The Transformation Distance Problem Revisited

Behshad Behzadi and Jean-Marc Steyaert

LIX, École Polytechnique
Palaiseau cedex 91128, France

e-mail: {behzadi,steyaert}@lix.polytechnique.fr

Abstract. Evolution acts in several ways on biological sequences: either by mutating an element, or by inserting, deleting or copying a segment of the sequence. Varré et al. [VDR98] defined a transformation distance for the sequences, in which the evolutionary operations are copy, reverse copy and insertion of a segment. They also proposed an algorithm to calculate the transformation distance. This algorithm is $O(n^4)$ in time and $O(n^4)$ in space, where n is the size of the sequences. In this paper, we propose an improved algorithm which costs $O(n^2)$ in time and $O(n^2)$ in space. Furthermore, we extend the operation set by adding point deletions. We present an algorithm which is $O(n^3)$ in time and $O(n^2)$ in space for this extended case.

Keywords: dynamic programming, pattern matching

1 Introduction

Building models and tools to quantify evolution is an important domain of biology. Evolutionary trees or diagrams are based on statistical methods which exploit comparison methods between genomic sequences. Many comparison models have been proposed according to the type of physico-chemical phenomena that underly the evolutionary process [Do81]. Different evolutionary operation sets are studied. Mutation, deletion and insertion were the first operations dealt with [SaKr83]. Duplication and contraction were then added to the operation set [BeRi02, BeSt03]. All these operations were acting on single letters, representing bases, aminoacids or more complex sequences: they are called point transformations. Segment operations are also very important to study. In a number of papers [VDR97, VDR98, VDR99], Varré et al. have studied an evolutionary distance based on the amount of segment moves that Nature needed (or is supposed to have needed) to transfer a sequence from one species to the equivalent sequence in another one. Their model is concerned with segments copy with or without reversal and on segment insertion: it is thus a very simple and robust model which can easily be explained from biological mechanisms. They developed this study on DNA sequences, but the basic concepts and algorithms apply as well to proteins or satellites.

The algorithm they propose to compute the minimal transformation sequence is based on an encoding into a graph formalism, from which one can get the solution by computing shortest paths. This gives an $O(n^4)$ answer both in space and time[1].

[1] Even $O(n^6)$ in the last french version [Va00].

In fact it is possible to give a direct solution based on dynamic programming which costs only $O(n^2)$ in time and space. This solution is obviously more efficient for long sequences and makes the problem tractable even for very long sequences.

In the second section we describe the model and the problem description.

In the third section our algorithm for calculating the transformation distance is presented. Firstly, in the preprocessing part we show how to find efficiently the existence of all the substrings of one string in another one. Then the core of the algorithm is presented, which is basically a dynamic programming algorithm.

In section 4, we introduce the point deletions in our model and we give an algorithm to solve the transformation distance problem in presence of point deletions: this algorithm runs in time $O(n^3)$ and space $O(n^2)$.

Finally, section 5 is dedicated to conclusions and remarks.

2 Model and Problem Description

The symbols are elements from an alphabet Σ. The set of all finite-length strings formed using symbols from alphabet Σ is denoted by Σ^*. In this paper, we use the letters x, y, z,... for the symbols in Σ and S, T, P, R, ... for strings over Σ^*. The empty string is denoted by ϵ. The length of a string S is denoted by $|S|$. The *concatenation* of a string P and R, denoted PR, has length $|P| + |R|$ and consists of the symbols from P followed by the symbols from R.

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 $S[i..j] = S[i]S[i+1]\dots S[j]$. The *reverse* of a string S is denoted by S^{-1}. Thus, if n is the length of S, $S^{-1}[i..j] = S[(n-j+1)..(n-i+1)]^{-1}$ and $S[i..j]^{-1} = S^{-1}[(n-j+1)..(n-i+1)]$. We say that a string P is a *prefix* of a string S, denoted $P \sqsubseteq S$, if $S = PR$ for some string $R \in \Sigma^*$. Similarly, we say that a string P is a *suffix* of a string S, denoted by $P \sqsupseteq S$, if $S = RP$ for some $R \in \Sigma^*$. For brevity of notation, we denote the k-symbol prefix $P[1..k]$ of a string pattern $P[1..m]$ by P_k. Thus, $P_0 = \epsilon$ and $P_m = P = P[1..m]$. We recall the definition of a *subsequence*: Given a string $S[1..n]$, another string $R[1..k]$ is a *subsequence* of S, denoted by $R \prec S$, if there exists a strictly increasing sequence $< i_1, i_2, \dots, i_k >$ of indices of S such that for all $j = 1, 2, \dots, k$, we have $S[i_j] = R[j]$. For example, if $S = xxyzyyzx$, $R = zzxx$ and $P = xxzz$, then P is a subsequence of S, while R is not a subsequence of S. When a string S is a subsequence of a string T, T is called a *supersequence* of S, denoted by $T \succ S$. In the last example, S is a supersequence of P.

Varré et al. [VDR98, VDR99] propose a new measure which evaluates segment-based dissimilarity between two strings: the source string S and the target string T. This measure is related to the process of constructing the target string T with *segment operations*[2]. The construction starts with the empty string ϵ and proceeds from left to right by adding segments (concatenation), one segment per operation. The left-to-right generation is not a restriction but a fact that can be formally proved. A list of operations is called a *script*. Three types of segment operations are considered: the *copy* adds segments that are contained in the source string S, the *reverse copy* adds

[2] In this paper we use segment as an equivalent word for substring.

the segments that are contained in S in reverse order, and the *insertion* adds segments that are not necessarily contained in S. The measure depends on a parameter that is the *Minimum Factor Length (MFL)*; it is the minimum length of the segments that can be copied or reverse copied. Depending on the number of common segments between S and T, there exist several scripts for constructing the target T. Among these scripts, some are more likely; in order to identify them, we introduce a cost function for each operation. *InsertCost*($T[i..j]$) is the cost of insertion of substring $T[i..j]$. *CopyCost*($T[i..j]$) is the cost of copying the segment $T[i..j]$ from S if it is contained in S. Finally *RevCopyCost*($T[i..j]$) is the cost of copying substring $T[i..j]$ from S if the reverse of this substring is contained in the source S. The cost of a script is the sum of the costs of its operations. The *minimal scripts* are all scripts of minimum cost and the *transformation distance*[3] *(TD)* is the cost of a minimal script. The problem which we solve in this paper is the computation of the transformation distance. It is clear that it is also possible to get a minimal script.

3 Algorithm

In this section we describe the algorithm to determine the transformation distance between two strings. The algorithm consists of two parts. The first part is a preprocessing part in which we determine for each substring of target string T, whether it exists in the source string S or not. In the second part, which is the core algorithm, we determine the transformation distance with help of the information that we obtained in the preprocessing part. This core algorithm is a dynamic programming algorithm.

3.1 Preprocessing

Deciding whether a given substring exists in S or not, and finding its position in the case of presence, needs to apply a *string matching* algorithm. For this aim, we design an algorithm based on KMP (Knutt-Moris-Pratt) string matching algorithm with some changes. Let $FP[i,j]$ be the the first position of occurrence of the substring $T[i..j]$ in S if such an occurrence exists and ∞ otherwise. Similarly $FPR[i,j]$ is the first position of an occurrence of $T^{-1}[i..j]$ in S. We need to recall the definition of *prefix function* π (adapted from the original KMP one), which is needed for precomputation. Given a pattern $P[1..m]$, the prefix function for pattern P is the function $\pi : \{1,2,\ldots,m\} \to \{0,1,\ldots,m-1\}$ such that $\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupseteq P_q\}$. That is, π_q is the length of the longest prefix of P that is a proper suffix of P_q. We have the following lemma for the prefix functions.

Lemma 1 The prefix function of P_k is a restriction of prefix function of P to the set $\{1,2,\ldots,k\}$.

Proof: The proof is immediate by the definition of the prefix function because $\pi[i]$ for a given i can be obtained only from $P_{i-1} = P[1..(i-1)]$ and $P[i]$.

Although simple, this lemma is a corner-stone of the algorithm. It shows that, one can search for the presence of the prefixes of a pattern string in the source string, in the

[3]Although this measure is not a mathematical distance but we will use the term transformation distance which was introduced by Varré et al. [VDR98, VDR99].

```
Algorithm 1 Prefix-Matcher(A, S, P, index)   %% index = |T| + 1 − length of the
1.  n ← length[S]                            %%   suffix P being searched in S
2.  m ← length[P]                            %% A_[n×n] : A[i, i+q] ≠ ∞ iff the prefix
3.  q ← 0                                    %%      of P of length q+1 occurs in S
4.  for i ← 1 to n
5.  do  while q > 0 and P[q+1] ≠ S[i]
6.          do q ← π[q]
7.      if P[q+1] = S[i] then
8.          q ← q + 1
9.      if A[index, index+q] = ∞ then
10.         A[index, index+q] = i − q
11.     if q = m then
12.         Exit                             %% the suffix P has been discovered
```

Figure 1: Prefix-Matcher

```
Algorithm 2 PreProcessing(S, T)
1.  FillArray(FP, ∞)
2.  FillArray(FPR, ∞)
3.  n ← length[T]
4.  for k ← 1 to n
5.  do  P ← T[k..n]
6.      Prefix-Matcher(FP, S, P, k)          %% direct pattern
7.      PR ← T^{-1}[k..n]
8.      Prefix-Matcher(FPR, S, PR, k)        %% reverse pattern
```

Figure 2: PreProcessing

same time of searching for the complete pattern, without increasing the complexity of the search. The algorithm is given in pseudocode in figure 1 as the procedure **Prefix-Matcher**. The complexity of the Prefix-Matcher algorithm is $O(n)$ in time. For the proof of the complexity and correctness of this algorithm, see chapter 34.4 of [CLR90]. Prefix-Matcher finds the position of the first occurrence of all prefixes of a pattern string P in string S. In the **PreProcessing** algorithm (figure 2), we call the Prefix-Matcher with patterns $T[1..n], T[2..n], ..., T[n]$. Thus, we have the position of the first occurrences of all of the substrings of T in S. Similarly, the first position of all substrings of T^{-1} are found in S. The total complexity the preprocessing part is $O(n^2)$ in time and $O(n^2)$ in space.

3.2 Core Algorithm

As the scripts construct the target string T from left to right by adding segments, dynamic programming is an ideal tool for computing the transformation distance. The core part of the algorithm determines the transformation distance between S and T by a dynamic programming algorithm. Let $C[k]$ be the minimum production cost of $T[1..k]$ using the segments of S. The algorithm is given in figure 3. We make use of generic functions *CopyCost*, *RevCopyCost* and *InsertCost* as defined at the end of section 2. These functions are defined using the PreProcessing algorithm: arrays

Algorithm 3 TransformationDistance(S, T)

1. **PreProcessing**(S, T)
2. $C[0] \leftarrow 0$
3. **for** k $\leftarrow 1$ **to** $|T|$
4. $$C[k] \leftarrow \min_{0<i\leq k} \begin{cases} C[i-1] + CopyCost(T[i..k]) & \textbf{if } FP[i,k] < \infty \\ C[i-1] + RevCopyCost(T[i..k]) & \textbf{if } FPR[n-k+1, n-i+1] < \infty \\ C[i-1] + InsertCost(T[i..k]) & \\ \infty & \end{cases}$$
5. **return** $C[n]$

Figure 3: Transformation Distance: a dynamic programming solution

FP and *FPR*. In order to fix ideas, one can consider that these costs are proportional to the length of the searched segment (and ∞ if this segment does not occur in S). In fact any sub-additive function would be convenient.

Proposition 1 The recurrence relations of Algorithm 3, correctly determine the transformation distance of S and T.

Proof: We prove by induction on k that after the algorithm execution, $C[k]$ contains the minimum production cost of target $T[1..k]$ with the source string S. $C[0]$ is initialized to 0, because the cost of production of ϵ from S is zero.

Now, we suppose that $C[i]$ is calculated correctly for all $i < k$ for some positive value of k. Let us consider the calculation of $C[k]$. The last operation in a minimal script which generates $T[1..k]$, creates a suffix of $T[1..k]$. Let this suffix be $T[i..k]$. As the script is minimal, the script without its last operation is a minimal script for $T[1..(i-1)]$. The minimum cost of the script for $T[1..(i-1)]$ is $C[i-1]$ by induction hypothesis. If $T[i..k]$ exists in S and the last operation of the minimal script is a copy operation, the minimal cost of the script is $C[i-1] + CopyCost(T[i..k])$. Similarly, if the reverse of $T[i..k]$ exists in S and the last operation in the minimal script of $T[1..k]$ is a reverse copy operation, the minimal cost of the script is $C[i-1] + RevCopyCost(T[i..k])$. Finally, if the last operation in the minimal script of $T[1..k]$ is an insertion, the minimal cost of the script is $C[i-1] + InsertCost(T[i..k])$ (see figure 4). Thus, $C[n]$ is the minimum cost of production of $T = T[1..n]$ and the algorithm determines correctly the transformation distance of S and T.

Note that when the length of the substring $T[i..k]$ is smaller than MFL, *CopyCost*($T[i..k]$) and *RevCopyCost*($T[i..k]$) are equal to ∞.

The complexity of Algorithm 3 is $O(n^2)$ in time and $O(n)$ in space. So the total complexity of our algorithm (preprocessing + core algorithm) is $O(n^2)$ in time and $O(n^2)$ in space.

4 An Additional Operation: Point Deletion

In this section, we extend the set of evolutionary operations by adding the *point deletion* operation. During a point deletion (or simply deletion) operation, a symbol of the string which is under evolution is eliminated. This is an important operation from

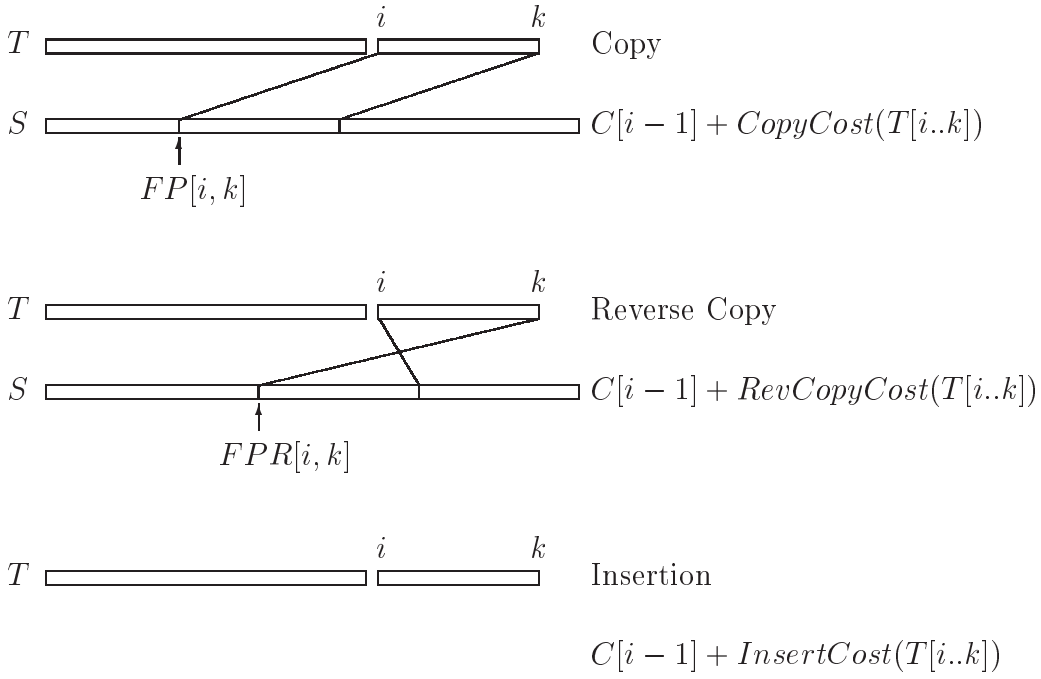

Figure 4: The three different possibilities for generation of a suffix of $T[1..k]$

the biological point of view; in the real evolution of biological sequences, in several cases after or during the copy operations some bases (symbols) are eliminated. We denote the cost of deletion of a symbol by *DelCost*. For simplicity, we suppose that the cost of deletion of every unique symbol is the same. Since we have only point deletions, deleting a segment of k symbols amounts to delete the k symbols one by one, which will cost $k \times DelCost$. As before, our objective is to find the minimum cost for a script generating a target string T, with the help of segments of a source string S. As the costs are independent of time, we consider that the deletions are applied only in the latest added segment (rightmost one), at any moment during the evolution. It should be clear that in an optimal transformation, deletions are not applied into an inserted substring (a substring which is the result of an insertion operation). Depending on the assigned costs, deletions can be used after the copy or reverse copy operations. We consider a copy operation together with all deletions which are applied to that copied segment as a unit operation. So we have a new operation called *NewCopy* which is a copy operation followed by zero or more deletions on the copied segment. In figure 5 a schema of a NewCopy operation is illustrated. Similarly, *NewRevCopy* is a reverse copy operation followed by zero or more deletions. Solving the extended transformation distance with the point deletions, amounts to solve the transformation distance with the following three operations: Insertion, NewCopy and NewRevCopy. A substring $T[i..j]$ of the target string can be produced by a unique NewCopy operation if and only if $T[i..j]$ is a subsequence string of source S. Conversely, $T[i..j]$ can be produced by a unique NewRevCopy operation if and only if $T[i..j]^{-1}$ is a subsequence string of the source S. In a preprocessing part, the algorithm determines the minimum generation cost by a NewCopy or NewRevCopy operation, for any substring of the target string T. Very similar to the last section

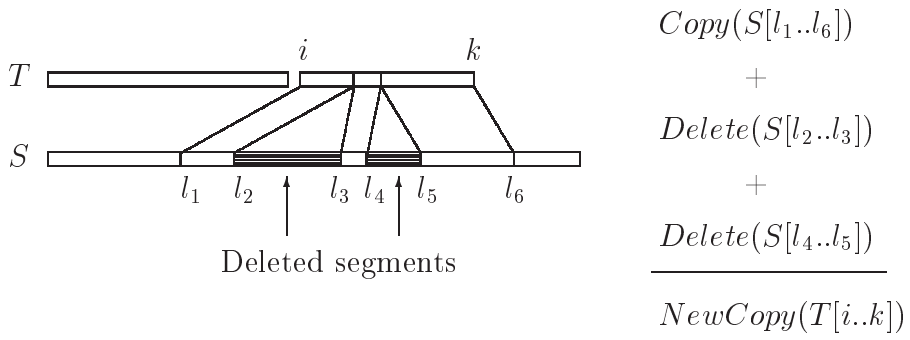

Figure 5: The illustration of NewCopy operation: A copy operation + zero or more deletions

algorithm, a dynamic programming algorithm calculates the extended transformation distance in the new core algorithm.

4.1 New Preprocessing

In the preprocessing part, we compute the costs of these new operations for any substring of the target: $NewCopyCost[i,j]$ is the minimum cost of generating the $T[i..j]$ by a NewCopy operation. Similarly, $NewRevCopyCost[i,j]$ is the minimum cost of generating $T[i..j]$ by a NewRevCopy operation. Computing the $NewCopyCost[i,j]$ amounts to find the shortest substring (with minimum length) of the source string which contains $T[i..j]$ as a subsequence string. By this way, the number of deletions which are needed for this NewCopy operation is minimized. For $NewRevCopyCost[i,j]$, we need to find the shortest substring in S^{-1} which contains $T[i..j]$ as a subsequence.

In the **NewPreProcessing** algorithm listed in figure 6, the cost tables *NewCopyCost* and *LastOcc* are initially filled with ∞ (lines 1-2). The algorithm scans the source from left to right to find the shortest supersequence for each segment of the target. The algorithm uses an auxiliary table *LastOcc* for this aim.

After the k-th letter of S is processed (loop of line 3), the following is true: $LastOcc[i,j]$ is the largest $l \leq k$ such that $S[l..k]$ is a supersequence of $T[i..j]$. The loop on T (line 4) is processed with decreasing indices for memory optimization. Whenever the letter $S[k]$ occurs in j-th position in T (line 5), then there is an opportunity of obtaining a better supersequence for some of $T[i..j]$'s, $i \leq j$. $LastOcc[i,j]$ takes the value $LastOcc[i,j-1]$ (computed for $k-1$) since $S[LastOcc[i,j-1]..k]$ is now the rightmost supersequence for $T[i..j]$ (line 9). Its cost is compared to the cost of the best previous one; if better, the new cost is stored in *NewCopyCost* (lines 11-13). One should observe that rightmost sequences are updated only when a new common letter is scanned. This is necessary and sufficient as stated in the following lemma:

Lemma 2 If $S[l..k]$ is the best supersequence for $T[i..j]$ over $S[1..N]$, then it is the rightmost supersequence for $T[i..j]$ on $S[1..k]$.

Proof: $S[l..k]$ is the best sequence for $T[i..j]$ over $S[1..k]$ then it is better than all $S[l'..k]$ for $l' < l$ and no $S[l''..k]$ can be a supersequence for $l'' < l$.

Algorithm 4 NewPreProcessing(S, T)

1. **FillArray**$(NewCopyCost, \infty)$
2. **FillArray**$(LastOcc, \infty)$ %% LastOcc is a sub-diagonal array: $LastOcc[i,j] = \infty$ for $i > j$
3. **for** $k \leftarrow 1$ **to** $|S|$ %% Source scanned left to right
4. **for each** $j \leftarrow |T|$ **downto** 1 %% find matches in T for S[k]

%% for a fixed k: $LastOcc[i,j]$ =largest l such that $S[l..k] \succ T[i..j]$

5. **if** $S[k] = T[j]$ **then**
6. $LastOcc[j,j] \leftarrow k$
7. $NewCopyCost[j,j] \leftarrow CopyCost(T[j])$ %% deletions are not needed
8. **for** $i \leftarrow 1$ **to** $j-1$ %% for all suffixes of T[1..j]
9. $LastOcc[i,j] \leftarrow LastOcc[i,j-1]$ %% $S[LastOcc[i,j-1]..k-1]] \succ T[i..j-1]$
10. $NumDel \leftarrow k - LastOcc[i,j] - i - j$ %% difference in lengths
11. $ThisCost \leftarrow DelCost \times NumDel + CopyCost(S[LastOcc[i,j]..k])$
12. **if** $ThisCost < NewCopyCost[i,j]$ **then**
13. $NewCopyCost[i,j] \leftarrow ThisCost$

Figure 6: NewPreProcessing (simplified: reverse copies have been omitted)

Algorithm 5 NewTransformationDistance(S, T)

1. **NewPreProcessing**(S,T)
2. $C[0] \leftarrow 0$
3. **for** k $\leftarrow$ 1 **to** n

4. $$C[k] \leftarrow \min_{0<i\leq k} \begin{cases} C[i-1] + NewCopyCost[i,k] & \textbf{if } FP[i,k] < \infty \\ C[i-1] + NewRevCopyCost[i,k] & \textbf{if } FPR[i,k] < \infty \\ C[i-1] + InsertCost(T[i..k]) & \\ \infty & \end{cases}$$

5. **return** $C[n]$

Figure 7: New Transformation Distance: dynamic programming

4.2 New Core Algorithm

In the core algorithm, the minimum generation costs of the prefixes of the target string T are determined from left to right. This is realized by a dynamic programming algorithm: Let $C[k]$ be the minimum production cost of $T[1..k]$ using the segments of S. The algorithm is given in figure 7. The proof of the following proposition is very similar to the proof of proposition 1:

Proposition 2 The recurrence relations of Algorithm 5, correctly determine the extended transformation distance of S and T.

The complexity of the preprocessing part, is $O(n^3)$ in time and $O(n^2)$ in space. The complexity of the core algorithm is $O(n^2)$ both in time and space. Therefore, the whole complexity of the new algorithm for the calculation of extended transformation distance is $O(n^3)$ in time and $O(n^2)$ in space.

Remarks and Conclusion

In this paper, we presented a new improved algorithm for calculation of the transformation distance problem. We also gave an algorithm for the transformation distance problem in presence of the deletion operations. In this version, costs have been given a special additive form for clarity. In fact a number of variations are possible within our framework: the main property needed on costs seems to be their subadditivity.

In this paper, we state that Algorithm 3 complexity is $O(n^3)$; this stands for the worst case complexity; in fact only a small proportion of pairs $(S[k], T[j])$ imply running the inner loop. Under certain additional statistical hypotheses the average complexity could be less than $O(n^3)$.

References

[BeSt03] Behzadi B. and Steyaert J.-M.: An Improved Algorithm for Generalized Comparison of Minisatellites. CPM 2003.

[BeRi02] Bérard, S., Rivals, E.: Comparison of Minisatellites. Proceedings of the 6th Annual International Conference on Research in Computational Molecular Biology. ACM Press, 2002.

[CLR90] Cormen, T.H., Leiserson, C.E., Rivest R.L.: Introduction to Algorithms. MIT Press, 1990.

[Do81] Doolittle, R.F.: Similar amino acid sequences: chance or common ancestry?, Science,214,149-159, 1981.

[SaKr83] Sankoff, D. and Kruskal, J.B: Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison. Addison-Wesley, 1983.

[Va00] Varré, J.S.: Concepts et algorithmes pour la comparaison de séquences génétiques : une approche informationnelle. PhD thesis, 2000.

[VDR99] Varré, J.S., Delahaye, J.P., Rivals, E.: Transformation Distances: a family of dissimilarity measures based on movements of segments. Bioinformatics, vol. 15, no. 3, pp 194-202, 1999.

[VDR98] Varré, J.S., Delahaye, J.P., Rivals, E.: The Transformation Distance : A Dissimilarity Measure Based On Movements Of Segments,German Conference on Bioinformatics, Koel - Germany, 1998.

[VDR97] Varré, J.S., Delahaye, J.P., Rivals, E.: The Transformation Distance. Genome Informatics Workshop, Tokyo, Japan, 1997.