B.Sc. Engg. Thesis

Prefix transpositions on binary and ternary strings

By
Amit Kumar Dutta

Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology (BUET)
Dhaka-1000.

Declaration

This is to certify that the work presented in this thesis entitled "**Prefix transpositions on binary and ternary strings**" is the outcome of the investigation carried out by me under the supervision of Dr. Masud Hasan, Assistant Professor, Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology (BUET), Dhaka.

– – – – – – – – – – – – – –

(Supervisor)
Dr. Masud Hasan
Assistant Professor
Department of Computer Science
and Engineering (BUET), Dhaka-1000

– – – – – – – – – – – – – –

(Author)
Amit Kumar Dutta
Student No.: 0405071
Department of Computer Science
and Engineering (BUET), Dhaka-1000.

Abstract

Genome rearrangement algorithms are powerful tools to analyze gene orders in molecular evolution. Analysis of genomes evolving by reversals and transpositions leads to a combinational problem of sorting by reversals and transpositions, the problem of finding a shortest sequence of reversals and transpositions that sorts one genome into the other. Given a permutation π, the application of prefix reversal $f^{(i)}$ to π reverses the order of the fisrt i elements of π. The problem of Sorting By Prefix Reversals (also known as $pancakeflipping$), asks for the minimum number of prefix reversals required to sort the elements of a given permutation. A variant of this problem was also studied where the prefix reversals act not on permutations but on strings over a fixed size alphabet. The minimum number of prefix reversals required to sort binary and ternary strings, with polynomial-time algorithms for these sorting problems is significant in Bioinformatics.

The problem sorting by Prefix Transpositions asks for the minimum number of prefix transpositions required to sort the elements of a given permutation. In this thesis, we study a variant of this problem where the prefix transpositions act not on permutations but on strings over a fixed size alphabet. We determine the minimum number of prefix transpositions required to sort the binary and ternary strings, with polynomial time algorithms for these sorting problems. We also considered *grouping* and give polynomial-time algorithms for optimally grouping binary and ternary strings.

Acknowledgments

First of all I would like to thank my supervisors, Dr. Masud Hasan and Dr. M. Sohel Rahman, for introducing me to the amazingly interesting world of bioinformatics and teaching me how to perform research work. Without their continuous supervision, guidance and valuable advice, it would have been impossible to complete the thesis. I am especially grateful to them for allowing me greater freedom in choosing the problems to work on, for their encouragement at times of disappointment, and for their patience with my wildly sporadic work habits. I am grateful to all other friends for their continuous encouragement and for helping me in thesis writing. I would like to express my gratitude to all my teachers. Their motivation and encouragement in addition to the education they provided meant a lot to me. Last but not least, I am grateful to my parents and to my families for their patience, interest, and support during my studies.

Contents

Chapter 1

Introduction

1.1 Genome Rearrangement

Genome rearrangement problems are widely studied in computational molecular biology to derive functional relationship between genes. One of the most studied genome rearrangement problems is to sort one genome sequence into another. Here the basic problem is, given two genome sequence, to transform one into another by using minimum number of pre specified rearrangement operations.

The study of evolutionary distance between two organisms using genomic data requires reconstruction of the sequence of evolutionary events that transform one genome into the other. Sequence comparison in computational molecular biology is a powerful tool for deriving evolutional and fundamental relationships among genes. But classical alignment algorithms take into account only local mutations (insertions, deletions and substitutions of nucleotides) and ignore global rearrangements (reversals, transpositions, fusions, fissions, etc. of long fragments). When estimating the evolutionary distance between two organisms using genomic data one wishes to reconstruct the sequence of evolutionary events that transformed one genome into

other. In the 1980's, evidence was found that some species have essentially the same set of genes, but their gene order differs [17, 25]. This suggest that global rearrangement events such as reversals and transposition of genome segments, can be used to trace the evolutionary path between genomes. As opposed to local point mutations(i.e., insertions, deletions and substitutions of nucleotides) global rearrangements are rare and may therefore provide more accurate and robust clues to the evolution.

Every genome rearrengement results in a change of gene ordering, and a series of these rearrangements can alter the genomic architecture of a species. Analyzing the rearrangement history of mammalian genomes is a challenging problem, even though a recent analysis of human and mouse genomes implies that fewer than 250 genomic rearrangements have occurred since the divergence of humans and mice approximately 80 million years ago. Every study of genome rearrangements involves solving the combinational puzzle of finding a series of rearrangements that transform one genome into another. The elementary rearrangement event can be reversal, transposition or block interchange etc.

Biologists are interested in the most parsimonious evolutionary scenario, that is, the smallest number of rearrangement events. While there is no guarantee that this scenario represents an actual evolutionary sequence, it gives us a lower bound on the number of rearrangements that have occurred and indicates the similarity between two species. For more background on genome rearrangement the reader is referred to [22, 12, 24].[1]

[1]In fact, a sequence of rearrangements such as reversals that transforms the X chromosome of mouse into the X chromosome of man does not even represent an evolutionary sequence, since humans are not descended from the present-day mouse. However, biologists believe that the architecture of the X chromosome in the human-mouse ancestor is about the same as the architecture of the human X chromosome.

1.2 Reversals and Transpositions

In the last decade, a large body of work was devoted to genome rearrangement problems. Genomes are represented by permutations, with the genes appearing as elements. Circular genomes (such as bacterial and mitochondrial genomes) are represented by circular permutations. The basic task is, given two permutations to find a shortest sequence of operations that transforms one permutation into other. Assuming that one of the permutations is the identity permutation, the problem is to find the shortest way of sorting a permutation using a given rearrangement operation, or set of operations.

While studying similarities among genome sequences of different species, biologists consider several rearrangement operations including reversals [23, 20, 3, 7], transpositions [19, 27], transreversals, fission and fusion etc. Among them, the problem of sorting permutations by reversals has been studied extensively. It was shown to be NP-hard [2], and several approximation algorithms have been suggested [28, 26, 11]. On the other hand, for signed permutations (every element of the permutation has a sign, + or -, which represents the direction of the gene) a polynomial time algorithm for sorting by reversals was first given by Hannenhalli and Pevzner [26]. Subsequent work improved the running time of the algorithm, and simplified the underlying theory [16, 1, 10, 15]

Like many other biology motivated problems, the problem of sorting permutations has deep theoretical and algorithmic nature, Within 1992-1995, this problem was first introduced to the algorithm community by Kececioglu and Sankoff [18], who studied the problem of sorting by reversals and by Bafna and Pevzner [28], who studied the problem of sorting by reversals and the problem of sorting by transpositions. Since then researchers continue to work on genome sorting and specially on sorting by reversals and transpositions. While Bafna and Pevzner were motivated by the biological application

of genome sorting, later the motivation was not confined only within biological domain. Rather, many variations of the problem were surfaced up due to their deep theoretical and algorithmic nature, until now they themselves continue to form a very rich and interesting class of computationally challenging problems.

Sorting permutations by transpositions is an important problem in genome rearrangements. There has been significant less progess on the problem of sorting by transpositions. 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.

1.2.1 Sorting Strings as a variant of Sorting Permutation

A natural variant of the sorting problems is to consider them not on permutations but on strings over fixed size alphabets. In the context of genome comparison, duplicate genes can occur, so that permutation model is not always the appropriate one. The shift from permutations to strings alters the problem universe somewhat. Indeed, papers by Christie and Irving [9], Radcliffe, Scott and Wilmer [6] and [5] explore the consequences of switching from permutations to strings; they both consider arbitrary (substring) reversals, and *transpositions*. It has been noted that, viewed as a whole, such

rearrangement operations on strings have bearing on the study of orthologus gene assignment [8], especially where the level of symbol repetition in the strings is low. There is also somewhat surprising link with the relatively unexplored family of *string partitioning* problems[4]. To put our work in context, we briefly describe the most relevant (for this thesis) results from [5].

In [5], *Grouping* a weaker form of sorting where identical symbols need only be grouped together, while the groups can be any order. For grouping on binary and ternary strings, they give a complete characterization of the minimum number of reversals(flips) required to group a string and provide polynomial-time algorithms for computing such an optimal sequence of flips. (The complexity of grouping over larger fixed size alphabets remains open but as an intermediate result they described how a PTAS can be constructed for each such problem.) Grouping aids in developing a deeper understanding of sorting which is why we tackle it first. It was also mentioned as a problem of interest in its own right by Eriksson et al. [14].

Chapter 2

Preliminaries

2.1 Definitions

In this section we will describe some of preliminaries regarding Prefix transpositions over a finite alphabet.

2.1.1 Alphabet

We start with the notion of an alphabet: a finite set of symbols. An example is, naturally, the English alphabet a,b,. . .,z. In fact, any object can be in an alphabet; from a formal point of view, an alphabet is simply a finite set of any sort. For simplicity, however, we use as symbols only letters, numerals, and other common characters such as $, or #. Binary alphabet can be written as 0,1 and ternary as 0,1,2.

2.1.2 String

A string over an alphabet is a finite sequence of symbols from the alphabet. 011011100 is string over binary alphabet 0,1, Note that, sting can have repeated characters while permutation do not allow repetitions.

2.1.3 Fully String

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 $[k]$.

2.1.4 Prefix

A prefix of string $T = t_1 \ldots t_n$ is $T^p = t_1 \ldots t_m$ where $m \leq n$. A proper prefix of a string is not equal to the string itself($0 \leq m < n$).The string *ban* is equal to a prefix of the string *banana*.

2.1.5 Transposition

A *transposition* involves swapping two adjacent substrings of permutation or string. More formally, the transposition $\pi(i, j, k)$ $(1 \leq i < j < k \leq n + 1)$ transforms the permutation π of $\{1, 2, \ldots, n\}$ into π^{t}, where

$$\pi^{\mathrm{t}}(m) = \begin{cases} \pi(m + j - i) & \text{if } i \leq m < i + k - j, \\ \pi(m - k + j) & \text{if } i + k - j \leq m < k, \\ \pi(m) & \text{otherwise.} \end{cases}$$

As an example, we can show a simple transposition operation below:

$$\left(\begin{array}{l} 1 \ldots i-1 \boxed{\textbf{i i+1 .....j-2 j-1}} \boxed{\text{j...k-1}} \, k \ldots n \\ 1 \ldots i-1 \boxed{\text{j...k-1}} \boxed{\textbf{i i+1 .....j-2 j-1}} k \ldots n \end{array} \right).$$

2.1.6 Prefix Transposition

In case of *prefix transposition*, between the two adjacent substrings , one is a prefix of that permutation or string. More formally, a prefix transposi-

tion $f(1, i, j)$, where $1 < i < j \leq (n + 1)$, is an rearrangement event that transforms s into $f\{s\} = [s[i] \dots s[j-1]s[1] \dots s[i-1]s[j] \dots s[n]]$.

As an example, we can show that

$$\left(\begin{array}{|l|l|l} \hline \textbf{1...i-1} & \text{i i+1 .....j-2 j-1} & j \dots n \\ \hline \text{i i+1 .....j-2 j-1} & \textbf{1...i-1} & j \dots n \\ \hline \end{array} \right).$$

2.1.7 Prefix transposition distance

The prefix transposition distance $d_s(s)$ of a string s is defined as the number of prefix transpositions required to sort the string.

2.1.8 Grouping

Grouping is weaker than sorting. In sorting symbols of a string need to be sorted. But in grouping, any order is acceptable. That means, all sorted string is a grouped string but not all grouped string is a sorted string. We give the example using ternary strings. Such as: 012 is both sorted and grouped. 102 is not sorted but grouped.

2.1.9 Grouping Distance

The grouping distance $d_g(s)$ of a fully k ary string s is defined as the minimum number of prefix transposition required to reduce the string to one of length k.

2.2 Equivalence Class

After a transposition operation is performed, the two adjacent symbols of the corresponding string may be 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 *reduction*. As representatives of the *equivalence classes* we take the shortest string of each class. Clearly, these are the strings where adjacent symbols always differ. The process of transforming a string into the representative string of its equivalence class is sometimes referred to as *normalization*. Therefore, the process of normalization basically comprises of repeated transposition and reduction.

As an example, let $s = baaabbabaabaaab$ and we want to apply the operation $f(1, 3, 7)$. Then,

$$
\begin{aligned}
& baaabbabaabaaab \\
\Rightarrow\ & \boxed{\mathbf{ba}}\ \boxed{\mathbf{aabb}}\ abaabaaab \\
\Rightarrow\ & \boxed{\mathbf{aabb}}\ \boxed{\mathbf{ba}}\ abaabaaab \\
\Rightarrow\ & aab\ \boxed{\mathbf{bb}}\ \boxed{\mathbf{aa}}\ baabaaab \\
\Rightarrow\ & aab\ \mathbf{ba}\ baabaaab
\end{aligned}
$$

2.2.1 2 transposition

We will denote a prefix transposition as a *2 transposition* if that operation decreases the string length by 2. We give an example. Let $s = 1012020120$. Then,

$$
\begin{aligned}
\Rightarrow\ & 1012020120 \\
\Rightarrow\ & \boxed{\mathbf{1012}}\ \boxed{\mathbf{0201}}\ 20 \\
\Rightarrow\ & \boxed{\mathbf{0201}}\ \boxed{\mathbf{1012}}\ 20 \\
\Rightarrow\ & 020\ \boxed{\mathbf{11}}\ 01\ \boxed{\mathbf{22}}\ 0 \\
\Rightarrow\ & 020\mathbf{1}01\mathbf{2}0
\end{aligned}
$$

2.2.2 1 transposition

We will denote a prefix transposition as a 1 transposition if that transposition operation decreases the string length by 1. Let $s = 10121$. Then,

$$
\begin{aligned}
&\Rightarrow 10121 \\
&\Rightarrow \boxed{\mathbf{1012}}\,\boxed{\mathbf{1}} \\
&\Rightarrow \boxed{\mathbf{1}}\,\boxed{\mathbf{1012}} \\
&\Rightarrow \boxed{\mathbf{11}}012 \\
&\Rightarrow \mathbf{1}012
\end{aligned}
$$

We give another example of 1 transposition where the first symbol occurs exactly once in the string. Let $s = 1020202$. Then,

$$
\begin{aligned}
&\Rightarrow 1020202 \\
&\Rightarrow \boxed{\mathbf{10}}\,\boxed{\mathbf{2}}\,0202 \\
&\Rightarrow \boxed{\mathbf{2}}\,\boxed{\mathbf{10}}\,0202 \\
&\Rightarrow 21\,\boxed{\mathbf{00}}202 \\
&\Rightarrow 21\mathbf{0}202
\end{aligned}
$$

2.2.3 0 transposition

When the string reaches to the length equal to its arity, we can not decrease it's length by giving any prefix transposition as the string is a fully string. But sometimes we need to perform such prefix transpositions. This is called 0 transposition. This is done mainly to order the string. Let $s = 120$. Then,

$$
\begin{aligned}
&120 \\
&\Rightarrow \boxed{\mathbf{12}}\,\boxed{\mathbf{0}} \\
&\Rightarrow \boxed{\mathbf{0}}\,\boxed{\mathbf{12}} \\
&\Rightarrow 012
\end{aligned}
$$

Chapter 3

Grouping

In this chapter we will discuss about grouping binary and ternary strings. We will find grouping distance for these strings and also provide algorithms, time complexity of the algorithms to find the grouping distance.

3.1 Purpose of Grouping

The task of sorting a string can be divided into two subproblems: *grouping* identical symbols together and putting the groups of identical symbols in the right *order*. Notice that first grouping and then ordering may not be the most efficient way to sort strings. Although grouping appears to be slightly easier than the sorting problem, essentially the same questions remain open as in sorting. Grouping binary strings is trivial and in Section 3.2 we discuss about that. In Section 3.3 we give the grouping distances of all ternary strings. As a result we give polynomial time algorithm for binary and ternary grouping in Section 3.4. For larger alphabets the grouping problem remains open. While the problems of grouping and sorting are closely related for strings on small alphabets, the problem diverge when alphabet size approaches the string length, with the permutation being the limit.

3.2 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.

3.2.1 Grouping distance for Binary strings

Theorem 3.2.1. (Bound for Binary strings) *Let s be a fully binary string. Then, $d_g(s) = \lceil \frac{n-k}{2} \rceil$.*

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 $d_g(s) = \lceil \frac{n-k}{2} \rceil$. □

We illustrate the above result with the help of examples. Let $s = 101010$. Here $|s|$ is even. So we will be able to give 2 transpositions each time. We can continue as follows:

$\Rightarrow 101010$
$\Rightarrow \boxed{\mathbf{10}}\ \boxed{\mathbf{1}}\ 010$
$\Rightarrow \boxed{\mathbf{1}}\ \boxed{\mathbf{10}}\ 010$
$\Rightarrow \boxed{\mathbf{11}}\ \boxed{\mathbf{00}}\ 10$
$\Rightarrow \mathbf{10}10$
$\Rightarrow \boxed{\mathbf{10}}\ \boxed{\mathbf{1}}\ 0$
$\Rightarrow \boxed{\mathbf{1}}\ \boxed{\mathbf{10}}\ 0$
$\Rightarrow \boxed{\mathbf{11}}\ \boxed{\mathbf{00}}$
$\Rightarrow \mathbf{10}$

Here, we need two 2 transpositions to group this string. So, $d_g(s) = 2$.

Now we take a odd length string s =0101010. In this case we need to

give one 1 transpositions. We show the detail analysis below:

$\Rightarrow 0101010$
$\Rightarrow \boxed{\mathbf{01}}\,\boxed{\mathbf{0}}\,1010$
$\Rightarrow \boxed{\mathbf{0}}\,\boxed{\mathbf{01}}\,1010$
$\Rightarrow \boxed{\mathbf{00}}\,\boxed{\mathbf{11}}\,010$
$\Rightarrow \mathbf{01}010$
$\Rightarrow \boxed{\mathbf{01}}\,\boxed{\mathbf{0}}\,10$
$\Rightarrow \boxed{\mathbf{0}}\,\boxed{\mathbf{01}}\,10$
$\Rightarrow \boxed{\mathbf{00}}\,\boxed{\mathbf{11}}\,0$
$\Rightarrow \mathbf{01}0$
$\Rightarrow \boxed{\mathbf{0}}\,\boxed{\mathbf{1}}\,0$
$\Rightarrow \boxed{\mathbf{1}}\,\boxed{\mathbf{0}}\,0$
$\Rightarrow \boxed{\mathbf{1}}\,\boxed{\mathbf{00}}$
$\Rightarrow 1\mathbf{0}$

Here, n=7 and we need 3 prefix transpositions (i.e two 2 transpositions and one 1 transpositions). Thus it holds the bound.

3.3 Grouping Ternary Strings

In the previous section, we gave the bound for grouping binary strings using prefix transpositions. 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 3.3.1. *In a fully ternary string, we can always give a 1 transposition.*

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

3.3.1 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 seen from Theorem 3.3.2 below, the upper bound differs. We first give an easy but useful lemma.

Theorem 3.3.2. (Bound for Ternary strings) *Let s be a fully ternary string. Then, $\lceil \frac{n-k}{2} \rceil \leq d_g(s) \leq \lceil \frac{n-k}{2} \rceil + 1$ where n is the length of the string and k is the arity.*

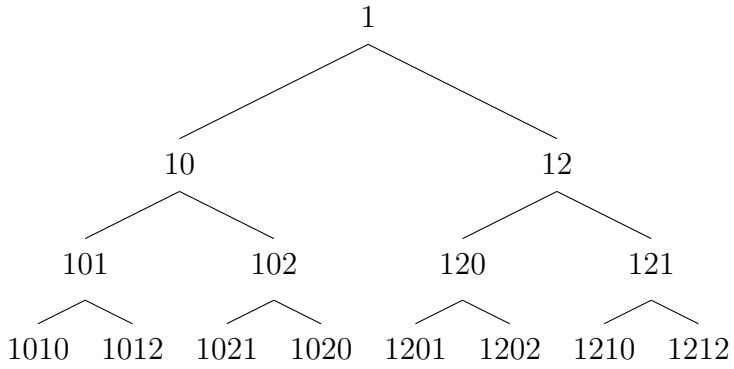
Proof. First we prove the lower bound. Here, $k = 3$. If we can always give a 2 transposition, then at each operation the string length is decremented by 2 and if $|s|$ is odd, we need an extra 1 transposition. Hence, we have $\lceil \frac{n-k}{2} \rceil \leq d_g(s)$.

Let us concentrate on the upper bound. As strings are fully ternary, we don't need to work with $n \leq 3$. Now, if we apply the upper bound for $n = 4, 5$ and 6, we have $d_g(s) = 2, 2$ and 3 respectively. For $n = 4$, we need only one 1 transposition. So, if 2 transpositions not exists, we can give 2,3 1 transpositions for $n = 5, 6$ respectively. Thus the upper bound is proved for $n < 7$.

Now we consider $n \geq 7$. 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 following eight prefixes of length 4 (see Figure 1):

$$1012, 1010, 1021, 1020, 1201, 1202, 1210 \text{ and } 1212. \tag{3.1}$$

Here we give the tree diagram of all strings starting with 1:



Now note that, the upper bound of Theorem 3.3.2 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)[1]. For a n length string, if we can give a 2 transposition, the resulted reduced string may start with 1,0 or 2. For the latter two cases, i.e., 0 or 2, 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 3.1. Hence, it suffices to prove the bound considering each of the prefixes of List 3.1. We will now follow the following strategy:

> We will take each of the prefixes of List 3.1 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

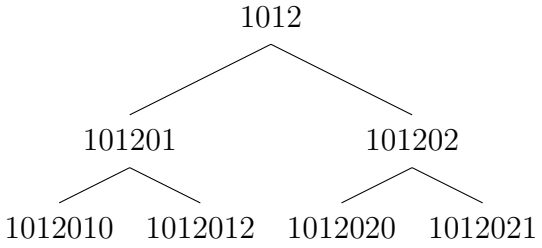
[1]If we could give a 2 transposition at each step, we would get the bound of $\lceil \frac{n-k}{2} \rceil$.

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 can't 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. Therefore, we can safely exclude them from the following discussion. In what follows, when we refer to the prefixes of List 3.1, we would actually mean all the prefixes excluding 1010 and 1212. Now, to expand the prefixes, if we add 10 or 12, all of them would be able to give a 2 transposition . Therefore, in what follows, we consider the other cases. Now we analyze each of the prefixes below.

1012

We first give the tree diagram of all string having prefix 1012



If we add 01, we can only add[2] 0 or 2. 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

[2]By 'add' we mean append.

becomes 1012020 or 1012021. The string 1012020 satisfies the bound as follows:

1012020
$\Rightarrow \boxed{\mathbf{1}}\ \boxed{\mathbf{01}}\ 2020$
$\Rightarrow \boxed{\mathbf{01}}\ \boxed{\mathbf{1}}\ 2020$
$\Rightarrow 0\ \boxed{\mathbf{11}}\ 2020$
$\Rightarrow 0\mathbf{1}2020$
$\Rightarrow \boxed{\mathbf{012}}\ \boxed{\mathbf{0}}\ 20$
$\Rightarrow \boxed{\mathbf{0}}\ \boxed{\mathbf{012}}\ 20$
$\Rightarrow \boxed{\mathbf{00}}\ 1\ \boxed{\mathbf{22}}\ 0$
$\Rightarrow \mathbf{0}1\mathbf{2}0$
$\Rightarrow \boxed{\mathbf{0}}\ \boxed{\mathbf{12}}\ 0$
$\Rightarrow \boxed{\mathbf{12}}\ \boxed{\mathbf{0}}\ 0$
$\Rightarrow 12\ \boxed{\mathbf{00}}$
$\Rightarrow 12\mathbf{0}$

Here, $n = 7$ and we need only 3 transpositions holding the bound true. Now if we add 1, the string becomes 10120201. For this one the bound holds

as follows:

10120201
$\Rightarrow$ $\boxed{\mathbf{1}}$ $\boxed{\mathbf{01}}$ 20201
$\Rightarrow$ $\boxed{\mathbf{01}}$ $\boxed{\mathbf{1}}$ 20201
$\Rightarrow$ 0 $\boxed{\mathbf{11}}$ 20201
$\Rightarrow$ 0**1**20201
$\Rightarrow$ $\boxed{\mathbf{01}}$ $\boxed{\mathbf{2020}}$ 1
$\Rightarrow$ $\boxed{\mathbf{2020}}$ $\boxed{\mathbf{01}}$ 1
$\Rightarrow$ 202 $\boxed{\mathbf{00}}$ $\boxed{\mathbf{11}}$
$\Rightarrow$ 202**01**
$\Rightarrow$ $\boxed{\mathbf{20}}$ $\boxed{\mathbf{2}}$ 01
$\Rightarrow$ $\boxed{\mathbf{2}}$ $\boxed{\mathbf{20}}$ 01
$\Rightarrow$ $\boxed{\mathbf{22}}$ $\boxed{\mathbf{00}}$ 1
$\Rightarrow$ **20**1

Here, n =8 and we needed 3 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.

Now we consider 1012021. The bound holds for this one as well as follows:

1012021
$\Rightarrow \boxed{\mathbf{1}}\ \boxed{\mathbf{01}}\ 2021$
$\Rightarrow \boxed{\mathbf{01}}\ \boxed{\mathbf{1}}\ 2021$
$\Rightarrow 0\ \boxed{\mathbf{11}}\ 2021$
$\Rightarrow 0\mathbf{1}2021$
$\Rightarrow \boxed{\mathbf{012}}\ \boxed{\mathbf{0}}\ 21$
$\Rightarrow \boxed{\mathbf{0}}\ \boxed{\mathbf{012}}\ 21$
$\Rightarrow \boxed{\mathbf{00}}\ 1\ \boxed{\mathbf{22}}\ 1$
$\Rightarrow 0121$
$\Rightarrow \boxed{\mathbf{01}}\ \boxed{\mathbf{2}}\ 1$
$\Rightarrow \boxed{\mathbf{2}}\ \boxed{\mathbf{01}}\ 1$
$\Rightarrow 20\ \boxed{\mathbf{11}}$
$\Rightarrow 20\mathbf{1}$

Now, strings like $1012(02)^*1$ can also be handled similarly hence the bound holds for them as well. Adding anything with $1012(02)^*1$ will also give a 2 transposition. Figure 2 shows the tree diagram. For the other cases, tree diagram is almost similar and hence we will not include them in other cases.

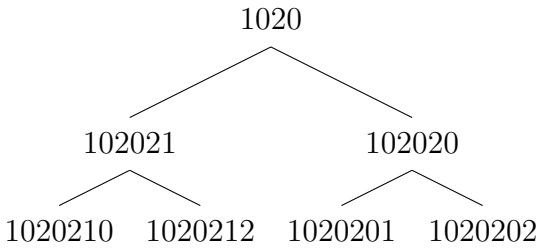
1021

This prefix is ending with 1 and adding anything will give a 2 transposition.

1020

We first add 21 with this prefix. Then, adding anything with 102021 will give a 2 transposition. 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 3.2.1). Also, adding anything with $102(02)^+1$ will give a 2 transposition.



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:

1020201
$\Rightarrow$ $\boxed{\mathbf{1}}$ $\boxed{\mathbf{02020}}$ 1
$\Rightarrow$ $\boxed{\mathbf{02020}}$ $\boxed{\mathbf{1}}$ 1
$\Rightarrow$ 02020 $\boxed{\mathbf{11}}$
$\Rightarrow$ 02020**1**
$\Rightarrow$ $\boxed{\mathbf{02}}$ $\boxed{\mathbf{0}}$ 201
$\Rightarrow$ $\boxed{\mathbf{0}}$ $\boxed{\mathbf{02}}$ 201
$\Rightarrow$ $\boxed{\mathbf{00}}$ $\boxed{\mathbf{22}}$ 01
$\Rightarrow$ **02**01
$\Rightarrow$ $\boxed{\mathbf{02}}$ $\boxed{\mathbf{0}}$ 1
$\Rightarrow$ $\boxed{\mathbf{0}}$ $\boxed{\mathbf{02}}$ 1
$\Rightarrow$ $\boxed{\mathbf{00}}$ 21
$\Rightarrow$ **0**21

All strings like $1020(20)^+1$ can also be handled similarly and hence the bound holds for them as well. Adding anything with $1020(20)^+1$ will give a

2 transposition.

1020201
$\Rightarrow \boxed{\mathbf{1}}\ \boxed{\mathbf{02020}}\ 1$
$\Rightarrow \boxed{\mathbf{02020}}\ \boxed{\mathbf{1}}\ 1$
$\Rightarrow \boxed{\mathbf{02020}}\ \boxed{\mathbf{11}}$
$\Rightarrow 020201$
$\Rightarrow \boxed{\mathbf{02}}\ \boxed{\mathbf{0}}\ 201$
$\Rightarrow \boxed{\mathbf{0}}\ \boxed{\mathbf{02}}\ 201$
$\Rightarrow \boxed{\mathbf{00}}\ \boxed{\mathbf{22}}\ 01$
$\Rightarrow \mathbf{02}01$
$\Rightarrow \boxed{\mathbf{02}}\ \boxed{\mathbf{0}}\ 1$
$\Rightarrow \boxed{\mathbf{0}}\ \boxed{\mathbf{02}}\ 1$
$\Rightarrow \boxed{\mathbf{00}}\ 21$
$\Rightarrow \mathbf{0}21$

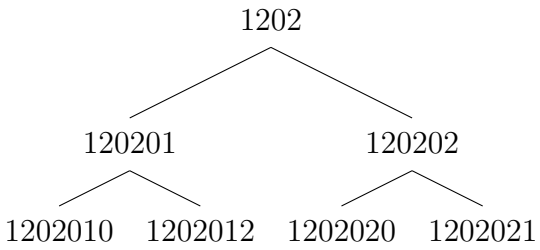
Now, string like $10202(02)^+$ can be handled similarly and hence the bound holds. For $10202(02)^+1$, we need a 1 transposition to move the initial 1 at the end. Now the upper bound holds because the rest of the string is binary (Theorem 3.2.1). Also adding anything with $10202(02)^+1$ will give a 2 transposition.

1201

This prefix is ending with a 1 and adding anything will give a 2 transposition.

1202

Tree diagram for strings having a prefix 1202 is given below:



We first add 01 with it. Then adding anything with 120201 will also give a 2 transposition. Now for 120(20)*1, we need a 1 transposition to move the initial 1 at the end and the rest of the string is binary. So, the upper bound holds. Also adding anything with 120(20)*1 will also give a 2 transposition. Now we add 02 with the prefix and get 120202. Next we add 1 or 0 and get 1202021 and 1202020 respectively. For 1202021

1202021
$\Rightarrow \boxed{\mathbf{1}}\ \boxed{\mathbf{20202}}\ 1$
$\Rightarrow \boxed{\mathbf{20202}}\ \boxed{\mathbf{1}}\ 1$
$\Rightarrow 20202\ \boxed{\mathbf{11}}$
$\Rightarrow 20202\mathbf{1}$
$\Rightarrow \boxed{\mathbf{20}}\ \boxed{\mathbf{2}}\ 021$
$\Rightarrow \boxed{\mathbf{2}}\ \boxed{\mathbf{20}}\ 021$
$\Rightarrow \boxed{\mathbf{22}}\ \boxed{\mathbf{00}}\ 21$
$\Rightarrow \mathbf{20}21$
$\Rightarrow \boxed{\mathbf{20}}\ \boxed{\mathbf{2}}\ 1$
$\Rightarrow \boxed{\mathbf{2}}\ \boxed{\mathbf{20}}\ 1$
$\Rightarrow \boxed{\mathbf{22}}\ 01$
$\Rightarrow \mathbf{2}01$

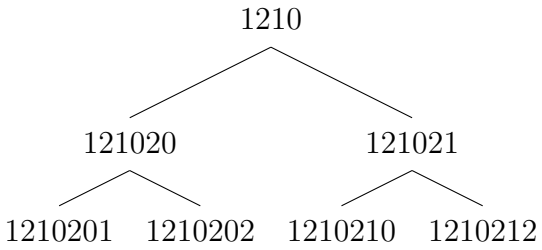
Here, $d_g(s) = 3$ for $n = 7$. All strings like $1202(02)^+1$ will also need similar transpositions to follow the bound and adding anything with $1202(02)^+1$ will also give a 2 transposition. Now for 1202020,

1202020
$\Rightarrow \boxed{\mathbf{12}}\,\boxed{\mathbf{0}}\,2020$
$\Rightarrow \boxed{\mathbf{0}}\,\boxed{\mathbf{12}}\,2020$
$\Rightarrow 01\,\boxed{\mathbf{22}}\,020$
$\Rightarrow 012020$
$\Rightarrow \boxed{\mathbf{012}}\,\boxed{\mathbf{0}}\,20$
$\Rightarrow \boxed{\mathbf{0}}\,\boxed{\mathbf{012}}\,20$
$\Rightarrow \boxed{\mathbf{00}}\,1\boxed{\mathbf{22}}\,20$
$\Rightarrow (0)1\mathbf{2}0$
$\Rightarrow \boxed{\mathbf{0}}\,\boxed{\mathbf{12}}\,0$
$\Rightarrow \boxed{\mathbf{12}}\,\boxed{\mathbf{0}}\,0$
$\Rightarrow 12\,\boxed{\mathbf{00}}$
$\Rightarrow 12\mathbf{0}$

Again $d_g(s) = 3$ for $n = 7$. All strings like $12020(20)^+$ will follow the similar steps. For $12020(20)^+1$, we need a 1 transposition to move the initial 1 at the end and rest of the string is binary. Now, if we add anything with $12020(20)^+1$, we can also give a 2 transposition.

1210

If we add 21, string becomes 121021, we next add 0 or 2 and get 1210210 or 1210212. In both cases, we can give 2 transposition. If we add 20, string becomes 121020,we next add 1 or 2 and get 1210201 or 1210202. For 1210202,



1210202
$\Rightarrow \boxed{\mathbf{1}}\ \boxed{\mathbf{2}}\ 10202$
$\Rightarrow \boxed{\mathbf{2}}\ \boxed{\mathbf{1}}\ 10202$
$\Rightarrow 2\ \boxed{\mathbf{11}}\ 0202$
$\Rightarrow 2\mathbf{1}0202$
$\Rightarrow \boxed{\mathbf{210}}\ \boxed{\mathbf{2}}\ 02$
$\Rightarrow \boxed{\mathbf{2}}\ \boxed{\mathbf{210}}\ 02$
$\Rightarrow \boxed{\mathbf{22}}\ 1\ \boxed{\mathbf{00}}\ 2$
$\Rightarrow \mathbf{2}1\mathbf{0}2$
$\Rightarrow \boxed{\mathbf{2}}\ \boxed{\mathbf{10}}\ 2$
$\Rightarrow \boxed{\mathbf{10}}\ \boxed{\mathbf{2}}\ 2$
$\Rightarrow 10\ \boxed{\mathbf{22}}$
$\Rightarrow 10\mathbf{2}$

$d_g(s) = 3$ for $n = 7$. Again adding a 1 we have, 12102021. Then,

12102021
$\Rightarrow \boxed{\mathbf{1}}\ \boxed{\mathbf{21}}\ 02021$
$\Rightarrow \boxed{\mathbf{21}}\ \boxed{\mathbf{1}}\ 02021$
$\Rightarrow 2\ \boxed{\mathbf{11}}\ 02021$
$\Rightarrow 2\mathbf{1}02021$
$\Rightarrow \boxed{\mathbf{210}}\ \boxed{\mathbf{2}}\ 021$
$\Rightarrow \boxed{\mathbf{2}}\ \boxed{\mathbf{210}}\ 021$
$\Rightarrow \boxed{\mathbf{22}}\ 1\ \boxed{\mathbf{00}}\ 21$
$\Rightarrow \mathbf{2}1\mathbf{0}21$
$\Rightarrow \boxed{\mathbf{21}}\ \boxed{\mathbf{02}}\ 1$
$\Rightarrow \boxed{\mathbf{02}}\ \boxed{\mathbf{21}}\ 1$
$\Rightarrow 0\ \boxed{\mathbf{22}}\ \boxed{\mathbf{11}}$
$\Rightarrow 0\mathbf{21}$

Here we have discussed for all the strings starting with 1. For any other string we can use relabeling to follow the bound in the abobe way. And this completes our proof. □

3.4 Algorithm to group fully binary and ternary strings

In this section we present the algorithm to group fully binary and ternary strings. We also analyze the running time later.

3.4.1 Description of the Algorithm

In this algorithm, we take all possible prefixes and try to find a 2 transposition in the suffix. If no 2 transposition is found (This will never occur in case of fully binary string), the algorithm gives a 1 transposition with the current prefix which is always possible according to 3.3.1. If the input string is fully binary, it runs until the length is 2 and if the input is fully ternary, runs until the length is 3. A variable count is initialized to zero and after the algorithm runs, it contains total number of prefix transpositions required.

Algorithm 2 performs 1 transpositions. As we have discussed in 2.2.2, there are 2 ways of performing a 1 transposition. If at Algorithm 1 we find that no 2 transposition is available with the current prefix, Algorithm 2 is called with the current string. Algorithm 2 performs a 1 transposition (which is always available as discussed in Lemma 3.3.1) and the returned string is assigned to the main string of Algorithm 1.

Algorithm 1 GroupByPrefixTransposition(s:input string)

Require: s is a fully binary or ternary string

```
initialize variables
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 if
    end for
    if twoTranspDone == false then
      perform a 1 transposition using Algorithm 2
      s ⇐ return value from 2
    end if
  end for
  twoTranspDone ⇐ false
end while
```

Algorithm 2 Do1Transposition(s:input string)

Require: 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
    count ⇐ count + 1
    forward ⇐ true
end if
if forward == false then
    for p = 1; p < (| s | −1); p++ do
        take a string suffix, substring of input string form (i + 1) to end
        if p^th symbol occurs at suffix then
            perform a 1 transposition
            count ⇐ count + 1
        end if
    end for
end if
return s
```

3.4.2 Time complexity analysis

There are 3 main loops in Algorithm 1. First one checks whether the string length reaches to the arity. If the string length reaches to its arity we don't need to proceed any further as we are dealing with fully strings. The second for loop supplies a prefix and the third loop checks whether any prefix transpositon is available or not. If not, then second loop supplies the next prefix and third loop checks again. We discuss with an example here: let we have a string s=1021212 which is a fully ternary string. Now, second loop will take 10 as a prefix and third loop will check whether it can perform a 2 transpositon. If yes, then the loop breaks there and computation goes on similar manner. If not, then the second loop will take 102 as prefix and then third loop will try to give a 2 transpositon again. In this case, the algorithm can perform a 2 transposition with prefix 102. By this way, all possible prefix are checked for a 2 transposition in this manner. If no two transposition can be given with any prefix, the boolean variable $twoTranspDone$ will be false and the algorithm will perform one 1 transposition (as it is always available according to 3.3.1) using Algorithm 2. Now we know from Section 2.2.2 that there are 2 kinds of 1 transposition available. Algorithm 2 gives one 1 transposition which is available first among the two type of 1 transposition. By this way, Algorithm 1 deals with the input string. As there are three main loops, we can say that the overall running time is $O(n^3)$.

Chapter 4

Sorting

In this Chapter we present results similar to those on grouping in the previous section. Grouping is an easier problem than sorting and we wanted to formulate grouping first as it will help us in sorting. The sorting distance $d_s(s)$ of a fully k ary string s is defined as the minimum number of prefix transposition required to sort the string to one of length k. We again consider normalized strings.

4.1 Sorting Binary Strings

Sorting binary string is straightforward. Strings can be sorted in any order: ascending or descending. We give the bound for sorting binary strings here.

Theorem 4.1.1. (Bound for Binary strings) *Let s be a fully binary string. Then, $d_s(s) \leq \lceil \frac{n-k}{2} \rceil$.*

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

4.2 Sorting Ternary Strings

We use Grouping to formulate ternary strings. We discuss the bounds in this section.

Theorem 4.2.1. (Bound for Ternary strings) *Let s be a fully Ternary string. Then, upper bound for sorting ternary string is $d_g(s) \leq \lceil \frac{n-k}{2} + 2 \rceil$.*

Proof. After grouping a ternary string, we have the following grouped strings for different fully ternary 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. We show the 0 transpositions for each case below:

For $s = 021$

$$\begin{aligned} &021 \\ &\Rightarrow \boxed{\mathbf{0}}\,\boxed{\mathbf{21}} \\ &\Rightarrow \boxed{\mathbf{21}}\,\boxed{\mathbf{0}} \\ &\Rightarrow 210 \end{aligned}$$

For $s = 102$

$$\begin{aligned} &102 \\ &\Rightarrow \boxed{\mathbf{10}}\,\boxed{\mathbf{2}} \\ &\Rightarrow \boxed{\mathbf{2}}\,\boxed{\mathbf{10}} \\ &\Rightarrow 210 \end{aligned}$$

For $s = 120$

$$\begin{aligned} &120 \\ &\Rightarrow \boxed{\mathbf{12}}\,\boxed{\mathbf{0}} \\ &\Rightarrow \boxed{\mathbf{0}}\,\boxed{\mathbf{12}} \\ &\Rightarrow 012 \end{aligned}$$

For $s = 201$

$$
\begin{aligned}
&201 \\
&\Rightarrow \boxed{\mathbf{2}}\,\boxed{\mathbf{01}} \\
&\Rightarrow \boxed{\mathbf{01}}\,\boxed{\mathbf{2}} \\
&\Rightarrow 012
\end{aligned}
$$

Hence the result follows. Now, this extra prefix transposition works on a string having length equal to its arity.□

4.3 Algorithm to sort fully binary and ternary strings

Now we discuss about the algorithm to sort binary and ternary strings.

4.3.1 Description of the Algorithm

We first apply Algorithm 1 of Section 3.4 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.

4.3.2 Time complexity analysis

A careful look at Algorithm 3 says us that we need only one 1 transposition more in case of ternary string. This 1 transposition will not reduce the string length, just arrange the symbols in correct order. Then, it can be done in constant time and thus the overall running time is $O(n^3)$, which is similar to the case of grouping.

Algorithm 3 SortByPrefixTransposition(s:input string)

Require: s is a fully binary or ternary string

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

Chapter 5

Conclusion and Future Works

5.1 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 [5], 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 $d_g(s) = \lceil \frac{n-k}{2} \rceil$ and for ternary string we have $\lceil \frac{n-k}{2} \rceil \leq d_g(s) \leq \lceil \frac{n-k}{2} \rceil + 1$, where n is the length of the string and k is the arity. On the other hand, for sorting binary and ternary strings the sorting distance $d_s(s)$ is upper bounded by $\lceil \frac{n-k}{2} \rceil$. and $\lceil \frac{n-k}{2} \rceil + 2$ respectively. We further have presented a polynomial time algorithm for grouping binary and ternary strings. Then we specify how to use the idea of grouping algorithm to sort strings.

5.2 Open problems and Future works

In this study we have unearthed many rich (and surprisingly difficult) combinational questions which deserves further analysis. We discuss some of

them here. The main unifying, "umbrella" suggestion is that, to go beyond ad-hoc (and case based) proof techniques, it will be necessary to develop deeper, more structural insights into the action of flips on strings over fixed size alphabets.

5.3 Grouping and sorting on higher arity alphabets

We have shown how to group and sort optimally binary and ternary strings, but characterizations and algorithms for quaternary (and higher) have not been discussed. Our algorithm for grouping in Section 3.4 works with all possible prefixes with a brute force approach. This gives an insight that for higher arity strings the algorithm should be similar. But to find the bound we need to think more deeper in the problem. Related problems include: for all fixed k, are there polynomial algorithms to optimally sort (optimally group) k-ary strings? Is grouping strictly easier than sorting, in a complexity issue? In [5], the authors asked how grouping will work under transpositions. In this thesis, we have discussed about that.

5.4 Signed strings

The problem of sorting signed permutations by prefix reversal (the *burnt* pancake problem) is well known [13, 29, 21], but in this thesis we have not yet attempted ti analyze the action of prefix transposition on signed, fixed size alphabet strings. Obviously, analogous of all the problems described in this paper exist for signed strings.

5.5 Complexity/approximation

In the presence of hardness results it is interesting to explore the complexity of restricted instances, and to develop algorithms with guaranteed approximation bounds. For example, [6] gives a PTAS for dense instances, the development of approximation algorithms is also a useful intermediate strategy where the complexity of a problem remains elusive. In particular, this requires the development of improved lower bounds.

Bibliography

[1] A.Bergeron. A very elementary presentation of the hannenhalli-pevzner theory. *In Combinationatorial Pattern Matching (CPM'01)*, pages 106–117, 2001.

[2] A.Capara. Sorting permutations by reverssals and eulerian cycle decompositions. *SIAM Journal on Discrete Mathematics*, 12(1):91, February 1999. 110.

[3] A.Capara and R.Rizzi. Improved approaximation for breakpoint graph decomposition and sorting by reversals. *In Proc. of 10th European Symposium on Algorithms*, 6(2):75–83, 2002.

[4] J.Zheng A.Goldstein, P.Kolman. Minimum common string partition problem: hardness and approaximations. *Electron. J. Combin*, 2005.

[5] Cor A.J.Hurkens, Leo van Iersel, Judith Keijsper, Steven Kelk, Leen Stougie, and John Tromp. Prefix reversals on binary and ternary strings. *SIAM J. Discrete Math.*, 21(3):592–611, 2007.

[6] A.J.Radcliffe and E.L.Wilmer A.D.Scott. Reversals and transpositions over finite alphabets. *SIAM J. Discrete Math.*, 2006.

[7] B.Chitturi and I.H.Sudborough. Bounding prefix transposition distance for strings and permutations. *In Proc. of 41st Hawaii International Conference on Systems Science*, 6(2):468, 2008.

[8] Xin Chen, Jie Zheng, Zheng Fu, Peng Nan, Yang Zhong, Stefano Lonardi, and Tao Jiang. Assignment of orthologous genes via genome rearrangement. *IEEE/ACM Trans. Comput. Biology Bioinform.*, 2(4):302–315, 2005.

[9] David A. Christie and Robert W. Irving. Sorting strings by reversals and by transpositions. *SIAM J. Discrete Math.*, 14(2):193–206, 2001.

[10] B.M.E.Moret D.A Bader and M.Yan. A linear-time algorithm for computing inversion distance between signed permutations with an experimental study. *Journal of Computational Biology,*, 8(5):438–491, 2001.

[11] D.A.Christie. Genome rearrangement problems. *Phd thesis, University of Glasgow*, 1999.

[12] D.Sankoff and N.El-Mabrouk. Genome rearrangement. in current topics in computational molecular biology. *MIT Press*, 2002.

[13] M.Blum D.S.Cohen. On the problem of sorting burnt pancakes. *Discrete Appl. Math*, 61(2):105–120, 1995.

[14] Henrik Eriksson, Kimmo Eriksson, Johan Karlander, Lars J. Svensson, and Johan Wästlund. Sorting a bridge hand. *Discrete Mathematics*, 241(1-3):289–300, 2001.

[15] E.Tanier and M.F.Sagot. Sorting by reversals in subquadratic time. *In Combinatorial Pattern Matching (CPM'04)*, 3109:1–13, 2004.

[16] R.Shamir H.Kaplan and E.Tarjan R. Faster and simpler algorithm for sorting signed permutations by reversals. *SIAM Journal of Computing*, 29(3):880, 2000. 892.

[17] J.D.Palmer and L.A.Herbon. Tricircular mitochondrial genomes of brassica and raphanus: reversal of repeat configurations by inversion. *Nucleic Acids Research*, 14:9755–9764, 1986.

[18] J.Kececioglu and D.Sankoff. Exact and approaximation algorithms for the inversions distance between two permutations. *In Proc. of 4th Annual Symposium on Combinatorial Pattern Matching, (CPM'93)*, 684:87–105, 1995.

[19] L.Elias and T.Hartman. A 1.375-approaximation for sorting by transpositions. *In Proc. of 5th International Workshop on Algorihms in Bioinformatics*, pages 204–214, October 2005.

[20] M.Sagot M.D.V.Braga, C.Scornavacca, and E.Tannier. The solution space of sorting by reversals. *In Proc. of 10th European Symposium on Algorithms*, pages 293–304, 2007.

[21] I.H.Sudborough M.H.Heydari. On the diameter of the pancake network. *J. Algorithms*, 25:67–94, 1997.

[22] P.A.Pevzner. Computational molecular biology: An algorithmic approach. *MIT Press*, 2000.

[23] S.Hannanhalli P.Berman and M.Karpinski. 1.375-approaximation algorithm for sorting by reversals. *In Proc. of 10th European Symposium on Algorithms (ESA'02)*, 25(2):200–210, 1996. Springer, 2002. LNCS 2461.

[24] R.Shamir. Algorithms in molecular biology: Lecture notes. 2002. Available at http://www.math.tau.ac.it/ rshamir/algmb/01/algmb01.html.

[25] S.B.Hoot and J.D.Palmer. Structural rearrangements, including parallel inversions, within the chloroplast genome of anemone and related genera. *J. Molecular Evolution*, 38(274-281), 1994.

[26] S.Hannenhalli and P.Pevzner. Transforming cabbage into turnip: Polynomial algorithm for sorting signed permutations by reversals. *Journal of the ACM*, 46(1-27), 1999.

[27] T.Hartman and R.Sharan. A simpler 1.5 approaximation algorithm for sorting by transpositions and transreversals. *In Proc. of 4th International Workshop on Algorihms in Bioinformatics*, pages 50–61, 2004.

[28] V.Bafna and P.A.Pevzner. Genome rearrangements and sorting by reversals. *SIAM Journal on Computing*, 25(2):272–289, 1996.

[29] W.H.Gates and C.H.Papadimitriou. Bounds for sorting by prefix reversal. *Discrete Math.*, (27):47–57, 1979.

Index