ELSEVIER

Theoretical Computer Science 145 (1995) 357–369

Theoretical Computer Science

Note

A space efficient algorithm for finding the best nonoverlapping alignment score☆

Gary Benson*

Department of Mathematics, University of Southern California, 1042 W. 36th Place, DRB 155, Los Angeles, CA 90089-1113, USA

Received March 1994; revised November 1994
Communicated by M. Crochemore

Abstract

Repeating patterns make up a significant fraction of DNA and protein molecules. These repeating regions are important to biological function because they may act as catalytic, regulatory or evolutionary sites and because they have been implicated in human disease. Additionally, these regions often serve as useful laboratory tools for such tasks as localizing genes on a chromosome and DNA fingerprinting. In this paper, we present a *space efficient* algorithm for finding the maximum alignment score for any two substrings of a single string T under the condition that the substrings do not overlap. In a biological context, this corresponds to the largest repeating region in the molecule. The algorithm runs in $O(n^2 \log^2 n)$ time and uses only $O(n^2)$ space.

1. Introduction

DNA and proteins are long linear molecules made up of several kinds of individual units. In DNA, there are four kinds of units (*bases* or *nucleotides*); in proteins there are 20 kinds of units (*amino acids*). Because of their linear structure, these molecules can be thought of as strings over a finite alphabet.

Repeating patterns make up a significant fraction of DNA and protein molecules. The exact function of many of these repeating regions is unknown. In some cases (e.g. the protein collagen), the repetition produces a structural attribute. But, in many

☆ A preliminary version of this paper appeared in the *Proceedings of CPM '94*, Springer, Berlin, Lecture Notes in Computer Science, Vol. 807, 1994. Submission of this paper in final form to *Theoretical Computer Science* was invited in view of the strong endorsement contained in the conference reviews.

* Supported by NSF grants DMS-87-20208, DMS-90-05833 and NIH grant GM-36230. Email: gbenson@hto.usc.edu.

SSDI 0304-3975(94)00003-8

others, the repetition may function as a catalytic, regulatory or evolutionary site. For example, the centromeric region of DNA controls the movement of the chromosome during cell division. This region, termed a *satellite*, consists of many contiguous copies of a species specific pattern and may serve as a protein binding site.

In still other cases, repeating regions have been implicated in human disease. A region consisting of a three-nucleotide repeat on the human X chromosome is sometimes replicated incorrectly, causing the number of repeats to balloon from 50 to hundreds or thousands. Individuals with this defect suffer from fragile-X mental retardation. Several other diseases are also now known to have their basis in huge expansions of different trinucleotide repeats.

Besides their importance in understanding protein and DNA function, repeating regions are useful laboratory tools. For example, the number of copies of a pattern at a particular site on a chromosome is often variable among individuals (*polymorphic*). Such polymorphic regions are helpful in localizing genes to specific regions of the chromosome and also in determining the probability of a match between two samples of genetic material (DNA fingerprinting).

Given their importance and given the exponential growth in the size of the DNA and protein databases, efficient methods for detecting repeating regions are required. Additionally, because the really efficient methods are heuristic, a method that has both greater sensitivity and exactitude is required to corroborate and extend the results.

Due to the action of evolutionary mutation, repeating regions rarely consist of exact repeats. Rather they are approximate repeats contaminated with substitutions, deletions and insertions. It is thus natural to consider approximate string matching techniques when designing algorithms for detecting repeats.

Let $T = t_1 \ldots t_n$ and $W = w_1 \ldots w_m$ be two strings over an alphabet Σ. Let $T[i,j] = t_i \ldots t_j$ and $W[g,h] = w_g \ldots w_h$ be two substrings. An *alignment* of $T[i,j]$ and $W[g,h]$ is a sequence Q of *edit operations* [5] that transforms substring $T[i,j]$ into substring $W[g,h]$. The allowed operations are: insert a symbol into $T[i,j]$, delete a symbol from $T[i,j]$ and replace a symbol in $T[i,j]$ with a (possibly identical) symbol in $W[g,h]$. If a weighting function δ is defined for each possible edit operation [9], then we can compute a *score* for an alignment by adding the weights assigned to each operation in Q.

In the *global alignment problem*, we seek the optimal cost alignment for T and W. In the *local alignment problem*, initial deletions and terminal insertions have zero cost. This has the effect of permitting a global alignment for any two substrings $T[i,j]$ and $W[g,h]$. Either problem can be solved in $O(nm)$ time by dynamic programming. Typically, in the biological domain, δ is negative for all operations except replacement of similar symbols and the object is to maximize the alignment score.

In this paper, we consider the problem of finding the maximum alignment score for any two substrings of a single string T under the condition that the substrings do not overlap, that is, the maximum alignment score between two substrings $T[g,h]$ and $T[i,j]$ such that $g \leqslant h < i \leqslant j$. In a biological context, this corresponds to the largest repeating region in the molecule.

In [6], Miller observed that for a general weighting function δ, the problem can be solved in $O(n^3)$ time and $O(n^2)$ space by a modification of the Smith–Waterman algorithm [8]. That time was improved by Kannan and Myers [2] to $O(n^2 \log^2 n)$ in a rather complicated recursive algorithm. Unfortunately, their algorithm requires $O(n^2 \log n)$ space. They considered reducing the space to $O(n^2)$ to be an important open problem.

In a similar vein, Landau and Schmidt [4] gave an algorithm for identifying approximate *tandem* or contiguous repeats. Their algorithm uses a very restricted weighting function for the edit operations. Either insertions and deletions have infinite negative weight (Hamming distance) or each edit operation has a weight of one (edit distance). The time for their algorithm is $O(kn \log (n/k))$ for a Hamming distance of at most k and $O(kn \log k \log n)$ for an edit distance of at most k. Note that this matches the time of the Kannan–Myers algorithm when the edit distance is at most n.

The *main contribution of this paper* is a new, *space efficient algorithm* for finding the maximum alignment score for two nonoverlapping substrings of a sequence T. Our algorithm is *simpler* than the algorithm of [2], uses the same time and *uses only* $O(n^2)$ *space*.

The remainder of the paper is organized as follows, In Section 2, we formally define our problem. In Section 3 we briefly discuss *edit graphs* and two algorithms by other authors that we use as subroutines. In Section 4 we present an overview of our algorithm. In Section 5 we introduce the idea of ranks and show how they can be built and used efficiently and in Section 6 we present a new algorithm satisfying the time and space bounds we claim.

2. Problem description

Let $T = t_1 \ldots t_n$ be a sequence and δ a weighting function. Let $S([g,h],[i,j])$ be the best alignment score for substrings, $T[g,h]$ and $T[i,j]$. We seek to find

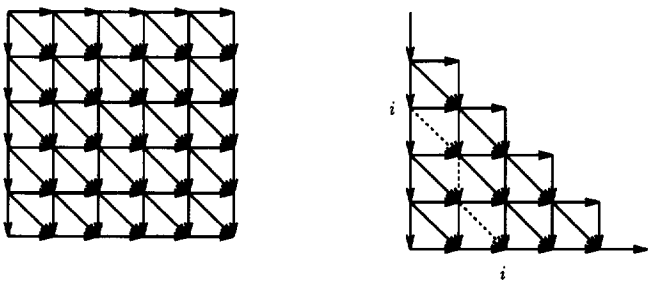
$$H = \max_{1 \leqslant g \leqslant h < i \leqslant j \leqslant n} \{S([g,h],[i,j])\},$$

that is, the maximum alignment score for two *nonoverlapping* substrings of T. Miller [6] calls such nonoverlapping regions *twins* and we adopt this nomenclature.

Without loss of generality, we will assume that n is a power of 2. This can be accomplished by padding the sequence if necessary. Although in this paper we only discuss finding the best score, we can additionally find the substrings, and once the substrings are determined the alignment can be computed in time and space $O(n^2)$.

3. Preliminaries

The best local alignment score for a string T versus itself can be computed by the Smith–Waterman [8] dynamic programming algorithm. If we exclude the trivial



a b

Fig. 1. (a) Edit graph for the scoring matrix; (b) the lower triangular part of the edit graph and a twin (dotted) bounded by i.

alignment of T with itself, the resulting alignment consists of two (possibly overlapping) substrings of T. Let $S(i,j) = S([*,i],[*,j])$ be the best scoring alignment between any substring ending at T_i and any substring ending at T_j. The recurrence is simple:

$$S(i,j) = \max \begin{cases} S(i,j-1) + \delta(\text{insert } T_j), \\ S(i-1,j) + \delta(\text{delete } T_i), \\ S(i-1,j-1) + \delta(\text{replace } T_i \text{ with } T_j), \\ 0. \end{cases}$$

The final option, which restricts the scores to nonnegative values, permits *local alignment*, that is, the starting indices of the substrings are not fixed.

Note that computing a single entry in the scoring matrix requires knowing the value of only three other entries. Because of this, the scoring matrix can be viewed as a *weighted edit graph* [3, 7] where the entries are the nodes and the weights δ are assigned to the edges (Fig. 1). An alignment consists of a path through the edit graph and its score is the sum of the edge weights along the path.

Throughout the remainder of this paper, we will think of our problem in terms of finding high scoring paths in the edit graph.

As has been observed [2], a path in the edit graph is a twin iff it lies entirely within a rectangle bounded by row i and column i for some i, $1 < i < n$, and because of this, we need consider only the lower triangular part of the edit graph.

In our algorithm, we will use two other algorithms as subroutines. They are (1) the DIST table construction algorithm from [1] and (2) the Propagate algorithm from [2]. In the remainder of this section, we give a brief overview of each algorithm.

3.1. DIST table construction

Let G be an $n \times m$ edit graph. Let LT be the set of nodes on the left and top edges of G. Similarly, let RB be the set of nodes on the right and bottom edges of G. We seek to

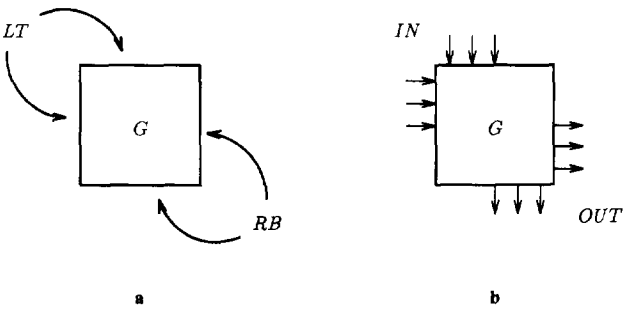


Fig. 2. (a) The DIST table for G contains best scores from every node in LT to every node in RB; (b) algorithm Propagate finds the best score $OUT(y)$ for every node $y \in RB$ given initial values $IN(x)$ on every node $x \in LT$.

construct a table $DIST_G[x, y]$ that gives the best score through G from every node $x \in LT$ to every node $y \in RB$ (Fig. 2). In [1], it was shown that the time to construct such a table is $O((n+m)^2 \log(n+m))$. The construction is done recursively by dividing G into four equal-sized subgraphs, finding the tables for each subgraph and then combining the tables. Combining takes $O((n+m)^2)$ time and relies on the fact [1] that for a given node $x \in LT$ and two nodes $y, y' \in RB$ *ordered counterclockwise*, the "leftmost" best scoring path from x to y cannot cross the leftmost best scoring path from x to y'.

3.2. *Propagate algorithm*

Let G be an $n \times m$ edit graph as above. Suppose we assign a value $IN(x)$ to each node $x \in LT$. We want to determine the following maximum values $OUT(y)$:

$$\forall y \in RB, \quad OUT(y) = \max_{x \in LT} \{IN(x) + DIST_G[x, y]\},$$

that is, the best value that comes out of y given the initial values on x (Fig. 2).

Using the same property of noncrossing best scoring paths, [2] show how to compute $OUT(y)$. Start by imposing a clockwise ordering on the nodes $x \in LT$ and a counterclockwise ordering on the nodes $y \in RB$. The method is to first find the best score for y_{mid}, the middle $y \in RB$. The sums $IN(x) + DIST_G[x, y_{\text{mid}}]$ are computed for all $x \in LT$ and the lowest ordered node x_j which maximizes the sum is chosen. The best scores are then found recursively for all y ordered below y_{mid} using only $x_1, \ldots x_j$ and for all y ordered above y_{mid} using only $x_j, \ldots, x_{n+m}$. The time is $O((n+m)\log(n+m) + (n+m))$ which becomes $O(n \log n)$ when G is a square.

4. Algorithm overview

Our algorithm will work as follows (Fig. 5):

1. Divide the edit graph into rectangles and, within each, use dynamic programming [8] to find the best score to every node. The rectangles are arranged so that alignments are nonoverlapping. This handles some but not all possible nonoverlapping alignments.
2. Adjacent rectangles form an upper and lower *panel* where a nonoverlapping alignment can begin and end. Find the best remaining nonoverlapping scores from nodes in a panel $[k,j]$ of rows to nodes in a panel $[k,j]$ of columns. Note that the union of these panels can be divided into an upper rectangle A, a lower rectangle B and a triangle T.
   (a) For alignments with at least one end in either rectangle A or B, we jump best values through a series of DIST tables.
   (b) For alignments *strictly within* the triangle T, we recurse.

Our algorithm is able to make an improvement in the space requirement because of several new ideas:

- In step 1, we let the rectangles be geometrically decreasing in size.
- In step 2, we interleave the jumping of the values and the construction of larger DIST tables.
- In step 2, we observe that by maintaining nonincreasing lists of scores while jumping, we can limit the number of scores that must be considered by each node.

These ideas are elaborated upon in the discussion below.

5. Ranks of tables

In this section, we show how to efficiently jump a set of scores through a series of DIST tables. Let G be an $n \times m$ edit graph with $n \leqslant m$. Let there be m/n consecutive $n \times n$ DIST tables $D_1, \ldots, D_{m/n}$ covering G (see Fig. 3). We call such a collection of tables a *rank*. The *size* of a rank is the length of a side of a DIST table in the rank. Here the rank has size n.

We begin by showing that a set of IN values can be efficiently jumped across G using the rank of tables. A similar idea appears in algorithm Mesh_propagate in [2].

Lemma 1. *Let G be an $n \times m$ edit graph with $n \leqslant m$. Let a size n rank of DIST tables cover G as above. Let each node $x \in LT_G$ have an associated score $IN(x)$. Then the best score to each element $y \in RB_G$ that originates anywhere in LT_G can be computed in time $O(m \log n)$.*

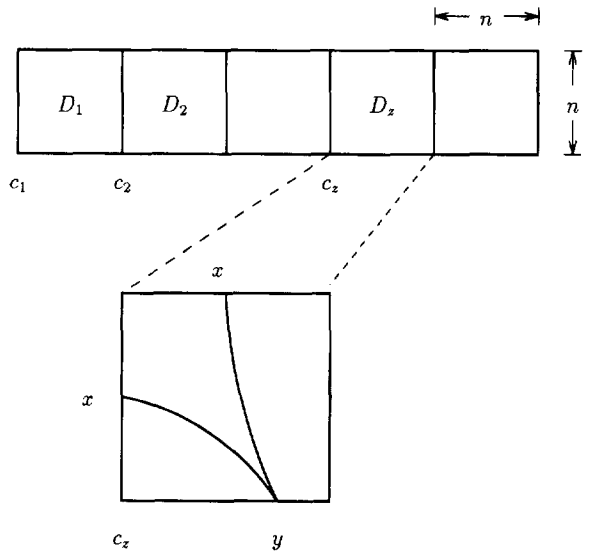


Fig. 3. Computing the values $OUT(y)$ with a rank of DIST tables.

Proof (see Fig. 3). Let c_z be the column on the left side of DIST table D_z and let RB_z be the set of nodes on the right or bottom edges of D_z. We show by induction that the best score from LT_G to an element $y \in RB_z$ is

$$\forall y \in RB_z, \quad OUT(y) = \max_{x \in LT_z} \{VALUE(x) + D_z[x, y]\},$$

where

$$VALUE(x) = \begin{cases} IN(x) & \text{if } x \in LT_G, \\ OUT(x) & \text{if } x \in RB_{z-1}. \end{cases}$$

That is, the best score to a node $y \in RB_z$ consists of the maximum sum of (1) the score in node $x \in LT_z$ and (2) the edge-to-edge score of x to y. Clearly, this is true for D_1. Inductively, suppose it is true for D_k. In D_{k+1}, the best score comes either from the nodes x on the top of D_{k+1} or from the nodes to the left of c_{k+1}. The definition correctly selects a maximum from the nodes on the top. For a node x to the left, note that a best scoring path can be partitioned into a best scoring path from x to c_{k+1} and a best scoring path from c_{k+1} to y. Again, the definition correctly selects a maximum.

To compute the best scores on row n, we compute for each table D_z in order for $z = 1, 2, \ldots$, the best scores to RB_z using the algorithm Propagate [2]. The time for one table is $O(n \log n)$. There are m/n tables so the total time is $O(m \log n)$. □

Lemma 2. *The space required to store a rank of DIST tables is* O(*the area that the tables cover*).

Proof. In a rank of size k, each table is square, so the edge-to-edge information occupies $O(k^2)$ space, which is O(the area covered by the table). □

5.1. *Building the ranks*

Next we consider how to efficiently build and discard a sequence of DIST table ranks. Consider a block of n rows in an edit graph. The DIST table ranks are built for each row *in order* from n up to 1 so that at each row i there are at most $\log n$ ranks between rows i and n. When we build the ranks for row i, we assume that the ranks for row $i+1$ are already available. For illustrative purposes, label each row i with $n-i$ (Fig. 4). In the binary representation of each label, the ones indicate exactly the number and size of ranks between rows i and n. For example, for row n with label 0, there are 0 ranks. For row $n-7$ with label $7 = 0111_2$, there are three ranks, the bottom rank of size 4, the next rank of size 2 and the top rank of size 1. Notice that for row $n-8$, the next row up, the label is $8 = 1000_2$ and there is a single rank of size 8 between this row and row n.

The order of the construction of the ranks can be determined by examining the label bits from right to left, stopping after the first 1 is reached. Each time we build a rank from smaller ranks, the smaller ranks are discarded. Consider, again, the case of row $n-8$. Reading the label from right to left, we encounter three zeros. The first indicates

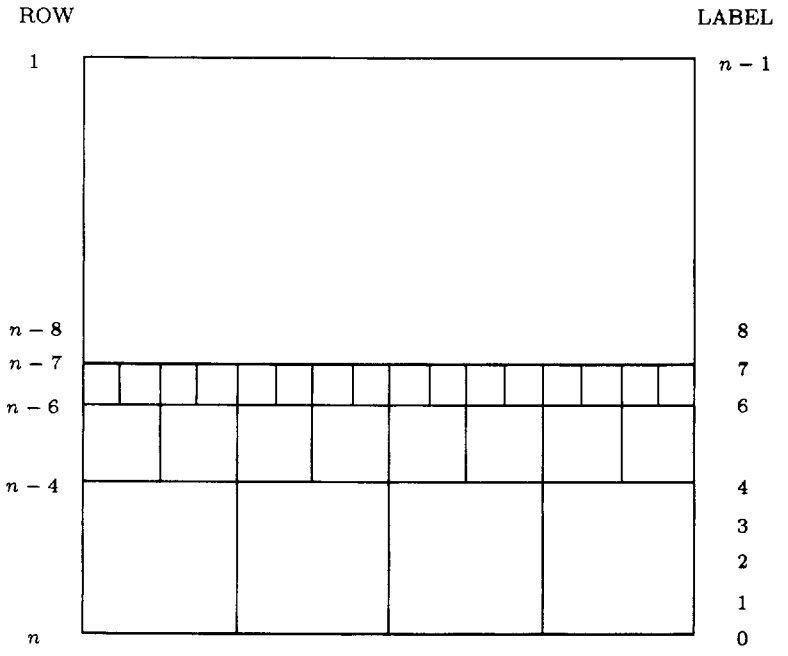


Fig. 4. Three ranks lie between rows $n-7$ and n.

a rank between rows $n-8$ and $n-7$ and the next a rank between rows $n-8$ and $n-6$. The last indicates a rank between rows $n-8$ and $n-4$. The 1 indicates a rank between rows $n-8$ and $n-0$, and this final rank is the only one not discarded.

Lemma 3. *Let G be an $n \times m$ edit graph with $n \leqslant m$. A sequence of DIST table ranks as described above can be constructed in time $O(nm \log n)$ and space $O(nm)$.*

Proof. By Lemma 2, the space for a table is O(the area it covers). Since smaller DIST tables are discarded, the tables are nonoverlapping. Therefore, the space is at most $O(nm)$. Each table in a rank is square and is constructed from four smaller tables as in the algorithm of [1]. The time to build all the tables is equivalent to the time to build m/n tables of size $n \times n$ or $O(mn \log n)$. □

5.2. *Jumping values*

Finally, for an edit graph G of size $n \times m$ we show how to efficiently calculate, for each k, the best score to every node $x \in RB_G$ beginning anywhere in rows k to n.

The values can be computed by the following algorithm which is implemented with the ranks construction just described. Note especially that we interleave the jumping of scores with the rank table construction. This is a key idea which permits our gain in space complexity:

Algorithm Jumps

1. For each node $x \in RB_G$ create a list $VALUES(x)$. Initialize the score at the top of each list of be zero.
2. For each row k from n up to 1 do
   (a) Build the rank tables for row k.
   (b) Starting with $IN(y) = 0$ for every node y in row k, compute $OUT(x)$ for every node $x \in RB_G$ by jumping values across the ranks using the method of Lemma 1. (Nodes in RB_G in rows $1, \ldots, k-1$ get no values from this computation.)
   (c) At each node $x \in RB_G$, compare $OUT(x)$ with the score at the top of $VALUES(x)$. If $OUT(x)$ is larger, add it to the top of $VALUES(x)$. Otherwise, discard $OUT(x)$ and add a duplicate of the previous score to the top. (This ensures that the scores are in nonincreasing order from the top.)

Let a node $x \in RB_G$ be in row n (or more generally x could also be in row k on the right edge of G). The final scores in $VALUES(x)$ are the best scores to node x. The important point here is that the filling of $VALUES(x)$ in step 2(c) serves as a checkpoint to guarantee that the scores are nonincreasing from the top. Therefore, the scores actually represent the best score obtained from a range of rows, rather than from a single row. The best score from rows 1 to n is on top, the best score from rows 2 to n is next, etc. This becomes important in our final algorithm presented in the next section.

Theorem 4. *Let G be an $n \times m$ edit graph with $n \leqslant m$. Using algorithm Jumps, for each k ($1 \leqslant k \leqslant n$), the best score to every node $x \in RB_G$ beginning anywhere in rows k to n, can be calculated in time $O(mn \log^2 n)$ and space $O(nm)$.*

Proof. *Time*: Building the ranks takes total time $O(mn \log n)$ (Lemma 3). Transferring values across one rank takes $O(m \log n)$ time (Lemma 1). Each row i must transfer its values across at most $\log n$ ranks for a time of $O(m \log^2 n)$ per row. The time for all n rows is therefore $O(mn \log^2 n)$.

Space: The space to store the ranks is $O(mn)$ (Lemma 3). Each of the $n + m$ lists $VALUES(x)$ holds at most n scores. The total space is therefore $O(mn)$. □

6. An $O(n^2)$ space algorithm

Here, we outline the $O(n^2)$ space and $O(n^2 \log^2 n)$ time algorithm. It has some of the flavour of an $n^{2.5} \log^{0.5} n$ time algorithm described in [2]. We adopt some of the nomenclature from that paper.

Algorithm Best Scores

- Partition the interval $[n, 1]$ into $\log n$ *panels* P_i of geometrically decreasing size (Fig. 5). The panels are $[n, n/2]$, $[n/2, n/4], \ldots, [2, 1]$ with partition indices $n/2^i$. Each partition index defines a rectangle R_i ($T(1, n/2^i)$ versus $T(n/2^i, n)$) of the edit graph.

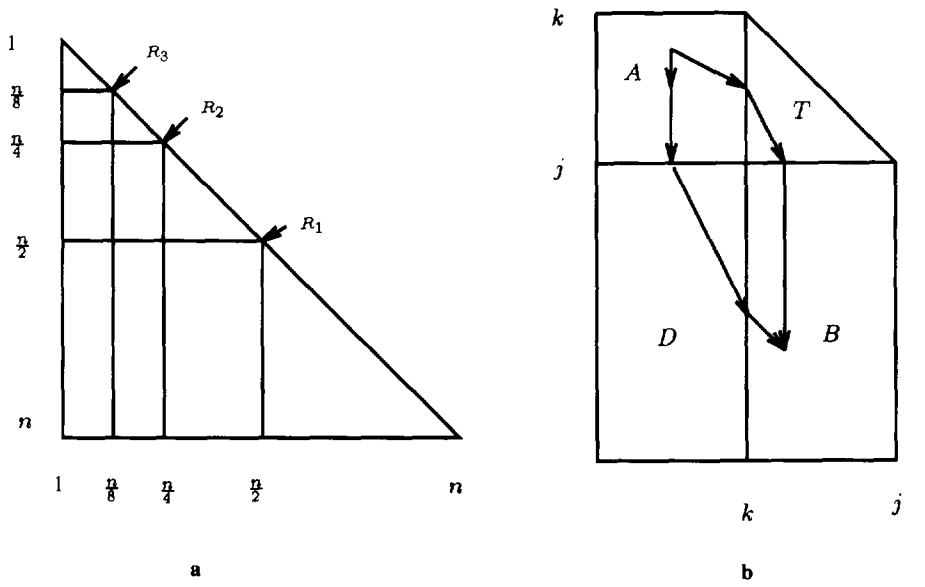


Fig. 5. (a) Partition indices and rectangles: (b) step 1, scores from nodes in A to nodes in B.

- Run the Smith–Waterman algorithm [8] in each rectangle R_i of the edit graph. Each node saves its best score.
- For each panel, P_i, $i = \log n, \ldots, 1$ do the following: Let $[k,j] = [n/2^i, n/(2^{i+1})]$ and associate three parts of the edit graph with the panel: (1) the rectangular part, A, of rows k to j, (2) the rectangular part, B, of columns k to j, and (3) the triangular part, T. Also, associate a DIST table rank D.

  At this point, we have to compute four classes of scores.

  (1) Scores that originate in A and end in T.

  (2) Scores that originate in T and end in B.

  (3) Scores that originate in A and end in B.

  (4) Scores that originate and end in T.

  1. *Class* (2) *and* (3): Run algorithm Jumps on the $n/2^i$ *rows* in A and T combined to get the best scores from nodes in A and T to row j. Although Jumps is described for rectangles, we can use appropriate weightings on the edges outside the lower triangle to preclude using those edges.

  Using DIST table rank D, jump the values on row j at the bottom of A to column k. We now have lists $VALUES(x)$ for every node in the LT border of B.

  Run a variation of algorithm Jumps on the $n/2^i$ *columns* in B. This time, we use Jumps to carry a set of scores on the LT border of B to a column in B. The variation builds the ranks for columns from k to j (i.e. from k to $k+1$, from k to $k+2$, etc.). At each iteration i, the top score in each list $VALUES(x)$ is removed and used as the value $IN(x)$.

  2. Class (1): Similar to step 1 above. Run algorithm Jumps on the $n/2^i$ *rows* in A to get the best scores from nodes in A to column k. Then run the variation of Jumps on the $n/2^i$ *columns* in T. (During iteration i, an appropriately large negative value can be used as $IN(x)$ for x in rows $1, \ldots, i-1$.)

  3. *Class* (4): Recursively run Best Scores on the triangle T.

  4. Each node picks the best of

     (1) the scores computed in steps 1, 2 or 3 above and

     (2) the score from the Smith–Waterman algorithm.
- Pick the best score over all the nodes.

Note again that the scores in $VALUES(x)$ represent the best score from a range of rows. When we run the variation of Jumps to carry scores out to *column* i, we do not want to jump individual scores from each of *rows* i to j because this would take too long. Instead, we already have the single best score from the entire range of rows i to j in each of the nodes on column k. We only need to jump one score from each of these nodes to column i, thus preserving our time bounds.

Theorem 5. *Algorithm Best Scores runs in time* $O(n^2 \log^2 n)$ *and space* $O(n^2)$ *assuming the DIST table ranks D for all the panels can be constructed in these bounds.*

Proof. The algorithm operates on a sequence of triangles of geometrically decreasing area. Excluding the DIST table ranks D, the time and space recursions are

$$T(n) \leqslant t(n) + \sum_{i=1}^{\log n} T\left(\frac{n}{2^i}\right),$$

$$S(n) \leqslant s(n) + \sum_{i=1}^{\log n} S\left(\frac{n}{2^i}\right),$$

where $t(n)$ and $s(n)$ are respectively (within just the largest triangle) the time for the Smith–Waterman algorithm in the rectangles plus the time for algorithm Jumps (steps 1 and 2) and the space for the ranks and the lists.

For the largest rectangle, the time for Smith–Waterman is $O(n^2)$ and the time for Jumps is $O(n^2 \log^2 n)$. Since the rectangles and panels are of geometrically decreasing size, $t(n) = O(n^2 + n^2 \log^2 n)$. Similarly, $s(n)$ is $O(n^2)$ since the DIST tables we construct within the panels (for steps 1 and 2) never overlap. The solution then for $T(n)$ is $O(n^2 \log^2 n)$ and for $S(n)$ is $O(n^2)$. □

7. Constructing the DIST table ranks D

In this section, we show how to construct the DIST table ranks D in the time and space bounds of Theorem 5.

Theorem 6. *The DIST table ranks D can be constructed in* $O(n^2 \log n)$ *time and* $O(n^2)$ *space.*

Proof. Consider first the original lower triangular graph. We start with the rank for the smallest panel $P_{\log n}$. Note that each table in this rank has size 2. We can build this rank, and then complete algorithm Best Scores for the smallest panel. Then, *and this is the key point*, we can use the rank for panel $P_{\log n}$ to build the rank for the next largest panel $P_{\log n - 1}$ and then discard the rank for panel $P_{\log n}$. We will never use it again. Each table in the new rank has size 4, so we can use the algorithm of [1] as in Lemma 3. Continuing on in this way, the time for all the tables is $O(n^2 \log n)$, the time to build the largest rank. We discard smaller tables after we use them to build the next largest tables, so the tables are nonoverlapping. Since the recursively processed triangles are geometrically decreasing in area, the total space is $O(n^2)$. □

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

[3] Z.M. Kedem and H. Fuchs, On finding several shortest paths in certain graphs, in: *Proc. 18th Allerton Conf. on Communication Control and Computing* (1980) 677–683.
[4] G. Landau and J. Schmidt, An algorithm for approximate tandem repeats, in: *Proc. 4th Ann. Symp. on Combinatorial Pattern Matching*, Lecture Notes in Computer Science, Vol. 648 (Springer, Berlin, 1993) 120–133.
[5] V.I. Levenshtein, Binary codes capable of correcting deletions, insertions and reversals, *Soviet Phys. Dokl.* **10** (1966) 707–710.
[6] W. Miller, An algorithm for locating a repeating region, manuscript, 1992.
[7] E. Myers, An O(*ND*) difference algorithm and its variants, *Algorithmica* **1** (1986) 251–266.
[8] T.F. Smith and M.S. Waterman, Identification of common molecular sequences, *J. Mol. Biol.* **147** (1981) 195–197.
[9] R.A. Wagner and M.J. Fisher, The string to string correction problem, *J. ACM* **21** (1974) 168–173.