($a$) to $a$ that breaks the only optimal path from First($d$) to $e$. The corresponding alignment is $[\text{a}][\text{r}][\text{e}][\text{d}][\text{c}][\text{a}][\text{r}][\text{e}]$, which scores 6 but contains the pair $[\text{a}][\text{r}]$. This would affect our algorithm because
When the Myers-Miller procedure is invoked to find an optimal path between two given vertices, there is no effective way of controlling which optimal path is found.

If First is defined as in Section 3 (using <, to break ties, then First(v) = (1,0). Then the only optimal path from First(1) to u spells the alignment \[\text{First}(v) = \text{First}(u) = (1,0).\] The second best non-intersecting alignment is spelled by a path extending \(\frac{1}{2}\) of the way down the main diagonal of \(G_u\), i.e., \[\text{First}(v) = (0,1),\] which scores 4.

Figure 7 also provides an example of the non-unique-ness phenomenon mentioned at the end of Section 2, i.e., where two sequences of \(k\) best non-intersecting paths have different scores. For \(k = 2\), the paths

\[(0,0) \rightarrow (1,1) \rightarrow (2,2) \rightarrow (3,3) \rightarrow (4,4)\]

\[(1,0) \rightarrow (2,1) \rightarrow (3,2)\]

constitute two best non-intersecting paths, the second of which scores 5.

The paths

\[(1,0) \rightarrow (2,1) \rightarrow (3,2) \rightarrow (3,3) \rightarrow (4,4)\]

\[(0,0) \rightarrow (1,1) \rightarrow (2,2) \rightarrow (3,3)\]

also constitute two best non-intersecting paths, the second of which scores 4.
5. Discussion

This section compares the execution time of the new algorithm and that of the algorithm of Huang et al. [6]. Both algorithms were implemented in the C programming language. DNA sequences containing the human and rabbit β-like globin gene clusters were used as test sequences. These sequences share many similar regions, which tends to favor the old algorithm. Use of random sequences would perhaps unfairly favor the new algorithm because such sequences usually contain only short regions of similarity, hence the new algorithm would spend very little time recomputing parts of the matrix.

In all tests, a matching aligned pair scored 1.0 and a mismatched pair scored −1.5. The running time of the old algorithm is not significantly affected by the choice of weights. Thus, we conducted only one run using one set of weights for the old algorithm. On a 73,360-nucleotide sequence containing the human β-globin gene cluster and a corresponding 44,594-nucleotide rabbit sequence, the old algorithm took about 23 days to compute 100 best non-intersecting alignments with a gap-open penalty \( g = 6.0 \) and a gap-extension penalty \( c = 0.2 \). (That is, each deletion or insertion pair is scored −0.2.) On the 11,950-nucleotide human αβ region and the 12,400-nucleotide rabbit \( \alpha - \beta \) region, the old algorithm took 5½ h to compute 20 best non-intersecting alignments with \( g = 6.0 \) and \( c = 0.2 \).

Times for the new program are shown in Tables I and II. Table I reports the time to compute 100 best non-intersecting alignments between the human and rabbit sequences for various gap penalties. Table II reports the time to compute 20 best non-intersecting alignments between the shorter \( \alpha - \beta \) regions. The total column shows the total time (in hours) taken by our program. The time distribution over the five major parts of the program is also given. Where main is the complete forward pass, locate is the backward pass determining a rectangle, update is recomputation of the rectangle, list is maintained at LIST, and align is the Myers–Miller global alignment procedure. Notice that using small gap penalties can

<table>
<thead>
<tr>
<th>g</th>
<th>c</th>
<th>Total</th>
<th>Main</th>
<th>Locate</th>
<th>Update</th>
<th>List</th>
<th>Align</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.0</td>
<td>0.5</td>
<td>15.5</td>
<td>71.3%</td>
<td>12.9%</td>
<td>11.9%</td>
<td>1.4%</td>
<td>2.3%</td>
</tr>
<tr>
<td>3.0</td>
<td>1.0</td>
<td>12.5</td>
<td>91.1%</td>
<td>4.2%</td>
<td>3.1%</td>
<td>0.4%</td>
<td>1.5%</td>
</tr>
<tr>
<td>8.0</td>
<td>0.2</td>
<td>17.3</td>
<td>66.7%</td>
<td>14.3%</td>
<td>12.4%</td>
<td>2.2%</td>
<td>3.4%</td>
</tr>
<tr>
<td>8.0</td>
<td>0.2</td>
<td>13.3</td>
<td>82.7%</td>
<td>7.5%</td>
<td>0.8%</td>
<td>1.6%</td>
<td>1.2%</td>
</tr>
</tbody>
</table>
substantially increase the sizes of recomputed regions, which has a noteworthy impact on the total execution time.

REFERENCES