The Minisatellite Transformation Problem Revisited: A Run Length Encoded Approach

Behshad Behzadi and Jean-Marc Steyaert

LIX, Ecole Polytechnique, Palaiseau cedex 91128, France
{Behzadi,Steyaert}@lix.polytechnique.fr

Abstract. In this paper we present a more efficient algorithm for comparison of minisatellites which has complexity $O(n'^3 + m'^3 + mn'^2 + nm'^2 + mn)$ where n and m are the lengths of the maps and n' and m' are the sizes of run-length encoded maps. We show that this algorithm makes a significant improvement for the real biological data, dividing the computing time by a factor 30 on a significant set of data.

1 Introduction

Comparing sequences is a long-addressed problem in computer science as well as in biology. Numerous algorithms have been designed starting from `diff` in Unix and ending (for the moment) at the subquadratic algorithm of Crochemore et al. (see [13]). Our interest in this paper is devoted to a structured comparison of sequences when complex operations can be used to transform strings. These notions intervene naturally in the algorithmic study and treatment of minisatellites – a very important concept in biology. These genomic subsequences are commonly used to understand the dynamic of mutations in particular for interallelic gene conversion-like processes at autosomal loci [8, 9]. Jobling et al. [7] have characterized the Y-specific locus MSY1, a haploid minisatellite, which is composed of 48 to 114 copies of a repeat unit of length 25, rich in AT and predicted to form stable hairpin structures. These sequences are of great interest since they constitute markers for Y chromosome diversity: therefore they allow to trace male descendence proximity in populations.

Modelling minisatellite evolution is therefore necessary in order to provide biologists with a rigorous tool for comparing sequences and establishing likely conclusions as to their proximity. Bérard and Rivals [6] have proposed a combinatorial algorithm to solve the edit distance problem for minisatellites: they considered the five operations – amplification, contraction, mutation, insertion, deletion – with symmetric costs for each type of operation and designed an $O(n^4)$ algorithm. We showed in [4] that it is possible to take into account the generalized cost model, and we designed an algorithm which runs in time $O(n^3)$, thus being more efficient even in a more involved context. In this paper we propose a new enhancement of the method, by making use of a renewed vision based on the run-length encoding.

A string s is called *run length encoded* if it is described as an ordered sequence of pairs (x, i), often denoted x^i, where x is an alphabet symbol and i is an integer.

I. Jonassen and J. Kim (Eds.): WABI 2004, LNBI 3240, pp. 290–301, 2004.

Each pair corresponds to a *run* in s consisting of i consecutive occurrences of x. For example, the string *aaaabbbbcccabbbbcc* is encoded as $a^4b^4c^3a^1b^4c^2$. Letters of two consecutive runs are different. Such a run-length encoded string can be significantly shorter than the standard string representation. Run-length encoding is a usual image compression technique, since many images typically contain large runs of identically-valued pixels. Among biological sequences, minisatellites are ideal ones for using the run-length encoded technique, because basically they consist of a large number of tandem repeats.

Different algorithms have been developed for comparing run-length encoded (RLE) strings. Bunke and Csirik [3] as well as Apostolico, Landau, and Skiena [1] present algorithms for the LCS problem on RLE strings. Mäkinen, Navarro and Ukkonen in [5], Arbell, Landau and Mitchell in [2] and Crochemore, Landau and Ziv-Ukelson in [13] presented algorithms for edit distance of RLE strings.

In this paper, we extend these algorithms and propose an algorithm for computing the transformation distance between two RLE minisatellite maps. The framework we propose has its full generality; operation costs can be almost arbitrary with the only feature that amplifications and contractions are of low cost.

In Section 2, we describe the mathematical model for the minisatellite evolution, and we state the problem in its general form.

In Section 3, we state different lemmas which are essential to prove the correctness of our algorithm.

In Section 4, we show how our method can be adapted for the simplest transformation distance using the arch concept developed by Bérard and Rivals [6].

Section 5 is dedicated to the algorithm. It consists of two parts: Preprocessing and the Core algorithm both of which use the dynamic programming method.

In section 6, we discuss about the performance of the new algorithm compared with the previous ones on randomly generated data and real biological data. We show that our new algorithm works much faster on the real minisatellite data.

2 Model Description

The symbols are elements from a finite alphabet Σ. We will use the letters x, y, z,... for the symbols in Σ and s, t, u, v,... for strings[1] over Σ. The empty string is denoted by ϵ. We will denote by $s[i]$ the symbol in position i of the string s (the first symbol of a string s is $s[1]$). The substring of s starting at position i and ending at position j is denoted by $s[i..j] = s[i]s[i+1]\dots s[j]$. A substring $s[i..j]$ is called a *run* if $s[i-1] \neq s[i] = s[i+1] = s[i+2]...s[j-1] = s[j] \neq s[j+1]$. A string obtained by replacing each of the runs of string s by a letter of that run is called the *compact* representation of s and is denoted by s'.

[1] Throughout the paper we use the word string to designate what biologists call sequences or maps [6]. The word sequence will refer to a sequence of operations on a string.

In the evolutionary model, five elementary operations are considered on strings. These operations are mutation (replacement), insertion, deletion, amplification and contraction. The first three are the well-known string edit distance operations (see for example [12]). The last two are new operations which are significant in the study of the evolution of minisatellite strings and more generally whenever large substrings in the genome are multiply repeated or deleted. Amplification of a symbol x in a string s amounts to repeating this symbol after one of its occurrences in s. A p-plication of a symbol x which is an amplification of *order* p amounts to $p-1$ times repeat symbol x *after* the initial symbol x. Conversely, the p-contraction of a symbol x means to delete $p-1$ consecutive symbols x provided that the symbol just *before* them is also an x. Given two strings s and t, there are infinitely many sequences of elementary operations which transform the string s into the string t. Among this infinity, some evolution sequences are more likely; in order to identify them, we introduce a cost function for each elementary operation depending on the symbols involved in the operation: $I(x)$ and $D(x)$ are the costs of an insertion or a deletion of symbol x. $M(x,y)$ is the cost of the replacement of symbol x by symbol y in the string. For $p>1$, $A_p(x)$ is the cost of a p-plication of symbol x in the string and finally $C_p(x)$ is the cost of a p-contraction of a symbol x. In this paper we consider only the amplifications (and contractions) of order 2. Whenever we use the terms amplification and contraction, we mean duplication and 2-contraction. Note that the costs can be non symmetric ($I(x)$ may be different from $I(y)$, etc.). We suppose that the mutation cost function satisfies the triangle inequality property: $M(x,y)+M(y,z) \geq M(x,z)$ for all different x,y,z in $\Sigma \cup \{\epsilon\}$. In addition, $M(x,x)=0$ for any symbol x and all other values of all of the cost functions are strictly greater than zero. These hypotheses do not reduce the generality of our statements. The main hypothesis to consider in comparison of minisatellites is the following:

Hypothesis 1 *The cost of duplications (and contractions) is less than the cost of all other operations.*

A *transformation* of s into t amounts to applying a sequence of operations on s transforming it into t. When s is transformed into t by a sequence of operations we write by $s \xrightarrow{*} t$ and when s is transformed into t in one elementary operation we use the notation $s \rightarrow t$. The cost of a transformation is the sum of the costs of its operations. The *transformation distance* from s into t is the minimum cost for a possible transformation from s into t. The transformation which gives this minimum is called *optimal transformation* (it is not necessarily unique). Our objective in this paper is to find this distance between two strings and one of their optimal transformations. In the next section we will study the optimal transformation properties.

It will be convenient to add an extra special symbol \$ to the alphabet and to consider that the value of all the functions with \$ as one of their variables is ∞ (with exception of $M(\$,\$)=0$). Whenever we are asked to find the transformation distance between strings s and t, we will compute the optimal transformation of $\$s$ into $\$t$. By our assumption these two values are equal. This is a

way to forbid any insertion (and deletion) at the beginning of strings. (So from now without loss of generality we suppose that the insertions (and deletions) are allowed *only after* symbols.)

3 Optimal Transformation Properties

A transformation applies a sequence of operations on a string s and the result will be a string t. This sequence is called *transformation sequence* from s into t. In a transformation of s into t, each symbol of s, generates a substring of t: this substring can be the empty string. The symbols of s which generate a non-empty substring of t are called *generating symbols* and the other symbols are called *vanishing symbols*.

Now consider the transformation of one symbol x to a non-empty string s: This transformation is called *generation*. The generations which use only mutations, amplifications and insertions are called non-decreasing generations. A non-decreasing generation can be represented by a tree. The tree construction rules are the following:
1) The root of the tree has label x.
2) For any duplication of a symbol y, add two new nodes with label y as children of that node.
3) The insertion of a letter z after a symbol y is shown by adding two children to the corresponding node y which have labels y and z from left to right.
4) The mutation of a symbol y into z is represented by a single child with label z for the node with label y.

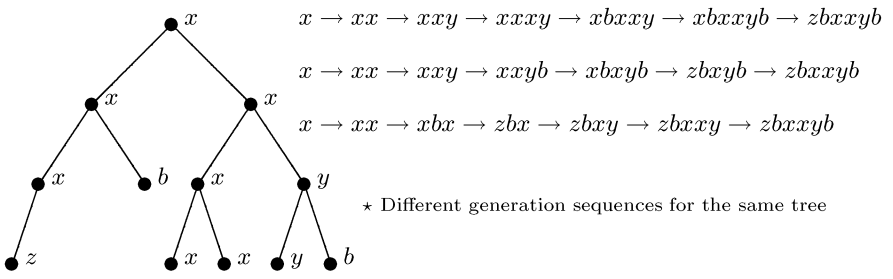

Fig. 1. The tree representation of a non-decreasing generation.

Each internal node in the tree corresponds to an operation. As shown in the Figure 1, different generation sequences can have the same tree representation. They differ by the order of operations but their costs are the same. A distinguished generation sequence that one can construct for a given tree is the sequence which is obtained by a *left depth first search* of the tree (visit the children of a node from left to right). This sequence is called *left-to-right* generation

sequence. We recall two lemmas about the optimal transformation properties from [4]. The proofs are given in [4].

Lemma 1. *(The generation lemma):*
The optimal generation of a non-empty string s from a symbol x can be achieved by a non-decreasing generation.

Lemma 2. *(The independency of contractions)*
There exists an optimal transformation of a string s into string t in which all the contractions are done before all the amplifications.

Symmetrically, the optimal *reduction* of a non-empty string s to a single symbol x can be obtained by using only mutations, deletions and contractions. Now we study the properties of runs in an optimal transformation. The next lemma considers a transformation of a single run string into another single run string.

Lemma 3. *(Transformation of x^k into y^l)*
The cost of the optimal transformation of the string $s = x^k$ into $t = y^l$ is:
For $k \leq l$:
(1) $(k-1) \times C_2(x) + M(x,y) + (l-1) \times A_2(y)$ *if* $M(x,y) \geq A_2(y) + C_2(x)$
(2) $k \times M(x,y) + (l-k) \times A_2(y)$ *if* $M(x,y) < A_2(y) + C_2(x)$
For $k > l$:
(3) $(k-1) \times C_2(x) + M(x,y) + (l-1) \times A_2(y)$ *if* $M(x,y) \geq A_2(y) + C_2(x)$
(4) $k \times M(x,y) + (k-l) \times C_2(x)$ *if* $M(x,y) < A_2(y) + C_2(x)$

Proof: Let u be the number of x's which are mutated into y. Then $k-u$ contractions on $s = x^k$ are necessary to delete the extra symbols. If $l \geq k$ then $l-u$ duplications should be applied after the mutations. The total cost can be expressed as a function of u: $f(u) = u \times M(x,y) + (l-u) \times A_2(y) + (k-u) \times C_2(x)$. This linear function of u is minimized at $u = 1$ or $u = k$ depending on the sign of $M(x,y) - A_2(y) - C_2(x)$. These two possible cases for the minimum are the expressions (1) and (2) of the lemma. A similar proof can be considered in the case $l < k$ for the expressions (3) and (4). Note that $M(x,y) = 0$ if $x = y$. □

Hypothesis 1 leads us to the following fact:

Fact 1 *There exists an optimal generation of a non-empty string t from a single symbol x in which for every run of size $k > 1$ in t, the $k-1$ right symbols of the run are generated by duplications of the leftmost symbol of the run.*

Lemma 4. *(First run lemma)*
There exists an optimal transformation of string s into string t in which for any generating symbol x the following is true: If x generates $t[i..j]$ and $t[i]$ is not the first symbol of a run in t then the first run in $t[i..j]$ has length one.

Proof: Consider an optimal transformation of s into t. Let x be the rightmost symbol which violates the statement of the lemma. This means that the first run in $t[i..j]$ generated by x is a run of length at least two and $t[i-1] = t[i]$. If we move $t[i]$ from this generated substring to the substring immediately generated

at its left, the total cost remains the same (Fact 1). By iterating this operation the first run generated by x will have length one. By iterating the whole procedure all the symbols will satisfy the lemma statement. □

Lemma 4 is important for the design of our algorithm. In fact in an optimal transformation of string s into string t, if $s[n]$ is a generating symbol it will generate a suffix of t and there are $O(m')$ candidate suffixes in t (and not $O(m)$) for this generation.

Lemma 5. *(Generating and vanishing symbols of a run)*
There exists an optimal transformation of s to t such that for any run in s, all the vanishing symbols are at the right of all the generating symbols.

Proof: Consider a run of size $k = k_v + k_g$ such that k_v symbols are vanishing and k_g symbols are generating. If $k_g = 0$ the statement of the lemma is correct. If $k_g > 0$, the reduction of k_v symbols costs minimum when they have an identical symbol at their left because all these symbols can use contractions for their deletions. □

We call a substring of s a *vanishing substring* if all the symbols of the substring are vanishing symbols. A vanishing substring is maximal if there is no other vanishing symbol just before or just after it. By lemma 5, there exists an optimal transformation of s into t in which the last symbol of all maximal vanishing substrings of s is the last (rightmost) symbol of some run in s. This is an important fact for the design of our algorithm. In a transformation of string s (with compact form s') into string t, the maximum number of maximal vanishing substrings is not more than $n' = |s'|$ and the rightmost symbol of any of these vanishing substring is one of the n' special positions in s.

Consider a maximal vanishing substring $s[i..j]$ and let i' be the smallest number in the interval $[i, j]$ which is a first symbol of a run in s if such a number exists. If i' exists the evolution can be considered as two reductions: First $s[(i'-1)..j]$ is reduced into $s[i'-1]$ and then $s[(i-1)..(i'-1)]$ is reduced into $s[i-1]$. The rightmost symbol of both of these reductions is a rightmost symbol of some run in s, so the whole number of these reductions in a transformation is $O(m')$. As a conclusion we have the following lemma.

Lemma 6. *(Reduction types)*
There exists an optimal transformation of s into t, in which the last symbol of any reduced substring (reduction) is the last symbol of a run in s and the reduction is one of the two following types:
(a) Some complete runs of s vanish.
(b) A suffix of one of the runs of s vanishes.

4 Arch Representation

Bérard and Rivals [6] use the notion of arches in order to represent the duplications (and contractions). For a given string s an arch is identified by a pair of

integers (i, j) such that $i < j$ and $s[i] = s[j]$. Two arches are called *compatible* if both of the duplications (or contractions) can happened together in an evolution history. Formally, two arches (i, j) and (i', j') where $i \leq i'$ are *incompatible* if $i < i' < j$ and $j' \geq j$. Two arches are compatible if they are not incompatible (Figure 2).

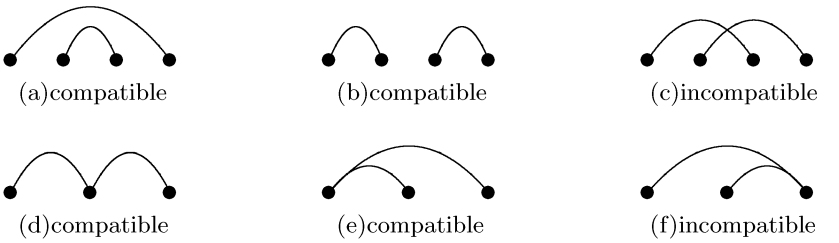

Fig. 2. Different cases for compatible and incompatible arches.

In their simple model, where the cost of operations does not depend on the symbols, they show that the minimum generation cost of a substring can be computed from the maximum number of compatible arches in the substring. The following proposition shows how one can compute the maximum number of the compatible arches faster using the runs idea:

Proposition 1 *Let s' be the string obtained by replacing each run of the string s by a single letter. The maximum number of compatible arches in s equals the maximum number of compatible arches in s' plus the difference of lengths of the strings s and s'.*

Proof: Let n and n' be the lengths of the strings s and s' respectively. The proof consists of two parts.

Let k' be the maximum number of compatible arches in s'. Let $\mathcal{A}'$ be a set of k' compatible arches in s'. We construct a set $\mathcal{A}$ of arches in s in the following way: For each pair (i', j') in $\mathcal{A}'$, add an arch connecting the last symbol of the i'-th run to the first symbol of the j'-th run (*inter-run arches*). Add all arches of the form $(i, i+1)$ into $\mathcal{A}$ (*in-run arches*). The number of in-run arches is $n - n'$. All of the arches in $\mathcal{A}$ are compatible; hence we have at least $k = k' + (n - n')$ compatible arches in s.

Let k be the maximum number of compatible arches in s and $\mathcal{A}$ be a set of k compatible arches in s. If the number of in-run arches of form $(i, i+1)$ in $\mathcal{A}$ is less than $n - n'$, then there exists an in-run arch of form $(i, i+1)$ which is not included in $\mathcal{A}$. Let $(i, i+1)$ be the leftmost such arch. By definition of the incompatible arches, in $\mathcal{A}$ there is at most one arch with $i+1$ as the right endpoint (Figure 2(f)). So there is at most one arch in $\mathcal{A}$ incompatible with $(i, i+1)$. By maximality of $\mathcal{A}$, there is exactly one arch in $\mathcal{A}$ incompatible with $(i, i+1)$. We replace this arch by $(i, i+1)$ in $\mathcal{A}$. By iterating the same procedure, we arrive to a set of k compatible arches such that $n - n'$ are in-run arches of

form $(i, i+1)$. Since all the possible in-run arches of form $(i, i+1)$ are present in this set, there is at most one inter-run arch between two given runs. If we reduce every run to a single symbol and keep only the inter-run arches, we obtain a set of $k' = k-(n-n')$ compatible arches for s'. This completes the proof.□

As a result of this proposition, instead of computing the maximum number of compatible arches in a string s, one can compute the maximum number of compatible arches in s' (which is eventually much smaller than s) and add $n-n'$ to the computed value. Let $MA[i,j]$ be the maximum number of compatible arches in substring $s'[i..j]$. The simple dynamic programming algorithm given in Figure 3 computes the maximum number of compatible arches for all substrings of s'. The proof is easy and is omitted. The time complexity is $O({n'}^3)$.

Algorithm 1 Compatible Arches

If $i \geq j$ **Then** $MA[i,j] = 0$

Else $MA[i,j] = \max \begin{cases} MA[i+1,j] \\ MA[i,k-1] + MA[k,j] + 1 \quad \textit{if } s'[i] = s'[k],\ \forall i < k \leq j \end{cases}$

Fig. 3. Recurrence relations for Maximum Compatible arches calculation.

5 The Algorithm

In this section we describe an algorithm to compute the transformation distance from a string s into a string t. Firstly, in the preprocessing part we determine the cost of generation of substrings of the target string t from any given symbol in the alphabet. Then we will compute the transformation distance from s to t by applying a dynamic programming algorithm using the preprocessing results.

5.1 Pre-processing

Let $t[i..j]$ be the string generated by a symbol x. The compact representation of this substring is $(t[i..j])'$. By Fact 1 and the proof of Proposition 1, the optimal cost of $x \xrightarrow{*} t[i..j]$ can be obtained by adding all duplication costs of two identical neighbor symbols in $t[i..j]$ into the optimal cost of the generation $x \xrightarrow{*} (t[i..j])'$. Let $b(i)$ denotes the number of the runs including $t[i]$. $NDup[i]$ is defined as below:

$$NDup[1] = 0 \text{ and } \forall\, 1 < i \;\; NDup[i] = NDup[i-1] + \begin{cases} 0 & \text{if } t[i] \neq t[i-1] \\ A_2(t[i]) & \text{if } t[i] = t[i-1] \end{cases}$$

If t' is the compact representation of t, the minimum generation cost of $t[i..j]$ from a symbol x denoted by $G_t(x,i,j)$ is equal to $(NDup[j] - NDup[i]) + G_{t'}[x, b[i], b[j]]$. The table $G_{t'}$ is computed by dynamic programming algorithm

given in Figure 4. This algorithm is a simplified version of the pre-processing algorithm given in [4]. The proof is identical to the proof in [4].
Symmetrically to $G_t(x,i,j)$, we compute $K_s(x,i,j)$ which is the minimum cost of reduction of $s[i..j]$ into the symbol x. The complexity of the preprocessing part for a fixed alphabet is $O(n+m+{n'}^3+{m'}^3)$ in time and $O(m+n+{n'}^2+{m'}^2)$ in space.

Algorithm 2 Generation_Costs

Initialization:
$\forall\, 0 < i \leq m',\ \forall x \in \Sigma : G_{t'}[x,i,i] = M(x,t'[i])$

Recurrence:
$\forall i < j\ \forall x \in \Sigma :$

1) $T[x,i,j] = \min \begin{cases} A_2(x) + \min_{i<k\leq j}\{G_{t'}[x,i,k-1] + G_{t'}[x,k,j]\} \\ G_{t'}[y,i+1,j] + I(y) \qquad\qquad \text{if } t'[i] = x,\ \forall y \in \Sigma \end{cases}$

2) $G_{t'}[x,i,j] = \min_{y\in\Sigma}\{T[y,i,j] + M(x,y)\}$

Fig. 4. Recurrence relations for generation costs.

5.2 Core Algorithm: Dynamic Programming Algorithm

An optimal transformation which satisfies the lemmas 4, 5 and 6 is called a *good optimal transformation*. Dynamic programming is the tool to find a good optimal transformation of string s into string t. Let $TD[i..j]$ be the minimum cost for a transformation of $s[1..i]$ into $t[1..j]$ which can be extended into a good transformation of $s[1..n]$ into $t[1..m]$; n' and m' denote the number of runs in strings s and t respectively. Let $e_s(k)$ be the position of the last element of the k-th run in s for any $1 \leq k \leq n'$. E_s is the set of all values of $e_s(k)$ for $1 \leq k \leq n'$. For each $i \leq n$, $e_s^*(i)$ is defined as the largest $e_s(k) < i$ for $k \geq 1$ if such a k exists and 0 otherwise; $e_t(l)$, E_t and $e_t^*(j)$ are defined similarly for $1 \leq l \leq m'$ and $1 \leq j \leq m$. All these functions can be computed in linear time, before the execution of the core algorithm. The core algorithm is given in Figure 5.

Proposition 2 *The Algorithm given in Figure 5 determines correctly the transformation distance of s into t.*

Proof Sketch: Let us explain how the recurrence relations given in Figure 5 determine the cost of a good optimal transformation. The first lines correspond to the fact there is a special symbol \$ in the head of the strings. We have four different cases for the recurrence relation. Each case corresponds to different positions of the current indices w.r.t a run in s and t; then we apply repeatedly Lemmas 4, 5 and 6 to the situation and evaluate systematically the costs of all possible analyses that can be done. □

Algorithm 3 Transformation_Distance(S,T)

$TD[0,0] = 0$

For $i \leftarrow 1$ **to** n **do** $TD[i,0] = \infty$

For $j \leftarrow 1$ **to** m **do** $TD[0,j] = \infty$

For $i \leftarrow 1$ **to** n **do**

For $j \leftarrow 1$ **to** m **do**

$$TD[i,j] = \begin{cases} \textbf{A) if } i \notin E_s \textbf{ and } j \in E_t \\ \min\{TD[i\text{-}1,l] + G_t(s[i], l+1, j)\}_{l \le j:\ l \in E_t \text{ or } l+1 \in E_t \text{ or } l=j-1} \\ \\ \textbf{B) if } (i \notin E_s \textbf{ and } i\text{-}1 \notin E_s) \textbf{ and } j \notin E_t \\ \min\{TD[i\text{-}1, j\text{-}1] + G_t(s[i], j, j),\ TD[i\text{-}1, e_t^*(j)] + G_t(s[i], e_t^*(j)+1, j)\} \\ \\ \textbf{C) if } (i \in E_s \textbf{ or } i\text{-}1 \in E_s) \textbf{ and } j \notin E_t \\ \quad \min \begin{cases} TD[i\text{-}1, j\text{-}1] + G_t(s[i], j, j) \\ TD[i\text{-}1, e_t^*(j)] + G_t(s[i], e_t^*(j)+1, j) \\ \{TD[i\text{-}1,l] + G_t(s[i], l+1, j)\}_{l \le j:\ l \in E_t \text{ or } l+1 \in E_t} \quad \textbf{if } i\text{-}1 \in E_s \\ \{TD[k,j] + K_s(s[k], k, i)\}_{k<i:\ k \in E_s \text{ or } k > e_s^*(i)} \end{cases} \\ \\ \textbf{D) if } i \in E_s \textbf{ and } j \in E_t \\ \quad \min \begin{cases} \{TD[i\text{-}1,l] + G_t(s[i], l+1, j)\}_{l \le j:\ l \in E_t \text{ or } l+1 \in E_t \text{ or } l=j-1} \\ \{TD[k,j] + K_s(s[k], k, i)\}_{k<i:\ k \in E_s \text{ or } k > e_s^*(i)} \end{cases} \end{cases}$$

Fig. 5. Recurrence relations for Transformation Distance.

The complexity of the algorithm is $O(mn + mn'^2 + nm'^2)$ in time and $O(mn)$ in space. Analysis of the complexity can be done by computing the total complexity of each single line in the recurrence relation separately and then adding the results.

6 Discussion

In this section, we compare the running time of the algorithms presented in this paper with the algorithms presented in [4] and [6]. For this aim we firstly compare the running time on randomly generated strings on an alphabet Σ. In a second part we run the three algorithms on a real biological database of minisatallites.

Let us just remark that the number of runs R in a random sequence of length n with equally distributed letters follows a Bernoulli law: $P\{R = k\} = \binom{n-1}{k-1}(\frac{|\Sigma|-1}{|\Sigma|})^{k-1}(\frac{1}{|\Sigma|})^{n-k}$. The mathematical expectation of the number of runs for a string of size n is $1 + (n-1)\frac{|\Sigma|-1}{|\Sigma|}$.

The number of runs of a string which is generated randomly on an alphabet Σ grows linearly with n. The compact representations of randomly generated strings have size proportional to n on the average. We observe on biological minisattelite samples that the compact representation is much shorter than the original string.

Type 1: CACAATATACATGATGTATATTATA
Type 2: CATAATATACATGATGTATATTATA
Type 3: CACAATATACATCATGTATATTATA
Type 4: CATAATATACATCATGTATATTATA
Type 5: CATAATATACATGATGTATAATATA

0	1	1	2	2
1	0	2	1	1
1	2	0	1	3
2	1	1	0	2
2	1	3	2	0

Fig. 6. Five variant MSY1 repeats identified and a simple mutation table on them based on the number of different nucleotides.

Table 1. Running times of different algorithms on random and biological datasets.

Algorithm	Random sequences		minisatellites data	
	PreProcessing	Core	PreProcessing	Core
Bérard & Rivals(2002)	15.90 sec	1058.44 sec	16.37 sec	1062.23 sec
Behzadi & Steyaert(2003)	2.51 sec	1014.32 sec	2.49 sec	1012.38 sec
This paper	2.14 sec	810.93 sec	0.03 sec	32.54 sec

The MSY1 repeats are AT rich (75%-80%) sequences. Five variant repeats designated were identified. Each of these repeats contains 25 bp. Repeat types 1-4 differ at two sites, a C/T transition at position 3, and a C/G transition at position 13. The single type 5 repeat, differs from the type 2 repeat by a transition at position 21.

We used a dataset provided by M. Jobling in which minisattelite maps for 690 Y chromosome from worldwide population samples were determined. The length of each of these 690 sequences is between 48 and 118. We compute the transformation distances between each pair of these strings by the three mentioned algorithms. The running times are given in table 1. The PreProcessing part is executed once for each of these strings and the core algorithm is considered for any pairs of these strings (690×690 pairs). The given times correspond to the total time needed for all these computations. Note that the random sequences have the same lengths as the sequences in the database and the alphabet is the same.

7 Conclusion

We have also considered a model in which we have amplifications (and contractions) of order greater than two. The results can be easily generalized.

As a final remark, let us just point out again, that the minisatellite problem is an instance of the general problem of transforming a chain into another and the framework is now at its maximum generality.

References

1. Apostolico, A., Landau, G.M. and Skiena, S.: Matching for Run Length Encoded Strings. Journal of Complexity, 15, 1, 4-16 (1999).
2. Arbell, O., Landau, G.M., Mitchell, J.S.B: Edit Distance of Run-Length Encoded Strings. Information Processing Letter, 83(6), 307-314, 2002.
3. Bunke, H. and Csirik, J.: An Improved Algorithm for Computing the Edit Distance of Run Length Coded Strings. Information Processing Letters, 54, 93-96 (1995).
4. Behzadi B. and Steyaert J.-M.: An Improved Algorithm for Generalized Comparison of Minisatellites. Proc. of 14th CPM. Lecture Notes in Computer Science (2003).
5. Mäkinen, V., Navarro, G., Ukkonen, E.: Approximate Matching of Run-Length Compressed Strings. Proc. of 12th CPM, Lecutre Notes in Computer Science 2089, Springer-Verlag, 31-49 (2001).
6. Bérard, S., Rivals, E.: Comparison of Minisatellites. Proceedings of the 6th Annual International Conference on Research in Computational Molecular Biology. ACM Press, (2002).
7. Jobling, M.A., Bouzekri, N., Taylor, P.G.: Hypervariable digital DNA codes for human paternal lineages: MVR-PCR at the Y-specific minisatellite, MSY1(DYF155S1). Human Molecular Genetics, Vol. 7,No. 4. (1998)643–653.
8. Bouzekri, N., Taylor, P.G., Hammer M.F, Jobling, M.A.: Novel mutation processes in the evolution of haploid minisatellites, MSY1: array homogenization without homogenization. Human Molecular Genetics,Vol. 7, No. 4. (1998)655–659
9. Jeffreys, A.J., Tamaki, K., Macleod, A., Monckton, D.G., Neil, D.L and Armour, J.A.L: Complex gene conversion events in germline mutation at human minisatellites. Nature Genetics, 6. (1994)136–145.
10. Brión, M., Cao, R., Salas, A., Lareu M.V., Carracedo A.: New Method to Measure Minisatellite Variant Repeat Variation in Population Genetic Studies. American Journal of Human Biology, Vol. 14.(2002) 421–428.
11. Elemento, O., Gascuel, O., Lefranc, M.-P.: Reconstructing the duplication history of tandemly repeated genes. Molecular Biology and Evolution,vol 19(3). (2002) 278–288.
12. Sankoff, D. and Kruskal, J.B: Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison. Addison-Wesley. (1983).
13. Crochemore, M., Landau, G. M., Ziv-Ukelson, M.: A sub-quadratic sequence alignment algorithm for unrestricted cost matrices. SODA'2002. ACM-SIAM. (2002)679–688.