Information Processing Letters 113 (2013) 265–270

Contents lists available at SciVerse ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

Prefix transpositions on binary and ternary strings

Amit Kumar Dutta, Masud Hasan *, M. Sohel Rahman

Department of Computer Science & Engineering, Bangladesh University of Engineering and Technology, Dhaka, Bangladesh

ARTICLE INFO

Article history:
Received 1 May 2012
Received in revised form 15 January 2013
Accepted 21 January 2013
Available online 7 February 2013
Communicated by Ł. Kowalik

Keywords:
Algorithms
Combinatorial problems
Prefix transposition
Ternary strings
Binary strings
Genome rearrangement

ABSTRACT

The problem of sorting by prefix transpositions asks for the minimum number of prefix transpositions required to sort the elements of a given permutation. In this paper, we study a variant of this problem where the prefix transpositions act not on permutations but on strings over an alphabet of fixed size. Here, we determine the minimum number of prefix transpositions required to sort the binary and ternary strings, with polynomial time algorithms for these sorting problems.

1. Introduction

The *transposition distance* between two permutations (and the related problem of *sorting by transposition*) is used to estimate the number of global mutations between genomes and can be used by molecular biologists to infer evolutionary and functional relationships. A transposition involves swapping two adjacent substrings of the permutation. In a prefix transposition, one of them must be a prefix. *Sorting by prefix transposition* is the problem of finding the minimum number of prefix transpositions needed to transform a given permutation into the identity permutation. In the literature, other interesting problems include sorting by other operations like reversals, prefix reversals, block interchange etc.

A natural variant of the aforementioned sorting problems is to consider them not on permutations but on strings over fixed size alphabets. This shift is inspired by the biological observation that multiple "copies" of the same gene can appear at various places along the genome [4]. Indeed, recent works by Christie and Irving [2], Radcliffe et al. [5] and Hurkens et al. [4] explore the consequences of switching from permutations to strings. Notably, such rearrangement operations on strings have been found to be interesting and important in the study of orthologous gene assignment [1], especially if the strings have only low level of symbol repetitions.

Chen et al. [1] presented for both reversals and transpositions, polynomial-time algorithms for computing the minimum number of operations to sort a given binary string. They also gave exact constructive diameter results on binary strings. Radcliff et al. [5] on the other hand gave refined and generalized reversal diameter results for non-fixed sized alphabets. Additionally, they gave a polynomial-time algorithm for optimally sorting a ternary (3 letter alphabet) string with reversals. Finally, Hurkens et al. [4] introduced *grouping* (a weaker form of sorting), where identical symbols need only be grouped together, while a group can be in any order. In the sequel, they gave a complete characterization of the minimum number of prefix reversals required to group (and sort) binary and ternary strings.

In this paper, we follow up the work of [4] and consider prefix transposition to group and sort binary and ternary strings. Notably, as a future work in [4], the authors raised the issue of considering other genome arrangement operators. In particular, here, we find the minimum number of

* Corresponding author.
E-mail address: mhasan2010@gmail.com (M. Hasan).

0020-0190/$ – see front matter
http://dx.doi.org/10.1016/j.ipl.2013.01.017

prefix transpositions required to group and sort binary or ternary strings. It may be noted that, apart from being a useful aid for sorting, grouping itself is a problem of interest in its own right [3].

The rest of the paper is organized as follows. In Section 2, we discuss the preliminary concepts and discuss some notations we use. Section 3 is devoted to grouping, where we present and prove the corresponding bounds. In Section 4, we present an algorithm to group the strings and in Section 5, we classify the strings along two different dimensions. Section 6 extends the results on grouping to get the corresponding bounds for sorting. Finally, we briefly conclude in Section 7.

2. Preliminaries

We follow the notations and definitions used in [4], which are briefly reviewed below for the sake of completeness. We use $[k]$ to denote the first k non-negative integers $\{0, 1, \ldots, k-1\}$. A k-ary string is a string over the alphabet $\Sigma = [k]$. Moreover, a string $s = s[1]s[2]\ldots s[n]$ of length n is said to be *fully k-ary*, or to have *arity k*, if the set of symbols occurring in it is $[k]$.

A prefix transposition $f(1, x, y)$ on a string s of length n, where $1 < x < y \leqslant (n+1)$, is a rearrangement event that transforms s into $[s[x]\ldots s[y-1]s[1]\ldots s[x-1]s[y]\ldots s[n]]$. The prefix transposition distance $d_s(s)$ of s is defined as the number of prefix transpositions required to sort the string. Note that, after a transposition operation is performed, the two adjacent symbols of the corresponding string may become identical. We consider two strings to be equivalent if one can be transformed into the other by repeatedly duplicating (by transposing) symbols and eliminating adjacent identical symbols. This elimination of adjacent identical symbols gives us a *reduced string*, i.e., a string of reduced length and this process is referred to as the *reduction*. A string having no identical adjacent symbols is said to be a *normalized string*. As representatives of the *equivalence classes* we take the shortest string of each class. Clearly, these are normalized strings where adjacent symbols always differ.

For example, let $s = bababab$ and we want to apply operation $f(1, 3, 6)$. Now, $s[x]\ldots s[y-1] = s[3]\ldots s[5] = bab$, $s[1]\ldots s[x-1] = s[1]\ldots s[2] = ba$, $s[y]\ldots s[n] = s[6]\ldots s[7] = ab$. Therefore, after applying the operation, we get, $s = s[3]\ldots s[5]s[1]\ldots s[2]s[6]\ldots s[7] = \overline{bab}\underline{ba}ab = ba\mathbf{bbaa}b = ba\mathbf{ba}b = bababab$.

A transposition that decreases the string length by p (possibly after some reduction) is called a p-transposition. So, if $p = 0$, then we have a 0-transposition. The above example illustrates a 2-transposition.

3. Grouping

The task of sorting a string can be divided into two subproblems, namely, *grouping* the identical symbols together and then putting the groups of identical symbols in the right order. The grouping distance $d_g(s)$ of a fully k-ary string s is defined as the minimum number of prefix transpositions required to reduce the string to one of length k. In what follows, we deduce different bounds for the grouping distance of fully binary and ternary strings and our bounds are deduced in terms of normalized strings. However, note that, the bounds deduced are equally applicable to a general (i.e., non-normalized) string having identical adjacent symbols, because, such a string can be easily reduced to a normalized string through *reduction* and the process of *reduction* (i.e., elimination of adjacent identical symbols) does not increase or decrease the number of prefix transpositions.

3.1. Grouping binary strings

As strings are normalized, only 2 kinds of binary strings are possible, namely, $010101\ldots010$ and $101010\ldots101$. The grouping of binary strings seems to be quite easy and obvious. The following bound is easily achieved.

Theorem 1 *(Bound for Binary strings). Let s be a fully binary normalized string and* $|s| > 2$. *Then,* $d_g(s) = \lceil \frac{n-2}{2} \rceil$.

Proof. We can always have a 2-transposition if $|s|$ is even. However, if $|s|$ is odd, we need an extra 1-transposition. So, the upper bound is $d_g(s) = \lceil \frac{n-2}{2} \rceil$. □

We illustrate the above result with the help of an example. Let $s = ababab$. Then we can continue as follows: $s = ababab = \overline{ab}\underline{a}bab = \mathbf{aabb}ab = abab = \overline{ab}\underline{a}b = \mathbf{aabb} = ab$. Here, we need two 2-transpositions to group this string. So, $d_g(s) = 2$.

3.2. Grouping ternary strings

In this section, we focus on ternary strings. As it seems, grouping ternary strings is not as easy as grouping binary strings. We start with the following theorem.

Lemma 1. *In a fully ternary normalized string, we can always perform a* 1*-transposition.*

Proof. We take a ternary string s of length $n > 3$. Now, we take a prefix $a[1\ldots k]$ of length k. If $a[1]$ occurs at the suffix at position i, we can transpose $a[1]\ldots a[i-1]$ with $a[i]$. Then, a[1] and a[i] are adjacent and we can eliminate one of the two. Otherwise, if $a[k]$ occurs at the suffix at position i, then we can transpose $a[1]\ldots a[k]$ with $a[k+1]\ldots a[i-1]$. Then $a[k]$ and $a[i]$ are adjacent and as before, we can eliminate one of them. Since, one of the above cases always occurs for ternary strings, the result follows. □

To perform a 2-transposition we take a prefix of length at least 2. So, instead of taking a prefix of length at least 2, we can always take a prefix of length 1 (i.e. with the first character), and perform a 1-transposition. Thus, presence of a 2-transposition always ensures that there is also a 1-transposition. For example, let us take 01201. There is a 2-transposition: $\overline{01}\underline{20}1 \Rightarrow 201$. We could also get a 1-transposition as follows: $\bar{0}\underline{120}1 \Rightarrow 1201$.

3.3. Grouping distance for Ternary strings

The lower bound for the *grouping* of a ternary string remains the same as that of binary strings; but, as can be

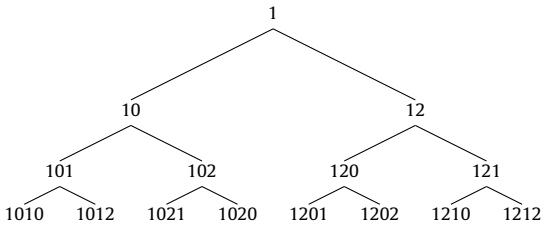

Fig. 1. Tree diagram for all strings starting with 1.

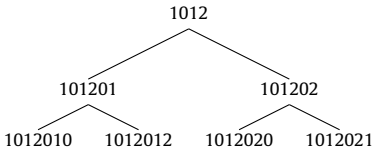

Fig. 2. Strings with prefix 1012.

seen from Theorem 2 below, the upper bound differs. We first give an easy but useful lemma.

Lemma 2. *Suppose $s[1..n]$ is a fully ternary normalized string. If we have a prefix $s[1..i]$, $1 < i \leqslant n-2$, such that $s[1] = s[n-1]$ and $s[i] = s[n]$, then we have a 2-transposition.*

The proof of Lemma 2 is very easy and hence is omitted.

Theorem 2 *(Bound for Ternary strings). Let s be a fully ternary normalized string. Then, $\lceil \frac{n-3}{2} \rceil \leqslant d_g(s) \leqslant \lceil \frac{n-3}{2} \rceil + 1$ where n is the length of the string and $n \neq 4$.*

Proof. For lower bounds, we consider the best cases where we will be able to give 2-transpositions in every step of the reduction. If $|s|$ is odd, the string length will reduce to 3 (e.g. for $|s| = 7$, we get $7 \Rightarrow 5 \Rightarrow 3$ length strings in each step). If $|s|$ is even, string length will reduce to 4 and we require an additional 1-transposition (e.g. for $|s| = 8$, with 2 transposition in each step, we get $8 \Rightarrow 6 \Rightarrow 4$ and then, an extra 1-transposition).

Let us now concentrate on the upper bound. As strings are fully ternary, we don't need to work with $n \leqslant 3$. Now, if we apply the upper bound for $n = 5$ and 6, we have $d_g(s) = 2$ and 3 respectively. It is easy to realize that, by Lemma 1, we can always satisfy the above upper bound. Thus the upper bound is proved for $n < 7$.

Now we consider $n \geqslant 7$. In what follows, we only consider strings starting with 1. This doesn't lose the generality since we can always use relabeling for strings starting with 0 or 2. Now, note carefully that for any string starting with 1, we can only have one of the eight prefixes of length 4 shown in list (1) below.

$$1012, 1010, 1021, 1020, 1201, 1202, 1210 \quad \text{and} \quad 1212. \tag{1}$$

In Fig. 1, we give the tree diagram of all strings starting with 1.

Now note that, the upper bound of Theorem 2 tells us that we can give at most three 1-transpositions when n is even (i.e., $n-k$ is odd) and two 1-transpositions when n is odd (i.e., $n-k$ is even). Note that, if we could give a 2-transposition at each step, we would get the bound of $\lceil \frac{n-k}{2} \rceil$. For an n length string, if we can give a 2-transposition, the resulting reduced string may start with 1, 0 or 2. For the latter two cases, we can use relabeling as mentioned before. Therefore we can safely state that the reduced string will have any of the 8 prefixes of list (1). Hence, it suffices to prove the bound considering each of the prefixes of list (1). We will now follow the following strategy:

"We will take each of the prefixes of list (1) and expand it (by adding symbols) to construct all possible strings of length greater than or equal to 7. Strictly speaking, we will not consider all possible strings; rather we will continue to expand until we get a 2-transposition, since afterwards, any further expansion would also guarantee a 2-transposition. Suppose s is one such string. We will take s and try to give a 2-transposition with any of its prefix. If we succeed, then, clearly, we are moving towards the best case and we only need to work with the reduced string. If we cannot give a 2-transposition, we specifically deal with s and show that the bound holds. Now if we can give a 2-transposition, the reduced string will have any of the 8 prefixes (using relabeling if needed) and we will show that all strings of these cases will follow the bound".

Firstly it is easy to note that, the prefixes 1010 and 1212 themselves have 2-transpositions (Lemma 2). Therefore, we can safely exclude them from the following discussion. In what follows, when we refer to the prefixes of list (1), we would actually mean all the prefixes excluding 1010 and 1212. Now, to expand any of the prefixes, if we add 10 or 12, it would be able to give a 2-transposition (Lemma 2). Therefore, in what follows, we consider the other cases. Now we analyze each of the prefixes below. In what follows, when we say we 'add' to a string we basically mean to 'append' to it.

1012 We first give a tree diagram of strings having the prefix 1012 in Fig. 2.

If we add 01, we can only add 0 or 2 subsequently. The resulting expanded string becomes 1012010 or 1012012. In both cases, we can give a 2-transposition. On the other hand, if we add 02, we can add 0 or 1 next and the string becomes 1012020 or 1012021.

The string 1012020 satisfies the bound as follows: $\overline{1}\underline{01}2020 \Rightarrow 012020 \Rightarrow \overline{012}\underline{0}20 \Rightarrow 0120 \Rightarrow \overline{0}\underline{12}0 \Rightarrow 120$. Here, $n = 7$ and we need a total of three transpositions only holding the bound true. Now if we further add 1, the string becomes 10120201 (not shown in Fig. 2). For this one as well the bound holds as follows: $\overline{1}\underline{01}20201 \Rightarrow 0120201 \Rightarrow \overline{01}\underline{2020}1 \Rightarrow 20201 \Rightarrow \overline{20}\underline{2}01 \Rightarrow 201$. Here, $n = 8$ and we need a total of three transpositions. Now, it can be easily checked that strings like 1012020(20)* or 1012020(20)*1, the bound holds using the same strategy as shown above. Adding anything with 1012020(20)*1 will also give a 2-transposition as follows. Clearly, we first need to add either 0 or 2 and immediately Lemma 2 would apply.

Now we consider 1012021. The bound holds for this one as well as follows: $\overline{1}\underline{01}2021 \Rightarrow 012020 \Rightarrow \overline{012}\underline{0}21 \Rightarrow 0121 \Rightarrow \overline{01}\underline{2}1 \Rightarrow 201$. Now, strings like 1012(02)*1 can also be handled similarly and hence the bound holds for them as well. Adding anything with 1012(02)*1 will also give a 2-transposition (Lemma 2).

1021 This prefix is ending with 1 and adding anything will give a 2-transposition (Lemma 2).

1020 We first add 21 with this prefix. Subsequently, adding anything (i.e., adding anything with 102021) will give a 2-transposition (Lemma 2). Now, for $102(02)^+1$, we need a 1-transposition to move the initial 1 at the end. Now the upper bound holds because the rest of the string is binary (Theorem 1). Also, adding anything with $102(02)^+1$ will give a 2-transposition (Lemma 2).

Now, if we add 20 with the prefix we get 102020. Next we add 1 or 2 and get 1020201 or 1020202 respectively. For 1020201, the bound holds as follows: $\overline{1}\underline{020201} \Rightarrow 020201 \Rightarrow \overline{02}\underline{0}201 \Rightarrow 0201 \Rightarrow \overline{0}\underline{2}01 \Rightarrow 201$. All strings like $1020(20)^+1$ can also be handled similarly and hence the bound holds for them as well. And adding anything with $1020(20)^+1$ will give a 2-transposition (Lemma 2).

Now, for 1020202, the bound holds as follows: $\overline{10}\underline{2}0202 \Rightarrow 210202 \Rightarrow \overline{210}\underline{2}02 \Rightarrow 2102 \Rightarrow \overline{2}\underline{10}2 \Rightarrow 102$. Now, strings like $10202(02)^+$ can be handled similarly and hence the bound holds. For $10202(02)^+1$, we need a 1-transposition to move the initial 1 at the end. Now the upper bound holds because the rest of the string is binary (Theorem 1). Also adding anything with $10202(02)^+1$ will give a 2-transposition (Lemma 2).

1201 This prefix is ending with a 1 and adding anything will give a 2-transposition (Lemma 2).

1202 Here, we can employ relabeling and map 2 to 0 and 0 to 2 to get 1020. Now recall that we have already considered 1020 before and hence we are done.

1210 Here we can again employ relabeling and map 2 to 0 and 0 to 2 to get 1012. And since we have already considered 1012, we are done.

This completes the proof. □

Lemma 3. *Let s be a fully ternary normalized string of length $n = 4$. Then, $d_g(s) = 1$.*

Proof. For the fully ternary normalized strings of length 4, we only require one 1-transposition to reduce it's length to three. Hence $d_g(s) = 1$. □

4. An algorithm to group fully binary and ternary strings

In this section we present Algorithm 1 to group fully binary and ternary strings that satisfy the proposed bound. We take all possible prefixes and try to find a 2-transposition in the suffix. If no 2-transposition is found, the algorithm gives a 1-transposition with the current prefix.

Algorithm 1: GroupByPrefixTransposition (s:input string).

```
Input: s, a fully binary or ternary string
initialization;
k ⇐ 2 if s is binary;
k ⇐ 3 if s is ternary;
count ⇐ 0;
twoTranspDone ⇐ false;
while |s| >k do
    for i = 1; i <| s |; i = i + 1 do
        take the first symbol of input string;
        take the i-th symbol of the input string;
        append these two symbols;
        call this string temp;
        check whether this string is a substring of the current
        suffix;
        for j = i + 1; j < (|s| − 1); j = j + 1 do
            take the substring named consecutive of input string
            from j to j + 2;
            if consecutive == temp then
                perform a 2-transposition;
                count ⇐ count + 1;
                twoTranspDone ⇐ true;
                break;
            end
        end
        if twoTranspDone == false then
            perform a 1-transposition using Algorithm 2;
            s ⇐ return value from 2;
            count ⇐ count + 1;
        end
    end
    twoTranspDone ⇐ false;
end
```

It runs until the length is 2 and 3 for fully binary and fully ternary strings respectively. A variable *count* is initialized to zero and after the algorithm runs, it contains total number of prefix transpositions required. Algorithm 1 uses Algorithm 2 to perform 1-transpositions (which is always available as discussed in Lemma 1). Algorithm 2 is simple. It first checks if the first character exists in the suffix. If so, it performs a 1-transposition. If not, it increases the length of the prefix and checks if the last character of the prefix exists in the suffix. If so, it performs a 1-transposition with that prefix and suffix pair.

Before formally proving the correctness of our algorithm, it will be useful to provide some classifications of the fully binary and ternary strings. In the following section, we will first classify the fully binary and ternary strings along two different dimensions and then discuss the correctness of our algorithm. Notably, apart from being useful in proving the correctness of our algorithm, the classifications provided below would be interesting in their own right.

5. Classification

5.1. Classes of ternary strings

In this section, we will identify two classes of fully ternary strings. We have already shown that it needs at most $\lceil \frac{n-3}{2} \rceil + 1$ prefix transpositions to group ternary strings. We will discuss two classes; the first one consists

Algorithm 2: Do1Transposition (s:input string).

```
Input: s is a fully binary or ternary string passed as a parameter
       from Algorithm 1
forward ⇐ false;
take the suffix of input string leaving the initial symbol of s;
if the first symbol exists in the suffix then
    perform a 1-transposition to s;
    forward ⇐ true;
end
if forward == false then
    for p = 1; p < (|s| − 1); p++ do
        take a string suffix, substring of input string form (p + 1)
        to end;
        if p-th symbol occurs at suffix then
            perform a 1-transposition;
        end
    end
end
return s;
```

of the strings that require $\lceil \frac{n-3}{2} \rceil$ prefix transpositions (Class 1) and the other consists of the strings that need $\lceil \frac{n-3}{2} \rceil + 1$ (Class 2). We are not able to classify all the fully ternary strings; however below we provide classification of a significant number of strings:

- From Lemma 3, we find that $d_g(s) = 1$ for all fully ternary strings of length four. So, these strings satisfy $\lceil \frac{n-3}{2} \rceil$ bound, so we put them in Class 1.
- $(10)^+102$, $(10)^+210$, $(10)^+212$, $(12)^+010$, $(12)^+012$, $(12)^+120$ belong to Class 1. We can apply relabeling on these to find strings starting with 0 or 2 and they also belong to Class 1.
- If a ternary string can be reduced to any one of the previous strings by a series of 2-transpositions, then that particular string also belongs to Class 1.

In Class 2, we put the strings that need $\lceil \frac{n-3}{2} \rceil + 1$ prefix transpositions to be grouped.

- 10120, 10121, 10201, 10202, 12021, 12020, 12101, 12102 and those we get by applying relabeling on them are in Class 2. There is no 2-transposition available for these strings.
- $(10)^+120$, $(10)^+121$, $(10)^+201$, $(10)^+2(02)^+$, $(12)^+021$, $(12)^+0(20)^+$, $(12)^+101$, $(12)^+102$, $(10)^+1201$, $(10)^+2021$, $(12)^+0201$, $(12)^+1021$ and those strings that we get by applying relabeling on these strings belong to Class 2.
- Ternary strings reduced to any of these by a series of 2-transpositions will also be placed in Class 2.

Now, we will categorize all the fully ternary strings from a different perspective. We call a string "Good string" if Algorithm 1 gives the optimal result. All the strings in Class 1 and Class 2 specified earlier are "Good strings". There are strings for which Algorithm 1 will not give optimal results. Those strings are called "Hard strings". $(02)^+10202, (10)^+21010, (21)^+02121$ are examples of "Hard strings". To elaborate, let us take a fully ternary string 0210202 as an example. Algorithm 1 will give the following set of prefix transpositions: $\overline{02}10202 \Rightarrow \overline{10}202 \Rightarrow \overline{2}102 \Rightarrow 102$. Here we need 3 prefix transpositions which satisfy the bound. However, an optimal sequence will need 2 prefix transpositions to group this string $\overline{0210202} \Rightarrow \overline{02}102 \Rightarrow 102$. So, 0210202 is a "Hard string".

5.2. Correctness of Algorithm 1

With the above classification of fully ternary strings at our hand, we are now ready to prove the correctness of our algorithm. We prove the following theorem.

Theorem 3. *Given a fully binary and ternary normalized string, Algorithm 1 is always able to group the string satisfying the bound given in Theorems 1 and 2.*

Proof. The proof is simple for fully binary strings because Algorithm 1 will always give optimal results for fully binary strings. So we need only to prove that Algorithm 1 will always satisfy the proposed bounds for ternary strings. Since, by definition, Algorithm 1 provides optimal results for "Good strings", we only need to consider "Hard strings" and prove that Algorithm 1 will satisfy the upper bound while dealing with the "Hard strings". Consider a hard string. Algorithm 1 may choose a 2-transposition which may not be optimal. Then it will reduce the string into any one of the following: $(10)^+120$, $(10)^+121$, $(10)^+201$, $(10)^+2(02)^+$, $(12)^+021$, $(12)^+0(20)^+$, $(12)^+101$, $(12)^+102$, $(10)^+1201$, $(10)^+2021$, $(12)^+0201$, $(12)^+1021$ or any fully ternary string of length 4 and those which we get by applying relabeling any of these. These strings satisfy the upper bound. So, overall Algorithm 1 will satisfy the upper bound. □

6. Sorting

The sorting distance $d_s(s)$ of a fully k-ary string s is defined as the minimum number of prefix transpositions required to sort the string to one of length k.

6.1. Sorting binary strings

Theorem 4 *(Bound for Binary strings). Let s be a fully binary normalized string. Then,* $d_s(s) \leqslant \lceil \frac{n-2}{2} \rceil$.

Proof. As binary strings have only 2 letters, after grouping they are already sorted (in ascending or descending order). So, the upper bound is $d_s(s) \leqslant \lceil \frac{n-2}{2} \rceil$. □

6.2. Sorting ternary strings

Theorem 5 *(Bound for Ternary strings). Let s be a fully ternary normalized string. Then, upper bound for sorting ternary string is* $d_g(s) \leqslant \lceil \frac{n-3}{2} + 2 \rceil$.

Proof. After grouping a ternary string, we have the following grouped strings: $012, 021, 102, 120, 201$ and 210. Among these, 012 and 210 are already sorted. We need

one more 0-transposition to sort $021, 102, 120$ and 201. Hence the result follows. □

6.3. Algorithm to sort fully binary and ternary strings

We first apply Algorithm 1 of Section 4 on the input string. Then, if the string is binary, it is already sorted. If it is ternary and the after grouping we have any one of $021, 102, 120$ and 201, we need 1 more prefix transposition to sort the string. The algorithm is formally presented in the form of Algorithm 3.

Algorithm 3: SortByPrefixTransposition (s:input string).

Input: s is a fully binary or ternary string

```
run Algorithm 1 on s;
if s is binary then
    s is sorted after grouping;
    finish;
end
else
    if s is in 021, 102, 120, 201 then
        perform one 0-transposition to sort s;
    end
    else
        s is already sorted;
    end
end
```

7. Conclusion

In this paper, we have discussed grouping and sorting of fully binary and ternary strings when the allowed operation is prefix transposition. Following the work of [4], we have handled grouping by prefix transpositions of binary and ternary strings first and extended the results for sorting. In particular we have proved that, for binary strings the grouping distance is $d_g(s) = \lceil \frac{n-2}{2} \rceil$ and for ternary string we have $\lceil \frac{n-3}{2} \rceil \leqslant d_g(s) \leqslant \lceil \frac{n-3}{2} \rceil + 1$, where $n \neq 4$ and $d_g(s) = 1$ when $n = 4$. On the other hand, for sorting binary and ternary strings the sorting distance $d_s(s)$ is upper bounded by $\lceil \frac{n-2}{2} \rceil$ and $\lceil \frac{n-3}{2} \rceil + 2$ respectively. As has already been mentioned, we are now considering the higher-arity alphabets as an extension of our work.

References

[1] X. Chen, J. Zheng, Z. Fu, P. Nan, Y. Zhong, S. Lonardi, T. Jiang, Assignment of orthologous genes via genome rearrangement, IEEE/ACM Trans. Comput. Biology Bioinform. 2 (4) (2005) 302–315.
[2] D.A. Christie, R.W. Irving, Sorting strings by reversals and by transpositions, SIAM J. Discrete Math. 14 (2) (2001) 193–206.
[3] H. Eriksson, K. Eriksson, J. Karlander, L.J. Svensson, J. Wästlund, Sorting a bridge hand, Discrete Math. 241 (1–3) (2001) 289–300.
[4] C.A.J. Hurkens, L. van Iersel, J. Keijsper, S. Kelk, L. Stougie, J. Tromp, Prefix reversals on binary and ternary strings, SIAM J. Discrete Math. 21 (3) (2007) 592–611.
[5] A.J. Radcliffe, A.D. Scott, E.L. Wilmer, Reversals and transpositions over finite alphabets, SIAM J. Discrete Math. (2006).