RAIRO-Inf. Theor. Appl. **39** (2005) 175-189
DOI: 10.1051/ita:2005011

A SPARSE DYNAMIC PROGRAMMING ALGORITHM FOR ALIGNMENT WITH NON-OVERLAPPING INVERSIONS

Alair Pereira do Lago[1], Ilya Muchnik[2] and Casimir Kulikowski[2]

Abstract. Alignment of sequences is widely used for biological sequence comparisons, and only biological events like mutations, insertions and deletions are considered. Other biological events like inversions are not automatically detected by the usual alignment algorithms, thus some alternative approaches have been tried in order to include inversions or other kinds of rearrangements. Despite many important results in the last decade, the complexity of the problem of alignment with inversions is still unknown. In 1992, Schöniger and Waterman proposed the simplification hypothesis that the inversions do not overlap. They also presented an $O(n^6)$ exact solution for the alignment with non-overlapping inversions problem and introduced a heuristic for it that brings the average case complexity down. (In this work, n is the maximal length of both sequences that are aligned.) The present paper gives two exact algorithms for the simplified problem. We give a quite simple dynamic program with $O(n^4)$-time and $O(n^2)$-space complexity for alignments with non-overlapping inversions and exhibit a sparse and exact implementation version of this procedure that uses much less resources for some applications with real data.

Mathematics Subject Classification. 05C85, 68R15, 90C27, 90C39.

1. Introduction

In evolution history, some biological events introduce changes in the DNA sequences. Some typical biological events include *mutations*, in which a nucleotide is substituted by another one, *deletions* and *insertions* of nucleotides. Hence, any

[1] Universidade de São Paulo, Brasil; alair@ime.usp.br
[2] Rutgers University

```
actaga-tcagtc-a      actaga-tcagtca
| | || || | | |      | | || ||****|
a-ttgaatc-gacta      a-ttgaatcgacta
```

FIGURE 1. Two alignments for *actagatcagtca* and *attgaatcgacta*.

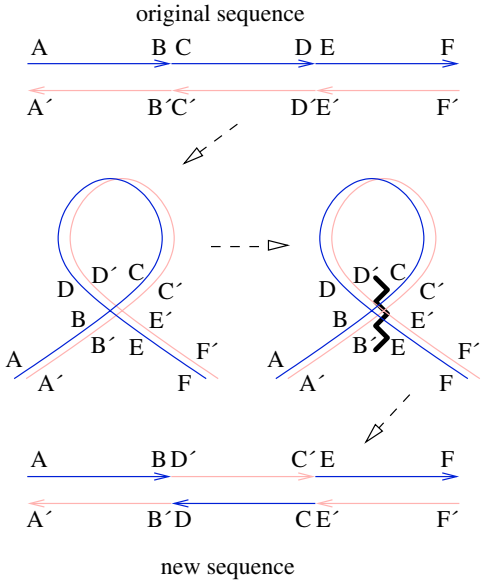

FIGURE 2. Example of DNA inversion.

sequence comparison must take into consideration the possibility of these events if it is expected to identify high similarity between two sequences. Typical alignment procedures try to identify which parts do not change and where these biological events are, after exhibiting one best alignment according to some optimization criteria.

For instance, in Figure 1, we see two alignments of *actagatcagtca* against *attgaatcgacta*. From left to right, one can detect a deletion of c, a mutation from a to t and an insertion of a in both alignments. At the rightmost part, the first alignment reports a deletion of a, a mutation from t to a and an insertion of t. In contrast, the second alignment highlights the inversion of *agtc* to its reverse[1] complement[2] *gact*. Since common aligners do not take *inversions* into consideration, they would report the first alignment. Figure 2 shows how an inversion can occur and why one segment is substituted by its reverse complement sequence.

Alignments can be associated with a set of edit operations that transform one sequence to the other. Usually, the only edit operations that are considered are

[1] The order of symbols is reversed.

[2] Symbols a and t are swapped and c and g are swapped as well.

the *substitution* (mutation) of one symbol by another one, the *insertion* of one symbol and *deletion* of one symbol. If costs are associated with each operation, there is a classic $O(n^2)$ dynamic program that computes a set of edit operations with minimal total cost and exhibits the associated alignment, which has good quality and high likelihood for realistic costs.

Consider three new possible edit operations:

- the 2-*reversion*, which reverses the order of *two consecutive symbols*;
- the *reversion* operation, which reverses the order of *any segment* of symbols instead of a segment of length 2;
- the *inversion* operation, which substitutes any segment by its *reverse complement* sequence. The inversion operation is the operation that is interesting in molecular biology. (Diagram in Fig. 2 represents the inversion of a DNA segment. The DNA sequence is coded by two complementary tapes and any tape has an orientation from 5' to 3'.)

Associated with any of these three operations, we can define new alignment problems. For instance, given two sequences and fixed costs for each kind of edit operation, the *alignment with inversions* problem is an optimization problem that queries the minimal total cost of a set of edit operations that transforms one sequence to the other. Moreover, one may also be interested in the exhibition of its correspondent alignment and/or edit operations. Similarly, we can define the *alignment with* 2-*reversions* and the *alignment with reversions* problems.

In 1975, Wagner [22] studied the alignment with 2-reversions problem and proved that it admits a polynomial solution if the cost of a 2-reversion is null. On the other hand, he also proved that obtaining an optimal solution is *NP-hard*, if any operation has a constant positive cost.

To the best of our knowledge, the computational complexities of alignment with reversions and alignment with inversions problems are unknown.

In order to deal with alignments with inversions, three main approaches have been considered through the years:

(1) non-overlapping inversions;
(2) sorting unsigned permutations by reversals and;
(3) sorting signed permutations by reversals.

Before we proceed with the results of this paper, we give a brief summary of these three approaches.

The first approach was introduced in 1992, by Schöniger and Waterman [20], when they introduced a *simplification hypothesis*: *all regions involved in the inversions do not overlap*. This led to the *alignment with non-overlapping inversions* problem. They presented an $O(n^6)$ solution for this problem and also introduced a *heuristic* for it that reduced the running-time. This heuristic uses the algorithm by Waterman and Eggert [23] that reports the K best non mutual intersecting local alignments in order to reduce the running time to something between $O(n^2)$ and $O(n^4)$, depending on the data.

A second approach has been tried in order to study inversions. This approach applies well for alignment of *sequences of genes* and has been very used with

mitochondrial genomes. It does not apply for sequences of nucleotides nor for sequences of aminoacids because *no repetitions* of symbols are allowed. (Repeated genes and paralogs are not allowed.) Moreover, *no insertion* and *no deletion* are considered and the only admitted operation is the reversal, where a *reversal* is defined as transforming a sequence like $(1, 2, 3, 4, 5)$ into $(1,\ 4, 3, 2,\ 5)$.

The problem solved by this approach, also called *sorting unsigned permutations by reversals*, means the computation of a kind of "edit distance" of two permutations where *only the reversal operation is allowed*. In this case, the data are two permutations of $(1, 2, 3, \ldots, n)$, where n is the number of genes. Kececioglu and Sankoff [18] gave a quadratic 2-approximation algorithm in 1995 and Christie [3] gave a 3/2-approximation algorithm in 1998. Caprara [2] proved in 1999 that this problem is in fact NP-hard.

The third interesting approach is the problem called *sorting signed permutations by reversals*. This is the same problem as sorting unsigned permutations by reversals up to the fact that signs are also attributed to a gene and a reversal also flips its sign. For instance, one reversal could transform $(1, 2, 3, 4, 5)$ to $(1, -4, -3, -2, 5)$. This sign is usually associated with the direction of the gene (which DNA strand it belongs to).

Hannenhalli and Pevzner [13, 14] gave the first polynomial algorithm for the problem in 1995 and started a sequence of papers based on this approach. Hannenhalli and Pevzner's algorithm was $O(n^4)$ and it was improved to $O(n^2)$ by Kaplan, Shamir and Tarjan [16, 17] in 1997. In 2001, Bader, Moret and Yan [1] gave an algorithm that computes the minimal number of reversals distance in $O(n)$ (finding the sequence of reversals still requires $O(n^2)$). These studies have been applied to philogenetic reconstruction studies.

In 2000, El-Mabrouk [7, 8] studied the inclusion of two operations: insertions and deletions of gene segments. She obtained partial results and gave an exact polynomial solution for one case and a polynomial heuristic with a polynomial tester for optimality in the other case. Repeated symbols are still not allowed. In 2002, El-Mabrouk [9] also obtained some partial results on considering reversals and duplications.

This paper gives two exact algorithms for the alignment with non-overlapping inversions problem, which is the first approach used to attack the problem of alignments with inversions. Algorithm 1 is an $O(n^4)$ solution that uses $O(n^2)$ space and Algorithm 2 is a sparse dynamic implementation of it that reduces the resources usage if there are $o(n^2)$ matches. This is often expected if the cardinality of the alphabet is large and it is true for the kind of application we have in mind, where the letters are DNA fragments of fixed length.

2. Basic definitions

Let A be any *alphabet*, a set of *letters*. Any finite sequence on A is also called a *word on* A or simply a *word* if the alphabet is clear. Let A^* be the set of all words on A, including the *empty word* denoted by 1. We identify words of length 1 to

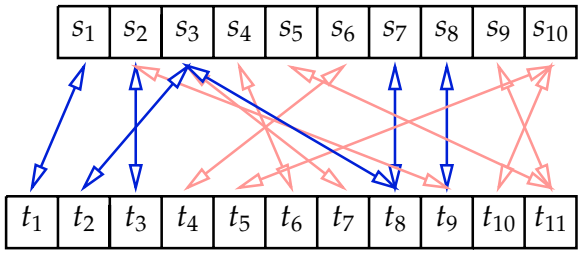

FIGURE 3. Example of a matching graph.

the letters they contain. The concatenation $\cdot$ of words is an associative operation defined over A^* and it will be often omitted. Let $w = w_1 w_2 \dots w_k$ be a word. We denote by $|w|$ the *length* k of w. We will also denote w_i, the i-th letter of w, by $w[i]$. Let $x, y, z \in A^*$. We denote by xA^* the set $\{xy | y \in A^*\}$, we denote by A^*x the set $\{yx | y \in A^*\}$ and we denote by A^*xA^* the set $\{yxz | y, z \in A^*\}$. We say that x is a *prefix of* w if $w \in xA^*$, we say that x is a *suffix of* w if $w \in A^*x$ and we say that x is a *factor of* w if $w \in A^*xA^*$. For $1 \leq i \leq j \leq k$, the factor $w_i w_{i+1} \cdots w_j$ of w is also represented by $w[i\,..\,j]$.

Let $-$, also called *inversion*, be any operation on A^* that satisfies the following properties:

(1) $\overline{a} \in A, \forall a \in A$;
(2) $\overline{x \cdot y} = \overline{y} \cdot \overline{x}, \forall x, y \in A^*$.

Notice that the inversion operation on A^* is defined by its values on the letters of A. For instance, let $A = \{a, c, t, g\}$ and let $s \in A^*$ be any DNA sequence. If the inversion is defined by $\overline{a} = a$, $\overline{t} = t$, $\overline{c} = c$ and $\overline{g} = g$, it maps s to its *reverse sequence*. On the other hand, if the inversion is defined by $\overline{a} = t$, $\overline{t} = a$, $\overline{c} = g$ and $\overline{g} = c$, it maps s to its *reverse complement sequence*. The last case is the interesting case for DNA sequences in molecular biology.

Let $\omega : A \times A \to \mathbb{R} \cup \{-\infty\}$ be any *weight* function. We say that $(a, b) \in A \times A$ is a *match* if $\omega(a, b) \neq -\infty$.

Let $s = s_1 s_2 \cdots s_k$ and $t = t_1 t_2 \cdots t_{k'}$ be two words. We define the *matching graph of* s *and* t as the weighted and colored bipartite graph $G = G(s, t, \omega, -) = (V, E)$. The vertex set is a set of symbols $V = \{s_1, \dots, s_k, t_1, \dots, t_{k'}\}$ and has size $|s| + |t|$. (We do not identify two symbols s_i and t_j even in the case the letters s_i and t_j are the same letter in A.) The edge set E is a double copy of $K_{|s|,|t|}$: for any pair of vertices s_i and t_j we link them with one *blue/dark-gray edge of weight* $\omega(s_i, t_j)$ and one *red/light-gray edge of weight* $\omega(s_i, \overline{t_j})$. (In fact, edges with weight $-\infty$ will not be used for our purposes and could be deleted.) An edge with weight that is not $-\infty$ is called a *direct match* if it is blue and it is called *inverted match* if it is red. In Figure 3 we have an example of a matching graph. In this figure, as in others, we do not draw edges with weight $-\infty$.

Given $u \in V^*$ a nonempty factor of $s_1 s_2 \cdots s_k$ and $v \in V^*$ a nonempty factor of $t_1 t_2 \cdots t_{k'}$, we call $B = (u, v)$ a *block*. Given a block $B = (u, v)$, there exist integers

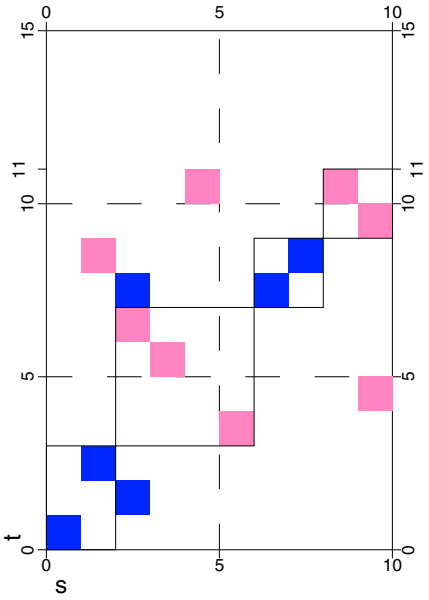

FIGURE 4. Bicoloured incidence matrix.

$1 \leq i' \leq i \leq |s|$ and $1 \leq j' \leq j \leq |t|$ such that $u = s[i' \,..\, i]$ and $v = t[j' \,..\, j]$. Hence, we can associate $([i' \,..\, i] \times [j' \,..\, j])$, the *rectangle associated with the block* B, with B. In Figure 4 we have an incidence matrix of the matching graph of Figure 3 and only matches are shown. Cells at position (s_i, t_j) are colored according to the color of the edge linking these vertices. (We could use purple if we had both matches for the same cell.) We can also see four rectangles corresponding to the blocks $(s_1 s_2, t_1 t_2 t_3)$, $(s_3 s_4 s_5 s_6, t_4 t_5 t_6 t_7)$, $(s_7 s_8, t_8 t_9)$ and $(s_9 s_{10}, t_{10} t_{11})$. Cells inside a rectangle correspond to edges with vertices in the factors that form the respective block.

Consider an edge that links s_i to t_j and an edge that links $s_{i'}$ to $t_{j'}$. We say that they *cross* each other if $(i - i')(j - j') < 0$, we say that they *touch* each other if $(i-i')(j-j') = 0$ and we say that they are *parallel* if $(i-i')(j-j') > 0$. Let $M \subseteq E$ be any set of edges in a matching graph. Recall that M is called a *matching* if any two edges of M do not touch each other. Moreover, M is called a *direct matching* if it has only direct matches and any two of them are parallel. Furthermore, M is called an *inverted matching* if it has only inverted matches and any two of them cross each other. The *restriction of* M *to a block* $B = (s[i' \,..\, i], t[j' \,..\, j])$ is the submatching of all edges of M with vertices in $s[i' \,..\, i]$ and $t[j' \,..\, j]$). Finally, M is called a *blockwise inverted matching*, or simply a *bimatching*, if, considering words in V^*, there exists $l \geq 1$ such that:

- there are l blocks $B_i = (u_i, v_i)$ such that $s = s_1 s_2 \dots s_k = u_1 u_2 \dots u_l$ and $t = t_1 t_2 \dots t_{k'} = v_1 v_2 \dots v_l$;

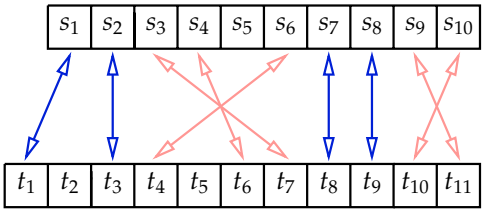

FIGURE 5. Example of a bimatching with four blocks.

- for any $i \in \{1, \ldots, l\}$, the restriction M_i of M to the block B_i is either a direct matching or an inverted matching;
- $M = \cup_{i=1}^{l} M_i$.

Given a bimatching M, the number of blocks l is not unique since any direct submatching M_i can also be split as a union of smaller direct submatchings. However, the number of blocks such that the corresponding restriction M_i is an inverted matching does not change. This is the *number of inversions of* M, $\imath(M)$. Given a bimatching M, the smallest possible value for l is called the *number of blocks of* M.

One can notice that in Figure 5 we have a bimatching M with four blocks which are those associated with the blocks of Figure 4. There are $\imath(M) = 2$ maximal inverted submatchings.

3. Two algorithms for optimal bimatching

Given a bimatching M, one can easily deduce an alignment with non-overlapping inversions associated with it. One could naturally define the weight of a matching in a matching graph to be the sum of the weights of its edges. However, it is common in alignments to give penalties for biological events like mutations, insertions or deletions. In order to be more general, we attribute a *inversion penalty* $I \geq 0$ for every inverted submatching M_i. (We deal with penalties for insertions and deletions in Sect. 4.) Hence, we define the *weight of a bimatching* M as

$$\omega(M) = \sum_{e \in M} \omega(e) - \imath(M)I.$$

Hence, the following is the optimization problem we are interested in solving.

Problem 1. *Given a matching graph $G = G(s, t, \omega, -)$, we want to compute the maximal weight $\omega(M)$ for all possible bimatchings M. As usual, we may also be interested in a bimatching of maximal weight.*

Such a bimatching M^* of maximal weight is called an *optimal bimatching* of s and t and its weight $\omega(M^*)$ is denoted by $\mathrm{BIM}(s, t)$. The weight of an *optimal direct matching* is denoted by $\mathrm{DM}(s, t)$. Next lemma proves some recurrences on BIM.

Lemma 2. *Let A be an alphabet, let $a, b \in A$ be any letters, let $u, v \in A^*$ be any words on A, let us give an inversion operation $-$ on A^* and a weight function ω and let I be an inversion penalty. Hence, the following are true:*

(1) $\mathrm{BIM}(1, v) = \mathrm{BIM}(u, 1) = 0$;

(2) $$\mathrm{BIM}(ua, vb) = \max \left\{ \begin{array}{l} \mathrm{BIM}(u, v) + \omega(a, b) \\ \mathrm{BIM}(ua, v) \\ \mathrm{BIM}(u, vb) \\ B - I \end{array} \right\}$$
where $B = \max\{\mathrm{BIM}(u_1, v_1) + \mathrm{DM}(u_2, \overline{v_2}) | ua = u_1 u_2, vb = v_1 v_2, u_2 v_2 \neq 1\}$.

Proof. By definition, $\mathrm{BIM}(1, v) = \mathrm{BIM}(u, 1) = 0$. There are four cases for an optimal bimatching of ua and vb: it includes the blue edge (a, b), which leads to $\mathrm{BIM}(ua, vb) = \mathrm{BIM}(u, v) + \omega(a, b)$; it does not include any edge using b, which leads to $\mathrm{BIM}(ua, vb) = \mathrm{BIM}(ua, v)$; it does not include any edge using a, which leads to $\mathrm{BIM}(ua, vb) = \mathrm{BIM}(u, vb)$; its last block is inverted, which leads to $\mathrm{BIM}(ua, vb) = -I + \max\{\mathrm{BIM}(u_1, v_1) + \mathrm{DM}(u_2, \overline{v_2}) | ua = u_1 u_2, vb = v_1 v_2, u_2 v_2 \neq 1\}$. □

Lemma 2 leads us to the dynamic programming algorithm given by Algorithm 1. In fact, if we associate the cases seen in the proof of Lemma 2 with the lines in Algorithm 1, the first case is considered in line 8 and the next two cases are considered in line 7. Every subcase considered in the max operation of fourth case is considered at line 15. Notice that $1 \leq i' \leq i \leq |s|$, in contrast to $1 \leq j \leq j' \leq |t|$. One can notice that the maximum B of fourth case is not computed first in order to compute other B's later. Partial computations are done in advance in Algorithm 1 in order to factorize better the computations of $L[i', j'] = \mathrm{DM}(s[i' \,..\, i], \overline{t[j \,..\, j']})$. These facts lead to the correctness of Algorithm 1. Considering the four loops and the dimension of the two matrices L and B, one can then prove Theorem 3.

Theorem 3. *Algorithm 1 computes in $B[|s|, |t|]$ the maximal weight of a bimatching $\mathrm{BIM}(s, t)$ with time complexity $|s|^2|t|^2$ and space complexity $|s||t|$.*

As usual, by tracking any maximality choice, one can obtain also an optimal bimatching. The numbers present in positions (s_i, t_j) of the incidence matrix of Figure 6 show the corresponding values $B[i, j]$ computed for most important cells in Algorithm 1 if all matches have weight 1. One can therefore obtain the bimatching of Figure 5 as the optimal bimatching for the matching graph of Figure 3. It should be said that this algorithm was first published in 2003 [6] and a preliminary version of Algorithm 2 has been used since 2001 [4]. Gao *et al.* independently published a time complexity $|s|^2|t|^2$ and space complexity $|s||t|$ for alignments with non-overlapping inversions in 2003 [12]. This work does not include a sparse implementation like Algorithm 2.

3.1. A sparse dynamic program for $\mathrm{BIM}(s, t)$

In Algorithm 2, we improve the computational resources usage required by Algorithm 1 when there are $o(|s||t|)$ matches. We use techniques that appeared

Algorithm 1. An $O(n^4)$-time and $O(n^2)$-space algorithm for BIM.

```
BIM(s, t)
 1 ▷ Compute the table B[i, j] = BIM(s[1 . . i], t[1 . . j])
 2 Let B[i, j] be 0 for i = 0 or j = 0
 3 for i from 1 to |s| do
 4     for j from |t| downto 1 do
 5         B[i, j] ← −∞
 6     for j from 1 to |t| do
 7         B[i, j] ← max(B[i, j], B[i − 1, j], B[i, j − 1])
 8         B[i, j] ← max(B[i, j], B[i − 1, j − 1] + ω(s[i], t[j]))
 9         ▷ Compute L[i′, j′] = DM(s[i′ . . i], t[j . . j′]) and set B[i, j′]
10         Let L[i′, j′] be 0 for i′ = i + 1 or j′ = j − 1
11         for j′ from j to |t| do
12             for i′ from i downto 1 do
13                 L[i′, j′] ← max(L[i′, j′ − 1], L[i′ + 1, j′])
14                 L[i′, j′] ← max(L[i′ + 1, j′ − 1] + ω(s[i′], t[j′]), L[i′, j′])
15                 B[i, j′] ← max(B[i, j′], L[i′, j′] + B[i′ − 1, j − 1] − I)
16 return B
```

in a work by Hunt and Szymanski [15] in 1977 for the computation of the length of the longest common subsequence (LCS) (A good survey can be found in [21]). The name *Sparse Dynamic Programming* was adopted by Eppstein *et al.* [10, 11] in a well known work published in 1992. The main idea behind these techniques is that only cells associated with a match are "visited", and this is done here.

Before we proceed, we need a few more definitions. We say that a rectangle $([i\,..\,j]\times[k\,..\,l])$ *dominates* the rectangle $([i'\,..\,j']\times[k'\,..\,l'])$ if $j \le j'$ and $l \le l'$. The pair (j, l) is called the *right upper corner of* $([i\,..\,j] \times [k\,..\,l])$ and the domination relation depends only on the right upper corners. We will give direction signs for rectangles according to whether we admit a direct or an inverted matching as the restriction of a bimatching to the corresponding block. We define the *rank* of a signed rectangle $([i\,..\,j] \times [k\,..\,l])$ to be the maximal weight of bimatchings that are restricted to the block $(s[a\,..\,j], t[c\,..\,l])$ and admit a block decomposition where the last block is $(s[i\,..\,j], t[k\,..\,l])$ and the restriction to it has the right direction. We say that a signed rectangle is *dominant* if any other signed rectangle that dominates it has either a smaller rank or the same right upper corner. The dominance relation is a partial quasi-order where the equivalence classes have always the same rank and the same right upper corner.

The only considered signed rectangles in Algorithm 2 are all direct rectangles $([i\,..\,i] \times [j\,..\,j])$ such that (i, j) is a direct match and all inverted rectangles $([i\,..\,j] \times [k\,..\,l])$ such that both (j, k) and (i, l) are inverted matches. We store only dominant signed rectangles (one for each equivalence class). This naturally gives us the rank of their right upper corner positions. (See function UPDATE in Algorithm 3.)

Algorithm 2. An $O(m^2(\log^2 k + \log m))$-time and $O(m + k)$-space algorithm for BIM, where $m = O(|s||t|)$ is the number of matches and $k = O(|s||t|)$ is the number of dominant rectangles

BIM$(s[a\,..\,b], t[c\,..\,d], M, \omega)$

1 ▷ Compute R, an AVL-tree of AVL-trees R_r of dominant rectangles
2 ▷ Any AVL-tree $R[r] = R_r$ has only rectangles of rank r, for $r \in \mathbb{R}$
3 ▷ In $R[r]$, rectangles $([i'\,..\,i] \times [j\,..\,j'])$ are ordered by i
4 ▷ In R, AVL-trees $R[r]$ are ordered by the rank r
5 ▷ $([x\,..\,y] \times [u\,..\,v]) \in R[r]$ dominates $([x'\,..\,y'] \times [u'\,..\,v']) \in R[r']$ implies $r \le r'$
6 ▷ RANK$(R, i, j) =$ BIM$(s[a\,..\,i], t[c\,..\,j])$
7 ▷ M: list of matches with signs $(i, j, d') \in [a\,..\,b] \times [c\,..\,d] \times \{+1, -1\}$
8 ▷ M is ordered by i as first key, then by j as secondary key, then by d'
9 $\delta \leftarrow$ the reflection-rotation defined by $\delta(i, j, d') = (j, |s| + 1 - i, -d')$
10 $R \leftarrow \emptyset$
11 $R[-\infty] \leftarrow \emptyset$
12 ▷ $R[-\infty]$ receives a rectangle that dominates any rectangle
13 UPDATE$(R, -\infty, ([a-1\,..\,a-1] \times [c-1\,..\,c-1]), +1)$
14 $R[+\infty] \leftarrow \emptyset$
15 **for** $(i, j, d') \in M$ **do**
16 $\quad M' \leftarrow \delta(M \cap [a\,..\,i] \times \mathbb{Z} \times \{-1\})$ ▷ all inverted matches mapped by δ
17 $\quad$ Sort all $(i, j, +1) \in M'$ by i as first key, then by j as secondary key
18 $\quad$ **if** $d' = +1$ **then**
19 $\quad\quad r \leftarrow \omega(s[i], t[j]) +$ RANK$(R, i-1, j-1)$
20 $\quad\quad$ **if** $r >$ RANK(R, i, j) **then**
21 $\quad\quad\quad$ UPDATE$(R, r, ([i\,..\,i] \times [j\,..\,j]), +1)$
22 $\quad$ **else** $M'' \leftarrow M' \cap [j\,..\,d] \times \mathbb{Z} \times \{+1\}$
23 $\quad\quad$ ▷ $[j\,..\,d] \times [|s| + 1 - i\,..\,|s| + 1 - a] \times \{+1\} = \delta([a\,..\,i] \times [j\,..\,d] \times \{-1\})$
24 $\quad\quad \varphi \leftarrow$ function defined by $\varphi(f, g) = \omega(\overline{g}, \overline{f})$
25 $\quad\quad B \leftarrow$ BIM$((t[j\,..\,d], \overline{s}[|s| + 1 - i\,..\,|s| + 1 - a], M'', \varphi))$
26 $\quad\quad$ **for** $\delta(i', j', -1) \in M''$ **do**
27 $\quad\quad\quad$ ▷ DM$(s[i'\,..\,i], \overline{t[j\,..\,j']}) +$ BIM$(s[a\,..\,i'-1], t[c\,..\,j-1]) - I$
28 $\quad\quad\quad r \leftarrow$ RANK$(B, j', |s| + 1 - i') +$ RANK$(R, i'-1, j-1) - I$
29 $\quad\quad\quad$ **if** $r >$ RANK(R, i, j') **then**
30 $\quad\quad\quad\quad$ UPDATE$(R, r, ([i'\,..\,i] \times [j\,..\,j']), -1)$
31 **return** R

The ranks of other positions are "automatically propagated" from dominant positions. (See function RANK in Algorithm 3.)

Comments in the algorithms explain well the used data structure. One can notice that the computation of DM$(s[i'\,..\,i], \overline{t[j\,..\,j']})$ that was done in Algorithm 1

is now recursively done at line 25 of Algorithm 2. If M is the matches list applied to the first call of BIM, the recursive call will only take inverted matches of M. In fact, these inverted matches are reflected and rotated by the δ bijective transformation and any recursive call uses a subset M'' of these transformed matches. These matches are all direct and no inner recursive call is performed. This reflection-rotation is performed in order to use the same function BIM for the computation of DM. Taking advantage of the orderings of the lists M and M' we can efficiently perform the intersection at line 22 since M'' is a tail of M'. Taking advantage of the orderings of the lists M and M' we can efficiently perform the intersection at line 16 and the ordering at line 17. Considering the four cases seen in the proof of Lemma 2, the first case is treated at lines 19–21. The next two cases are automatically propagated from dominant positions by the function RANK. The fourth case is treated at lines 28–30.

Algorithm 3. Functions RANK and UPDATE used in Algorithm 2.

```
RANK(R, i, j)
 1 ▷ Return the rank of position (i, j)
 2 W ← maximal weight assumed by ω
 3 a ← −∞
 4 b ← +∞
 5 m ← the rank of the root of R
 6 while there is a third rank between a and b do
 7     ▷ ∃([x..y] × [u..v]) ∈ R[a] that dominates ([i..i] × [j..j])
 8     ▷ ∄([x..y] × [u..v]) ∈ R[b] that dominates ([i..i] × [j..j])
 9     if ∃m' ∈ [m..m + W) such that
       ∃([x..y] × [u..v]) ∈ R[m'] that dominates ([i..i] × [j..j]) then
10         a ← m'
11         while m ≤ a do
12             m ← right child of m
13     else b ← m
14         m ← left child of m
15 return a

UPDATE(R, r, ([i'..i] × [j..j']), d)
 1 ▷ Insert the rectangle ([i'..i] × [j..j']) of rank r > RANK(R, i, j') in R
 2 ▷ ([x..y] × [u..v]) ∈ R[r] dominates ([x'..y'] × [u'..v']) ∈ R[r'] im-
     plies r < r'
 3 foreach ([x..y] × [u..v]) dominated by ([i'..i] × [j..j']) do
 4     if ([x..y] × [u..v]) ∈ R[r'] for some r' ≤ r then
 5         Remove ([x..y] × [u..v]) from R[r']
 6 Insert ([i'..i] × [j..j']) in R[r]    ▷ We had r > RANK(R, i, j')
```

In function RANK in Algorithm 3, we perform a binary-search-like in R in order to find out the rank of position (i, j) and a and b are the limits of this search. Unfortunately, it may happen that $\nexists([x..y] \times [u..v]) \in R[b]$ that dominates

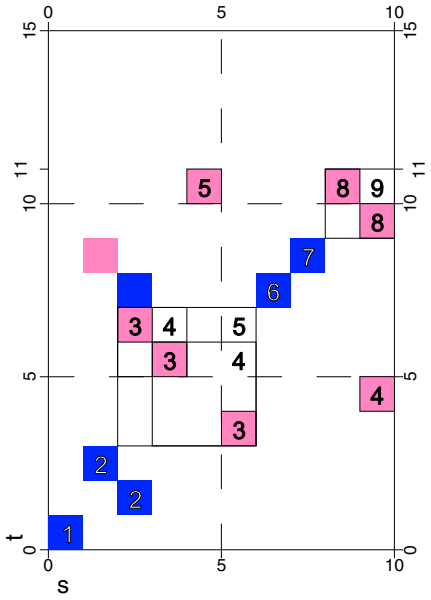

FIGURE 6. Dominant rectangles and their ranks.

$([i\,..\,i] \times [j\,..\,j])$, but $\exists b' > b$ such that $\exists([x'\,..\,y'] \times [u'\,..\,v']) \in R[b']$ that dominates $([i\,..\,i] \times [j\,..\,j])$. Since one can always delete one edge from the boundary from the rectangle $([x'\,..\,y'] \times [u'\,..\,v'])$, one can suppose $b' < b + W$ where W is the maximal weight assumed by ω. This is checked at line 9. Notice that the intervals $[m\,..\,m + W)$ only contains m if all matches have weight 1 and the penalty I is integer. In function UPDATE in Algorithm 3, using the pattern of inserted rectangles, the rectangles $([x\,..\,y] \times [u\,..\,v]) \in R[r']$ analyzed at loop at line 3 have $y = i$ and can be kept in a linked list ordered by v (and hence by r', too).

Figure 6 shows dominant rectangles with their corresponding ranks in the right upper corners as computed in Algorithm 2 (all matches have weight 1). Figure 7 shows an example of possible data that have been applied to the algorithm BIM. Since the number of matches is much smaller than the size of the matrix, the sparse implementation is much faster, despite non-sparse implementation simplicity. The DNA sequences correspond to two syntenic regions from two bacteria (*Xylella fastidiosa*, genes 0227-0239, and *Pseudomonas aeruginosa*, genes 4727-4746). These sequences were split in fragments of length 100 in order to form the alphabet. A match between two fragments is assumed if the alignment score is above an adequately chosen threshold.

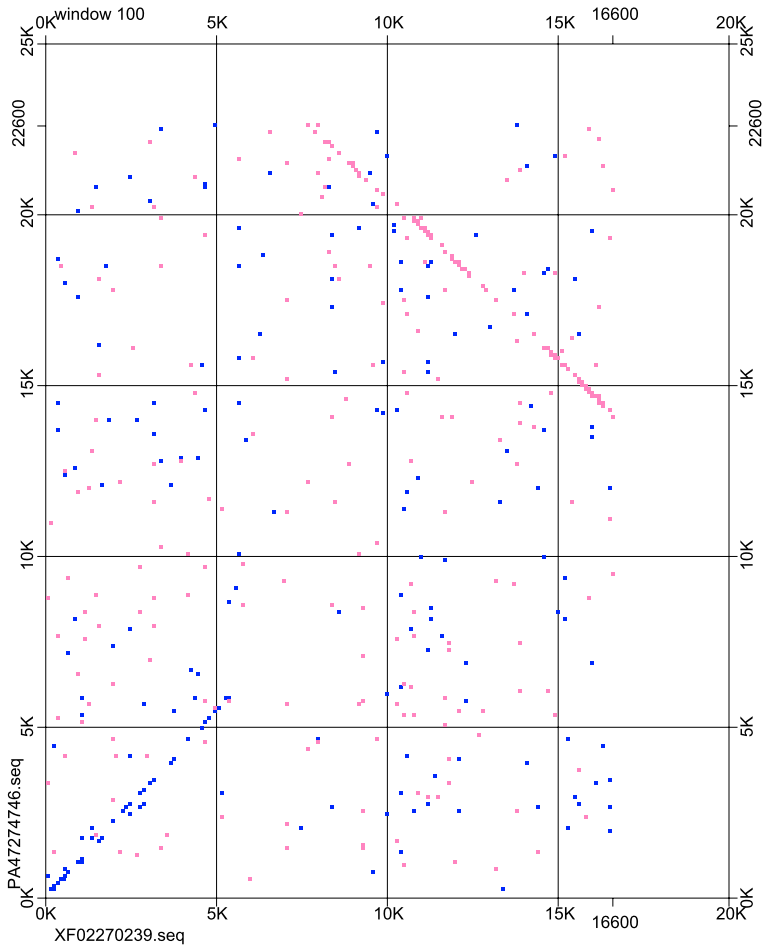

FIGURE 7. Some real data for BIM.

4. Gap penalties

Usually, biologists are also interested in penalties for events like insertions and deletions in the same way inversion penalty I was considered before. Many consecutive insertions (deletions) are called a gap and many gap functions (in terms of the number of symbols involved) are considered for a gap penalty and none of them are completely accepted to be realistic. The simplest considered gap function, and a quite satisfactory one, is the linear function. This leads to a *gap penalty* $g \geq 0$ that is subtracted from the weight of the bimatching $\omega(M)$ for every insertion (a symbol t_j that is not matched in s by the bimatching) and for every deletion (a symbol s_i that is not matched in t by the bimatching).

In Algorithms 1 and 2, a gap penalty $g = 0$ was considered. The case $g > 0$ could easily be managed by few simple modifications in Algorithm 1 (one could subtract g at lines 7 and 13), but such approach would make things harder for Algorithm 2. Hopefully, the case $g > 0$ can easily be reduced to the case $g = 0$ by applying Algorithms 1 or 2 for a new weight function φ defined by $\varphi(a, b) = \omega(a, b) + 2g$. In fact, one can easily prove the following lemma

Lemma 4. $\mathrm{BIM}_{\omega,g}(s, t) = \mathrm{BIM}_{\varphi,0}(s, t) - g|s| - g|t|$.

A discussion on local alignments is out of the scope of this paper. Appropriate initialization modifications are enough for dealing with them.

5. Conclusion and open problems

We gave new algorithms for the alignment with non-overlapping inversions problem, improving the time complexity of an exact solution from $O(n^6)$ to $O(n^4)$ in Algorithm 1. In Algorithm 2, we also gave a sparse dynamic programming implementation that gives the exact solution and can be speeded up even further if we have $o(n^2)$ matches. This is quite common for applications with large alphabets. Both algorithms can deal with linear gap functions as we have seen and very small modifications need to be introduced in order to manage local alignments with non-overlapping inversions.

Let $\mathrm{LCS}(u, v)$ denote the length of the longest common subsequence of u and v. Motivated by these algorithms, one of the authors recently [5] proposed the following open problem: given two words s and t of length n, one can pre-process both words in such a way that any query $\mathrm{LCS}(u, v)$ can be answered in time $O(1)$, for u a factor of s and v a factor of t. This pre-processing can be done in $O(n^4)$. Can we do it in $O(n^2)$? In $O(n^3)$? Although a little stronger, one can think of a version with $\mathrm{DM}(u, v)$ queries instead of $\mathrm{LCS}(u, v)$ queries.

As far as we know, alignment with non-restricted inversions is an open problem.

Acknowledgements. We would like to thank Joachim Messing, Marie-France Sagot and Augusto Fernandes Vellozo for their comments. This work was partially supported by FAPESP # 2000/08039-3, Project PRONEX-FAPESP # 03/09925-5, Alfred P. Sloan Foundation # 99-10-8, NSF DBI 99-82983, NSF # 9975618, and Rutgers SROA #2-7472.

References

[1] D.A. Bader, B.M.E. Moret and M. Yan, A linear-time algorithm for computing inversion distance between signed permutations with an experimental study, in *Algorithms and data structures (Providence, RI, 2001)*, *Lect. Notes Comput. Sci.* **2125** (2001) 365–376.

[2] A. Caprara, Sorting permutations by reversals and Eulerian cycle decompositions. *SIAM J. Discrete Math.* **12** (1999) 91–110 (electronic).

[3] D.A. Christie, A 3/2-approximation algorithm for sorting by reversals, in *Proc. of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (San Francisco, CA, 1998)*. ACM, New York (1998) 244–252.

[4] A.P. do Lago, C.A. Kulikowski, E. Linton, J. Messing and I. Muchnik, Comparative genomics: simultaneous identification of conserved regions and their rearrangements through global optimization, in *The Second University of Sao Paulo/Rutgers University Biotechnology Conference*, Rutgers University Inn and Conference Center, New Brunswick, NJ (August 2001).

[5] A.P. do Lago, *A sparse dynamic programming algorithm for alignment with inversions*. Workshop on Combinatorics, Algorithms, and Applications (2003).

[6] A.P. do Lago, I. Muchnik and C. Kulikowski, An $o(n^4)$ algorithm for alignment with non-overlapping inversions, in *Second Brazilian Workshop on Bioinformatics, WOB 2003*, Macaé, RJ, Brazil (2003) `http://www.ime.usp.br/~alair/woob03.pdf`

[7] N. El-Mabrouk, Genome rearrangement by reversals and insertions/deletions of contiguous segments, in *Combinatorial pattern matching (Montreal, QC, 2000). Lect. Notes Comput. Sci.* **1848** (2000) 222–234.

[8] N. El-Mabrouk, Sorting signed permutations by reversals and insertions/deletions of contiguous segments. *J. Discrete Algorithms* **1** (2000) 105–121.

[9] N. El-Mabrouk, Reconstructing an ancestral genome using minimum segments duplications and reversals. *J. Comput. Syst. Sci.* **65** (2002) 442–464. Special issue on computational biology (2002).

[10] D. Eppstein, Z. Galil, R. Giancarlo and G.F. Italiano, Sparse dynamic programming. I. Linear cost functions. *J. Assoc. Comput. Mach.* **39** (1992) 519–545.

[11] D. Eppstein, Z. Galil, R. Giancarlo and G.F. Italiano, Sparse dynamic programming. II. Convex and concave cost functions. *J. Assoc. Comput. Mach.* **39** (1992) 546–567.

[12] Y. Gao, J. Wu, R. Niewiadomski1, Y. Wang, Z.-Z. Chen and G. Lin, A space efficient algorithm for sequence alignment with inversions, in *Computing and Combinatorics, 9th Annual International Conference, COCOON 2003. Lect. Notes Comput. Sci.* **2697** 57–67 (2003).

[13] S. Hannenhalli and P.A. Pevzner, Transforming cabbage into turnip: polynomial algorithm for sorting signed permutations by reversals, in *ACM Symposium on Theory of Computing*, Association for Computing Machinery (1995) 178–189.

[14] S. Hannenhalli and P.A. Pevzner, Transforming cabbage into turnip: polynomial algorithm for sorting signed permutations by reversals. *J. ACM* **46** (1999) 1–27.

[15] J.W. Hunt and T.G. Szymanski, A fast algorithm for computing longest common subsequences. *Comm. ACM* **20** (1997) 350–353.

[16] H. Kaplan, R. Shamir and R.E. Tarjan, Faster and simpler algorithm for sorting signed permutations by reversals, in *Proc. of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (New Orleans, LA, 1997)*, ACM, New York (1997) 344–351.

[17] H. Kaplan, R. Shamir and R.E. Tarjan, A faster and simpler algorithm for sorting signed permutations by reversals. *SIAM J. Comput.* **29** (2000) 880–892 (electronic).

[18] J. Kececioglu and D. Sankoff, Exact and approximation algorithms for sorting by reversals, with application to genome rearrangement. *Algorithmica* **13** (1995) 180–210.

[19] I. Muchnik, A. Pereira do Lago, V. Llaca, E. Linton, C.A. Kulikowski and J. Messing, Assignment-like optimization on bipartite graphs with ordered nodes as an approach to the analysis of comparative genomic data. *DIMACS Workshop on Whole Genome Comparison* (2001) `http://dimacs.rutgers.edu/Workshops/WholeGenome/`

[20] M. Schöniger and M.S. Waterman, A local algorithm for DNA sequence alignment with inversions. *Bull. Math. Biology* **54** (Jul. 1992) 521–536.

[21] I. Simon, Sequence comparison: some theory and some practice, in *Electronic Dictionaries and Automata in Computational Linguistics*, edited by M. Gross and D. Perrin. Springer-Verlag. Berlin, *Lect. Notes Comput. Sci.* **377** (1989) 79–92.

[22] R. Wagner, On the complexity of the extended string-to-string correction problem, in *Seventh ACM Symposium on the Theory of Computation*. Association for Computing Machinery (1975).

[23] Waterman and Eggert, A new algorithm for best subsequence alignments with application to tRNA-rRNA comparisons. *J. Molecular Biology* **197** (1987) 723–728.