M.Sc. Engg. Thesis

GENOME REARRANGEMENT
OPERATIONS ON
PERMUTATIONS AND STRINGS

by

Md. Khaledur Rahman
Student No.: 0413052039

Submitted to

Department of Computer Science & Engineering
in partial fulfillment of the requirements for the degree of
Masters of Science in Computer Science and Engineering

Department of Computer Science & Engineering
Bangladesh University of Engineering & Technology (BUET)
Dhaka-1000, Bangladesh

April 2016
Declaration of Authorship

This is to certify that the work presented in this thesis titled "GENOME REARRANGEMENT OPERATIONS ON PERMUTATIONS AND STRINGS" is the outcome of the investigation carried out by me under the supervision of Dr. M. Sohel Rahman, Professor, Department of Computer Science and Engineering (BUET), Dhaka. It is also declared that neither this thesis nor any part thereof has been submitted or being currently submitted anywhere else for the award of any degree or diploma.

Md. Khaledur Rahman
Student No.: 0413052039
Department of Computer Science and Engineering (BUET)
Dhaka-1000, Bangladesh
The thesis titled "GENOME REARRANGEMENT OPERATIONS ON PERMUTATIONS AND STRINGS" submitted by Md. Khaledur Rahman, Roll No. 0413052039, Session April 2013, to the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, has been accepted as satisfactory in partial fulfillment of the requirements for the degree of Master of Science in Computer Science and Engineering and approved as to its style and contents. Examination held on April 20, 2016.

Dr. M. Sohel Rahman  
Professor & Head  
Department of Computer Science and Engineering  
BUET, Dhaka 1000

Dr. Mohammad Kaykobad  
Professor  
Department of Computer Science and Engineering  
BUET, Dhaka 1000

Dr. Mohammed Eunus Ali  
Professor  
Department of Computer Science and Engineering  
BUET, Dhaka 1000

Dr. Swakkhar Shatabda  
Assistant Professor  
Department of Computer Science and Engineering  
United International University, Dhaka 1209
Acknowledgements

I show my heartfelt gratitude toward my supervisor Dr. M. Sohel Rahman, who was very helpful during the entire span of my research work. Without his direction, support and advice, this work would not have been possible. I am especially grateful to him for allowing me greater freedom in choosing the problems to work on, for his encouragement at times of disappointment, and for his patience with my wildly sporadic work habits.

I would like to thank Bangladesh University of Engineering and Technology for its generous support. The university also provided me with its library facilities and online resource facilities. I would also like to thank Department of Computer Science and Engineering for its support with resources and materials during the research work.

I also remember my teachers and colleagues who earnestly provided me with encouragement and inspiration for achieving this goal. Last but not the least, I am thankful to my parents, family and friends for their support and tolerance.
Abstract

The problem of sorting by a genome rearrangement event asks for the minimum number of that event required to sort the elements of a given permutation. In this thesis, we study some variants of the rearrangement event called prefix and suffix transreversal, prefix and suffix block-interchange, and prefix and suffix versions of transposition and reversal. A transreversal is an operation which reverses the first block before exchanging two adjacent blocks in a permutation. A prefix (suffix) transreversal always reverses and moves a prefix (suffix) of the permutation to another location. Interestingly, we will apply transreversal not on permutations but on strings over an alphabet of fixed size. We determine the upper bound for prefix and suffix transreversals required to sort the binary and ternary strings, with polynomial time algorithms for these sorting problems. Then we focus on applying prefix block-interchanges on binary and ternary strings. A block-interchange operation exchanges two blocks of a permutation which are not necessarily adjacent and in a prefix block-interchange, one block is always the prefix of that permutation. We present the upper bounds to group and sort a given binary/ternary string. We also provide upper bounds for a different version of block-interchange operation which we call ‘restricted prefix block-interchange’. When we apply restricted prefix block-interchange operations on binary strings, we observe that our obtained upper bound is better than that of other genome rearrangement operations to group fully normalized binary strings. Consequently, we provide a linear-time algorithm to solve the problem of grouping binary normalized strings by restricted prefix block-interchanges. We also provide a polynomial time algorithm to group normalized ternary strings. We provide a classification for ternary strings. Finally, we study the variations of transposition and reversal events which we call prefix and suffix breakpoint removal by transposition and reversal operations, and present a new (2 + \( \epsilon \))-approximation algorithm for this problem. We propose a new sufficient condition to improve approximation ratio for sorting an unsigned permutation using genome rearrangement events. We also present experimental results to support our analyses with a naive polynomial time algorithm which runs in \( O(n^2) \), where \( n \) is the number of elements in the permutation. We observe from experimental results over both real genomic data and simulated genome (in silico) data that the approximation ratio of our algorithm is far better under the proposed conditions. We further present an improved implementation technique to reduce the running time of our proposed algorithm.
The contributions of this research are either published or under review or prepared to submit to journals and conferences as mentioned in the following list:


## Contents

1 Introduction ........................................... 1
   1.1 Genome Rearrangement on Strings .................. 3
   1.2 Genome Rearrangement on Permutation ................. 4
   1.3 Summary of Contributions .......................... 6
   1.4 Organization of the Thesis ........................ 7
      1.4.1 Preliminaries ............................... 7
      1.4.2 Sorting Strings By Prefix and Suffix TransReversals ........................ 7
      1.4.3 Sorting Strings By Prefix Block-Interchanges ............. 8
      1.4.4 Sorting Permutation By Genome Rearrangement Operations ........................................ 8
      1.4.5 Conclusion ................................ 8

2 Preliminaries ........................................... 9
   2.1 Notations for String Problem ....................... 9
   2.2 Notations for Permutation Problem ................. 12

3 Sorting Strings By Prefix and Suffix Transreversals .... 15
   3.1 Grouping ......................................... 15
      3.1.1 Grouping Binary Strings ...................... 15
      3.1.2 Grouping Ternary Strings .................... 18
      3.1.3 Grouping distance for Ternary strings ........ 18
   3.2 Sorting ........................................... 32
      3.2.1 Sorting Binary Strings ....................... 32
      3.2.2 Sorting Ternary Strings ..................... 32
   3.3 Classification ................................... 33

4 Sorting Strings by Prefix Block-Interchanges ............ 36
   4.1 Grouping and Sorting Binary Strings ................. 36
      4.1.1 Grouping Binary Strings by Restricted Prefix Block-interchanges ......................... 38
   4.2 Grouping and Sorting Ternary Strings ................. 41
4.3 Classification .................................................. 51

5 Sorting Permutation by Genome Rearrangement Operations 53
  5.1 Scenarios of the Breakpoint Graph ....................... 53
  5.2 The \((2 + \epsilon)\)-Approximation Algorithm ............... 57
  5.3 Improved Approximation Algorithms with forward march ... 58
      5.3.1 Conditions to Improve the Approximation Ratio ..... 59
  5.4 Experimental Results ...................................... 64

6 Conclusion .................................................... 71
  6.1 Findings .................................................. 71
  6.2 Future Direction .......................................... 72
List of Figures

2.1 Breakpoint graph $G(\pi)$ of $\pi = [0 \ 2 \ 6 \ 3 \ 1 \ 4 \ 7 \ 3 \ 5 \ 8]$ ........... 14
3.1 Tree diagram of 4 length strings (placed at leaf nodes) having prefix 1. ......................... 20
3.2 Tree diagram of 7 length strings (placed at leaf nodes) having prefix 1010. ..................... 22
3.3 Tree diagram of 7 length strings (placed at leaf nodes) having prefix 1012. ..................... 23
3.4 Tree diagram of 7 length strings (placed at leaf nodes) having prefix 1020. ..................... 24
3.5 Tree diagram of 7 length strings (placed at leaf nodes) having prefix 1021. ..................... 25
4.1 Tree diagram for normalized ternary strings of length 4 starting with 1 ......................... 42
4.2 Tree diagram of 7 length strings (placed at leaf nodes) having prefix 1012. ..................... 45
4.3 Tree diagram of 7 length strings (placed at leaf nodes) having prefix 1020. ..................... 46
4.4 Tree diagram of 7 length strings (placed at leaf nodes) having prefix 1021. ..................... 47
5.1 Different scenarios of breakpoint graphs. ....................... 54
5.2 Breakpoint graph $G(\pi)$ of $\pi = [0 \ 1 \ 2 \ 3 \ 5 \ 4 \ 8 \ 7 \ 6 \ 9 \ 10]$ and its sorting steps. ............... 55
5.3 Linear changes of required number of operations that remove two breakpoints at each step with total number of breakpoints. 63
5.4 Average approximation ratio of randomly generated data for different permutation size. ........... 65
5.5 Average approximation ratio of real genome data for different permutation size. ...................... 66
5.6 Average approximation ratio of real genome data for different breakpoints. .............................................. 67
5.7 # of breakpoints for different size of permutations. .............. 67
5.8 Running time analyses for simple implementation and improved implementation of our proposed algorithm. ........... 68
List of Tables

1.1 A list of results on grouping and sorting strings by Genome Rearrangement (GR) operations. Here, \( n \) is the length of the string. Other notations are as follows: \( X \)-Not Available, \( PR \)-Prefix Reversal, \( PT \)-Prefix Transposition, \( PSTR \)-Prefix and Suffix TransReversal, \( PBI \)-Prefix Block-Interchange and \( RPBI \)-Restricted Prefix Block-Interchange. . . . . . . . . . . 4

4.1 This table shows the derivation of the grouping distance for normalized ternary strings of length 5 and 6 deduced from List (4.1). \( n \) is the length of normalized ternary strings and \( d_{pbi}^g(s) \) is the grouping distance. Here, binary strings like 101010 or 121212 are omitted. . . . . . . . . . . . . . . . . . . . . . . . . 44

4.2 This table shows the derivation of the grouping distance for normalized ternary strings of length 5 and 6 deduced from List (4.1). \( n \) is the length of normalized ternary strings and \( d_{rpbi}^g(s) \) is the grouping distance applying restricted prefix block-interchanges. Here, binary strings like 101010 or 121212 are omitted. . . . . 52

5.1 Minimum number of operations \( (k_1, k_2, k_3, k_4) \) that removes two breakpoints at each step for different values of \( b(\pi) \). \( k_1 \) represents required number of operations to acquire approximation ratio 1.667, \( k_2 \) represents required number of operations to acquire approximation ratio 1.5, \( k_3 \) represents required number of operations to acquire approximation ratio 1.375 and \( k_4 \) represents required number of operations to acquire approximation ratio 1.25. . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
Chapter 1

Introduction

One of the most promising ways to trace evolutionary distance between two species is to compare the order of appearances of identical genes in two different genomes. Nowadays, the goal of the scientists is to explain better the evolutionary history of a set of species. Interestingly, although the organizations of the molecules of different species differ dramatically, the content of the DNA molecules from one species to another are found to be often similar. Genome rearrangements refer to the mutations that affect this organization. Now, the role of computer scientists (or bioinformaticians), in this regard, lies basically in the efficient formulation of the evolutionary events to a combinatorial problem and then to provide an efficient solution for it. More specifically, a computer scientist would use mathematical model and combinatorial tools to reconstruct rearrangement scenarios to explain better the evolutionary history of a set of species.

Genome rearrangement has become a popular combinatorial problem in the field of computational biology which helps to trace the evolutionary distance between two species. The genome is structurally specific to each species, and it changes only slowly over time. Though the gene content of two genomes is almost identical, gene order can be quite different which results in different species [6]. In addition, we can apply genome rearrangement to find genetic disorder in case of cancer or any other diseases. We can also use genome rearrangement operations to apply drug on specific point and
to identify location of gene that is responsible for disorder. For example, gene implicated in the waardenburg’s syndrome disease was linked to human chromosome 2 but it was not clear where exactly it is located on chromosome 2. A breed of mice (with splotch gene) had similar symptoms caused by the same type of gene as in humans. Scientists succeeded in identifying location of gene responsible for disorder in mice. Finding the gene in mice gives clues to where the same gene is located in humans. So these are some applications which motivate us to work in computational field of genome rearrangement. There are several genome rearrangement events such as reversal, transposition, block-interchanges, etc. [27, 28, 5] that contribute much to such biological diversity. In this thesis, we will discuss variations of reversal and transposition, and block-interchanges.

Given a permutation, a transposition operation involves swapping two adjacent substrings of it while a reversal involves reversing a single substring. A transreversal operation is another biologically motivated operation that combines a transposition and a reversal. In the transreversal operation, before executing a transposition operation between the adjacent substrings, the first substring is cut out of the permutation and reversed. Then it is swapped with the adjacent substring. In a prefix transreversal, one of the two substrings must be a prefix of the permutation. The transreversal distance between two permutations (and the related operations of transreversals) may be used to estimate the number of global mutations between genomes and by molecular biologists to infer evolutionary and functional relationships.

In addition to rearrangements restricted to the prefix of a permutation, it is also possible to consider their suffix version. It is reasonable to believe that it is more probable to break a genome at one point rather than at two or more. Besides, if this happens, either the first or the second part could be reversed; thus, characterizing the prefix/suffix reversals. The same analogy can be used for the prefix/suffix transpositions since it would require breaking a genome at two points, which happens with less probability, but it is still easier than breaking at three points, as a transposition would. However, notice that if a problem involves only prefix rearrangements, there is no need to study a problem that allows only the suffix versions of the same rearrangements,
since they are equivalent. Hence, this paper will study the problem of sorting a permutation by transreversals involving both prefix and suffix versions of them. Formally, *Sorting by prefix and suffix transreversal* is the problem of finding the minimum number of prefix and suffix transreversals needed to transform a given permutation into the identity permutation.

A prefix block-interchange is a variation of the block-interchange operation where one of the two blocks must be a prefix of the permutation. The prefix block-interchange distance between two permutations can be used to estimate the number of global mutations between genomes and can be used by molecular biologists to infer evolutionary and functional relationships. *Sorting by prefix block-interchanges* is the problem of finding the minimum number of prefix block-interchange operations needed to transform a given permutation into the *identity* permutation.

### 1.1 Genome Rearrangement on Strings

A natural variant of the previously mentioned sorting problems is to apply genome rearrangement operations not on permutations but on strings over fixed size alphabets. A string over the alphabet $\sum$ is a sequence of symbols where repetitions of symbols in $\sum$ are allowed. This shift is inspired by the biological observation that multiple “copies” of the same gene can appear at various places along the genome [6]. In general, the grouping problem of normalized strings provides upper bound i.e., the number of required operations to group all symbols of a string and the sorting problem computes the number of operations required to sort all grouped symbols of that normalized string by genome rearrangement operations.

Indeed, recent works by Christie and Irving [27], Radcliffe et al. [1], Hurkens et al. [3] and Dutta et al. [2] have explored 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 [13], especially if the strings have only low level of symbol repetition. We provide a summary of results on sorting strings by genome rearrangement operations in Table 1.1.
Table 1.1: A list of results on grouping and sorting strings by Genome Re-arrangement (GR) operations. Here, \( n \) is the length of the string. Other notations are as follows: \( X \)-Not Available, \( PR \)-Prefix Reversal, \( PT \)-Prefix Transposition, \( PSTR \)-Prefix and Suffix TransReversal, \( PBI \)-Prefix Block-Interchange and \( RPBI \)-Restricted Prefix Block-Interchange.

Chen et al. [13], presented polynomial-time algorithms for computing the minimum number of reversals and transpositions operations to sort a given binary string. They also gave exact constructive diameter results on binary strings. Radcliffe et al. [1] on the other hand gave refined and generalized reversal diameter results for non fixed size alphabets. Hurkens et al. [3] introduced grouping (a weaker form of sorting), where identical symbols need only to 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. Their proposed upper bound for grouping binary string is \( n - 2 \), where \( n \) is the length of the binary string \( s \). Subsequently, Dutta et al. [2] followed up their work [3] on binary and ternary strings to apply prefix transpositions introducing relabeling. They gave a complete characterization of the minimum number of prefix transpositions required to group (and sort) binary and ternary strings. Their proposed upper bound to group a binary string is \( \lceil \frac{n - 2}{2} \rceil \), where \( n \) is the length of the string. It may be noted that, apart from being a useful aid for sorting, grouping itself is a problem of interest in its own right [7]. Chou
et al. also proposed a bound only for binary strings applying prefix block-interchanges \[14\]. Their proposed bound to group binary strings is \(\lceil \frac{n-2}{3} \rceil\). They also proposed a linear time algorithm to sort binary strings.

1.2 Genome Rearrangement on Permutation

Now, we move from strings to permutations. Hannenhalli and Pevzner showed that sorting signed permutations by reversals can be solved in polynomial time \[20\]. A plethora of works regarding this operation exist in the literature where improved running time of the algorithm and simplified underlying theory are presented \[21\]. To date, sorting a signed permutation by reversals can be solved in subquadratic time \[22\]. On the other hand, sorting by reversals for unsigned linear permutations is NP-Hard \[23\]. The currently best known algorithm has an approximation ratio of 1.375 \[24\]. Meidanis et al. have shown that sorting by reversals for circular permutations is essentially equivalent to the same for linear permutations \[25\].

There are several 1.5-approximation algorithms proposed for the sorting by transpositions \[26, 27, 28\]. Elias and Hartman improved the approximation ratio to 1.375 \[29\]. Hartman and Shamir have proved that sorting by transposition for circular permutations is equivalent to the same for linear permutations \[28\]. The 1.5-approximation algorithm as well as the 1.375-approximation algorithms for sorting by transpositions have also been improved to run in \(O(n \log n)\) time \[30\]. Recently, Bulteau, Fertine and Rusu have shown that sorting by transposition problem is also NP-Hard \[32\]. In the literature, there exists a work of Firoz et al. on sorting by transposition where authors have presented an algorithm which runs in \(O(n \log n)\) having approximation ratio 1.375 \[31\]. Very recently Cunha et al. have disproved their algorithm \[33\].

A number of variations (prefix reversal, suffix transposition, etc.) and different combinations (transposition and translocation, transposition and reversal, etc.) of rearrangement events have also been contributed to the literature. Walter, Dias and Meidanis have given a 2-approximation algorithm for sorting signed linear permutations by reversals and transpositions \[34\].
Gu and Peng have given a 2-approximation algorithm for sorting signed linear permutations by transpositions and transreversals [35]. Lin and Xue have introduced the operation \( \text{revrev} \), and given a 1.75-approximation algorithm for sorting signed linear permutations by reversals, transpositions, transreversals and \( \text{revrevs} \) [36]. Hartman and Sharan have given a 1.5-approximation algorithm for sorting signed circular permutations by transpositions and transreversals [37]. Their algorithm can be used to sort linear permutations if the operation \( \text{revrev} \) is allowed. For unsigned linear permutations, Walter et al. have given a 3-approximation algorithm which allows reversals and transpositions [34]. After that Rahman, Shatabda and Hasan ([38]) have proposed a \( (4 - \frac{2}{\ell}) \)-approximation algorithm and improved the result to \( 2.5909 + \delta \) by putting the best known value of \( k \) from [41].

In the literature, the problems of sorting by different rearrangement operations have been tackled from at least two different perspectives. In one direction, researchers have been trying to improve the approximation ratio and in the other, efforts have been made to improve the running time of the algorithm keeping the approximation ratio intact. In this thesis, we are concerned with both directions. To the best of our knowledge, for sorting by prefix transpositions, prefix reversals, suffix transpositions and suffix reversals, the best-known algorithms have an approximation ratio of 2 [15, 17, 18].

### 1.3 Summary of Contributions

In this thesis, we follow up the works of [2, 3, 14] and consider prefix and suffix transreversal operations, and prefix block-interchange operations to group and sort binary and ternary strings. Notably, as a future work in [3], the authors raised the issue of considering other genome rearrangement operators. In particular, here, we find the minimum number of prefix and suffix transreversals required to group and sort binary or ternary strings. Then we find the number of prefix block-interchanges required to group and sort binary or ternary strings. In addition, we also present restricted prefix block-interchange operations to group binary strings. We deduce that when we apply restricted prefix block-interchanges, the upper bound is \( \lceil \frac{n-2}{4} \rceil \) to group fully binary
normalized string $s$, where $n$ is the length of the string. We clearly see that this bound is better than other proposed bounds for other rearrangement operations in the literature. We also provide a linear-time algorithm to group a binary normalized string by restricted prefix block-interchanges. Then, we deduce the upper bound, $\lceil \frac{n-2}{2} \rceil$ to group fully normalized ternary strings by prefix block-interchanges, where $n$ is the length of the string. We also provide a polynomial time algorithm to group normalized ternary strings. We briefly present the classification of normalized ternary strings. Finally, we present a sufficient condition to improve approximation ratio for sorting unsigned permutations by genome rearrangement operations. We also discuss different scenarios of breakpoint graph and propose a new approximation ratio where we restrict our focus on removing prefix and suffix breakpoint by applying transposition and reversal operations. Then we deduce an algorithm using unsophisticated data structure to run our proposed technique in $O(n^2)$ time, where $n$ is the size of permutation. We also provide experimental results to support our analyses using both real genomic data and simulated data. We further discuss an improved implementation technique to reduce the running time.

1.4 Organization of the Thesis

A brief overview of the organization of the rest of this thesis is presented in this section.

1.4.1 Preliminaries

Genome rearrangement is an important field of computational biology to trace evolutionary distance between two species. Before going in depth, it is quite necessary to understand the biological context so that they can smartly be applied in computational context. In Chapter 2, we discuss the preliminary concepts and several genome rearrangement operations. We also explain some notations that have been used throughout the paper.
1.4.2 Sorting Strings By Prefix and Suffix TransReversals

In Chapter 3, we discuss the applications of prefix and suffix transreversal operations on binary and ternary strings. We deduce upper bounds for both binary and ternary strings applying prefix and suffix transreversal operations and prove the corresponding bounds to group and sort fully normalized binary and ternary strings. We also provide several polynomial time algorithms to group and sort both binary and ternary strings. Then we provide classification of ternary strings.

1.4.3 Sorting Strings By Prefix Block-Interchanges

In Chapter 4, we discuss the applications of prefix block-interchange operations on binary and ternary strings. We deduce upper bounds for both binary and ternary strings applying prefix block-interchange operations and prove the corresponding bounds to group and sort fully normalized binary and ternary strings. We introduce a restricted version of prefix block-interchange and show that we achieve better upper bound for normalized binary strings. We provide several polynomial time algorithms to group and sort both binary and ternary strings. We also present a brief classification of the strings based on the relevant equivalence relation.

1.4.4 Sorting Permutation By Genome Rearrangement Operations

We discuss the application of genome rearrangement operations on permutation in Chapter 5. Here, we show several scenarios of breakpoint graph that help to reduce approximation ratio of sorting permutation by genome rearrangement operations. Then we deduce some suitable conditions to achieve particular approximation ratio algorithms. We also show experimental results to support our deduced conditions.
1.4.5 Conclusion

We conclude in Chapter 6 briefing our contributions. Some future research directions are also identified and briefly discussed.
Chapter 2

Preliminaries

In this chapter, we discuss some terminology, concepts and relevant notations that are used throughout the thesis. The terms discussed here, belong to both biological and algorithmic studies. The discussion in this chapter will help the readers to grasp the significance and importance of our study and will aid in properly comprehending the meaning of studied problem which are discussed throughout the subsequent chapters.

2.1 Notations for String Problem

We follow the notations and definitions used in [2] and [3], 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 transreversal $f_p(1, x, y)$ on a string $s$ of length $n$, where $1 < x < y \leq (n+1)$, is a rearrangement event that transforms $s$ into $s[x] \ldots s[y-1]s[x-1] \ldots s[1]s[y] \ldots s[n]$. Similarly, a suffix transreversal $f_s(x+1, y+1, n+1)$ on a string $s$ of length $n$, where $1 \leq x < y < (n+1)$, is a rearrangement event that transforms $s$ into $s[1] \ldots s[x]s[n] \ldots s[y]s[x+1] \ldots s[y-1]$. The prefix transreversal distance $d_p(s)$ of $s$ is defined as the number of prefix
transreversals required to sort the string and the suffix transreversal distance $d_s(s)$ of $s$ is defined as the number of suffix transreversals required to sort the string. We define the prefix and suffix transreversal distance $d_{ps}(s)$ of $s$ as the number of prefix and suffix transreversals required to sort the string. We also define another notation called $d_{gps}(s)$ which indicates the total number of pairs of prefix and suffix transreversals required to sort the string. For binary strings, we count a pair of prefix-suffix tranreversal as one operation when a prefix transreversal is bound to be followed by a suffix transreversal. But when a prefix transversal may be followed by either of the operations, we count prefix transversal separately.

We define block-interchange $\beta(w, x, y, z)$ on a string $s$ of length $n$, where $1 \leq w \leq x < y \leq z \leq n$ and $s = [s[1] \ldots s[w] \ldots s[x] \ldots s[y] \ldots s[z] \ldots s[n]]$, as a rearrangement event that transforms $s$ into $[s[1] \ldots s[w−1]s[y] \ldots s[z]s[x+1] \ldots s[y−1]s[w] \ldots s[x]s[z+1] \ldots s[n]]$. When we consider prefix block-interchange, we actually perform $\beta(1, x, y, z)$. So, a prefix block-interchange event transforms $s$ into $[s[y] \ldots s[z]s[x+1] \ldots s[y−1]s[1] \ldots s[x]s[z+1] \ldots s[n]]$. The prefix block-interchange distance $d_{psb}(s)$ of $s$ is defined as the minimum number of prefix block-interchanges required to sort the string $s$. Similarly, we define $d_{gps}(s)$ as the minimum number of prefix block-interchanges required to group all symbols of the string $s$. Note that, after a prefix block-interchange operation is performed, the two adjacent symbols of the corresponding string may be identical.

Note that, after a genome rearrangement operation has been performed, the two adjacent symbols of the corresponding string may be identical. We refer to this phenomenon as ‘duplicating’. We consider two strings to be equivalent if one can be transformed into the other by 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. A string having no consecutive identical 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. In our work, we only consider normalized strings. This does not lose generality because any string can be easily
converted to the respective normalized string, i.e., the representative of the class it belongs to. Let us explain the *equivalence class* with an example. Suppose, we have the strings: 111001, 1111101 and 1111000011111. Now, if we remove the consecutive identical characters then all the strings will give us 101. So, these three strings belong to the same *equivalence class* and 101 is the representative string of this class, because it is the shortest string thereof. Therefore, the equivalence relation here is that if we remove the adjacent identical symbols, all strings of a class reduce to the same normalized string (i.e., the representative of the class). We remark that the concepts of normalization, reduction and equivalence class have been borrowed from the previous works of [2, 3].

Here, we provide an example to demonstrate the scenario after performing a rearrangement operation on a binary string. Let us consider \( s = 1010101 \) and we want to apply a prefix transreversal operation \( f_p(1, 3, 5) \). Now, \( s[x] \ldots s[y - 1] = s[3] \ldots s[4] = 10, \ s[x - 1] \ldots s[1] = s[2] \ldots s[1] = 01, \ s[y] \ldots s[u] = s[5] \ldots s[7] = 101 \). Therefore, after applying the operation, we get, \( s = s[3] \ldots s[4]s[2] \ldots s[1]s[5] \ldots s[7] = \overline{1010101} = 1001101 = 10101 \).

A reduction that decreases the string length by \( l \) after applying a prefix or a suffix transreversal is called an \( l \)-transreversal. So, if \( l = 0 \), then we have a 0-transreversal. The above example illustrates a 2-transreversal. Similarly, a reduction that decreases the string length by \( l \) after applying a prefix block interchange is called an \( l \)-pblockInterchange. So, if \( l = 0 \), then we have a 0-pblockInterchange.

A *token* is a string of one or more characters that is significant as a group. Every element of a *token* will be treated as a *symbol*. A *quantifier* is a type of determiner that indicates quantity and it is basically applied after a *token*. Such as ‘+’ is a *quantifier* which indicates the one or more occurrences of the *token* after which ‘+’ is applied. An *extension* refers to a string where a *quantifier* is not applied. Consider the string (10) + 201. Here, 10 is a *token*, ‘1’ and ‘0’ are symbols of the token 10, ‘+’ is the *quantifier* and 201 is an *extension*. 

12
2.2 Notations for Permutation Problem

In this section, we briefly present the notions and notations that would be used throughout the report. Notably, most of these notations follow from [18, 19].

Let \( \pi = [\pi_0, \pi_1, \ldots, \pi_n, \pi_{n+1}] \) be a permutation of \( n + 2 \) distinct elements where \( \pi_0 = 0, \pi_{n+1} = n + 1 \) and \( 1 \leq \pi_i \leq n \) for each \( 1 \leq i \leq n \) (the middle \( n \) elements of \( \pi \) are to be sorted). A transposition \( \tau = \tau(i, j, k) \) for some \( 1 \leq i < j \leq n + 1 \) and some \( 1 \leq k \leq n + 1 \) such that \( k \notin [i, j] \) cuts the elements \( \pi_i, \ldots, \pi_{j-1} \) and pastes between \( \pi_{k-1} \) and \( \pi_k \) and thus transforms \( \pi \) into permutation \( \pi \cdot \tau = [\pi_0, \ldots, \pi_{i-1}, \pi_i, \pi_{i+1}, \ldots, \pi_{j-1}, \pi_k, \ldots, \pi_{n+1}] \) when, \( k > j \) or, into permutation \( \pi \cdot \tau = [\pi_0, \ldots, \pi_{k-1}, \pi_i, \pi_{i+1}, \ldots, \pi_{j-1}, \pi_k, \ldots, \pi_{i-1}, \pi_j, \ldots, \pi_{n+1}] \) when, \( k < i \). A reversal \( \beta = \beta(i, j) \) for some \( 1 \leq i < j \leq n + 1 \) applied to \( \pi \) reverses the elements \( \pi_i, \ldots, \pi_{j-1} \) and thus transforms \( \pi \) into permutation \( \pi \cdot \beta = [\pi_0, \ldots, \pi_{i-1}, \pi_{j-1}, \ldots, \pi_i, \pi_j, \ldots, \pi_{n+1}] \).

We now define an important concept related to permutations, the breakpoint [16]. A breakpoint for this problem is a position \( i \) of a permutation \( \pi \) such that \( |\pi_i - \pi_{i-1}| \neq 1 \), for \( 1 \leq i \leq n + 1 \). We denote by \( b(\pi) \) the number of breakpoints in a permutation \( \pi \). An identity permutation, \( \iota_n \) has no breakpoint. Note that \( \pi = [0, n, n - 1, n - 2, \ldots, 1, n + 1] \) has exactly two breakpoints. A strip is a subsequence \( \pi(i \ldots j) \) of \( \pi \) where \( i \leq j \) such that \( i \) and \( j + 1 \) are breakpoints and there are no breakpoints between positions \( i \) and \( j \). We will not consider first (last) element (\( \pi_1 \) or \( \pi_n \)) to perform an operation rather we will consider the position of first (last) breakpoint (i.e., after (before) first (last) strip) to apply transposition or reversal operations. A \((a, b)\)-move represents that applying first operation on \( \pi \) removes \( a \) breakpoints and the second operation removes \( b \) breakpoints consecutively.

A prefix breakpoint removal transposition \( \tau_p = \tau_p(x, j, k) \) for some \( 2 \leq j \leq n, |\pi_x - \pi_{x+1}| \neq 1 \) and some \( x \leq k \leq n + 1 \) such that \( k \notin [x, j] \) and \( x \geq 1 \) cuts the elements \( \pi_x, \ldots, \pi_{j-1} \) and pastes between \( \pi_{k-1} \) and \( \pi_k \) and thus transforms \( \pi \) into permutation \( \pi \cdot \tau_p = [\pi_0, \ldots, \pi_{x-1}, \pi_j, \ldots, \pi_{k-1}, \pi_x, \ldots, \pi_{j-1}, \pi_k, \ldots, \pi_{n+1}] \). Similarly, a prefix breakpoint removal reversal \( \beta_p = \beta_p(x, j) \) for some \( x \leq j \leq n + 1 \) and \( |\pi_x - \pi_{x+1}| \neq 1 \), applied to \( \pi \) reverses the elements \( \pi_x \ldots \pi_{j-1} \) and
thus transforms \( \pi \) into \( \pi \cdot \beta_p = [\pi_0, \ldots, \pi_{x-1}, \pi_{j-1}, \ldots, \pi_x, \pi_j, \ldots, \pi_{n+1}] \). For both of these operations, \( x \) is the position of first breakpoint in \( \pi \).

A suffix breakpoint removal transposition \( \tau_s = \tau_s(i, j, y) \) for some \( 1 \leq i \leq y-1, |\pi_y - \pi_{y+1}| \neq 1 \) and some \( 2 \leq j \leq y \) such that \( i \notin [j, y] \) and \( y \leq n \) cuts the elements \( \pi_{j+1}, \ldots, \pi_y \) and pastes between \( \pi_i \) and \( \pi_{i+1} \) and thus transforms \( \pi \) into permutation \( \pi \cdot \tau_s = [\pi_0, \ldots, \pi_i, \pi_{j+1}, \ldots, \pi_y, \pi_{i+1}, \ldots, \pi_j, \pi_{y+1}, \ldots, \pi_{n+1}] \).

Similarly, a suffix breakpoint removal reversal \( \beta_s = \beta_s(j, y) \) for some \( 2 \leq j \leq n \) and \( |\pi_y - \pi_{y+1}| \neq 1 \), applied to \( \pi \) reverses the elements \( \pi_{j+1}, \ldots, \pi_y \) and thus transforms \( \pi \) into permutation \( \pi \cdot \beta_s = [\pi_0, \ldots, \pi_j, \pi_y, \pi_{y-1}, \ldots, \pi_{j+2}, \pi_{j+1}, \pi_{y+1}, \ldots, \pi_{n+1}] \). For both of these operations, \( y \) is the position of last breakpoint in \( \pi \).

An identity permutation \( \iota_n \) is a permutation such that \( \pi_i = i \) for all \( 1 \leq i \leq n \). Given two permutations, the problem of transforming one permutation into another is equivalent to the problem of sorting a given permutation to the identity permutation. The prefix and suffix breakpoint removal transpositions and reversals distance \( d(\pi) \) between \( \pi \) and \( \iota_n \) is the minimum number of operations such that \( \pi \cdot o_1 \cdot o_2 \cdot \ldots \cdot o_d(\pi) = \iota_n \), where each operation \( o_i \) is either a prefix breakpoint removal transposition \( \tau_p \), a prefix breakpoint removal reversal \( \beta_p \), a suffix breakpoint removal transposition \( \tau_s \) or a suffix breakpoint removal reversal \( \beta_s \). The prefix or suffix breakpoint removal problem is as follows: given a permutation \( \pi \), find a combination of shortest sequence of prefix breakpoint removal transposition, prefix breakpoint removal reversal, suffix breakpoint removal transposition and suffix breakpoint removal reversal such that permutation \( \pi \) transforms into the identity permutation \( \iota_n \), i.e., finding the distance \( d(\pi) \). Note that we always get the first or prefix breakpoint after the first strip and last or suffix breakpoint before the last strip.

The breakpoint graph \( G(\pi) \) is an undirected multigraph whose \( n+2 \) vertices are \( \pi_i \) for \( 0 \leq i \leq n+1 \). \( G(\pi) \) has \( 2b(\pi) \) edges and they are of two types: gray and black. For each \( 0 \leq i \leq n \), the vertices \( \pi_i \) and \( \pi_{i+1} \) are joined by a black edge if \( |\pi_i - \pi_{i+1}| \neq 1 \). For \( 0 \leq i, j \leq n+1 \), there is a gray edge between \( \pi_i \) and \( \pi_j \) if \( \pi_i = \pi_j + 1 \).

In this thesis, we draw the vertices of \( G(\pi) \) horizontally in order \( \pi_0, \pi_1, \ldots, \)
\[ \pi_{n+1} \text{ from left to right. We show black edges by horizontal lines and the gray ones by dotted arcs (see Figure 2.1).} \]
Chapter 3

Sorting Strings By Prefix and Suffix Transreversals

In this chapter, we discuss grouping and sorting of normalized binary and ternary strings. First, we deduce grouping distance of both binary and ternary normalized strings for prefix and suffix transreversal operations where we present some algorithms in case of ternary strings. Second, we deduce sorting distance of both binary and ternary normalized strings. Finally, we provide classification of a large set thereof comprising infinite number of ternary normalized strings.

3.1 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 $d_{ps}^{s}(s)$ of a fully $k$ ary string $s$ is defined as the minimum number of prefix and suffix transreversals required to reduce the string to one of length $k$.

3.1.1 Grouping Binary Strings

A binary string can be of either even or odd length. As strings are normalized, only 4 kinds of binary strings are possible, namely, 01010...010, 10101...101
for odd length and 01010…101, 10101…010 for even length. The grouping of binary strings seems to be quite easy and obvious. The following bounds are easily achieved.

**Lemma 1.** (Distance for binary strings of prefix transreversals) Let $s$ be a fully binary normalized string and $|s| > 2$. Then, $d_p^g(s) = \lceil \frac{n-k}{2} \rceil$, when $n$ is odd and $d_p^g(s) = \lceil \frac{n-k}{2} \rceil + 1$, when $n$ is even, $k$ is the arity and for binary string, $k = 2$.

**Proof.** We can always have a 2-transreversal if $|s| > 4$. When $|s| = 3$, we need an extra 1-transreversal. But when $|s| = 4$, we need two 1-transreversals. For all the other cases (i.e. string length $|s| > 4$), we will get at least one 2-transreversal which will reduce the string length by 2. As a result we will recursively get the derived prefix transreversal distance. So, the distance is $d_p^g(s) = \lceil \frac{n-k}{2} \rceil$, when $n$ is odd and $d_p^g(s) = \lceil \frac{n-k}{2} \rceil + 1$, when $n$ is even. \hfill \Box

Following the procedure of Lemma 1, we can deduce a similar distance for the suffix transreversal distance.

**Lemma 2.** For a fully binary normalized string $s$, $|s| > 2$, $d_s^g(s) = \lceil \frac{n-k}{2} \rceil$, when $n$ is odd and $d_s^g(s) = \lceil \frac{n-k}{2} \rceil + 1$, when $n$ is even, $k$ is the arity and for binary string, $k = 2$. \hfill \Box

Now, when both of the prefix transreversal and suffix transreversal operations are present, we apply only one of them in each step on binary normalized string. So, we can easily deduce the following bound for binary normalized strings from Lemma 1 and Lemma 2.

**Corollary 1.** For a fully binary normalized string $s$, $|s| > 2$, $d_p^g(s) = \lceil \frac{n-k}{2} \rceil$, when $n$ is odd and $d_p^g(s) = \lceil \frac{n-k}{2} \rceil + 1$, when $n$ is even, $k$ is the arity and for binary string, $k = 2$. \hfill \Box

We illustrate the result of Lemma 1 with the help of two examples. We first consider a string of odd length. Let $s = 10101$. Then we can continue as follows: $s = 10101 = \overline{101}0101 = 1$ $0011$ $01 = 10101 = \overline{101}101 = 1$ $00$ $1 = 101 = \overline{101} = 0$ $11 = 01$. Here, we need two prefix 2-transreversals
and one prefix 1-transreversal to group this string. So, \( d^p(s) = 3 \). We present another example of fully binary string having even length. Let, \( s = 10101010 \). Then we can continue as follows:

\[
10101010 = 1010 = 1100 = 0110 = 0011 = 10 = 01 = 010 = 01010 = 01110 = 101010 = 10.
\]

Here, we need three prefix 2-transreversals and one prefix 1-transreversal to group this string. So, \( d^p(s) = 4 \). Now, we concentrate on applying both prefix and suffix transreversals on binary strings simultaneously. This is an observation for fully normalized binary string that if both prefix transreversals and suffix transreversals are performed in tandem (also called pair) then \( d_{gps}(s) \) will be minimized. For simplicity, we can consider that every prefix transreversal operation is performed with a reduction process followed by a suffix transreversal operation with a reduction process i.e., prefix transreversal and suffix transreversal are dependent on each other. But sometime, this is restricted by some conditions of the strings. We do not need to perform a suffix transreversal after the reduction process of a prefix transreversal when length of the binary string, \(|s| < 4\).

**Theorem 1.** (Distance of binary strings for prefix and suffix transreversals)

Let \( s \) be a fully binary normalized string and \( n = |s| \). Then, for \( n > 3 \),

\[
d_{gps}(s) = \left\lceil \frac{n-k}{4} \right\rceil + 1, \text{ when } n \text{ is odd and } d_{gps}(s) = \left\lfloor \frac{n-k}{4} \right\rfloor + 2, \text{ when } n \text{ is even,}
\]

\( k \) is the arity. For binary string, \( k = 2 \).

**Proof.** When we perform a suffix transreversal after performing a prefix transreversal each time, then combinedly, we get a 4-transreversal for string length, \( n > 8 \). For string length, \( n = 7 \) and \( n = 8 \), we get 3-transreversal and for string length, \( n = 5 \) and \( n = 6 \), we get 2-transreversal. For string length, \( n = 4 \), we need two 1-transreversals. This defines the distance for a binary string of even length having \( n > 3 \), which is \( d_{gps}(s) = \left\lceil \frac{n-k}{4} \right\rceil + 2 \).

But for string length, \( n = 3 \), we will need one 1-transreversal. This defines the distance for binary string of odd length having \( n > 3 \), which is \( d_{gps}(s) = \left\lfloor \frac{n-k}{4} \right\rfloor + 1 \).

\( \square \)
3.1.2 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 3.** In a fully ternary normalized string, we can always perform a 1-transreversal.

*Proof.* We take a ternary string $s$ of length $n > 3$. We can represent string $s$ as $s = s[1] \ldots s[l] \ldots s[i] \ldots s[n]$, here it is obvious that $1 \leq l < i \leq n$. Now, we take $s[1] \ldots s[l]$ as a prefix. As this will reverse, if $s[l]$ occurs at the suffix at position $i - 1$, we can transpose $s[l] \ldots s[1]$ with $s[l + 1] \ldots s[i - 1]$. Then, $s[l]$ and $s[i - 1]$ become adjacent and we can eliminate one of the two. Otherwise, if $s[1]$ occurs at the suffix at position $i$ (or $i - 1$), then we can transpose $s[1]$ with $s[2] \ldots s[i - 1]$. Then $s[1]$ and $s[i]$ ($s[i - 1]$) are adjacent and as before, we can eliminate one of the two. Since, one of the above cases always occurs for fully ternary strings, the result follows. 

Now, we assume that we can perform a 2-transreversal on a string. Note that, 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-transreversal. Thus, the presence of a 2-transreversal always ensures that there is also a 1-transreversal. For example, consider 10201. There is a 2-transreversal: 10201 $\Rightarrow$ 201. We could also get a 1-transversal as follows: 01201 $\Rightarrow$ 1201.

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

**Lemma 4.** Suppose $s[1 \ldots n]$ is a fully ternary normalized string. If we have a prefix $s[1 \ldots i], 1 \leq i < j \leq n$ such that $s[1] = s[j]$ and $s[i] = s[j - 1]$, then we have a 2-transreversal.
Proof. We have a fully normalized ternary string $s$. After performing a prefix transreversal $f_p(1, i, j)$ on $s$, $s[1]$ will be adjacent to $s[j]$ and $s[i]$ will be adjacent to $s[j-1]$. Then, we will be able to eliminate one of $s[1]$ or $s[j]$ and one of $s[j]$ or $s[i]$. This ensures that $s$ will be reduced by length 2 which completes the proof.

Following the procedure of Lemma 4, we can deduce a similar condition for the suffix transreversal distance.

**Lemma 5.** Suppose $s[1\ldots n]$ is a fully ternary normalized string. If we have a suffix $s[i\ldots n], 1 \leq i < j \leq n$ such that $s[n] = s[i]$ and $s[j] = s[i+1]$, then we have a 2-transreversal. □

From Lemma 4 and Lemma 5, we can easily deduce a trivial case that it is not possible to have a 3-transreversal in a fully binary or ternary normalized string. Now, we will concentrate on performing prefix and suffix transreversals. In this context, we will not perform a suffix transreversal after each prefix transreversal. Rather, we will apply suffix transreversal as and when appropriate i.e., both prefix transreversal and suffix transreversal will be applied independently.

**Theorem 2.** (Bound for ternary strings for prefix and suffix transreversals) Let $s$ be a fully ternary normalized string. Then, $\lceil \frac{n-k}{2} \rceil \leq d_{ps}(s) \leq \lceil \frac{n-k+1}{2} \rceil + 1$ where $n$ is the length of the string, $k = 3$ is the arity.

Proof. To prove this Theorem, we will first show that the given bound holds true for all ternary strings of length $n < 7$. Then, we will show that the bound also holds true for all ternary strings of length $n \geq 7$. For the lower bound, we consider the best cases where we will be able to give 2-transreversals in every step of the reduction. If $|s|$ is odd, the string length will reduce to 3 (e.g. if $|s| = 7$, we get $7 \Rightarrow 5 \Rightarrow 3$). If $|s|$ is even, string length will reduce to 4 and we will require an additional 1-transreversal. (e.g. for $|s| = 8$, with a 2-transreversal at each step, we get $8 \Rightarrow 6 \Rightarrow 4$ and a final 1-transreversal gives us the length 3 string).

Let us now concentrate on the upper bound. As strings are fully ternary, we don’t need to work with $n \leq 3$. For $n = 4$, we require a 1-transreversal and
that is optimal, hence $d_{ps}^0(s) = 1$, and the upper and lower bound is satisfied. 

Now, if we apply the upper bound for $n = 5$ and 6, we have $d_{ps}^0(s) = 3$ and 3 respectively. It is easy to realize that, by Lemma 3, we can always satisfy the given upper bound. 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:

1012, 1010, 1021, 1020, 1201, 1202, 1210 and 1212 (3.1)

We give the tree diagram of length 4 prefixes of all strings starting with ‘1’ in Figure 3.1.

The upper bound of Theorem 2 tells us that we will need to give at most three 1-transreversals when $n$ is even (i.e., $n - k$ is odd) and two 1-transreversals when $n$ is odd (i.e., $n - k$ is even). Note that, if we could give a 2-transreversal at each step, we would get the bound of $\lceil \frac{n-k}{2} \rceil$. For a $n$ length string, if we can give a 2-transreversal, the resulting reduced string may start with 1, 0 or 2. For the latter two cases, we can again use relabeling as mentioned before. Therefore we can safely assume 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 show lists for 7 length fully normalized ternary strings each of which will have a 4 length prefix.
from List (3.1). Then we will demonstrate the proof with examples for each of the prefixes from List (3.1). We will now approach 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. For the information, we will not consider all possible strings; rather we will continue to expand until we get a 2-transreversal, since afterwards, any further expansion would also guarantee a 2-transreversal. Let $s$ be one such string. We will take $s$ and try to give a 2-transreversal 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-transreversal, we specifically deal withs and show that the bound holds. Now if we can give a 2-transreversal, 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 are binary strings. So they take two prefix and suffix transreversals (Theorem 1). Therefore, we can safely exclude one of them from the following discussion. Now, if we relabel “0” to “2” and “2” to “0”, then we get 1201 from 1021. Similarly, after relabeling we get 1202 from 1020 and 1210 from 1012. So, we can safely exclude these also 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 1201, 1202, 1210 and 1212. 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.

1010

We first give the list of all strings of length 7 having prefix 1010 (see List (3.2) and tree diagram in Figure 3.2). The possible 5 length strings having this prefix are 10101 and 10102.

1010101, 1010102, 1010120, 1010121, 1010201, 1010202, 1010210 and 1010212 (3.2)
Here, all strings will have at least one 1-transreversal. When we have a prefix (suffix) 10, we need to lookup somewhere for a suffix (prefix) 01. Then we will be able to perform a 2-transreversal. This is also true for all the prefixes (suffixes) that has the form 1...0. Similarly, when we have a prefix (suffix) 1...2, we need to lookup somewhere for a suffix (prefix) 21. Then we will be able to perform a 2-transreversal (Lemma 5). If we add any symbol of ‘0’, ‘1’ or ‘2’ to a string of List (3.2), then we will be able to apply a 2-transreversal (Lemma 5). Thus the bound holds true.

Let us check this with examples. The string 1010121 satisfies the bound as follows: $\overline{1010121} \Rightarrow 10121 \Rightarrow 1012\overline{1} \Rightarrow 1012 \Rightarrow \overline{1012} \Rightarrow 012$. We see that, for $n = 7$, we need only three transreversals holding the bound true. However, if we plan to reduce the string length by at least 1 after each step, we may not achieve the desired bound. Let us see an example. Here, all strings satisfy the bound except 1010202. The reduction steps are as follows: $\overline{1010202} \Rightarrow 010202 \Rightarrow 0102\overline{0} \Rightarrow 01020 \Rightarrow 01\overline{0}20 \Rightarrow 0210 \Rightarrow 02\overline{1}0 \Rightarrow 012$. This takes four steps which violates the bound. But if we start with a 0-transreversal it is possible to have two consecutive 2-transreversals as follows: $10\overline{10202} \Rightarrow 1020102 \Rightarrow 10201\overline{0} \Rightarrow 10201 \Rightarrow \overline{10201} \Rightarrow 201$. Here, it needs three steps to group the ternary string holding the bound true. If we add 0 or 1 after 1010202, then the resulting string will be 10102020 or 10102021 in which case we will be able to perform 2-transreversal. So, adding any
length of ternary string(s) after 1010202 will also have 2-transreversal in the reduction steps.

1012

Here, again we first give the list of 7 length strings having prefix 1012 (see List (3.3) and corresponding tree diagram in figure 3.3). The possible 5 length strings of this prefix are 10120 and 10121.

1012010, 1012012, 1012020, 1012021, 1012101, 1012102, 1012120 and 1012121

In this case, we see that all strings having length, |s| = 7, will have 2-transreversal. When we have a prefix (suffix) 10, we need to lookup somewhere for a suffix (prefix) 01. Similarly, when we have a prefix (suffix) 102, we need to lookup somewhere for a suffix (prefix) 21. Then we will be able to perform a 2-transreversal which will reduce the string length by 2. Then according to Lemma 3, we will get at least two consecutive 1-transreversals which will satisfy the bound. This is also true for all the prefixes (suffixes) that has the form 1...0 or 1...2.

Let us illustrate this with an example. The string 1012120 satisfies the bound as follows: $1012120 \Rightarrow 10212 \Rightarrow 10212 \Rightarrow 2012 \Rightarrow 2012 \Rightarrow 012$. Here,
Figure 3.4: Tree diagram of 7 length strings (placed at leaf nodes) having prefix 1020.

\( n = 7 \) and we need only three transreversals holding the bound true. If we see carefully, here is another exception that needs a 0-transreversal first and it is 1012020. It satisfies the bound as follows: \( 1012020 \Rightarrow 1020120 \Rightarrow 102012 \Rightarrow 10201 \Rightarrow 1020 \Rightarrow 201 \). So we can say, if we add any length of ternary string(s) after 1012020, that will also have 2-transreversal. Basically, longer length of ternary string increases the number of 2-transreversal.

1020

Let us see the list of 7 length strings having prefix 1020 in List (3.4) and tree diagram in Figure 3.4.

\[
1020101, 1020102, 1020120, 1020121, 1020201, 1020202, 1020210 and 1020212
\]

Here, according to Lemma 4, all strings will have a 2-transreversal. When we have a prefix (suffix) 10, we need to lookup somewhere for a suffix (prefix) 01. This is also true for all the prefixes (suffixes) of the form 1...0. Similarly, when we have a prefix (suffix) of the form 1...2, we need to lookup somewhere for a suffix (prefix) 21. Then we will be able to perform a 2-transreversal. After that according to Lemma 9, the bound will be satisfied. Note that, if we add any symbol of ‘0’, ‘1’ or ‘2’ to a string of List (3.4), then
Figure 3.5: Tree diagram of 7 length strings (placed at leaf nodes) having prefix 1021.

We will also be able to apply a 2-transreversal. Thus the bound holds true.

1021

We will see the list of 7 length strings having prefix 1021 in List (3.5) and tree diagram in Figure 3.5.

1021010, 1021012, 1021020, 1021021, 1021101, 1021202, 1021210 and 1021212.

If we observe the 7 length strings carefully based on Lemma 4, we find that all strings have at least one 2-transreversal. Then it will be reduced to 5 length ternary strings in one step. According to Lemma 3, in the next two consecutive steps, these 5 length ternary strings will be reduced to 3 length strings after performing a 1-transreversal at proper location in each step. As a result, the bound holds true. Through the reduction steps, when we can perform a single 2-transreversal, then the bound holds true. So, if we add any length of ternary string after these 7 length ternary strings, the bound always holds true.

This completes the proof.

We have shown some algorithms (Algorithm 1, 2, 3, 4) that will be used by Algorithm 5 and Algorithm 6 to group fully binary and ternary normal-
ized strings respectively satisfying the proposed bound.

**Theorem 3.** Given a fully binary and ternary normalized string, Algorithm 5 and Algorithm 6 is always able to group satisfying the bound given in Theorem 1 and Theorem 2 respectively.

**Proof.** In Algorithm 5, we first check whether there is any prefix 2-transreversal. If there exists a prefix 2-transreversal, we perform a 2-transreversal using Algorithm 3. Again we search for whether there is any suffix 2-transreversal. If there exists a suffix 2-transreversal, we perform a 2-transreversal using Algorithm 4. If both prefix 2-transreversal and suffix 2-transreversal are found, then we do not need to do 1-transreversal. But if we miss any of the previous, we perform a corresponding 1-transreversal. This confirms that we perform a suffix transreversal after a prefix transreversal or vice versa. In the best case, it reduces the length of the string by 4 at each step which we have shown already in Theorem 1. We can easily get the number of steps required to group fully binary normalized strings for larger length recursively. Thus bound for binary strings holds true.

In Algorithm 6, we first check whether there is any prefix 2-transreversal. If there exists a prefix 2-transreversal, we perform a 2-transreversal using Algorithm 3. In this case, we do not search for a suffix 2-transreversal if a prefix 2-transreversal has already been done. Otherwise, if there exists suffix 2-transreversal, we perform a 2-transreversal using Algorithm 4. If any prefix 2-transreversal or suffix 2-transreversal is found, then we do not need to do a 1-transreversal. But if we miss both of the previous, we search for a 0-transreversal that will make two consecutive 2-transreversals. We will succeed in executing any of the previous two steps. However, after performing a 2-transreversal, we can perform a 1-transreversal in the next step. But we must perform a 2-transreversal after performing a 0-transreversal (except for 3 length strings) and it is always possible. In the best cases, it reduces the length of the string by 2 which we have shown already in Theorem 2. Thus bound for ternary strings holds true.
**Algorithm 1:** Do1PrefixTransReversal (s: input string)

**Input:**
- `s` is a fully binary or ternary normalized string, where length of `s`, `n ≥ k` and it is passed as a parameter from Algorithm 6

```plaintext
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-transreversal to `s`;
    forward ← true;
end
if forward == false then
    take first two symbols as prefix leaving rest of the string as suffix;
    2nd symbol will be identical to 4th symbol;
    perform a 1-transreversal to `s`;
end
return s;
```

**Algorithm 2:** Do1SuffixTransReversal (s: input string)

**Input:**
- `s` is a fully binary or ternary normalized string, where length of `s`, `n ≥ k` and it is passed as a parameter from Algorithm 6

```plaintext
forward ← false;
take the prefix of input string leaving the last symbol of `s`;
if the last symbol exists in the prefix then
    perform a 1-transreversal to `s`;
    forward ← true;
end
if forward == false then
    take last two symbols as suffix leaving rest of the string as prefix;
    (n − 1)th symbol will be identical to (n − 3)th symbol;
    perform a 1-transreversal to `s`;
end
return s;
```
Algorithm 3: Do2PrefixTransReversal (s: input string)

**Input:** $s$ is a fully binary or ternary normalized string, where length of $s$, $n \geq k$ and it is passed as a parameter from Algorithm 6

$\text{forward} \leftarrow \text{false}$;

take the suffix of input string leaving the first two symbols of $s$;

if the reverse of first two symbols exists consecutively in the suffix then

- perform a 2-transreversal to $s$;

  $\text{forward} \leftarrow \text{true}$;

end

if $\text{forward} == \text{false}$ then

  for $i = 3; i < |s| - 2; i = i + 1$ do

  - take the first symbol of input string;

  - take the $i^{th}$ symbol of the input string;

  - append these two symbols;

  - call this string $\text{temp}$;

  - check whether this string is a reverse substring of the current suffix;

  for $j = i + 1; j < (|s| - 1); j = j + 1$ do

    - take the reverse substring named $\text{consecutive}$ of input string from $j$ to $j + 2$;

    if $\text{consecutive} == \text{temp}$ then

      - perform a 2-transreversal to $s$;

        $\text{forward} \leftarrow \text{true}$;

        break;

    end

  end

  if $\text{forward} == \text{true}$ then break;

end

return $s$;
Algorithm 4: Do2SuffixTransReversal (s: input string)

Input: s is a fully binary or ternary normalized string, where length of s, \( n \geq k \) and it is passed as a parameter from Algorithm 6

\[ \text{forward} \leftarrow \text{false}; \]

take the prefix of input string leaving the last two symbols of s;

if the reverse of last two symbols exists consecutively in the prefix then

\[ \text{perform a 2-transreversal to } s; \]
\[ \text{forward} \leftarrow \text{true}; \]

end

if forward == false then

\[ \text{for } i = |s| - 2; i > 2; i = i - 1 \text{ do} \]

\[ \text{take the last symbol of input string;} \]
\[ \text{ take the } i^{th} \text{ symbol of the input string;} \]
\[ \text{ append these two symbols;} \]
\[ \text{ call this string } \text{temp}; \]
\[ \text{ check whether this string is a reverse substring of the current prefix;} \]

\[ \text{for } j = i - 1; j > 1; j = j - 1 \text{ do} \]

\[ \text{take the reverse substring named consecutive of input string from } j - 1 \text{ to } j + 1; \]
\[ \text{if consecutive == temp then} \]
\[ \text{ perform a 2-transreversal to } s; \]
\[ \text{forward} \leftarrow \text{true}; \]
\[ \text{break;} \]
\[ \text{end} \]
\[ \text{end} \]

if forward == true then break;

end

end

return s;
Algorithm 5: GroupByBinaryTransReversal \((s:\text{input string})\)

**Input:** \(s\), a fully binary normalized string, where length of \(s\), \(n \geq k\) initialization;

\(k \leftarrow 2\) as \(s\) is binary;

\(\text{count} \leftarrow 0;\)

\(\text{twoPrefixTransReversalDone} \leftarrow \text{false;}\)

\(\text{twoSuffixTransReversalDone} \leftarrow \text{false;}\)

**while** \(|s| > k\) **do**

**if** there is a prefix 2-transreversal **then**

perform a 2-transreversal using Algorithm 3;

\(s \leftarrow \text{return value from Algorithm 3;}\)

\(\text{twoPrefixTransReversalDone} \leftarrow \text{true;}\)

**end**

**if** there is a suffix 2-transreversal **then**

perform a 2-transreversal using Algorithm 4;

\(s \leftarrow \text{return value from Algorithm 4;}\)

\(\text{twoSuffixTransReversalDone} \leftarrow \text{true;}\)

**end**

**if** \(\text{twoPrefixTransReversalDone} == \text{false}\) **then**

if string length, \(|s| > 3\), perform a 1-transreversal using Algorithm 1;

\(s \leftarrow \text{return value from Algorithm 1;}\)

**end**

**if** \(\text{twoSuffixTransReversalDone} == \text{false}\) **then**

if string length, \(|s| > 2\), perform a 1-transreversal using Algorithm 2;

\(s \leftarrow \text{return value from Algorithm 2;}\)

**end**

\(\text{count} \leftarrow \text{count} + 1;\)

\(\text{twoPrefixTransReversalDone} \leftarrow \text{false;}\)

\(\text{twoSuffixTransReversalDone} \leftarrow \text{false;}\)

**end**

\(\text{return } s;\)
Algorithm 6: GroupByTernaryTransReversal (s:input string)

Input: s, a fully ternary normalized string, where length of s, n ≥ k
initialization;
k ⇐ 3 as s is ternary;
count ⇐ 0;
twoPrefixTransReversalDone ⇐ false;
twoSuffixTransReversalDone ⇐ false;
zeroTransReversalDone ⇐ false;
while |s| > k do
    if there is a prefix 2-transreversal then
        perform a 2-transreversal using Algorithm 3;
        s ⇐ return value from Algorithm 3;
twoPrefixTransReversalDone ⇐ true;
    end
    if there is a suffix 2-transreversal and
    twoPrefixTransReversalDone == false then
        perform a 2-transreversal using Algorithm 4;
        s ⇐ return value from Algorithm 4;
twoSuffixTransReversalDone ⇐ true;
    end
    if twoPrefixTransReversalDone == false and
    twoSuffixTransReversalDone == false then
        if there is 0-transreversal that ensure two consecutive
        2-transreversal then
            perform a 0-transreversal on suitable location;
zeroTransReversalDone ⇐ true;
        end
    end
    if zeroTransReversalDone == false then
        if there is a prefix 1-transreversal then
            perform a 1-transreversal using Algorithm 1;
            s ⇐ return value from Algorithm 1;
        end
        else
            perform a 1-transreversal using Algorithm 2;
            s ⇐ return value from Algorithm 2;
        end
    end
    count ⇐ count + 1;
twoPrefixTransReversalDone ⇐ false;
twoSuffixTransReversalDone ⇐ false;
zeroTransReversalDone ⇐ false;
end
return s;
3.2 Sorting

The sorting distance $d_{gps}^s(s)$ of a fully ternary string $s$ is defined as the minimum number of prefix and suffix transreversal required to sort the string to one of length 3. We again consider normalized strings. To measure binary sorting distance, we use the notation $d_{gps}^s(s)$.

3.2.1 Sorting Binary Strings

In this section, we present the following Theorem for sorting binary normalized strings.

**Theorem 4.** (Distance for Binary strings) Let $s$ be a fully binary normalized string. Then, for $n > 3$, $d_{gps}^s(s) = \lceil \frac{n-k}{4} \rceil + 1$, when $n$ is odd and $d_{gps}^s(s) = \lfloor \frac{n-k}{4} \rfloor + 2$, when $n$ is even and $k = 2$ is the arity.

*Proof.* As binary strings have only 2 letters, after grouping they are already sorted (in ascending or descending order). So, the upper bound is $d_{gps}^s(s) = \lceil \frac{n-k}{4} \rceil + 1$ for odd length and $d_{gps}^s(s) = \lfloor \frac{n-k}{4} \rfloor + 2$ for even length provided that length of string $s$, $n > 3$. Note also that, when $|s| = 3$, then it needs one extra 1-transreversal. \( \Box \)

3.2.2 Sorting Ternary Strings

In this section, we present the following Theorem for sorting ternary normalized strings. We present Algorithm 7 to sort binary/ternary strings.

**Theorem 5.** (Bound for Ternary strings) Let $s$ be a fully Ternary normalized string. Then, the upper bound for sorting ternary string is $d_{gps}^s(s) \leq \lceil \frac{n-k+1}{2} \rceil + 2$ and $k = 3$ is the arity.

*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 suffix or prefix 0-transreversal to sort 021, 102, 120 and 201. Hence the result follows. \( \Box \)
Algorithm 7: SortByPrefixSuffixTransReversal \((s:\text{input string})\)

**Input:** \(s\) is a fully binary or ternary normalized string, where length of \(s\), \(n \geq k\)

\[
\text{if } s \text{ is binary then}
\begin{align*}
\text{run Algorithm 5 on } s; \\
\text{ } \\
\text{s is sorted after grouping ;} \\
\text{ } \\
\text{finish;}
\end{align*}
\]

\[
\text{end}
\]

\[
\text{else}
\begin{align*}
\text{run Algorithm 6 on } s; \\
\text{if } s \text{ is in 021, 102, 120, 201 then}
\begin{align*}
\text{perform one 0-transreversal to sort } s ;
\end{align*}
\]

\[
\text{end}
\]

\[
\text{else}
\begin{align*}
\text{s is already sorted ;}
\end{align*}
\]

\[
\text{end}
\]

\[
\text{end}
\]

### 3.3 Classification

In this section, following the trend of [2], we will identify two classes of fully ternary normalized strings. We have already shown in Theorem 8 that it needs at most \(\lceil \frac{n-k}{2} \rceil + 1\) prefix and suffix transreversals to group fully ternary normalized strings. We will discuss two classes; the first one consists of the fully ternary normalized strings that require \(\lceil \frac{n-k}{2} \rceil\) prefix and suffix transreversals (Class 1) and the other need \(\lceil \frac{n-k+1}{2} \rceil + 1\) (Class 2). We are not able to classify all the fully ternary normalized strings; however, we provide below a classification of a large set thereof comprising infinite number of strings.

- All strings of length four satisfy the bound \(\lceil \frac{n-k}{2} \rceil\), so we put them in Class 1.
• $(10)^{+}201$, $(12)^{+}021$, $201(01)^{+}$, $021(21)^{+}$ 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 is reduced to any one of the previous strings by a series of consecutive 2-transreversals, then that particular string also belongs to Class 1.

In Class 2, we put the strings which need $\left\lceil \frac{n-k+1}{2} \right\rceil + 1$ prefix and suffix transreversals to be grouped.

• $10102$, $10120$, $10121$, $10210$, $10202$, $12102$, $12101$, $12012$, $12010$, $12020$ and ternary strings reduced to any of these by a series of 2-transreversals and/or 1-transreversals are in Class 2. Consecutive 2-transreversals are unavailable in these strings.

• $(10)^{+}102$, $(10)^{+}120$, $(10)^{+}121$, $(10)^{+}210$, $(10)^{+}212$, $(10)^{+}202$, $(12)^{+}120$, $(12)^{+}102$, $(12)^{+}101$, $(12)^{+}012$, $(12)^{+}010$, $(12)^{+}020$, $102(10)^{+}$, $120(10)^{+}$, $121(01)^{+}$, $210(10)^{+}$, $212(10)^{+}$, $202(10)^{+}$, $120(12)^{+}$, $120(12)^{+}$, $102(12)^{+}$, $101(21)^{+}$, $012(12)^{+}$, $010(12)^{+}$, $020(12)^{+}$ and ternary strings reduced to any of these by a series of 2-transreversals and/or 1-transreversals belong to Class 2.

• Ternary strings reduced to any of these by a series of 2-transreversals will also be placed in Class 2.

Now, we will categorize all the fully ternary normalized strings of Class 2 from a different perspective. If it is easy to look up a suitable location to perform a 2-transreversal in a string, we call all such of strings “Simple strings”. The following strings belonging to Class 2 are “Simple strings”.

• $(10)^{+}102$, $(10)^{+}120$, $(10)^{+}121$, $(10)^{+}210$, $(12)^{+}120$, $(12)^{+}102$, $(12)^{+}101$, $(12)^{+}012$, $102(10)^{+}$, $120(10)^{+}$, $121(01)^{+}$, $210(10)^{+}$, $212(10)^{+}$, $202(10)^{+}$, $120(12)^{+}$, $120(12)^{+}$, $102(12)^{+}$, $101(21)^{+}$, $012(12)^{+}$ and ternary strings reduced to any of these by a series of 2-transreversals and/or 1-transreversals.
There are strings for which it will be a bit difficult to find the a suitable location to get a 2-transreversal. We call those strings “Complex strings”. Thus, $(10)^+ 212$, $(10)^+ 010$, $(12)^+ 202$, $(12)^+ 020$, $212(10)^+$, $202(10)^+$, $010(12)^+$, $020(12)^+$ are examples of “Complex strings”. Notably, the extension of a “Simple string” contains two common symbols of a token after which a quantifier is applied. But, the extension of a “Complex string” contains one common symbol of a token after which a quantifier is applied.

To elaborate, let us take a fully ternary string $1212102$ as an example of “Simple strings”. We will get the following set of prefix and suffix transreversals: $1212102 = \rightarrow 12102 = \rightarrow 2102 = \rightarrow 012$. Here, we need 3 prefix and suffix transreversals which satisfy the bound. In this case, the length of the prefix (suffix) string involved in the transreversal operation will be one or two at most. Now, let us take another fully ternary string $1010212$ as an example of “Complex strings”. Here, we will get the following set of prefix and suffix transreversals: $1010212 = \rightarrow 021012 = \rightarrow 021102 = \rightarrow 021012 = \rightarrow 012$. In this case, the length of prefix (suffix) string will be greater than two.
Chapter 4

Sorting Strings by Prefix Block-Interchanges

In this chapter, we first deduce grouping distance of both binary and ternary normalized strings for prefix block-interchange operations where we present some algorithms. We also apply a restricted version of prefix block-interchange operations on binary normalized string where we achieve better upper bound. Second, we deduce sorting distance of both binary and ternary normalized strings. Finally, we provide classification of a large set thereof comprising infinite number of ternary normalized strings.

4.1 Grouping and Sorting Binary Strings

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 possible length of binary string is either even or odd. As strings are normalized, only 4 kinds of binary strings are possible, namely, \((01)^+, (10)^+, (01)^+0\) and \((10)^+1\). We achieve the following properties for binary strings.

Property 1. Let \(s\) be a fully binary normalized string of length \(n\), where \(n \geq 2\). Then, we will need to reduce \(n - 2\) symbols to group \(s\).

There will be at least one identical symbol in \(s\) if \(n > 2\). So, we need to
reduce those symbols applying prefix block-interchanges and then reducing
consecutive identical symbols which results in a normalized string. Finally,
after completion of grouping, we will get either 01 or 10, where 01 is already
sorted. If we get 10, we will need one extra operation to sort it provided that
we will then consider it as a permutation.

**Property 2.** Let $s$ be a fully binary normalized string of length $n$, where
$1 \leq x < y \leq z < n$ and $n \geq 4$. If we have a prefix block $s[1] \ldots s[x]$ and
another block $s[y] \ldots s[z]$ such that any two of the following conditions (1-3)
are satisfied, then we get a 2-pblockInterchange.

- Condition 1: $s[1] = s[y - 1]$
- Condition 2: $s[x + 1] = s[z]$
- Condition 3: $s[x] = s[z + 1]$

**Property 3.** Let $s$ be a fully binary normalized string of length $n$, where
$1 \leq x < y \leq z < n$ and $n \geq 5$. If we have a prefix block $s[1] \ldots s[x]$ and
another block $s[y] \ldots s[z]$ such that all three of the following conditions (1-3)
are satisfied, then we get a 3-pblockInterchange.

- Condition 1: $s[1] = s[y - 1]$
- Condition 2: $s[x + 1] = s[z]$
- Condition 3: $s[x] = s[z + 1]$

**Lemma 6.** Let $s$ be a fully normalized binary string of length $n$ such that $n \geq 5$. Then, we always get 3-pblockInterchange($s$) using prefix block-interchange
operations.

**Proof.** Let $s$ be a binary normalized string, where $n = 5$. Suppose, $s = ababa$
represents a class of binary normalized strings where either $a = 1$ and $b = 0$, or $a = 0$ and $b = 1$. Now, if we exchange $s[1]$ with $s[4]$ ($\beta(1, 1, 4, 4)$), we
get a normalized string of $ba$ which is a reduced string from the original one.
Here, the length of reduced string is 2 which is 3 less than the original string,
i.e., we have a 3-pblockInterchange here. Now, if we append any length of binary normalized string after $s$, then according to Property 3, we always get a 3-pblockInterchange if we always perform $\beta(1,1,4,4)$ provided that $n \geq 5$. So, the claim holds true.

Lemma 7. Let $s$ be a fully normalized binary string of length $n$ such that $n \geq 5$ and we always perform prefix block-interchange operations, then $d_{pbi}^n(s) = \lceil \frac{n-2}{3} \rceil$

Proof. According to lemma 6, as we always get a 3-pblockInterchange when $n \geq 5$, then the length of the string will be reduced by 3 after each operation. Thus, the bound holds true.

4.1.1 Grouping Binary Strings by Restricted Prefix Block-interchanges

Let us consider a different scenario where we always keep the first block of consecutive identical symbol(s) of a string fixed, i.e., we keep the first symbol of a normalized binary string fixed. So, while performing a prefix block-interchange operation we actually mean $\beta(2, x, y, z)$. We call this restricted prefix block-interchange operation. In case of a breakpoint (cycle) graph of linear permutation, we add 0 and $n + 1$ to two ends of the permutation [10]. The motivation for this apparently strange setting comes from the existence of circular genome. While handling a circular permutation (genome), we need to keep the first position fixed. There are some existing works where such circular permutations are considered [11, 12].

Property 4. Let $s$ be a fully binary normalized string of length $n$, where $2 \leq x < y \leq z < n$. If we have a prefix block $s[2] \ldots s[x]$ and another block $s[y] \ldots s[z]$ such that $s[1] = s[y]$, $s[x + 1] = s[z]$, $s[2] = s[y - 1]$ and $s[x] = s[z + 1]$, then we get a 4-pblockInterchange using restricted prefix block-interchange operations.

For example, let $s = 10101010$. Since, $s[1] = s[5]$, $s[3 + 1] = s[6]$, $s[2] = s[5 - 1]$ and $s[3] = s[6 + 1]$ (i.e., $x = 3$, $y = 5$ and $z = 6$), $\beta(2,3,5,6)$ can be applied on $s$ as follows: $10101\overline{0}10 = 11000110 = 1010 = 1010$. 39
Lemma 8. Let $s$ be a fully normalized binary string of length $n$ such that $n \geq 7$. Then, we always get 4-pblockInterchange(s) using restricted prefix block-interchange operations.

Proof. As $s$ is fully normalized binary string, there are only four kinds of such strings: 1010...101 of even length, 0101...010 of even length, 1010...101 of odd length and 0101...010 of odd length. So, for all these strings, we always get that the first symbol is same as the fifth symbol, the second symbol is same as the fourth symbol, the fourth symbol is same as the sixth symbol and the third symbol is same as the seventh symbol. Then, according to Property 4, we always get a 4-pblockInterchange for each of the four possible combinations (such that $n \geq 7$) which reduces the length of the string by 4. \hfill \Box

Theorem 6. Let $s$ be a fully binary normalized string of length $n$ such that $n \geq 2$ and we always perform restricted prefix block-interchange operations, then $d_{rpb}^p(s) = \lceil \frac{n-2}{4} \rceil$ where $d_{rpb}^p(s)$ is the number of restricted prefix block-interchanges required to group all symbols of the string $s$.

Proof. First, we prove the bound for all fully normalized binary strings where $n < 7$, then we prove for $n \geq 7$. When $n < 7$, we need one prefix block interchange operation to group all symbols in $s$. For this case, length of $s$ can be 3, 4, 5 or 6. We can easily group these strings as follow:

- $101 \xrightarrow{\beta(2,2,3,3)} 110 = 10$
- $1010 \xrightarrow{\beta(2,2,3,3)} 1100 = 10$
- $10101 \xrightarrow{\beta(2,2,5,5)} 11100 = 10$
- $101010 \xrightarrow{\beta(2,2,5,5)} 111000 = 10$.

Note that 1010 and 0101 are infact similar strings because we get the latter string from the former by relabeling ‘0’ to ‘1’ and 1 to ‘0’ as in [2, 8]. So, we need one restricted prefix block-interchange to group 010, 0101, 01010 and 010101 which satisfies the proposed bound. Now, when $n \geq 7$, we always get 4-pblockInterchange(s) to perform on $s$ (Lemma 8) which will reduce the length of $s$ by 4. Thus the proposed bound always holds true. \hfill \Box
Clearly, our proposed upper bound to group fully binary normalized string applying restricted prefix block-interchanges is better than other results presented in Table 1.1. After completion of grouping applying prefix block-interchanges, we get either 01 or 10, where 01 is already sorted. So, if we get 10 then we treat it as permutation and need one extra prefix block-interchange operation to sort it. Thus, \(d_{pbi}^s(s) = \lceil \frac{n-2}{3} \rceil + 1\), when we need to sort 10, otherwise \(d_{pbi}^s(s) = d_{pbi}^p(s) = \lceil \frac{n-2}{3} \rceil\).

**Algorithm 8:** Algorithm to group fully normalized binary string by restricted prefix block-interchanges

**Input:** \(s\) is a fully binary normalized string, where length of \(s\), \(n \geq 3\)

\[\text{count} = 0;\]

\[\text{while } (n \geq 7) \text{ do}\]
\[\quad \text{swap}(s[2], s[3], s[5], s[6]);\]
\[\quad \text{normalize}(s);\]
\[\quad n = n - 4;\]
\[\quad \text{count} = \text{count} + 1;\]
\[\text{end}\]

\[\text{if } (n == 3 || n == 4) \text{ then}\]
\[\quad \text{swap}(s[2], s[3]);\]
\[\text{end}\]

\[\text{if } (n == 5 || n == 6) \text{ then}\]
\[\quad \text{swap}(s[2], s[5]);\]
\[\text{end}\]

\[\text{normalize}(s);\]

\[d_{rpbib}^s(s) = \text{count} = \text{count} + 1;\]

\[\text{return } d_{rpbib}^p(s);\]

We also present Algorithm 8 to compute the restricted prefix block-interchange distance to group fully normalized binary string(s). In Algorithm 8, \textit{swap} and \textit{normalize} operations take constant time [14]. We can perform the \textit{swap} operation by exchanging one or two symbols with one or two symbols respectively. Similarly, \textit{normalize} operation can be done always by checking first to seventh symbols in \(s\). So, the \textit{while} loop will execute not
more than \( n \) times. Clearly, the time complexity of Algorithm 8 is \( O(n) \).

**Theorem 7.** The problem of grouping fully normalized binary strings by restricted prefix block-interchanges can be solved by a linear-time algorithm.

### 4.2 Grouping and Sorting Ternary Strings

In this section, we deduce both the grouping and sorting distances for ternary strings. It can be noted that grouping ternary strings is not as simple as grouping binary strings. We start with an easy lemma.

**Lemma 9.** In a fully normalized ternary string, we can always perform a 1-pblockInterchange.

**Proof.** Consider a normalized ternary string \( s \) of length \( n > 3 \). So, \( s = s[1] \ldots s[l] \ldots s[i] \ldots s[j] \ldots s[n] \), here it is obvious that \( 1 \leq l < i < j < n \).

Now, we take \( s[1] \ldots s[l] \) as a prefix. As this is a normalized ternary string, somewhere we will find a block \( s[i] \ldots s[j] \) such that any one of the following conditions (1-3) is satisfied; and consequently, we get a 1-pblockInterchange.

- **Condition 1:** \( s[1] = s[i - 1] \)
- **Condition 2:** \( s[j] = s[l + 1] \)
- **Condition 3:** \( s[l] = s[j + 1] \)

Since, one of the above cases always occurs for fully ternary strings when \( n > 3 \), the result follows.

Now, we assume that we can perform a 2-pblockInterchange on a string. Note that, instead of taking a prefix of length at least 2, we can always take a prefix of length 1 (i.e., the first symbol), and perform a 1-pblockInterchange. Thus, the presence of a 2-pblockInterchange always ensures that there is also a 1-pblockInterchange. For example, let us consider 10201. There is a 2-pblockInterchange: \( \overline{10201} \Rightarrow 00211 \Rightarrow 021 \). We could also get a 1-pblockInterchange as follows: \( \overline{10201} \Rightarrow 02011 \Rightarrow 0201 \).
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 8 below, the upper bound differs. At first, we give an easy but useful lemma.

**Lemma 10.** Suppose $s[1] \ldots s[n]$ is a fully normalized ternary string. If we have a prefix $s[1] \ldots s[i], 1 \leq i < j \leq k < n$ such that $s[1] = s[j - 1]$ and $s[i] = s[k + 1]$, then we have a 2-pblockInterchange.

**Proof.** We have a fully normalized ternary string $s$. After performing a prefix block-interchange $\beta(1, i, j, k)$ on $s$, $s[1]$ will be adjacent to $s[j - 1]$ and $s[i]$ will be adjacent to $s[k + 1]$. Then, we will be able to eliminate one of $s[1]$ or $s[j - 1]$ and one of $s[i]$ or $s[k + 1]$. This ensures that the length of $s$ will be reduced by 2 which completes the proof. \qed

**Theorem 8.** (Grouping ternary strings applying prefix block-interchanges) Let $s$ be a fully normalized ternary string. Then, $d_{\text{pbi}}(s) \leq \lceil \frac{n-2}{2} \rceil$ where $n$ is the length of the string.

**Proof.** At first, we will prove the bound for all normalized ternary strings where $n < 7$. Then, we will describe the proof for all normalized ternary strings where $n \geq 7$. In what follows, we only consider strings starting with 1. This does not lose the generality since we can always use relabeling for strings starting with 0 or 2.

![Tree diagram for normalized ternary strings of length 4 starting with 1](image)

As strings are fully ternary, we do not need to work with $n \leq 3$. For $n = 4$, we require a 1-pblockInterchange and that is optimal, hence $d_{\text{pbi}}(s) = 1$, and
the upper bound is satisfied. We give all fully normalized ternary strings of length 4 starting with 1 in List (4.1) (see tree diagram in Figure 4.2). Similarly, when \( n = 5 \), we can always perform either 1-pblockInterchange or 2-pblockInterchange and thus \( d_{pbi}^p(s) \leq 2 \). Now, we apply the upper bound for \( n = 6 \), we always get \( d_{pbi}^p(s) \leq 2 \). Table 4.1 shows the derivation of the grouping distance for fully normalized ternary strings of length 5 and 6. It is easy to realize that, by Lemma 9, we can always satisfy the given upper bound. Thus the upper bound is proved for \( n < 7 \).

\[
1012, 1010, 1021, 1020, 1201, 1202, 1210 \text{ and } 1212. 
\] (4.1)

Now we consider \( n \geq 7 \). Note carefully that for any string starting with 1, we can only have one of the eight prefixes of length 4 from List (4.1).

For a \( n \) length string, if we can give a 2-pblockInterchange, the resulting reduced string will start with its previous starting symbol and it may be 1, 0 or 2. For the latter two cases (0 and 2), we can again use relabeling as mentioned before. Therefore we can safely assume that the reduced string will have any of the 8 prefixes of List (4.1). Hence, it suffices to prove the bound considering each of the prefixes of List (4.1). We will show the lists for 7 length fully normalized ternary strings each of which will have a 4 length prefix from List (4.1). Then we will provide arguments for each of the prefixes from List (4.1).

Firstly, it is easy to note that, the prefixes 1010 and 1212 themselves are binary strings. So, they take one prefix block-interchange operation (Property 2). Therefore, we can safely exclude one of them from the following discussion. Now, if we relabel “0” to “2” and “2” to “0”, then we get 1201 from 1021. Similarly, after relabeling we get 1202 from 1020 and 1210 from 1012. So, we can safely exclude these also from the following discussion. In what follows, when we refer to the prefixes of List (4.1), we would actually mean all the prefixes excluding 1201, 1202, 1210 and 1212.
<table>
<thead>
<tr>
<th>$n$</th>
<th>derivation</th>
<th>$d_{pbi}^g(s)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>$10120 \xrightarrow{\beta(1,2,4,4)} 21100 = 210$</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>$10121 \xrightarrow{\beta(1,1,4,4)} 20111 = 201$</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>$10201 \xrightarrow{\beta(1,1,3,4)} 20011 = 201$</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>$10202 \xrightarrow{\beta(1,1,4,4)} 00212 = 0212 \xrightarrow{\beta(1,1,4,4)} 2210 = 210$</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>$10210 \xrightarrow{\beta(1,1,5,5)} 00211 = 021$</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>$10212 \xrightarrow{\beta(1,2,5,5)} 22110 = 210$</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>$101201 \xrightarrow{\beta(1,1,4,5)} 200111 = 201$</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>$101202 \xrightarrow{\beta(1,1,4,5)} 200112 = 2012 \xrightarrow{\beta(1,1,3,3)} 1022 = 102$</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>$101210 \xrightarrow{\beta(1,2,4,5)} 211100 = 210$</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>$101212 \xrightarrow{\beta(1,2,4,5)} 211102 = 2102 \xrightarrow{\beta(1,1,3,3)} 0122 = 012$</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>$102010 \xrightarrow{\beta(1,1,6,6)} 002011 = 0201 \xrightarrow{\beta(1,1,4,4)} 1200 = 120$</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>$102012 \xrightarrow{\beta(1,2,6,6)} 220110 = 2010 \xrightarrow{\beta(1,2,3,3)} 1200 = 120$</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>$102020 \xrightarrow{\beta(1,2,5,5)} 220100 = 2010 \xrightarrow{\beta(1,2,3,3)} 1200 = 120$</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>$102021 \xrightarrow{\beta(1,2,4,4)} 202011 = 20201 \xrightarrow{\beta(1,1,3,3)} 00221 = 021$</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>$102101 \xrightarrow{\beta(1,1,5,5)} 002111 = 021$</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>$102102 \xrightarrow{\beta(1,1,5,5)} 002112 = 0212 \xrightarrow{\beta(1,2,3,3)} 1022 = 102$</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>$102120 \xrightarrow{\beta(1,2,5,5)} 221100 = 210$</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>$102121 \xrightarrow{\beta(1,2,5,5)} 221101 = 2102 \xrightarrow{\beta(1,1,3,3)} 1022 = 102$</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 4.1: This table shows the derivation of the grouping distance for normalized ternary strings of length 5 and 6 deduced from List (4.1). $n$ is the length of normalized ternary strings and $d_{pbi}^g(s)$ is the grouping distance. Here, binary strings like 101010 or 121212 are omitted.

**1010**

We first give all possible strings of length 7 having prefix 1010 in List (4.2). The possible 5 length strings having this prefix are 10101 and 10102.

1010101, 1010102, 1010120, 1010121, 1010201, 1010202, 1010210 and 1010212.

In List (4.2), all strings will have at least one 1-pblockInterchange. When
we have a prefix symbol 1, we need to lookup somewhere for a symbol 1 so that we can perform a 1-pblockInterchange. Similarly, when we have a prefix block 10, we need to lookup somewhere for a suffix block 1...0. In that case, we will be able to perform either a 1-pblockInterchange or a 2-pblockInterchange. This is also true for longer prefix blocks.

Let us check this with examples. The string 1010121 satisfies the bound as follows:

\[
1010 \quad 121 \Rightarrow 0121 \Rightarrow 012 \quad 1 \Rightarrow 201.
\]

We see that, for \(n = 7\), we need only two prefix block-interchanges holding the bound true. However, if we can reduce the string length by at least 1 after each step, we will also achieve the desired bound. Here, all strings satisfy the bound. Let us see another example for string 1010212. The reduction steps are as follows:

\[
1010 \quad 212 \Rightarrow 01212 \Rightarrow 012 \Rightarrow 1202 \Rightarrow 1202 \Rightarrow 012.
\]

This takes three steps which satisfies the bound. Now, if we add 0 or 1 after 1010212, then the resulting string will be 10102120 or 10102121 in which case we will be able to perform a 2-pblockInterchange. So, adding any length of ternary string(s) after 1010212 will also have 2-pblockInterchange in the reduction steps.

**1012**

Here, again we first give the list of 7 length strings having prefix 1012 (see List (5.1)). Here, the possible 5 length strings of this prefix are 10120 and
Figure 4.3: Tree diagram of 7 length strings (placed at leaf nodes) having prefix 1020.

10121 (see tree diagram in Figure 4.2).

1012010, 1012012, 1012020, 1012021, 1012101, 1012102, 1012120 and 1012121.

In this case, we see that all strings having length 7, will have a 2-pblockInterchange as follows. When we have a prefix block 10, we need to lookup somewhere for a suffix block 1...0. Similarly, when we have a prefix block 1012, we need to lookup somewhere for a suffix block 1...2. Then we will be able to perform a 2-pblockInterchange which will reduce the string length by 2. Then according to Lemma 9, we will get at least two consecutive 1-pblockInterchange which will satisfy the bound. This is also true for higher length prefix blocks.

Let us illustrate this with an example. The string 1012120 satisfies the bound as follows: \(\underline{1012120} \Rightarrow 2120 \Rightarrow \overline{21210} \Rightarrow 120\). Here, \(n = 7\) and we need only two prefix block-interchanges to satisfy the bound. We can also illustrate this with another example where fully normalized ternary string is 1012020. It satisfies the bound as follows: \(\underline{1012020} \Rightarrow 21020 \Rightarrow \overline{21020} \Rightarrow 021\). So we can say, if we add any length of ternary string(s) after 1012020, that will also have a 2-pblockInterchange.
Figure 4.4: Tree diagram of 7 length strings (placed at leaf nodes) having prefix 1021.

1020

We present all strings of 7 length having prefix 1020 in List (4.4) and its corresponding tree diagram in Figure 4.3.

1020101, 1020102, 1020120, 1020121, 1020201, 1020202, 1020210 and 1020212.

(4.4)

Here, according to Lemma 10, all strings will have a 2-pblockInterchange. When we have a prefix block 102, we need to lookup somewhere for a suffix block 1...2. Similarly, when we have a prefix block 1, we need to lookup somewhere for a suffix block 1...1 so that after applying a prefix block-interchange, we get a 2-pblockInterchange. Then, according to Lemma 9, the bound will be satisfied. Thus, if we add any length of ternary string(s) after 7 length strings having prefix 1020, that will also have 2-pblockInterchange.
We present all strings of 7 length having prefix 1021 in List (4.5) and its corresponding tree diagram in Figure 4.4.

1021010, 1021012, 1021020, 1021021, 1021201, 1021202, 1021210 and 1021212.

If we observe the 7 length strings carefully based on Lemma 10, we find that all strings have at least one 2-pblockInterchange. Then it will be reduced to 5 length ternary strings in one step. According to Lemma 9, in the next two consecutive steps, these 5 length ternary strings will be reduced to 3 length strings after performing a 1-pblockInterchange appropriately in each step. As a result, the bound holds true. Through the reduction steps, if we can perform a single 2-pblockInterchange, the bound will hold true. Thus, if we add any length of ternary string after these 7 length normalized ternary strings, the bound always holds true.

This completes the proof.

We present Algorithm 9 to group fully normalized ternary strings. Here, $|s|$ denotes the length of the string $s$. The outer while loop iterates until all the symbols of the string $s$ are grouped. We make the string $firstblock$ appending the symbol of the first position and the symbol of the $i^{th}$ position. Then, we make another string $secondblock$ appending the symbol of the $k^{th}$ position and $(k+1)^{th}$ position such that $i < j < k$. Now, if $firstblock$ is equal to $secondblock$ then we actually get that $s[1]$ is equal $s[k]$ and $s[i]$ is equal to $s[k+1]$ which are basically the conditions of Lemma 10. So, according to Lemma 10, we will get a 2-pblockInterchange here and thus the length of the string will be reduced by 2. If a 2-pblockInterchange is not found, we will perform a 1-pblockInterchange which is always possible according to Lemma 9 and thus the length of the string will be reduced by 1. When all symbols are grouped, the length of the string will be 3 for fully ternary normalized string and so the while loop will terminate. Clearly, our presented algorithm (Algorithm 9) will take $O(n^4)$ time to execute, where $n$ is the length of the string.
We achieve a similar bound like Theorem 8, if we apply restricted prefix block-interchanges on normalized ternary strings. In Table 4.2, we only show the derivations of the grouping distance for normalized ternary strings of length 5 and 6 applying restricted prefix block-interchanges.

**Theorem 9.** (Sorting distance for ternary strings) Let $s$ be a fully normalized ternary string. Then, the upper bound for sorting ternary string is $d_{\text{pbi}}^*(s) \leq \left\lceil \frac{n-2}{2} \right\rceil + 2$ where $n$ is the length of the string.

*Proof.* After grouping a ternary string, we can have one of the following grouped strings: 012, 021, 102, 120, 201 and 210. Among these, 012 is already sorted. We need one more prefix 0-pblockInterchange to sort 210, 102, 120 and 201. We need two more prefix 0-pblockInterchanges to sort 021. Hence the result follows. \qed
**Algorithm 9**: Algorithm to group fully normalized ternary string by prefix block-interchanges ($s$: input string)

**Input**: $s$, a fully normalized ternary string

\[
\text{count} \leftarrow 0; \\
\text{twoBlockInterchangeDone} \leftarrow \text{false}; \\
\text{while } |s| > 3 \text{ do} \\
\quad \text{for } i = 1; i < |s|; i = i + 1 \text{ do} \\
\quad \quad \text{take the first symbol of input string} \\
\quad \quad \text{take the } i^{th} \text{ symbol of the input string} \\
\quad \quad \text{firstblock} \leftarrow s[1]s[i] \\
\quad \quad \text{check whether this string is a substring of the current suffix} \\
\quad \quad \text{for } j = i + 1; j < (|s| - 1); j = j + 1 \text{ do} \\
\quad \quad \quad \text{for } k = j + 1; k < (|s| - 1); k = k + 1 \text{ do} \\
\quad \quad \quad \quad \text{secondblock} \leftarrow s[k]s[k + 1] \\
\quad \quad \quad \text{if } \text{firstblock} == \text{secondblock} \text{ then} \\
\quad \quad \quad \quad \text{s} \leftarrow \text{perform a 2-pblockInterchange} \\
\quad \quad \quad \quad \text{count} \leftarrow \text{count} + 1 \\
\quad \quad \quad \quad \text{twoBlockInterchangeDone} \leftarrow \text{true} \\
\quad \quad \quad \quad \text{break} \\
\quad \quad \text{end} \\
\quad \text{end} \\
\text{if } \text{twoBlockInterchangeDone} == \text{false} \text{ then} \\
\quad \text{s} \leftarrow \text{perform a 1-pblockInterchange} \\
\quad \text{count} \leftarrow \text{count} + 1 \\
\text{end} \\
\text{end} \\
\text{twoBlockInterchangeDone} \leftarrow \text{false} ; \\
\text{end}
4.3 Classification

In this section, following the trend of [2, 8], we will briefly identify two classes of fully normalized ternary strings. We have already shown in Theorem 8 that it needs at most \( \lceil \frac{n-2}{2} \rceil \) prefix block-interchange operations to group fully normalized ternary strings. We will discuss two classes; the first one consists of the fully normalized ternary strings that require \( \lceil \frac{n-3}{2} \rceil \) prefix block-interchange operations (Class 1) and the other comprising strings that need \( \lceil \frac{n-2}{2} \rceil \) (Class 2). We are not able to classify all the fully normalized ternary strings; however we provide below a classification of a large set thereof comprising infinite number of strings:

- All strings of length 4 and 6 satisfy the bound \( \lceil \frac{n-3}{2} \rceil \), so we put them in Class 1 (see Table 4.1).

- \((10)^+210, (12)^+012, 120(10)^+, 102(12)^+, 102(10)^+\) 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 is reduced to any one of the previous strings by a series of consecutive 2-pblockInterchange, then that particular string also belongs to Class 1.

- \((102)^+021, (1020)^+21, 10(21)^+, 10(12)^+\) belong to Class 2. We can apply relabeling on these to find strings starting with 0 or 2 which belong to Class 2.

- Ternary strings reduced to any of these by a series of 2-pblockInterchange will also belong to Class 2.
Table 4.2: This table shows the derivation of the grouping distance for normalized ternary strings of length 5 and 6 deduced from List (4.1). \( n \) is the length of normalized ternary strings and \( d_{\text{rgbi}}(s) \) is the grouping distance applying restricted prefix block-interchanges. Here, binary strings like 101010 or 121212 are omitted.

<table>
<thead>
<tr>
<th>( n )</th>
<th>derivation</th>
<th>( d_{\text{rgbi}}(s) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>10120</td>
<td>( \beta(2,2,3,4) )</td>
<td>( 11200 = 120 )</td>
</tr>
<tr>
<td>10121</td>
<td>( \beta(2,2,5,5) )</td>
<td>( 11120 = 120 )</td>
</tr>
<tr>
<td>10201</td>
<td>( \beta(2,2,4,4) )</td>
<td>( 11200 = 120 )</td>
</tr>
<tr>
<td>10202</td>
<td>( \beta(2,2,5,5) )</td>
<td>( 12200 = 120 )</td>
</tr>
<tr>
<td>10210</td>
<td>( \beta(2,2,4,4) )</td>
<td>( 11200 = 120 )</td>
</tr>
<tr>
<td>10212</td>
<td>( \beta(2,2,4,5) )</td>
<td>( 11220 = 120 )</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>101201</td>
<td>( \beta(2,2,6,6) )</td>
<td>( 111200 = 120 )</td>
</tr>
<tr>
<td>101202</td>
<td>( \beta(2,2,3,5) )</td>
<td>( 112002 = 1202 )</td>
</tr>
<tr>
<td>101210</td>
<td>( \beta(2,2,5,5) )</td>
<td>( 111200 = 120 )</td>
</tr>
<tr>
<td>101212</td>
<td>( \beta(2,2,4,4) )</td>
<td>( 111202 = 120 )</td>
</tr>
<tr>
<td>102010</td>
<td>( \beta(2,3,5,6) )</td>
<td>( 110002 = 102 )</td>
</tr>
<tr>
<td>102012</td>
<td>( \beta(2,2,5,6) )</td>
<td>( 112200 = 120 )</td>
</tr>
<tr>
<td>102020</td>
<td>( \beta(2,2,5,5) )</td>
<td>( 122000 = 120 )</td>
</tr>
<tr>
<td>102021</td>
<td>( \beta(2,2,5,5) )</td>
<td>( 122001 = 1201 )</td>
</tr>
<tr>
<td>102101</td>
<td>( \beta(2,3,6,6) )</td>
<td>( 111002 = 102 )</td>
</tr>
<tr>
<td>102102</td>
<td>( \beta(2,3,4,5) )</td>
<td>( 110022 = 102 )</td>
</tr>
<tr>
<td>102120</td>
<td>( \beta(2,2,4,5) )</td>
<td>( 112200 = 120 )</td>
</tr>
<tr>
<td>102121</td>
<td>( \beta(2,3,6,6) )</td>
<td>( 111202 = 120 )</td>
</tr>
</tbody>
</table>

Table 4.2: This table shows the derivation of the grouping distance for normalized ternary strings of length 5 and 6 deduced from List (4.1). \( n \) is the length of normalized ternary strings and \( d_{\text{rgbi}}(s) \) is the grouping distance applying restricted prefix block-interchanges. Here, binary strings like 101010 or 121212 are omitted.
Chapter 5

Sorting Permutation by Genome Rearrangement Operations

In this chapter, we discuss different scenarios of breakpoint graph that help to reduce approximation ratio of permutation sorting by genome rearrangement operation. Then we deduce several sufficient conditions for different approximation ratio algorithms. We also conduct experiments on both real genomic data and simulated data, and show results to support our deduced conditions.

5.1 Scenarios of the Breakpoint Graph

We have deduced several scenarios of breakpoint graph. Figure 5.1 illustrates the possible scenarios that may arise in sorting a permutation. We will scan the permutation from left to right. We have also observed that the application of a particular operation (prefix breakpoint removal transposition, suffix breakpoint removal reversal, etc.) on a particular scenario results in a better approximation ratio. Based on our studies for all the scenarios of Figure 5.1, we have formulated the following cases.

Case 1: If we have scenario 1 or 6 in prefix (suffix), then we perform a $\tau_p$
Figure 5.1: Different scenarios of breakpoint graphs.

(\tau_s).

Case 2: If we have scenario 2, 3 or, 9 in suffix, then we perform a \tau_s.

Case 3: If we have scenario 4, 5 or, 8 in prefix, then we perform a \tau_p.

Case 4: If we have scenario 7 in prefix (suffix), then we perform a \beta_p (\beta_s).

Case 5: If we have scenario 11 in suffix, then we perform a \beta_s.

Case 6: If we have scenario 12 in prefix (suffix), then we perform a \beta_p (\tau_s).

This scenario emphasizes on (1, 2)-move i.e., 2 consecutive rearrangement operations remove 3 breakpoints.

Case 7: If we have scenario 10 in prefix, then we perform a \tau_p followed by a \beta_p.

Case 7 removes at least two breakpoints in two steps. Case 1 to Case 7
cover all scenarios of Figure 5.1. There may exist more than one scenario in the permutation but at least one scenario from the above cases is always present when $b(\pi) \geq 3$ because there will be at least three black edges. In fact, Figure 5.1 shows all possible combinations when there exists three black edges. When $b(\pi) < 3$, we perform a prefix or suffix breakpoint removal reversal that also removes at least one breakpoint. When more than one scenario arises in the permutation, we choose the operation that removes maximum breakpoints. We discuss about this choosing technique in the experimental section.

Now, we illustrate the steps permutation sorting using an example as shown in Figure 5.2. If we break a strip in a permutation, it may increase the number of breakpoints in that permutation. In this thesis paper, we restrict our focus on the position of a breakpoint while performing any operation so that at least one breakpoint is removed after performing a respective operation. In Figure 5.2(a), we see that there is a breakpoint between $\pi_3$ and
π₄. So, this is our first breakpoint and we will not change anything in the first strip which is π₀...π₃. We find a scenario 12 here. According to Case 7, we can perform either suffix breakpoint removal transposition (τₗ) or prefix breakpoint removal reversal (βₗ). If we apply τₗ, we will get the permutation, π = [0 1 2 3 8 7 6 5 4 9 10] as shown in Figure 5.2(b1). On the other hand, if we apply βₗ, we will get the permutation, π = [0 1 2 3 4 5 8 7 6 9 10] as shown in Figure 5.2(b2). There are two breakpoints in both of these permutations as shown in Figures 5.2(b1) and 5.2(b2). As mentioned earlier, we can now apply βₗ or βₛ operation. Note that the permutation shown in Figure 5.2(b2) has two breakpoints where one breakpoint is between π₅(= 5) and π₇(= 8), and another one is between π₇(= 6) and π₉(= 9). After performing a βₗ operation on both of these permutations we get the identity permutation as shown in Figure 5.2(c).

Lemma 11. For all the scenarios in Figure 5.1, at least one breakpoint is removed after performing a corresponding operation from Cases 1-7.

Proof. Let, π = π₀...πₖ−πₖ₊₁...πₗ−πₗ₊₁...πₙ−πₙ₊₁ be a permutation which represents different scenarios of Figure 5.1. For scenario 1, there will be gray edges among πₖ to πₖ₊₁, πₖ₊₁ to πₙ and πₗ to πₙ₊₁. When a τₗ is applied to π, πₖ comes in contact with πₖ₊₁, πₙ comes in contact with πₖ₊₁ and πₗ₊₁ comes in contact with πₙ. So two breakpoints will be removed. For scenario 2 of Figure 5.1, there will be gray edges between πₖ to πₗ₊₁ and πₖ₊₁ to πₙ₊₁. When a τₗ is applied to π, then πₖ₊₁ comes in contact with πₙ. As a result, one breakpoint is removed.

For scenario 3 of Figure 5.1, there will be gray edges between πₖ₊₁ to πₙ and πₗ to πₙ₊₁. When a τₗ is applied to π, then πₖ₊₁ comes in contact with πₙ and πₗ comes in contact with πₙ₊₁. As a result, at least one breakpoint is removed. For scenario 4, there will be gray edges between πₖ₊₁ to πₙ and πₗ₊₁ to πₙ₊₁. After applying a τₗ to π, πₖ₊₁ comes in contact with πₙ and πₗ₊₁ comes in contact with πₙ₊₁. As a result, at least one breakpoint is removed.

For scenario 5 of Figure 5.1, there will be gray edges between πₖ to πₗ₊₁ and πₖ₊₁ to πₙ. When a τₗ is applied to π, then πₖ comes in contact with πₖ₊₁ and πₖ₊₁ comes in contact with πₙ. As a result, at least one breakpoint

57
is removed. For scenario 6, there will be gray edges between $\pi_{i+1}$ to $\pi_{n+1}$ and $\pi_j$ to $\pi_n$. After applying a $\tau_s$ to $\pi$, $\pi_{i+1}$ comes in contact with $\pi_{n+1}$ and $\pi_j$ comes in contact with $\pi_n$. As a result, at least one breakpoint is removed.

For scenario 7 of Figure 5.1, there will be gray edges between $\pi_i$ to $\pi_n$ and $\pi_{i+1}$ to $\pi_{n+1}$. When a $\beta_p$ is applied to $\pi$, then $\pi_i$ comes in contact with $\pi_n$. As a result, one breakpoint is removed. For scenario 8, there will be gray edges in contact with $\pi_n$ and $\pi_{i+1}$ comes in contact with $\pi_{j+1}$. As a result, at least one breakpoint is removed.

For scenario 9 of Figure 5.1, there will be gray edges between $\pi_i$ to $\pi_j$ and $\pi_{i+1}$ to $\pi_n$. When a $\beta_p$ is applied to $\pi$, then $\pi_i$ comes in contact with $\pi_j$ and $\pi_{i+1}$ comes in contact with $\pi_n$. As a result, at least one breakpoint is removed. For scenario 10, there are gray edges between $\pi_i$ to $\pi_j$ and $\pi_{i+1}$ to $\pi_{n+1}$. When a $\tau_p$ is applied to $\pi$, then $\pi_{i+1}$ comes in contact with $\pi_{n+1}$. Then, we get $|\pi_{i+1} - \pi_{n+1}| = 1$ and one breakpoint is removed.

For scenario 11 of Figure 5.1, there will be gray edges between $\pi_i$ to $\pi_n$ and $\pi_j$ to $\pi_{n+1}$. When a $\beta_p$ is applied to $\pi$, then $\pi_i$ comes in contact with $\pi_n$ and $\pi_j$ comes in contact with $\pi_{n+1}$. As a result, at least one breakpoint is removed. For scenario 12, there are gray edges among $\pi_i$ to $\pi_j$, $\pi_{i+1}$ to $\pi_n$ and $\pi_{j+1}$ to $\pi_{n+1}$. When a $\beta_p$ is applied to $\pi$, then $\pi_i$ comes in contact with $\pi_j$ (or after applying a beta, $\pi_{j+1}$ comes in contact with $\pi_{n+1}$. Thus one breakpoint is removed.

This completes the proof. \qed

5.2 The $(2 + \epsilon)$-Approximation Algorithm

In this section, we present a $(2 + \epsilon)$-approximation algorithm for sorting permutation by both prefix and suffix breakpoint removal transpositions and reversals.

**Theorem 10.** $d(\pi) \geq \left\lfloor \frac{b(\pi) - 2}{2} \right\rfloor$.

**Proof.** This is the minimum sorting distance. At each step, we can perform either a prefix (suffix) breakpoint removal transposition or a prefix (suffix)

58
breakpoint removal reversal. In the best case, we know that it is always possible to perform prefix (suffix) breakpoint removal transposition and each such operation removes two breakpoints. In the last step, after which the permutation becomes sorted, removes three breakpoints at best case. So, the lower bound holds true.

**Theorem 11.** \( d(\pi) \leq b(\pi) - 1 \)

*Proof.* When \( b(\pi) \geq 3 \), at each step, we can perform either a prefix (suffix) breakpoint removal transposition or a prefix (suffix) breakpoint removal reversal. Any of these operation removes at least one breakpoint (Lemma 1). So, at any stage, we need at most \( b(\pi) - 2 \) steps if \( \pi \) is not sorted. When \( b(\pi) = 2 \), then we need one operation to sort the permutation which will remove the two remaining breakpoints. This is the worst case when we will need to perform a prefix/suffix breakpoint removal reversal operation that will remove two remaining breakpoints in the last step. By definition, we will have no breakpoints in identity permutation. So, we need \( b(\pi) - 2 + 1 = b(\pi) - 1 \) operations in total, thus the upper bound holds true.

It is clear that Theorem 10 and Theorem 11 provide a \((2 + \epsilon)\)-approximation algorithm for sorting permutation by both prefix and suffix breakpoint removal transpositions and reversals.

### 5.3 Improved Approximation Algorithms with forward march

In this section, we present some techniques to improve the approximation ratio.

**Lemma 12.** If one of the scenarios 1, 3, 5 (see Figure 5.1) exists in the permutation \( \pi \), then two breakpoints are removed after applying a \( \tau_p \) or \( \tau_s \).

*Proof.* Let, \( \pi = \pi_0\pi_1 - \pi_2 \ldots \pi_i - \pi_{i+1} \ldots \pi_j - \pi_{j+1} \ldots \pi_n \pi_{n+1} \) represents scenario 1 of Figure 5.1 and there are gray edges between \( \pi_1 \) to \( \pi_{i+1} \), \( \pi_2 \) to \( \pi_j \).
and \( \pi_i \) to \( \pi_{j+1} \). As we always remove first or last breakpoint in \( \pi \), after performing a \( \tau_p \) into \( \pi \), \( \pi_1 \) comes in contact with \( \pi_{i+1} \) and \( \pi_2 \) comes in contact with \( \pi_j \). As a result, two breakpoints are removed.

Now, let us consider \( \pi = \pi_0 \pi_1 \ldots \pi_i - \pi_{i+1} \ldots \pi_j - \pi_{j+1} \ldots \pi_{n-1} - \pi_n \pi_{n+1} \) represents scenario 3 of Figure 5.1 and there are gray edges between \( \pi_{i+1} \) to \( \pi_{n-1} \) and \( \pi_j \) to \( \pi_n \). After applying a \( \tau_s \) on \( \pi \), \( \pi_j \) comes in contact with \( \pi_n \) and \( \pi_{i+1} \) comes in contact with \( \pi_{n-1} \). So, two breakpoints are removed.

Again, let us consider \( \pi = \pi_0 \pi_1 - \pi_2 \ldots \pi_i - \pi_{i+1} \ldots \pi_j - \pi_{j+1} \ldots \pi_n \pi_{n+1} \) represents scenario 5 of Figure 5.1 and there are gray edges between \( \pi_1 \) to \( \pi_{i+1} \) and \( \pi_2 \) to \( \pi_j \). So, after performing a \( \tau_p \) to \( \pi \), \( \pi_1 \) comes in contact with \( \pi_{i+1} \) and \( \pi_2 \) comes in contact with \( \pi_j \). As a result, two breakpoints are removed.

We illustrate the proof of Lemma 12 with an example. Suppose, we have an unsigned permutation \( \pi = [0 - 2 1 - 6 7 - 5 4 3 - 8] \) and there are 4 breakpoints in this permutation (shown by ‘-’ between two elements). We see that scenario 3 is present in this permutation because there are grey edges between \( \pi_1 (=2) \) to \( \pi_7 (=3) \) and \( \pi_4 (=7) \) to \( \pi_8 (=8) \). So, after applying a suffix breakpoint removal transposition, \( \tau_s (1, 4, 8) \) on \( \pi \), we get the permutation \([0 - 5 4 3 2 1 - 6 7 8]\) where only 2 breakpoints are left but the permutation is not sorted. Clearly, we need one extra prefix (suffix) reversal operation to get the identity permutation. As we have mentioned earlier that we always focus on removing first or last breakpoint, we always get the last breakpoint before the last strip which consists of last three elements of \( \pi (\pi_6, \pi_7\) and \( \pi_8 \)) and consider the breakpoint between \( \pi_5 (=1) \) and \( \pi_6 (=6) \) to maintain the definition.

5.3.1 Conditions to Improve the Approximation Ratio

Suppose, \(|\pi| = n\), \(b(\pi)\) represents the number of breakpoints and \(k\) denotes the number of operations that removes two breakpoints at each step. So, these \(k\) operations will remove a total of \(2k\) breakpoints requiring a further \(b(\pi) - 2k - 2\) breakpoints to be removed. According to Lemma 11, we can sort this by at most \(b(\pi) - 2k - 2\) operations: the extra 1 is needed if we reach
the reverse sorted position. Then, the total number of operations becomes:

\[ k + b(\pi) - 2k - 2 + 1 = b(\pi) - k - 1. \]

Hence, we get the following approximation ratio:

\[
\rho = \frac{b(\pi) - k - 1}{\frac{k(\pi) - 2}{2} - 1} = \frac{2b(\pi) - 2k - 2}{b(\pi) - 4}
\]

(5.1)

Here, we basically deduce some bounds of \( k \) for which the approximation ratio will be better than 2. When there are \( b(\pi) \) breakpoints in permutation \( \pi \) and at least \( k_1 \) operations are applicable such that each of \( k_1 \) operations removes two breakpoints at each step then we get an approximation ratio of \( \frac{10}{6} \) or 1.667.

\[
\Rightarrow \frac{2b(\pi) - 2k_1 - 2}{b(\pi) - 4} \leq \frac{10}{6} \Rightarrow 6b(\pi) - 6k_1 - 6 \leq 5b(\pi) - 20 \Rightarrow 6k_1 \geq b(\pi) + 14
\]

\[
\Rightarrow k_1 \geq \frac{b(\pi) + 14}{6}
\]

So, we achieve an approximation ratio of 1.667 when the number of operations that removes two breakpoints \( (k_1) \) at each steps is at least one-sixth of the total number of breakpoints, \( b(\pi) \) plus 14. Similarly, when there are \( b(\pi) \) breakpoints in permutation \( \pi \) and at least \( k_2 \) operations are applicable such that each of \( k_2 \) operations removes two breakpoints at each step then we get an approximation ratio of \( \frac{3}{2} \) or 1.5.

\[
\Rightarrow \frac{2b(\pi) - 2k_2 - 2}{b(\pi) - 4} \leq \frac{3}{2} \Rightarrow 4b(\pi) - 4k_2 - 4 \leq 3b(\pi) - 12 \Rightarrow 4k_2 \geq b(\pi) + 8
\]

\[
\Rightarrow k_2 \geq \frac{b(\pi) + 8}{4}
\]

In the same way, we achieve an approximation ratio of 1.5 when the number of operations that removes two breakpoints \( (k_2) \) at each steps is at least one-fourth of the total number of breakpoints, \( b(\pi) \) plus 8. When there are \( b(\pi) \) breakpoints in permutation \( \pi \) and at least \( k_3 \) operations are applicable such that each of \( k_3 \) operations removes two breakpoints at each step then we get an approximation ratio of \( \frac{11}{8} \) or 1.375. We can derive the condition for this case as follows:
\[ \Rightarrow \frac{2b(\pi) - 2k_3 - 2}{b(\pi) - 4} \leq \frac{11}{8} \Rightarrow 16b(\pi) - 16k_3 - 16 \leq 11b(\pi) - 44 \Rightarrow 16k_3 \geq 5b(\pi) + 28 \]

\[ \Rightarrow k_3 \geq \frac{5b(\pi) + 28}{16} \]

We achieve an approximation ratio of 1.375 when the number of operations that removes two breakpoints \( k_3 \) at each steps is at least one-sixteenth of the total number of five times of breakpoints, \( b(\pi) \) plus 28. Similarly, when there are \( b(\pi) \) breakpoints in permutation \( \pi \) and at least \( k_4 \) operations are applicable such that each of \( k_4 \) operations removes two breakpoints at each step then we get an approximation ratio of \( \frac{5}{4} \) or 1.25. We can derive the condition for this case as follows:

\[ \Rightarrow \frac{2b(\pi) - 2k_4 + 2}{b(\pi) - 4} \leq \frac{5}{4} \Rightarrow 8b(\pi) - 8k_4 - 8 \leq 5b(\pi) - 20 \Rightarrow 8k_4 \geq 3b(\pi) + 12 \]

\[ \Rightarrow k_4 \geq \frac{3b(\pi) + 12}{8} \]

So, we achieve an approximation ratio of 1.25 when the number of operations that removes two breakpoints \( k_4 \) at each steps is at least one-eighth of the total number of three times of breakpoints, \( b(\pi) \) plus 12. In Table 5.1, we show the minimum value of \( k_1, k_2, k_3 \) and \( k_4 \) (theoretically) for different values of \( b(\pi) \). We see that all of \( k_1, k_2, k_3 \) and \( k_4 \) increase linearly with increasing values of \( b(\pi) \). We have shown this linearity in Figure 5.3.

We can generally propose same bounds for other genome rearrangement operations also. But in this thesis, we only consider prefix and suffix breakpoint removal transpositions and reversals to demonstrate the results. Now, we present a simple algorithm (see Algorithm 10) where we focus on the effectiveness of proposed bounds rather than efficiency i.e., we motivate to show that the presented condition is sufficient to improve approximation ratio and we do not emphasize much on reducing time complexity. In each single pass of our proposed algorithm (see Algorithm 10), we choose one operation from prefix breakpoint removal transposition, suffix breakpoint removal transposition, prefix breakpoint removal reversal or suffix breakpoint

62
<table>
<thead>
<tr>
<th>$b(\pi)$</th>
<th>$k_1(\rho = 1.667)$</th>
<th>$k_2(\rho = 1.5)$</th>
<th>$k_3(\rho = 1.375)$</th>
<th>$k_4(\rho = 1.25)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>11</td>
<td>15</td>
<td>18</td>
<td>21</td>
</tr>
<tr>
<td>100</td>
<td>19</td>
<td>27</td>
<td>33</td>
<td>39</td>
</tr>
<tr>
<td>200</td>
<td>36</td>
<td>52</td>
<td>65</td>
<td>77</td>
</tr>
<tr>
<td>500</td>
<td>86</td>
<td>127</td>
<td>158</td>
<td>189</td>
</tr>
<tr>
<td>1000</td>
<td>169</td>
<td>252</td>
<td>315</td>
<td>377</td>
</tr>
<tr>
<td>1500</td>
<td>253</td>
<td>377</td>
<td>471</td>
<td>564</td>
</tr>
<tr>
<td>5000</td>
<td>836</td>
<td>1252</td>
<td>1565</td>
<td>1877</td>
</tr>
<tr>
<td>5500</td>
<td>919</td>
<td>1377</td>
<td>1721</td>
<td>2064</td>
</tr>
<tr>
<td>10000</td>
<td>1669</td>
<td>2502</td>
<td>3127</td>
<td>3752</td>
</tr>
<tr>
<td>15000</td>
<td>2503</td>
<td>3752</td>
<td>4690</td>
<td>5627</td>
</tr>
<tr>
<td>50000</td>
<td>8336</td>
<td>12502</td>
<td>15627</td>
<td>18752</td>
</tr>
<tr>
<td>100000</td>
<td>16669</td>
<td>25002</td>
<td>31252</td>
<td>37502</td>
</tr>
</tbody>
</table>

Table 5.1: Minimum number of operations ($k_1, k_2, k_3, k_4$) that removes two breakpoints at each step for different values of $b(\pi)$. $k_1$ represents required number of operations to acquire approximation ratio 1.667, $k_2$ represents required number of operations to acquire approximation ratio 1.5, $k_3$ represents required number of operations to acquire approximation ratio 1.375 and $k_4$ represents required number of operations to acquire approximation ratio 1.25.

removal reversal. We search the first and last position of breakpoints in $\pi$ using a loop and store these into $x$ and $y$ respectively. To apply any of these operations, we scan a permutation from left to right. When a scenario $c$ is found, we check its corresponding number of breakpoint(s) removal and store it in $n_c$. If it is greater than $max_b$ (keeps maximum breakpoints removal for previous scenario), then it is stored in $Q$. To check maximum (at most 2) breakpoint removal, we just need to check two end points of the strip where an operation can be performed and this checking is done in constant time. After completion of scanning, we dequeue the scenario from $Q$ and perform an operation correspondingly. We use $\delta$ to store a permutation temporarily and $b(\delta)$ represents the total number of breakpoints in $\delta$. When $\delta$ is empty then we assume that $b(\delta)$ returns infinity. The value of $k$ represents the number of operations that remove two breakpoints. The inner for loop iterates at most $n$ times considering that there are $n$ elements in the permutation. The outer loop iterates at most $O(n)$ time considering that there are at most $n$ breakpoints. When perform a rearrangement operation that also costs $O(n)$
Figure 5.3: Linear changes of required number of operations that remove two breakpoints at each step with total number of breakpoints.

So, in total, the running time of our proposed algorithm is $O(n^2)$, where $n$ is the number of elements in a permutation.

**Theorem 12.** When $k \geq \frac{b(\pi) + 14}{b}$ in Algorithm 10, then we achieve an approximation ratio of 1.667 where $k$ is the minimum number of operations that removes two breakpoints at each step and $b(\pi)$ is the total number of breakpoints in $\pi$.

**Theorem 13.** When $k \geq \frac{b(\pi) + 8}{4}$ in Algorithm 10, then we achieve an approximation ratio of 1.5 where $k$ is the minimum number of operations that removes two breakpoints at each step and $b(\pi)$ is the total number of breakpoints in $\pi$.

**Theorem 14.** When $k \geq \frac{5b(\pi) + 28}{16}$ in Algorithm 10, then we achieve an approximation ratio of 1.375 where $k$ is the minimum number of operations that removes two breakpoints at each step and $b(\pi)$ is the total number of breakpoints in $\pi$. 

64
Theorem 15. When $k \geq \frac{3b(\pi)+12}{8}$ in Algorithm 10, then we achieve an approximation ratio of 1.25 where $k$ is the minimum number of operations that removes two breakpoints at each step and $b(\pi)$ is the total number of breakpoints in $\pi$.

Now, let us consider another example where $\pi = [01 \ 5674328]$. Clearly, we find the scenario 3 of Figure 5.1 here. If we apply a suffix transreversal operation [8], we get the identity permutation in only one step. But, we need two steps to get identity permutation using prefix and suffix breakpoint removal transpositions and reversals. As suffix transreversal is not our scope here, we conclude that it is possible to improve the approximation factor in such cases using combination of other genome rearrangement operations.

5.4 Experimental Results

In this section, we demonstrate some experimental results for our proposed bounds and also analyze the time complexity of our implemented algorithm. Here, we in fact show that when the number of operations that remove two breakpoints at each step meet a certain bound from any of Theorem 12-15 with the given total number of breakpoints ($b(\pi)$) then it acquires the corresponding approximation ratio. We conduct experiments on both real genomic data and simulated genome data. We collect real genomic data of Bacillus which are available in [42] obtained from NCBI. We generate simulated genome data using a customized program where we set a limit to generate a permutation inclusive within that limit. We calculate optimal distance by Theorem 10 where we first calculate the total number of breakpoints in original permutation. Then we estimate the approximation ratio where we divide the distance found from our algorithm by optimal distance. In case of simulated (in silico) genome data, we generate 20 permutations for each size of permutation. So, we run experiments 20 times for each size.

---

1A transreversal is a genome rearrangement operation which reverses the first block before exchanging two adjacent blocks in a permutation.
of the permutation and report average results\(^2\).

To perform all the experiments, we have used a Desktop PC configured as follows: Windows 7, 8GB RAM, Intel Core i3 3.2GHz processor having codeblocks 10.05 IDE with gnu gcc 4.4.1 compiler. We have also used JDK 1.7.0 to implement some of our customized program.

![Permutation Size vs. Approximation Ratio](image)

Figure 5.4: Average approximation ratio of randomly generated data for different permutation size.

We conduct experiments on two kinds of data sets namely simulated data (randomly generated data) and real genomic data. The experimental results are shown in Figures 5.4 and 5.5. Figure 5.4 shows the average approximation ratio for simulated data and Figure 5.5 shows the approximation ratio for real genomic data. We see from real genomic data that these data sets contain only a little variation in permutation size. To make a profound investigation, we generate larger random genome data from our customized program. Our generated random genome data satisfy the condition \( k \geq \frac{b(\pi)+14}{6} \). Sometimes, the value of \( k \) is much greater than \( \frac{b(\pi)+14}{6} \). When we use random genome data

\(^2\)We get similar result in each run that's why we do not report standard deviation seperately.
data, we run Algorithm 10 20 times for each size of permutation and take the average approximation ratio. We observe that average approximation ratio decreases (i.e., gets better) with the increase of the permutation size. This is due to the fact that higher permutation size increases the number of performing prefix (suffix) breakpoint removal transpositions i.e., the number of maximum (at most two) breakpoint removal increases. Our experimental results show that the approximation ratio is lower than 1.5 in more than 75% cases (see Figure 5.4). In all these cases, we also find that the number of breakpoints are higher i.e., $k$ is also higher. On the other hand, experiments on real genomic data show a little bit different results (see Figure 5.5). Of course, those datasets satisfy the condition of Theorem 12. For real genomic data, approximation ratio does not decrease with the increasing size of permutation. This is due to the fact that, there are lower number of breakpoints in permutations though the permutation size is high. So, we further investigate on breakpoints for real genomic data and find that approximation ratio decreases with increasing number of breakpoints (see Figure 5.6). Clearly,
Figure 5.6: Average approximation ratio of real genome data for different breakpoints.

Figure 5.7: # of breakpoints for different size of permutations.
approximation ratio depends not only on permutation size but also on the number of breakpoints. We report the permutation size of real genomic data and corresponding breakpoints in Figure 5.7. The same analysis holds true for running time of proposed algorithm.

![Running Time Analysis](image)

Figure 5.8: Running time analyses for simple implementation and improved implementation of our proposed algorithm.

In Figure 5.8, we report the comparative running time of our proposed algorithms for two kinds of implementations. We run each implementation five times and take average running time for each size of permutation. In the first kind of implementation which runs in $O(n^2)$, we do not use any sophisticated data structure. In the second kind of implementation, we use permutation...
tree data structure for prefix and suffix breakpoint removal transposition only and we see that running time also decreases. When we use simple implementation it takes $O(n)$ time to perform a prefix or suffix breakpoint removal transposition. On the other hand, when we use permutation tree, it takes at most $O(\log n)$ time to perform each transposition operation [30, 39]. As there are prefix reversals and suffix breakpoint removal reversal operations also, our second implementation does not run in $O(n \log n)$ time, but it reduces running time considerably.
Algorithm 10: ProposedAlgorithm ($\pi$: permutation)

**Input:** $\pi$ is a permutation where, $|\pi| = N$ and $\pi = \pi_1\pi_2\pi_3 \ldots \pi_N$

$k \leftarrow 0; \\
Q \leftarrow \emptyset;$

while $\pi$ is not sorted do

$\text{max}_b, x, y \leftarrow 0; \\
\text{for } i = 1 ; i \leq |\pi| ; i++ \text{ do} \\
\quad \text{if } |\pi_i - \pi_{i-1}| \neq 1 \text{ then} \\
\quad \quad x \leftarrow i; \\
\quad \quad \text{break;}
\text{end}
\text{end}
\text{for } i = |\pi| ; i \geq 0 ; i-- \text{ do} \\
\quad \text{if } |\pi_i - \pi_{i+1}| \neq 1 \text{ then} \\
\quad \quad y \leftarrow i; \\
\quad \quad \text{break;}
\text{end}
\text{end}
\text{for } i = 1 ; i \leq |\pi| ; i++ \text{ do} \\
\quad c_1 \leftarrow \text{Check for a scenario from } 1 - 12 \text{ among } \pi_x \text{ to } \pi_i; \\
\quad c_2 \leftarrow \text{Check for a scenario from } 1 - 12 \text{ among } \pi_i \text{ to } \pi_y; \\
\quad n_c \leftarrow \text{maximum number of breakpoints removal between scenario } c_1 \text{ and } c_2; \\
\quad \text{if } n_c > \text{max}_b \text{ then} \\
\quad \quad Q \leftarrow c; \\
\quad \quad \text{max}_b \leftarrow n_c;
\text{end}
\delta \leftarrow \text{apply operation on } \pi \text{ for scenario } Q; \\
\text{if } b(\pi) - b(\delta) == 2 \text{ then} \\
\quad k \leftarrow k + 1;
\text{end}
\text{if } b(\delta) < b(\pi) \text{ then} \\
\quad \pi \leftarrow \delta; \\
\quad Q \leftarrow \emptyset; \\
\text{end}
\text{else} \\
\quad \pi \leftarrow \text{apply a reversal operation on } \pi; \\
\quad Q \leftarrow \emptyset; \\
\text{end}
\text{end}
\text{end}
Chapter 6

Conclusion

This thesis deals with a topic which falls under the subfield of computational biology, namely genome rearrangement. The main idea of this study is to find the minimum number of genome rearrangement events that are required to sort a given string/permutation. We conclude the thesis presenting our findings from our research and future directions.

6.1 Findings

In this thesis, we have discussed grouping and sorting of fully binary and ternary normalized strings when the allowed operations are prefix transreversal and suffix transreversal. We have handled grouping and sorting binary and ternary normalized strings by prefix and suffix transreversal. In particular we have proved that, for binary strings the grouping distance is $d_{gps}^g(s) = \lceil \frac{n-k}{4} \rceil + 1$, when $n$ is odd and $d_{gps}^g(s) = \lfloor \frac{n-k}{4} \rfloor + 2$, when $n$ is even and for ternary string we have $\lceil \frac{n-k}{2} \rceil \leq d_{gps}^g(s) \leq \lceil \frac{n-k+1}{2} \rceil + 1$, where $n$ is the length of the string and $k$ is the arity. On the other hand, for sorting binary strings the sorting distance $d_{gps}^s(s)$ is same as its grouping distance and for ternary strings, $d_{gps}^s(s) \leq \lceil \frac{n-k+1}{2} \rceil + 2$.

Then we have deduced that, for binary strings the grouping distance is $d_{rpb}^g(s) = \lceil \frac{n-2}{4} \rceil$ using restricted prefix block-interchanges, which is better than existing results on binary strings applying other genome rearrangement
operations [2, 3, 8, 14]. In addition, we have also provided a linear-time algorithm to sort fully normalized binary strings by restricted prefix block interchanges. We have deduced upper bounds applying prefix block-interchanges for grouping and sorting binary normalized strings as $d_{pbi}^g(s) = \lceil \frac{n-2}{3} \rceil + 1$, when we need to sort 10, otherwise $d_{pbi}^g(s) = d_{pbi}^p(s) = \lceil \frac{n-2}{3} \rceil$. We have also deduced upper bounds for both grouping and sorting normalized ternary strings as $d_{pbi}^g(s) \leq \lceil \frac{n-2}{2} \rceil$ and $d_{pbi}^p(s) \leq \lceil \frac{n-2}{2} \rceil + 2$ respectively. We have also presented a $O(n^4)$ time algorithm to group normalized ternary strings, where $n$ is the length of the string.

Finally, we have presented an approximation ratio algorithm for sorting a permutation by both prefix and suffix breakpoint removal transpositions and reversals. We have also presented sufficient conditions to improve approximation ratio of sorting permutation by genome rearrangement events. In our discussion, $k$ is the number of operations that reduce two breakpoints at each steps and $b(\pi)$ is the total number of breakpoints. We have theoretically shown that we must have $k \geq \frac{b(\pi)+14}{6}$ to get an algorithm of approximation factor 1.667, $k \geq \frac{b(\pi)+8}{4}$ to get an algorithm of approximation factor 1.5, $k \geq \frac{3b(\pi)+28}{16}$ to get an algorithm of approximation factor 1.375 and $k \geq \frac{3b(\pi)+12}{8}$ to get an algorithm of approximation factor 1.25. We have also conducted experiments on our proposed 1.667-approximation algorithm over both real genomic data and simulated genomic data for large permutation. Our experimental results show that approximation ratio is infact lower than 1.5 in more than 75% cases. Our algorithm runs in $O(n^2)$ time. We have further proposed an improved running time algorithm that use permutation tree as a backbone data structure.

### 6.2 Future Direction

We can consider higher arity alphabets for strings and apply different combinations of genome rearrangement operations. Improving the upper bound can also be a future development task. Similarly, we can improve the running time of permutation problem applying sophisticated data structure. Besides, improvement of approximation ratio can also be a future development task.
Bibliography


