Annales UMCS Informatica AI IX, 1 (2009) 7–13
DOI: 10.2478/v10065-009-0001-4

Annales UMCS
Informatica
Lublin-Polonia
Sectio AI
http://www.annales.umcs.lublin.pl/

Transposition Rearrangement: Linear Algorithm for Length-Cost Model*

Łukasz Mikulski†

Faculty of Mathematics and Computer Science
Nicolaus Copernicus University, Toruń, Poland

Abstract

The contemporary computational biology gives motivation to study dependencies between finite sequences. Primary structures of DNA or proteins are represented by such sequences (also called words or strings). In the paper a linear algorithm, computing the distance between two words, is presented. The model operates with transpositions of single letters. The cost of a single transposition is equal to the distance which transposed letter has to cover. Other papers concerning the model give, as the best known, algorithms of time complexity $O(n \log n)$. The complexity of our algorithm is $O(nk)$, where k is the size of the alphabet, and $O(n)$ when the size is fixed.

1. Introduction

The problem of describing similarities, or differences, between two strings has been deeply studied over the years. One of the main motivation of this studies is the rapid development of computational biology. There is a need of good models to compare sequences of genes, nucleotides or aminoacids and fast algorithms computing distances in such models.

*The research supported by Ministry of Science and Higher Education of Poland, grant N N206 258035.

†*E-mail address:* frodo@mat.umk.pl

There are two well studied approaches. The first is based on erasing or changing letters. The classical measures are Levenstein distance or Hamming distance [**1, 2**]. Another one focuses on rearranging the order of letters. This second approach is closely related to an old problem of sorting sequences. In these two problems the same types of operation can be allowed. The best examples are reversals or transpositions [**3, 4, 5**].

In this paper we deal with strings composed of the same multisets of letters. Because of the equality of Parikh vectors of such strings, we say that they are Parikh equivalent. We consider one of the models for rearranging strings by transpositions, the length-cost model. This model of measuring distance between two strings was recently introduced in [**6**]. It supposes that shorter transposing is cheaper. In the simplest case, transposing letter from the i-th to the j-th position in string costs $|i - j|$. The authors give the solution of semilinear complexity. A similar problem for the interchange rearrangement was introduced in [**7**]. Once more, the model with the cost based on a simple difference of positions of the exchanged letters is an important special case. The given algorithm is quadratic and gives the description of rearrangement. The authors also claim that this measure could be computed in linear time. Both algorithms base on an observation that it is enough to exchange subsequent letters. Then, they solve a permutation case and broaden it to a general case, by setting a numeration of occurrences of different letters.

We look at the problem of string rearrangement. We start with the binary alphabet and show the linear algorithm for that case. Subsequently, we extend the binary alphabet case to the general case, using partial solution for all projections on the binary subalphabets. The algorithm has time complexity linearly dependent on the size of compared strings. However, it is also linearly dependent on the size of the alphabet.

2. Basic Notions and Definitions

We use some basic notions of mathematical language theory. By Σ we denote an arbitrary finite set, called *alphabet*. Elements of the alphabet are called *letters*. *Words* or *strings* are arbitrary sequences over the alphabet Σ, the empty word is denoted by ϵ. By u_n we mean the nth letter of the word u. The set of all finite words is denoted by Σ^*. By $|u|$ we denote the length of word u and by $|u|_a$ the number of letters a in the word u.

An useful operation on words is a *projection* $\Pi : \Sigma^* \times 2^\Sigma \to \Sigma^*$. The projection of a word u on the subalphabet $S \subseteq \Sigma$ is the word obtained by erasing from u all letters from $\Sigma \setminus S$. More precisely, we can give an inductive definition,

assuming that c is an arbitrary letter and u is an arbitrary word:

$$\begin{array}{rcll} \Pi(\epsilon, S) & = & \epsilon & \\ \Pi(c, S) & = & c & \text{for } c \in S \\ \Pi(c, S) & = & \epsilon & \text{for } c \notin S \\ \Pi(cu, S) & = & \Pi(c, S)\Pi(u, S) & \text{for } c \in \Sigma \text{ and } u \in \Sigma^* \end{array}$$

Instead of $\Pi(u, S)$ we will write $\Pi_S(u)$.

Definition 1. Let $u, v \in \Sigma^*$ be two Parikh equivalent words (i.e. $\forall_{a\in\Sigma}\ |u|_a = |v|_a$). Then the canonical permutation from the word u to the word v is a one to one function $P_{uv} : \{1, \ldots, |u|\} \to \{1, \ldots, |v|\}$ such that $v_{P_{uv}(i)} = u_i$ and $\forall_{i<j}\ u_i = u_j \Rightarrow P_{uv}(i) < P_{uv}(j)$. Whenever is not confusing, the index will be omitted. Moreover, whenever we tell about the canonical permutation from u to v, we suppose that the words u and v are the Parikh equivalent.

Definition 2. Let $u, v \in \Sigma^*$ and P be the canonical permutation from u to v. We say that a pair (i, j) is a reversed pair if and only if $i < j \wedge P(i) > P(j)$. A set of all reversed pairs for the words u and v is denoted by $RP(u, v)$. By $\#RP(u, v)$ we denote the number of elements in the set $RP(u, v)$.

Directly from the definitions, only indices of distinct letters could form a reversal pair. Moreover, there is a strict connection between the reversed pairs of two Parikh equivalent words and the reversed pairs of their projections to binary subalphabets. The two following facts describe this condition formally.

Lemma 1. *Projections to binary alphabets preserve existence of reversed pairs. In other words, indices of the n-th a and the m-th b form a reversed pair for the words u and v if and only if indices of the n-th a and the m-th b form a reversed pair for the corresponding projections, $\prod_{a,b} u$ and $\prod_{a,b} v$.*

Proof Sketch. Projections, as morphisms, preserve the order of appearances of letters. It means that the n-th a stays before the m-th b in a word u iff the n-th a is before the m-th b in the projection $\prod_{a,b} u$. The thesis of lemma follows from that simple observation. □

Proposition 1. *Then number of reversed pairs in two words u and v is equal to the sum of reversed pairs in their projections to all binary subalphabets. More formally:*

$$\#RP(u, v) = \sum_{a,b\in\Sigma} \#RP(\prod_{a,b} u, \prod_{a,b} v).$$

Proof Sketch. Let (i, j) be a reversed pair for the words u and v. Then, from Lemma 1, there is a corresponding reversed pair for $\Pi_{\{u_i,u_j\}}(u)$ and $\Pi_{\{u_i,u_j\}}(v)$. It means that

$$\#RP(u,v) \leq \sum_{a,b\in\Sigma} \#RP(\prod_{a,b} u, \prod_{a,b} v).$$

On the other hand, the reversed pairs counted from different projections are formed by different pairs of letters, so in the right side we count every pair at most once. It means that

$$\#RP(u,v) \geq \sum_{a,b\in\Sigma} \#RP(\prod_{a,b} u, \prod_{a,b} v),$$

which ends the proof. □

Example 1. Let us consider two strings $u = abac$ and $v = cbaa$.
These strings are the Parikh equivalent (both of them consist of two letters a, one letter b and one letter c).
The canonical permutation looks as follows:

$$\begin{array}{ccc} 1 & \to & 3 \\ 2 & \to & 2 \\ 3 & \to & 4 \\ 4 & \to & 1 \end{array}$$

The set of reversed pairs is $RP(u,v) = \{(1,2),(1,4),(2,4),(3,4)\}$.
The projections to binary subalphabets of Σ looks as follows:

$$\begin{array}{rcl} \prod_{a,b} u & = & aba \\ \prod_{a,b} v & = & baa \\ \Pi_{a,c}(u) & = & aac \\ \Pi_{a,c}(v) & = & caa \\ \Pi_{b,c}(u) & = & bc \\ \Pi_{b,c}(v) & = & cb \end{array}$$

The sets of reversed pairs for these projections are appropriately

$$\begin{array}{rcl} RP(\prod_{a,b} u, \prod_{a,b} v) & = & \{(1,2)\} \\ RP(\Pi_{a,c}(u), \Pi_{a,c}(v)) & = & \{(1,3),(2,3)\} \\ RP(\Pi_{b,c}(u), \Pi_{b,c}(v)) & = & \{(1,2)\} \end{array}$$

3. Binary Alphabet Case

Problem 1. Let u and v be two Parikh equivalent words over the binary alphabet $\Sigma = \{a, b\}$. Compute the minimum cost of transforming u into v by transpositions, when the cost of transposing a single letter through l positions is l.

According to [**6**], we have to label letters in both strings and compute the number of reversed pairs (in the sense of definitions 1 and 2). Every transposition of length l can be decomposed to l transpositions of length 1; cost of the long transposition is equal to the sum of costs of the short transpositions. We can consider only these short jumps over the single letter. The next observation refers to the reversed pairs. For each reversed pair, at least one jump has to be done to set this pair in a correct order. It is also possible to make whole transformation with transpositions of the length 1, where the number of such transpositions is equal to the number of the reversed pairs. In the case of binary alphabet, we can count the number of reversed pairs simply counting "how many b's are before each a" in the strings u and v. This way we get two vectors $U(u)$ and $V(v)$, of the length $|u|_a = |v|_a$. Then we compute the distance between two produced vectors using the Manhattan metrics ($|u - v| = \Sigma_{i=1}^{n} |u_i - v_i|$). The straightforward algorithm (without checking the correctness of data) is:

Algorithm 1. StringToVector(u)

```
1. a := 1, b := 0;
2. for i from 1 to sizeof(u) do
3.     begin
4.     if (u[i] = b)
5.         then b++;
6.         else U[a] := b; a++;
7.     end
8. return U;
```

BinaryLCMDistance(u,v)

```
1. d:=0;
2. U := StringToVector(u)
3. V := StringToVector(v)
4. for i from 1 to sizeof(U) do
5.     begin
6.     d += —U[i]-V[i]—
7.     end
8. return d;
```

Complexity. Both procedures use one loop looking through input strings letter by letter. Hence, the time complexity is linear.

4. Arbitrary Alphabet Case

Problem 2. Let u and v be two Parikh equivalent strings over alphabet Σ. Compute the minimum cost of transforming string u into v by transpositions when the cost of transposing a single letter through l positions is equal to l.

Similarly to the binary alphabet case, we can also label letters and fix attention on the number of reversed pairs in the processed strings u and v. Moreover, using Proposition 1, we can consider only projections to the binary subalphabets and sum up the results. It gives an algorithm that is linearly dependent on the length of the strings but also linearly dependent on the size of the alphabet Σ. If we treat the size of the alphabet as a constant, the algorithm is simply linear.

Algorithm 2. ComputeProjection(u,a,b)

```
1. u_{a,b} := ϵ;
2. j := 1;
3. for i from 1 to sizeof(u) do
4.     if (u[i] = a or u[i] = b)
5.         begin
6.         u_{a,b}[j] := u[i];
7.         j++;
8.         end
9. return u_{a,b};
```

LCMDistance(u,v)

```
1. d:=0;
2. for each pair a,b where a, b ∈ Σ
3.     begin
4.     u_{a,b} := ComputeProjection(u,a ,b);
5.     v_{a,b} := ComputeProjection(v,a ,b);
6.     d += BinaryLCMDistance(u_{a,b},v_{a,b});
7.     end
8. return d;
```

The first procedure computes the projection of a word u on the binary subalphabet $\{a, b\}$. Its linear complexity is obvious. In the second one, using the procedure that computes the cost of rearrangement for the binary alphabet case, the cost of rearrangement in the general case is computed. We use a loop that runs through all pairs of letters from the alphabet. It gives quadratic complexity with respect to the size of alphabet. However, small changes in the algorithm (computing all projections in one simple run) allow to establish the

time complexity to $O(nk)$, where n is the length of a rearranged word and k is the size of the alphabet.

5. Conclusions

The linear algorithm computing the cost of rearrangement of finite sequences is presented. The only allowed operations are transpositions, the cost of a single transposition is given by its range. The algorithm is dependent on the size of the alphabet. It is a serious disadvantage in the cases when the size of the alphabet is close to the length of sequence. However, many practical situations operate on small sets and long sequences. For instance, the DNA chains are the sequences over the set of cardinality four.

References

[1] Thomas H. Cormen and Charles E. Leiserson and Ronald L. Rivest *Introduction to Algorithms*, MIT Press, 1994.

[2] Gusfield D., *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1999.

[3] Radcliffe, Scott and Wilmer *Reversals and Transpositions Over Finite Alphabets*, SIJDM: SIAM Journal on Discrete Mathematics 19(1) (2005) 224.

[4] Bafna V. and Pevzner P. A., *Sorting by Transpositions*, SIAM Journal on Discrete Mathematics 11(2) (1998) 224.

[5] Hartman T. and Sharan R., *A 1.5-Approximation Algorithm for Sorting by Transpositions and Transreversals*. In Inge Jonassen and Junhyong Kim, editors, Proceedings of WABI 2004, Vol. 3240 of Lecture Notes in BI, 50–61. Springer, 2004.

[6] Kapah O., Landau G. M., Levy A. and Nitsan Oz *Interchange Rearrangement: The Element-Cost Model*. In SPIRE, vol. 5280 of Lecture Notes in Computer Science, 224–235. Springer, 2008.

[7] Amir A., Hartman T., Kapah O., Levy A. and Porat E., *On the Cost of Interchange Rearrangement in Strings*. In Lars Arge and Michael Hoffmann and Emo Welzl, editors, Proceedings of 15th Annual European Symposiom, vol. 4698 of Lecture Notes in Computer Science, 99–110. Springer, 2007.