

STRING ANALYSIS AND ALGORITHMS
WITH GENOMIC APPLICATIONS

by

Adiesha Lakshan Liyanage Liyana Ralalage

A dissertation submitted in partial fulfillment
of the requirements for the degree

of

Doctor of Philosophy

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

May 2024

©COPYRIGHT

by

Adiesha Lakshan Liyanage Liyana Ralalage

2024

All Rights Reserved

DEDICATION

In profound gratitude, I dedicate this work to my parents, Laksiri and Champika, for their unwavering support and encouragement. To my family members—my aunt Sudanthi, my sister Piyumi, and my loving girlfriend Supuni and her family—your unwavering support, encouragement, and belief in me have been consistent sources of inspiration.

To my friends and teachers, your encouragement and emotional support have sustained me through the challenges of my doctoral pursuit.

I express my profound gratitude to my advisor, Dr. Binhai Zhu, for your guidance, endless patience, and unwavering support in my career. Your mentorship has been instrumental in shaping my academic growth and personal development.

I dedicate this dissertation in loving memory of my dear friend Harish Thiruvalluvan, whose invaluable insights profoundly influenced my research journey.

To each of you who touched my life with your support, kind words, wisdom, and encouragement, I am truly grateful to you.

ACKNOWLEDGEMENTS

I am profoundly grateful to my advisor, Dr. Binhai Zhu, for his patient guidance and unwavering encouragement over the course of these five years. I am truly fortunate to have had such a dedicated mentor who consistently challenged and inspired me to strive for excellence.

I would also like to thank all my committee members, Dr. Brendan Mumey, Dr. Sean Yaw, and Dr. Lucia Williams, for their guidance and helpful comments.

I would also like to express my gratitude to my family and friends for their support and encouragement.

TABLE OF CONTENTS

1. INTRODUCTION	1
1.1 Motivation	1
2. THE COMPLEXITY OF FINDING COMMON PARTITIONS OF GENOMES WITH PREDEFINED BLOCK SIZES.	7
2.1 Introduction	7
2.2 Related Work	8
2.3 Preliminaries	9
2.4 F -Strip Recovery Problems	10
2.4.1 The Exact F -Strip Recovery With Fixed Block Sizes	10
2.5 GSR-F in Polynomial Time for Fixed Block Sizes and Alphabet	19
2.6 Fixed Alphabet with Unbounded Block Sizes Is NP-hard	26
2.7 Conclusion	28
2.7.1 Notes	29
3. THE LONGEST LETTER-DUPPLICATED SUBSEQUENCE PROBLEM	30
3.1 Introduction	30
3.2 Preliminaries	32
3.3 LLDS Problem	33
3.3.1 The Variants of LLDS	34
3.4 Hardness Result for LLDS+(d) and FT(d) When $d \geq 4$	35
3.4.1 FT(4) Is NP-complete	35
3.4.2 Inapproximability Results	40
3.5 Polynomial Time algorithms for LLDS+(d) and FT(d) When $d = 3$	42
3.5.1 Solving Feasibility Testing When $d = 3$	42
3.5.2 Optimization Version of LLDS+(3) When $d = 3$	43
3.6 Dynamic Programming Algorithm for Weighted-LDS	45
3.7 Conclusion	47
3.7.1 Notes	48
4. THE LONGEST SUBSEQUENCE-REPEATED SUBSEQUENCE PROBLEM	49
4.1 Introduction	49
4.2 Preliminaries	51
4.3 LSRS Problem	52
4.3.1 Solution for The LSRS Problem	53

TABLE OF CONTENTS – CONTINUED

4.4	LSRS+ Problem	53
4.5	LSRS+(3) Is Polynomially Solvable	54
4.6	FT(4) and LSRS+(4) Are NP-hard.....	58
4.7	An $O(n^6)$ Algorithm for The Longest Cubic Subsequences of All Substrings in a Given Sequence	62
4.8	Conclusion	68
4.8.1	Open Problems.....	69
4.8.2	Notes	69
5.	DISTANCE MEASUREMENT FOR MITOCHONDRIAL DNA COM- PARISON UNDER TDRL (TANDEM DUPLICATION RANDOM LOSS) MODEL FOR NON-PERMUTATION STRINGS	70
5.1	Introduction	70
5.2	Background.....	73
5.3	Heuristic Algorithms.....	78
5.3.1	Heuristic Algorithm for $\alpha = 1, \beta \leq \alpha$	78
5.3.2	Heuristic Algorithm for $\alpha \geq 2$	84
5.3.2.1	Algorithm to Create Stabilized Permutations Given Two Strings S, T	86
5.3.3	Experiments.....	91
5.3.3.1	Experiments on Real-World Data Sets.	91
5.4	Ancestral Gene Order Reconstruction Using The Feedback Arc Set Problem for Non-Permutation Strings.....	96
5.4.1	Reducing Ancestral Gene Order Reconstruction Into a Feedback Arc Set Problem	96
5.4.2	Experiments.....	100
5.5	Conclusion	108
6.	SUMMARY	110
7.	CONCLUSION	111
	REFERENCES CITED.....	114

LIST OF TABLES

Table	Page
3.1 Summary of results on LLDS+ and FT	48
5.1 Occurrences of each character $c \in \Sigma$ in S and T	86
5.2 The table presents the results obtained by running the heuristics algorithm with parameters $\alpha = 1$ and $\beta = 1$. It displays the distances from the vertebrate gene order to 43 different species, calculated using the heuristics algorithm.	93
5.3 Heuristics distance between vertebrate gene order and 44 species using the heuristics distance method, where $\alpha = 2$ and $\beta = 1$ parameters. The $d(S, S')$ column indicates the cost of transforming S to S' , while the $d^*(T', T)$ column indicates the cost of transforming T' to T	95
5.4 This table contains a summary of the ancestor gene order prediction tests. TD(A,S) and TD(A,T) columns represent the minimum tandem duplication needed to transform the ancestor (A) into S and T . The elapsed time column contains the time taken by the algorithm to generate the ancestor sequence (ms). The TD(S,T) column represents the minimum number of tandem duplications required to transform S to T	103

LIST OF FIGURES

Figure	Page
2.1 Partition of X_i and X'_i corresponding to $x_i = true$	13
2.2 Partition of X_i and X'_i corresponding to $x_i = false$	13
2.3 Strings A and B	27
3.1 Example Graph for the $(3^+, 1, 2^-)$ -SAT problem reduction.	40
4.1 Example Graph for the $(3^+, 1, 2^-)$ -SAT problem reduction.	58
4.2 An illustration of the subgrid containing $v_{0,b,c}$ cells. (the y -axis has the b coordinates, the z -axis the c coordinates). We can think of the cell (a, b, c) in the 3D-grid as the vertex $v_{a,b,c}$	64
5.1 Example for the mapping between two strings S and T . Note that there can be some characters in S that are not mapped or mapped to more than one character in T	97
5.2 Three possible directed acyclic graphs achievable by remov- ing the minimum number of blue and red edges, based on the example presented in Figure 5.1.	100
5.3 Ancestor gene order prediction between Siphonops Annula- tus and Trachyrincus Murrayi.	104
5.4 Ancestor gene order prediction between Siphonops Annula- tus and Ponticola Kessleri.	104
5.5 Ancestor gene order prediction between Hamaeleo Africanus and Siphonops Annulatus.	104
5.6 Ancestor gene order prediction between Siphonops annula- tus and Lampris Guttatus.	104
5.7 Ancestor gene order prediction between Rhineura Floridana and Siphonops Annulatus.	105
5.8 Ancestor gene order prediction between Siphonops Annula- tus and Hapalogenys Analis.	105
5.9 Ancestor gene order prediction between Normichthys Oper- osus and Pagothenia Borchgrevinki.	105

LIST OF FIGURES – CONTINUED

Figure	Page
5.10 Ancestor gene order prediction between <i>Dallia pectoralis</i> and <i>Muraenesox bagio</i>	105
5.11 Ancestor gene order prediction between <i>Dallia Pectoralis</i> and <i>Squalogadus Modificatus</i>	106
5.12 Ancestor gene order prediction between <i>Clinocottus Analis</i> and <i>Aulorhynchus Flavidae</i>	106
5.13 Ancestor gene order prediction between <i>Ventrifossa Gar-</i> <i>mani</i> and <i>Normichthys Operosus</i>	106
5.14 Ancestor gene order prediction between <i>Aulorhynchus Flavidae</i> and <i>Ponticola Kessleri</i>	106
5.15 Ancestor gene order prediction between <i>Rhineura Floridana</i> and <i>Chamaeleo Africanus</i>	107
5.16 Ancestor gene order prediction between <i>Hoplunnis Punc-</i> <i>tata</i> and <i>Ventrifossa Garmani</i>	107
5.17 Ancestor gene order prediction between <i>Hemisis Marmora-</i> <i>tus</i> and <i>Chamaeleo Arabicus</i>	107
5.18 Ancestor gene order prediction between <i>Hemisis Marmora-</i> <i>tus</i> and <i>Hyperolius Marmoratus</i>	107
5.19 Ancestor gene order prediction between <i>Hyperolius Mar-</i> <i>moratus</i> and <i>Trichobatrachus Robustus</i>	107
5.20 Ancestor gene order prediction between <i>Hyperolius Mar-</i> <i>moratus</i> vs <i>Breviceps Adspersus</i>	108

LIST OF ALGORITHMS

Algorithm	Page
2.1 Pseudocode for the computation of the dynamic programming.	25

ABSTRACT

In biology, genome rearrangements are mutations that change the gene content of a genome or the arrangement of the genes on a genome. Understanding how genome rearrangements occur in a genome can help us to understand the evolutionary history of extant species, improve genetic engineering, and understand the basis of genetic diseases. In this dissertation, we explored four problems related to genome partitioning and tandem duplication and deletion rearrangement operations. Our interest was focused on determining how difficult it is to solve these problems and identifying efficient algorithms to solve them. The proposed problems were formulated as string problems and then analyzed using complexity theory.

In the first chapter, we explored several variations of F -strip recovery problem called XSR-F and GSR-F and their complexity under different parameters. We proved that the XSR-F problem is hard to solve unless we restrict the allowed block sizes to one size. We provided a polynomial time algorithm for GSR-F under a fixed alphabet and fixed F . In the second and third chapters, we introduced two string problems named longest letter-duplicated subsequence (LLDS) and longest subsequence-repeated subsequence (LSRS)—formulated as alternative problem formulations for the tandem-duplication distance problem that allow to extract information about segments of genes that may have undergone tandem duplication—analyzed the complexity of their variations and devised efficient algorithms to solve them. We proved that constrained versions of LLDS and LSRS problems are NP-hard for parameter $d \geq 4$, while general versions were polynomially solvable which hints that any variations closer to the original tandem duplication distance problem are still hard to solve. In the final chapter, we delved into two heuristic algorithms designed to compute genomic distance between two mitochondrial genomes and a heuristic algorithm to predict ancestral gene order under the TDRL (tandem-duplication random loss) model. We improved the previously studied method developed for permutation strings by tweaking heuristic choices aimed at calculating the minimum distance between two genomes to apply to non-permutation strings. These heuristic algorithms were implemented and tested on a real-world mitochondrial genome data set.

CHAPTER ONE

INTRODUCTION

1.1 Motivation

During biological evolution, the gene content and the arrangement of the genes within the genomes of species have been modified by various kinds of mutations. A genome is made of a set of chromosomes, and each chromosome contains a set of genes. A chromosome represents a circular or linear string of DNA where genes are located. Each gene has an orientation depending on which of the two strands of DNA it is located on. A change or mutation that changes the arrangement of the genes on a chromosome is called a rearrangement mutation (or rearrangements). These rearrangements can occur in different ways; for example, inversion, which reverses the order and orientation of a subsequence of genes in a chromosome; transpositions, which move a segment of a chromosome to another location on the chromosome; duplications, which duplicate a part of the chromosome and add it to the chromosome; see reference [50]. Some other examples are fusion, which merges two chromosomes and forms a single chromosome, and fission, which splits a chromosome into two chromosomes. Basically, there are different types of rearrangement operations that can occur in nature. Some rearrangement operations do not change the content of the genome, while others do; for example, when a deletion occurs, a chromosome can lose some contents, whereas rearrangements like transpositions or translocations do not change the contents of the genome.

Gene order analysis helps to explain the differences in gene arrangement between extant species and helps to construct their phylogenetic evolution. For example, maximum

parsimony is used to analyze the gene order, in which it tries to explain the differences within a set of genomes by the minimum possible number of changes. One can ask what the minimum number of rearrangement operations is that we need to perform to transform one genome into another genome; this is called the genome sorting problem. The problem that asks for the smallest number of such mutations is called the distance problem. Such distance information can be used to construct the phylogenetic relationship between species [106].

Combinatorial and computational problems related to gene order analysis started with Sankoff and Blanchette [97] and Watterson et al. [109]. Some variations of the sorting problems were shown to be solved in polynomial time, such as sorting with inversions [47] and sorting with inversions plus fission and fusion [48]. Many other related problems in rearrangement space have been shown to be NP-complete, such as sorting with inversions for unsigned genomes (ignoring the orientation of genes) [20], the inversion median problem (the problem of finding the parsimonious common ancestral gene order from more than two given gene orders) [21], and the problem of sorting by transposition [16]. Therefore, a lot of research has been done on finding heuristic algorithms for such problems. The MGR algorithm for solving the inversion median problem [11] and approximation algorithms for sorting by transposition are some examples of heuristic algorithms [39].

Another important task in comparative genomics is to partition two genomes with duplicated genes into syntenic blocks, which are blocks that have the same gene content. Finding these blocks could help to identify ancestral gene order [2, 12], reconstruction of phylogenies [36, 45], gene tree reconciliation [32], the study of whole genome duplication [38] or prediction of orthologous genes [69, 77]. Extracting information about these syntenic blocks when comparing genomes would help us find duplications or deletions that occurred during the evolution of genomes. A popular problem to find these common syntenic blocks in genomes is the Minimum Common String Partition (MCSP) problem, where two genomes are represented as strings and the problem asks to find a minimum number of blocks in

a common partition. The purpose of this problem was to extract information about the orthology assignment [25]. Orthology assignment is the process of identifying orthologous genes (genes in different species that evolved from a common ancestor by speciation) among a set of sequences. Orthology assignment is useful for functional inference, comparative genomics, and phylogenetics [33]. It has been argued that MCSP does not necessarily give the best partition in terms of syntenic blocks due to the fact that MCSP may output syntenic blocks that are too small in order to minimize the number of blocks. Therefore, in [115] authors argued that small blocks should be treated as noise, then proposed an alternate problem formulation called *Strip Recovery*. This problem aims to find a common partition in which blocks must have at least length k , where k is an input parameter. This led to another alternate problem called the *Maximal Strip Recovery (MSR)* problem, in which the goal is to find the maximum length subsequences S' and T' of the input genomes S and T such that S' and T' admit a common string partition with the desired block sizes. The minimization version of this problem is called *Complementary MSR*, where one asks to delete the minimum number of genes such that two input genomes admit a common string partition with desired block sizes. However, MSR has not received as much attention as MCSP. Furthermore, most of the attention given to MSR was for the permutation model, in which genomes do not contain duplicates. Therefore, it would be interesting to dig further into the theoretical and algorithmic aspects of MSR with duplicated genes.

In biology, duplication is a major rearrangement operation in biological evolution. There are two types of duplications: arbitrary duplications (a segment in a gene is copied and inserted somewhere else in the gene) and tandem duplications (in the form of $X \rightarrow XX$), where X is a segment of the input sequence. The important property of tandem duplication is that the copied segment is inserted at the copied location. It is known that tandem duplications of 3 nucleotide *CAG* is closely related to the Huntington disease [82]. Whole-genome sequencing (WGS) data from tumors has revealed that massive rearrangements,

as in the tandem duplicator phenotype, are a specific cancer phenotype [54]. Tandem duplications occur in some cancers [103], particularly in ovarian and breast cancer genomes. A subset of ovarian cancer samples shares a marked tandem duplication phenotype with triple-negative breast cancers [84]. In 1980, Szostak and Wu showed evidence that gene duplication is the main driving force behind evolution and that the majority of duplications are tandem [104]. In the first sequenced human genome, around 3% are in the form of tandem repeats [73]. Tandem duplications were further studied in copying systems [37] and formal languages [13, 30, 110]. In 2020, Lafond et al. answered the open question regarding the complexity of computing the minimum tandem duplication distance between two sequences A and B (the minimum number of tandem duplications to convert A to B). The problem was shown to be NP-complete for the unbounded alphabet [71]. Later, Lafond et al. proved that the problem is NP-hard even when $\Sigma \geq 4$ by encoding each letter in the unbounded alphabet proof with square-free string over a new alphabet of size 4 (using the method from Leech’s construction in [78]). This is important since it covers the most relevant case in biology, where $\Sigma = \{A, C, G, T\}$ for DNA sequences and $\Sigma = \{A, C, G, U\}$ for RNA sequences [71]. Therefore, it would be interesting to find new problem formulations that could extract information about tandem duplications between two genomes. Note that in the evolution of genomes, there could be multiple rearrangement operations that occur. Suppose some genomes undergo tandem duplications and some other mutations like insertions and deletions; then, due to mutations in the genome, it would be harder to extract information about the segments that underwent duplications. Therefore, it would be interesting to propose new problem formulations that would allow us to extract information about duplicated segments under tandem duplications. For example, given two genomes with a starting sequence and a target sequence where the target sequence has undergone some rearrangements, it would be interesting to see which blocks have been duplicated by tandem duplications while disregarding some mutations (deletions, insertions, etc.) that may have

occurred.

This dissertation presents four problems related to genome rearrangement. In Chapter 2, we explored several variations of the Maximal Strip Recovery (MSR) problem, called the *F-Strip Recovery problem*. We proved that the Exact *F*-Strip Recovery problem is NP-hard for unbounded alphabets and unbounded *F* (block size list). Notably, we demonstrated that it is challenging to partition two genomes into blocks of sizes from any fixed list, unless we restrict ourselves to one size. However, we presented a polynomial-time algorithm for the optimization version of the GSR-F problem for fixed *F* and alphabets. In Chapter 3, we focused on the longest letter-duplicated subsequence problem and its variants. We introduced polynomial-time algorithms for the LLDS and weighted LLDS problems. Furthermore, we proved that the decision version of the LLDS+(*d*) problem is NP-complete, even when the maximum number of duplications *d* of each gene is 4. We also demonstrated that obtaining a bi-criteria approximation for LLDS+(4) is as difficult as approximating LLDS+(4). However, we found that when *d* = 3, the problem is polynomially solvable. In Chapter 4, we introduced the Longest Subsequence-Repeated Subsequence (LSRS) problem. We proved that the decision version of the LSRS+(*d*) problem is NP-complete, even when the maximum number of allowed duplications *d* of each gene is 4. However, we also demonstrated that when the maximum number of allowed duplications is 3, the problem is polynomially solvable. Additionally, we provided an algorithm to calculate the longest cubic subsequences of all substrings in a given sequence in $O(n^6)$ time. In chapter 5, we looked at a new distance measurement for calculating the distance between mitochondrial gene orders under the tandem duplication random loss (TDRL) model for non-permutation strings. We introduced two heuristic algorithms to calculate the distance between two mitochondrial gene orders under α, β parameters, which determines the cost of duplications and deletions under this model. Finally, we introduced a heuristics algorithm to calculate the ancestral gene order of two given non-permutation gene orders. Throughout this dissertation, we are interested

in the theoretical and algorithmic aspects of these problems, including complexity, decision algorithms, and optimization algorithms.

CHAPTER TWO

THE COMPLEXITY OF FINDING COMMON PARTITIONS OF GENOMES WITH
PREDEFINED BLOCK SIZES.2.1 Introduction

In comparative genomics, a common task is to partition two genomes with duplicated genes into *syntenic blocks*, which are blocks that have the same gene content and order. Finding these blocks can have many applications, such as inference of ancestral gene order [2, 12], reconstruction of phylogenies [36, 45] or prediction of orthologous genes [32]. The concept of synteny is a loosely defined concept; however, in most definitions, it requires that syntenic blocks have a minimum number of genes in common. If we are given two genomes, S and T then finding syntenic blocks can be modeled as finding common string partitions, where we need to split the genomes into two equal-sized sets of substrings. A popular formulation is the *Minimum Common String Partition* (MCSP), where given two genomes, we are asked to find the common string partitions such that the number of partitions is minimum. This problem was initially proposed [25] for the purpose of an orthology assignment. However, it is argued that MCSP does not necessarily find the best partition in regards to syntenic blocks. The reason is that it might infer blocks that are too small for syntenic blocks.

In [115], it was argued that these small blocks can be treated as noise in the context of genome comparison and proposed the *Strip Recovery* formulation. In this version, the goal is to find a common string partition such that every block has a length of at least k , where k is an input parameter. It could be that such a partition does not exist. This leads to the *Maximal Strip Recovery* (MSR) problem. In this version, the goal is to find the maximum-length subsequences S' and T' of S and T such that S' and T' are a common string partition with desired block sizes. The minimization version of this problem is called *Complementary*

Maximal Strip Recovery, which asks to delete a minimum number of genes from the two strings so that they admit a desired block partition.

However, MSR has received less attention compared to the MCSP problem. In fact, an open problem from [18] states that it is not known whether we could find a common partition into blocks of size 2 or more without deleting any gene. Furthermore, previous work on MSR mainly focused on the permutation model, where we consider genomes without duplicates. However, most of the applications deal with genomes with duplicates.

In this chapter, we are interested in understanding the theoretical and algorithmic aspects of MSR with duplicated genes. Here, we will provide several generalizations of the MSR problem. First, we allow the input to specify an arbitrary list F of allowed block sizes in the desired partition, with F possibly being infinite. With this generalization, the maximal strip recovery problem becomes a special case in which the allowed input block sizes are $F = \{2, 3, 4, \dots\}$. Furthermore, if we want the common partition with block sizes of at least size k then list F becomes $F = \{k, k + 1, \dots\}$. In fact, in this chapter, we will show that an infinite set F of block sizes can be reduced to a finite set F' that is not necessarily contiguous. Second, we allow the input to consist of two sets of strings, $\mathcal{W}_1$ and $\mathcal{W}_2$ instead of individual strings. The two sets can differ in size and total string length. We can use this generalization to analyze comparisons of multiple chromosomes or unrelated segments between two species.

2.2 Related Work

The decision version of the MSR and complementary MSR problems were shown to be NP-Complete by Wang and Zhu in [107]. Then MSR and CMSR were proved to be APX-hard in [15, 63]. Some heuristic algorithms based on the maximum independent set and maximum clique were used in [27, 115]. Chen et al. proposed a factor-4 approximation algorithm for MSR in [26]. Then, in [59], a factor-3 approximation algorithm for CMSR

and an $O^*(3^k)$ FPT algorithm were proposed, where k is the parameter representing the minimum number of deleted genes. Then the approximation factor was improved to 2 in [58]. In terms of fixed-parameter tractability, Jiang and Zhu obtained a linear kernel of size $78k$ for CMSR in [60]. This kernel was improved to $58k$ in [53] and recently to $42k$ in [80]. Combined with these kernel bounds, CMSR can be solved in $O(n^2 + 2.36^k k^2)$. The MCSP problem received some algorithmic improvements in [44, 80]. The best known approximation to minimize the number of blocks has a factor of $O(\log n)$ [42]. Then the fixed parameter tractability of MCSP was considered in [29]. For d -occurrence strings, the problem was shown to be FPT in [29, 61]. Lastly, in 2014, MCSP was shown to be FPT parameterized by solution size ℓ [17]. More recently, in 2021, Lafond and Zhu considered variants of MCSP when a permutation was given in advance [70].

2.3 Preliminaries

Let S be a string. We denote the size of S using $|S|$ and $S[i..j]$ for the substring of S that starts from position i and ends at position j . We denote the integer set $\{1, 2, 3, 4, \dots, n\}$ using $[n]$. In this chapter, each individual character is called a marker. Thus, S has $|S|$ distinct markers, and two markers may represent the same symbol but are considered distinct if they are located at different positions on S . We say that T is a subsequence of S if T can be obtained by deleting some set of markers from S . A permutation string is a string in which each symbol occurs exactly once.

A partition of the string S can be used to reconstruct S by concatenating them. The strings in the partition set are called blocks. If $\mathcal{W}$ is a set of strings, then the partition of $\mathcal{W}$ is the union of the partitions of each string $w \in \mathcal{W}$. For a given set of integers F , we say that partition M is an F -partition if the length of each block of M is in F . A common partition of two sets of strings $\mathcal{W}_1$ and $\mathcal{W}_2$ is a multiset of strings M such that M is a partition of both $\mathcal{W}_1$ and $\mathcal{W}_2$. M is also a common F -partition if M is also an F -partition.

2.4 F -Strip Recovery Problems

In this section, we are going to define the variants of the F -Strip Recovery problem that we are going to propose. Here we give definitions for two variants of the strip recovery problem. Assume that F is a fixed set of integers, possibly infinite.

Definition 1. The General F -Strip Recovery Problem ($GSR-F$)

Input: *Two sets of strings $\mathcal{W}_1$ and $\mathcal{W}_2$.*

Question: *Delete a minimum number of markers from the strings in $\mathcal{W}_1$ and $\mathcal{W}_2$ so that the resulting strings admit a common F -partition.*

Definition 2. The Exact F -Strip Recovery Problem ($XSR-F$)

Input: *Two sets of strings $\mathcal{W}_1$ and $\mathcal{W}_2$.*

Question: *Does there exist a common F -partition of $\mathcal{W}_1$ and $\mathcal{W}_2$?*

The original Strip Recovery problem can be considered a special case of the $GSR-F$, where $|\mathcal{W}_1| = |\mathcal{W}_2| = 1$ and $F = \{2, 3, \dots\}$. Note that the set F here is fixed, but we will consider the variant of $XSR-F$ in which F is unbounded but the alphabet is fixed in later sections.

2.4.1 The Exact F -Strip Recovery With Fixed Block Sizes

First, we will provide results for the $XSR-F$ problem, in which $XSR-F$ can be solved in polynomial time if all elements of F are multiples of its minimum; otherwise, it is NP-hard. The problem with an infinite set F is that: how does an algorithm use an infinite set to access its elements? The idea is to replace the infinite set with an equivalent finite set.

Definition 3. *For two sets F and F' , we say that $XSR-F$ and $XSR-F'$ are equivalent if $\mathcal{W}_1, \mathcal{W}_2$ is a yes-instance of $XSR-F$ if and only if $\mathcal{W}_1, \mathcal{W}_2$ is a yes-instance for $XSR-F'$*

Definition 4. For a set F of integers and an integer t , we say that t is expressible by F if there exists $\{g_1, \dots, g_k\} \subseteq F$ and positive integer coefficients $a_1, \dots, a_k$ such that $t = \sum_{i=1}^k a_i g_i$.

Lemma 1. $XSR-F$ and $XSR-F'$ are equivalent if and only if the set of integers expressible by F is equal to the set of integers expressible by F' .

Proof. First, let us assume that $XSR-F$ and $XSR-F'$ are equivalent. Then, consider $k \in F$. Then, if we look at an instance of $XSR-F$ that is $\mathcal{W}_1 = \mathcal{W}_2 = \{a^k\}$, where a^k is a string with k a 's. Then, if this is also a yes instance of $XSR-F'$, the only way this is possible is if k is expressible with F' . Since this holds for every $k \in F$, all elements are expressible with F' . Using the same idea, all elements in F' should be expressible by F as well.

On the other hand, suppose F and F' express the same set of elements. Then, let $\mathcal{W}_1, \mathcal{W}_2$ be a yes instance of $XSR-F$ and let M be the common F -partition. Now, if we take any block B in M , it is clear that $|B|$ could be expressed by F . Take the value that is used to express B in partition M ; this value is expressible by integers in the set F' (by the definition); therefore, B should be expressible by F' . If we apply the same idea to all the blocks in M , we prove that if $\mathcal{W}_1 = \mathcal{W}_2$ is a yes instance of $XSR-F$ then they are a yes instance of $XSR-F'$ as well. Thus, $XSR-F$ and $XSR-F'$ are equivalent. $\square$

Now, we show that when F is a set with infinite number of positive integers, it could be reduced to finite set $F' \subseteq F$ such that $XSR-F$ and $XSR-F'$ are equivalent.

Proposition 1. Let F be an infinite set of positive integers. Then there exists a finite set $F' \subseteq F$ such that $XSR-F$ and $XSR-F'$ are equivalent.

Proof. The first observation is that there must exist a finite set $G \subseteq F$ such that $\gcd(G) = \gcd(F)$. We can observe this by using the following procedure: Start with $G = \{\min(F)\}$. If

$\gcd(G) = \gcd(F)$, then we are done; otherwise, there must be some $k \in F$ such that $\gcd(G)$ does not divide k . Add k to G . Now observe that the gcd of $\{\min(F), k\}$ is strictly less than the gcd of $\{\min(F)\}$. Therefore, we can repeat this process until we get a set G such that $\gcd(G) = \gcd(F)$. Since at each step gcd decreases monotonically, the process finishes in $\min(F)$ steps.

Now let us use g to denote $g := \gcd(G) = \gcd(F)$. Since every $k \in F$ is a multiple of g , every integer expressible by F is a multiple of g . Let us argue that there exists an integer $t(G)$ such that every multiple of G greater than $t(G)$ is expressible by G . If $g = 1$, this means that every integer above some $t(G)$ is a linear combination of elements of G . In this case, one of Schur's theorems states that $t(G)$ is known to exist. If $g \geq 2$, let $G' = \{x/g : x \in G\}$. Then $\gcd(G') = 1$, as otherwise G would have the common divisor $g \cdot \gcd(G')$, which is larger than g . Thus, there is some $t(G')$ such that every integer greater than $t(G')$ is expressible by G' . This implies that every multiple of g greater than $t(G) := g \cdot t(G')$ is expressible by G .

Since $G \subseteq F$, every multiple of g greater than $t(G)$ can be expressed by F , and these are only expressible integers greater than $t(G)$. It follows that integers expressible by F and $F' = G \cup \{x \in F : x \leq t(G)\}$ are equal. By Lemma 1, $XSR - F$ and $XSR - F'$ are equivalent. $\square$

Proposition 2. *Let F be set of integers and let $\ell = \min(F)$. If every element of F is a multiple of ℓ , then $XSR - F$ is solved in polynomial time.*

Proof. Let $\mathcal{W}_1, \mathcal{W}_2$ be an instance of XSR-F. Then, by Lemma 1, $XSR - F$ and $XSR - \{\ell\}$ are equivalent. Therefore we can assume $F = \{\ell\}$. This means that all the string partitions must have the same length ℓ ; therefore, there is only one way to split $\mathcal{W}_1$ and $\mathcal{W}_2$. Then we can conclude that $\mathcal{W}_1$ and $\mathcal{W}_2$ admit a common F -partition if and only if each substring in the forced partitions appears with the same number of occurrences. $\square$

Theorem 1. *Let F be any set of integers in which some element is not a multiple of $\min$*

F. Then XSR-F is NP-Hard, even when each symbol appears at most 6 times in the input strings.

$$\begin{aligned} X_i &= \underline{\underline{P_i^1 \ P_i^2 \ \dots \ P_i^r \ P_i^* \ L_a \ M_a \ R_a \ Q_i \ L_b \ M_b \ R_b \ S_i \ L_c \ M_c \ R_c \ T_i^* \ T_i^1 \ T_i^2 \ \dots \ T_i^r}} \\ X'_i &= \underline{\underline{P_i^1 \ P_i^2 \ \dots \ P_i^r \ P_i^* \ L_a \ M'_a \ R_a \ Q_i \ L_b \ M'_b \ R_b \ S_i \ L_c \ M'_c \ R_c \ T_i^* \ T_i^1 \ T_i^2 \ \dots \ T_i^r}} \end{aligned}$$

Figure 2.1: Partition of X_i and X'_i corresponding to $x_i = \text{true}$.

$$\begin{aligned} X_i &= \underline{\underline{P_i^1 \ P_i^2 \ \dots \ P_i^r \ P_i^* \ L_a \ M_a \ R_a \ Q_i \ L_b \ M_b \ R_b \ S_i \ L_c \ M_c \ R_c \ T_i^* \ T_i^1 \ T_i^2 \ \dots \ T_i^r}} \\ X'_i &= \underline{\underline{P_i^1 \ P_i^2 \ \dots \ P_i^r \ P_i^* \ L_a \ M'_a \ R_a \ Q_i \ L_b \ M'_b \ R_b \ S_i \ L_c \ M'_c \ R_c \ T_i^* \ T_i^1 \ T_i^2 \ \dots \ T_i^r}} \end{aligned}$$

Figure 2.2: Partition of X_i and X'_i corresponding to $x_i = \text{false}$.

Proof. We reduce the Positive Cubic XSAT problem to the XSR-F problem. In this SAT problem, we are given a boolean formula ϕ in which each variable occurs exactly three times and each clause has exactly three literals that are all positive. The objective is to find whether there is an assignment such that each clause has exactly one of its variables that is true (the other two are false). When we say that ϕ is 1-in-3-satisfiable if there exists a truth assignment that completes the objective, this problem was shown to be NP-hard in [86].

Now, let ϕ be an instance of this problem, and let $x_1, x_2, \dots, x_n$ be its variables, and let $C_1, \dots, C_m$ be its clauses. Then let $\ell = \min F$ and let h be the smallest element of F such that h is not a multiple of ℓ . We assume that h exists and F is finite. If F is not fixed, we convert it to a finite set, F' . Therefore, we treat F as a finite set. Then we can find ℓ and h in constant time. Note that every element between ℓ and h in F is a multiple of ℓ , and by Lemma 1, we can ignore them and still have an equivalent instance. Therefore, we can assume h is the second-smallest element of F .

Then, let $r = \lfloor h/\ell \rfloor$ be the number of times we can fit ℓ into h . Let $s = \ell(r + 1) - h$, which is the excess we get if we try to fit one more ℓ . Since ℓ is not a divisor of h , we know that $0 < s < \ell$ ($h = r\ell + \ell - s$).

Now we define a set of strings that would act as the building blocks for the instance. Each string mentioned here is a permutation string; therefore, no symbol occurs more than once, and no symbol is shared among the other strings.

To each clause C_a in ϕ corresponds to four strings $\mathbb{L}_a, \mathbb{M}_a, \mathbb{M}'_a, \mathbb{R}_a$ (which stand for left, middle and right):

$\mathbb{L}_a$ and $\mathbb{R}_a$ are permutation strings of length s ,

$\mathbb{M}_a$ and $\mathbb{M}'_a$ are permutation strings of length $h - s$,

none of the $\mathbb{L}_a, \mathbb{M}_a, \mathbb{M}'_a, \mathbb{R}_a$ have a symbol in common.

The strings $\mathbb{L}_a\mathbb{M}_a$ and $\mathbb{M}'_a\mathbb{R}_a$ will represent the clause C_a being chosen by the variable, and strings $\mathbb{M}_a\mathbb{R}_a$ and $\mathbb{L}_a\mathbb{M}'_a$ will represent C_a not being chosen by a variable.

Let x_i be a variable. Let C_a, C_b, C_c be the clauses that contain the variable x_i . The x_i variable has two corresponding strings X_i and X'_i built as follows using strings from above and other substrings $P_i^j, P_i^*, Q_i, S_i, T_i^*$ and T_i^j where $j \in [r]$:

- for $j \in [r]$, P_i^j is a permutation string of length ℓ ,
- P_i^*, Q_i, S_i, T_i^* are permutation strings of length $\ell - s$,
- for $j \in [r]$, T_i^j is a permutation string of length ℓ .

These strings do not share symbols. Then let us define X_i and X'_i as

$$X_i = P_i^1 P_i^2 \dots P_i^r P_i^* \mathbb{L}_a \mathbb{M}_a \mathbb{R}_a Q_i \mathbb{L}_b \mathbb{M}_b \mathbb{R}_b S_i \mathbb{L}_c \mathbb{M}_c \mathbb{R}_c T_i^* T_i^1 T_i^2 \dots T_i^r,$$

$$X'_i = P_i^1 P_i^2 \dots P_i^r P_i^* \mathbb{L}_a \mathbb{M}'_a \mathbb{R}_a Q_i \mathbb{L}_b \mathbb{M}'_b \mathbb{R}_b S_i \mathbb{L}_c \mathbb{M}'_c \mathbb{R}_c T_i^* T_i^1 T_i^2 \dots T_i^r.$$

Note that each X_i and X'_i is a permutation string, but X_i and X'_i have symbols in common. The only difference is $\mathbb{M}_a, \mathbb{M}'_a, \mathbb{M}_b, \mathbb{M}'_b, \mathbb{M}_c, \mathbb{M}'_c$. Then we put $\mathcal{W}_1 = \{X_1, X'_1, X_2, X'_2, \dots, X_n, X'_n\}$. For $\mathcal{W}_2$, each of its strings has length ℓ or h . For each variable x_i , that is present in clauses C_a, C_b, C_c , we add the following strings to $\mathcal{W}_2$:

- $P_i^1, P_i^2, \dots, P_i^r$ (r strings of length ℓ),
- $P_i^1 P_i^2 \dots P_i^r P_i^*$ (one string of length $r\ell + \ell - s = h$),
- $P_i^* \mathbb{L}_a$ (one string of length $\ell - s + s = \ell$),
- $\mathbb{R}_a Q_i, Q_i \mathbb{L}_b, \mathbb{R}_b S_i, S_i \mathbb{L}_c$ (four strings of length $s + \ell - s = \ell$),
- $\mathbb{R}_c T_i^*$ (one string of length $\ell - s + s = \ell$),
- $T_i^* T_i^1 T_i^2 \dots T_i^r$ (one string of length $r\ell + \ell - s = h$),
- $T_i^1, T_i^2, \dots, T_i^r$ (r strings of length ℓ)).

Then for each clause C_a , we add the following strings to $\mathcal{W}_2$:

- $\mathbb{L}_a \mathbb{M}_a, \mathbb{M}'_a \mathbb{R}_a$ (two strings of length h),
- $\mathbb{M}_a \mathbb{R}_a, \mathbb{L}_a \mathbb{M}'_a, \mathbb{M}_a \mathbb{R}_a, \mathbb{L}_a \mathbb{M}'_a$ (Four strings of length h).

Note that $\mathbb{M}_a \mathbb{R}_a, \mathbb{L}_a \mathbb{M}'_a$ is included twice in the $\mathcal{W}_2$ while $\mathbb{L}_a \mathbb{M}_a, \mathbb{M}'_a \mathbb{R}_a$ included once. Since l and h are constants, we can build this instance in polynomial time. Note that for each $i \in [n]$, each variable substring $P_i^j, P_i^*, Q_i, S_i, T_i^j, T_i^*$ occurs only twice in $\mathcal{W}_1$ and twice in $\mathcal{W}_2$. The clause substrings $\mathbb{L}_a, \mathbb{M}_a, \mathbb{M}'_a, \mathbb{R}_a, \mathbb{L}_a, \mathbb{R}_a$ occur six times in $\mathcal{W}_1$, twice for each

variable that C_a contains. Each $\mathbb{M}_a, \mathbb{M}'_a$ occurs three times. The same holds for $\mathcal{W}_2$. It follows that the maximum number of occurrences of a symbol is six.

Now, we show that ϕ is 1-in-3 satisfiable if and only if there exists a common F -partition of $\mathcal{W}_1$ and $\mathcal{W}_2$.

Suppose that ϕ is 1-in-3 satisfiable using some assignment, then for each x_i that is true under this assignment, we partition the strings of $\mathcal{W}_1$ as follows: We partition X_i and X'_i as referred to in figure 2.1. If x_i is false, then we partition X_i and X'_i as in figure 2.2. We can see that each partition block is either ℓ or h .

For $\mathcal{W}_2$, we do not partition the strings. Since all the strings in the $\mathcal{W}_2$ are of size either ℓ or h , each string acts as a partition block. Now, we need to show that blocks that were created in $\mathcal{W}_1$ and $\mathcal{W}_2$ have a bijection. Let us consider the clause C_a which contains the variables x_i, x_j, x_k . Since we have a 1-in-3 satisfiable, we can assume that only one of the x_i, x_j, x_k is true. Suppose $x_i = \text{true}$ and $x_j = x_k = \text{false}$, then X_i is partitioned as in figure 2.1 and X_j, X_k is partitioned as in figure 2.2. Note that in $\mathcal{W}_1$, the substrings $\mathbb{L}_a\mathbb{M}_a, \mathbb{M}_a\mathbb{R}_a, \mathbb{M}_a\mathbb{R}_a, \mathbb{M}'_a\mathbb{R}_a$ only occur in $X_i, X'_i, X_j, X'_j, X_k, X'_k$. The way we partitioned the $\mathcal{W}_1$, we only have one block of $\mathbb{L}_a\mathbb{M}_a$ which is in X_i as we assumed $x_i = \text{true}$. Since $\mathbb{L}_a\mathbb{M}_a$ appears only once in $\mathcal{W}_2$, there should be only one block of $\mathbb{L}_a\mathbb{M}_a$ in $\mathcal{W}_1$. Then, the substrings $\mathbb{M}_a\mathbb{R}_a$ appear twice in the $\mathcal{W}_1$ from the way that we partitioned $\mathcal{W}_1$ which is in X_j and X_k . This can be mapped to two instances of $\mathbb{M}_a\mathbb{R}_a$ in $\mathcal{W}_2$. Similarly, $\mathbb{M}'_a\mathbb{R}_a$ is used once in X'_j and $\mathbb{L}_a\mathbb{M}'_a$ is chosen twice in X'_j and X'_k . This can also be mapped to its corresponding instances in $\mathcal{W}_2$. Now, we have to consider the rest of the strings in $\mathcal{W}_2$ that were added for each variable x_i , if $x_i = \text{true}$, then $P_i^1 P_i^2 \dots P_i^r P_i^*, \mathbb{R}_a Q_i, \mathbb{R}_b S_i, \mathbb{R}_c T_i^*$ and T_i^j is covered by X_i . The strings P_i^j for $j \in [r]$, $P_i^* \mathbb{L}_a, Q_i \mathbb{L}_b, S_i \mathbb{L}_c$ and $T_i^* T_i^1 \dots T_i^r$ is covered by X'_i . These strings are distinct for each x_i and we can put one-to-one correspondence in $\mathcal{W}_1$ and $\mathcal{W}_2$. If $x_i = \text{false}$, then the roles are reversed and the same strings are covered by X'_i and X_i . Therefore, a common partition exists.

Now, we must demonstrate that if there is a common F -partition, then a 1-in-3-satisfiable truth assignment for ϕ exists. Initially, we establish that none of the sequences in $\mathcal{W}_2$ can be split into smaller segments. This is because ℓ is the minimum value in F , implying that none of the ℓ -length sequences can be further partitioned. Let's examine a sequence of $\mathcal{W}_2$ that has a length of h . As per our assumption, h is the second-smallest value in F . Therefore, it cannot be partitioned any further because the only way to do so would be to divide the h -length string into substrings of length ℓ . This is unattainable because ℓ cannot divide h .

Let's analyze a sequence X_i from $\mathcal{W}_1$, where C_a, C_b, C_c are the clauses in which x_i occurs. We assert that X_i can only be partitioned in two ways, illustrated in figures 2.1 and 2.2. To demonstrate this, let us assume that W_m where $m \in [k]$ are the partition blocks of X_i . Since the strings in $\mathcal{W}_2$ have lengths ℓ and h , the only viable way to divide the first block (W_1) of X_i is by segmenting it into blocks of length ℓ or h . Consequently, W_1 must be either of size ℓ or h , which is $P_i^1 P_i^2 \dots P_i^r P_i^*$ or P_i^1 .

Assuming that $W_1 = P_i^1 P_i^2 \dots P_i^r P_i^*$, then the partition shown in figure 2.1 is obligatory for X_i . This is because in X_i , W_1 is succeeded by $\mathbb{L}_a$, and the only strings in $\mathcal{W}_2$ that start with $\mathbb{L}_a$ are $\mathbb{L}_a \mathbb{M}_a$ and $\mathbb{L}_a \mathbb{M}'_a$. Thus, we are obliged to include $\mathbb{L}_a \mathbb{M}_a$ from X_i . Then, the only substring in $\mathcal{W}_2$ that starts with a prefix of $\mathbb{R}_a$ and is in $\mathcal{W}_2$ is $\mathbb{R}_a Q_i$. Following the same logic, we are compelled to choose $\mathbb{L}_b \mathbb{M}_b$, then $\mathbb{R}_b S_i, \mathbb{L}_c \mathbb{M}_c, \mathbb{R}_c T_i^*$, and lastly, T_i^j 's separately. Hence, X_i is divided as shown in figure 2.1. If we examine the X'_i partitions, they must comprise $P_i^1, P_i^2, \dots, P_i^r, P_i^* \mathbb{L}_a, Q_i \mathbb{L}_b, S_i \mathbb{L}_c$, and $T_i^* T_i^1 \dots T_i^r$. These substrings contain distinct symbols from X_i and X'_i . As these were not included in the X_i partition, they must be included in the X'_i partition. Consequently, the X'_i partition follows the figure 2.1.

The second scenario is when $W_1 = P_i^1$. In this instance, the initial string in the partition of X'_i must be $P_i^1 \dots P_i^r P_i^*$. By symmetrically exchanging the roles of X_i and X'_i and utilizing the same reasoning as before, we can conclude that both X_i and X'_i are

partitioned, as depicted in figure 2.2.

Then we assign $x_i = \text{true}$ for X_i 's partitioned as in figure 2.1 and $x_i = \text{false}$ if the partition is as in figure 2.2. Since $\mathbb{L}_a\mathbb{M}_a$ and $\mathbb{M}'_a\mathbb{R}_a$ strings are in $\mathcal{W}_2$ exactly once, it follows that exactly one of X_i , X_j and X_k is partitioned as in figure 2.1. This implies that only one of the x_i, x_j, x_k is true. Therefore, ϕ is 1-in-3 satisfiable. $\square$

Next, we will also show that even if we restrict the problem to have only two input strings, the problem is NP-hard.

Theorem 2. *Let F be any set of integers in which at least one element is not a multiple of $\min F$. The $XSR - F$ problem is NP-hard, even when restricted to instances in which $\mathcal{W}_1, \mathcal{W}_2$ both contain only one string, and even if each symbol occurs at most 6 times.*

Proof. We begin by reducing the $XSR - F$ problem, which is known to be NP-hard by Theorem 1. In the $XSR - F$ problem, input sets can have more than one string, but occurrences are limited to at most six. Let $\mathcal{W}_1$ and $\mathcal{W}_2$ be an instance of this problem, with $\mathcal{W}_1 = \{A_1, \dots, A_n\}$ and $\mathcal{W}_2 = \{B_1, \dots, B_m\}$, and assume without loss of generality that $n \geq m$. Let ℓ be the minimum length of F . We define a new set Δ consisting of n new permutation strings, each of length ℓ and containing symbols that occur in no other string. These new strings are denoted as $X_1, Y_1, Z_1, X_2, Y_2, Z_2, \dots, X_n, Y_n, Z_n$.

We then construct a new instance of the $XSR - F$ problem, denoted by $\mathcal{W}'_1$ and $\mathcal{W}'_2$, by setting $\mathcal{W}'_1 = \mathcal{W}_1 \cup \Delta$ and $\mathcal{W}'_2 = \mathcal{W}_2 \cup \Delta$. Note that $\mathcal{W}_1, \mathcal{W}_2$ is a Yes-instance of $XSR - F$ if and only if $\mathcal{W}'_1, \mathcal{W}'_2$ is a Yes-instance of $XSR - F$, since the new strings in Δ must be matched. Therefore, we can use the instance $\mathcal{W}_1, \mathcal{W}_2$ for our reduction. Now we build the single strings S and T as follows:

$$S = A_1 X_1 Y_1 Z_1 A_2 X_2 Y_2 Z_2 \dots A_n X_n Y_n Z_n,$$

$$T = B_1 Z_1 Y_1 X_1 B_2 Z_2 Y_2 X_2 \dots B_m Z_m Y_m X_m Z_{m+1} Y_{m+1} X_{m+1} \dots Z_n Y_n X_n.$$

□

Note that the order of the Δ triples is reversed from S to T , meaning that $X_i Y_i Z_i$ appears in S , while $Z_i Y_i X_i$ appears in T . We will now demonstrate that $\mathcal{W}'_1$ and $\mathcal{W}'_2$ have a common F -partition if and only if S and T have a common F -partition.

Suppose that $\mathcal{W}'_1$ and $\mathcal{W}'_2$ have a common F -partition. It's important to note that S is a concatenation of the elements of $\mathcal{W}'_1$, while T is a concatenation of the elements of $\mathcal{W}'_2$. Therefore, this partition is also a common F -partition of S and T .

Conversely, assume that S and T have a common F -partition. Let's consider a substring X_i . Observe that X_i is preceded and succeeded by a different symbol in S and T . Specifically, X_i is preceded by some symbol of A_i in S and preceded by Y_i in T , while it is succeeded by Y_i in S and succeeded by some B_i symbol or Z_{i+1} in T . As a result, since X_i has a fixed length, any common F -partition must include X_i as a block. By the same logic, every substring Y_i and Z_i must also be a block. Consequently, the remaining blocks must form a partition of $A_1, \dots, A_n$ and $B_1, \dots, B_m$. Therefore, there is a common F -partition of $\mathcal{W}'_1$ and $\mathcal{W}'_2$.

2.5 GSR-F in Polynomial Time for Fixed Block Sizes and Alphabet

We will now focus on the optimization version $GSR - F$, which aims to minimize the number of characters that need to be deleted from input strings in order to allow for a common F -partition. It is worth noting that this problem is NP-hard, as shown in the previous section. However, the hardness reduction applies only to strings with alphabets of arbitrary size. We demonstrate that if F is a fixed set of integers and Σ is a fixed alphabet, then $GSR - F$ can be solved in polynomial time. Specifically, we prove that $GSR - F$ can be solved in $O(n^{|F||\Sigma|^k+3})$ time, where $k = \max(F)$.

This solution can also be applied to solve the $XSR - F$ problem under the same conditions. Furthermore, since Proposition 1 states that most infinite sets of interest reduce

to the finite case (including $F = 2, 3, \dots$, which can be replaced with $F = 2, 3$), requiring F to be finite is not a severe limitation for $XSR - F$.

Let B denote the set of strings in the alphabet Σ whose length is in F . It is noteworthy that $|B| = \sum_{c \in F} |\Sigma|^c \leq \sum_{c=1}^k |\Sigma|^c \leq |F| |\Sigma|^k$, which is a constant. Moreover, every block in any F -partition must be a string in B .

To obtain a polynomial-time algorithm, we need to compute the set of possible ways to split the input into blocks that are in B and check if there exists a common split between the input sets of strings. This can be accomplished using dynamic programming, which we explain in detail.

In order to achieve a polynomial-time algorithm, it is sufficient to calculate the possible ways of splitting the input into blocks that belong to B and then check if a common split exists between the input sets of strings. This can be accomplished using dynamic programming, which we will describe in detail.

We will use a block count table, denoted by $C : B \rightarrow \mathbb{N}$, that assigns to each $X \in B$ a value $C(X)$ representing the desired number of occurrences of X in a common F -partition. For a given set of strings $\mathcal{W}$ and a block count table C , we say that $\mathcal{W}$ is C -splittable if there exists an F -partition of $\mathcal{W}$ in which each string $X \in B$ occurs exactly $C(X)$ times.

In our dynamic programming approach, we arbitrarily order the strings in $\mathcal{W}_1$ and $\mathcal{W}_2$, and then process each string one marker at a time, from left to right. We will rely on a table that is described as follows:

Definition 5. *Let $\mathcal{W}$ be a set of strings and let C be a block count table. Then we denote by $D(\mathcal{W}, C)$ the minimum number of markers to delete from the strings in $\mathcal{W}$ so that the resulting strings are C -splittable (with $D(\mathcal{W}, C) = \infty$ if this is not possible).*

Our objective is to minimize $D(\mathcal{W}_1, C) + D(\mathcal{W}_2, C)$, where C is a block count table. Note that this value is always less than infinity, since we can take C with all zeroes in the

worst case, i.e., delete everything from $\mathcal{W}_1$ and $\mathcal{W}_2$. However, the number of C 's to consider is possibly infinite. We will consider the size of C later.

Let us introduce some definitions. For two strings X and Y , we write $X \prec Y$ to mean that X is a subsequence of Y . Furthermore, for a block count table C and $X \in B$, let $C - X$ denote the block count table in which $C(X)$ is reduced by one. Precisely, $C - X = C'$ where $C'(X) = C(X) - 1$ and $C'(Y) = C(Y)$ for all $Y \in B \setminus X$. Note that $C'(X) = -1$ is possible under this definition. Despite this, we still refer to C' as a block count table and allow negative values. It is important to observe that in this case, no string is C' -splittable.

We can now define a recurrence, as stated in Lemma 2, for $D(\mathcal{W}, C)$ where $\mathcal{W}$ is a general set of strings. The main idea is that to find a partition of $\mathcal{W}$ with counts C , we can select some $W_h \in \mathcal{W}$ and concentrate on the last block of W_h in this partition. For each suffix $W_h[j + 1..|W_h|]$ and each possible $X \in B$ that is a subsequence of this suffix, we compute $|W_h| - j - |X|$ deletions from that suffix to obtain X . After removing this suffix, we get a set of strings with a shorter total length. We can use a recurrence on this shorter set of strings, but with the target count table $C - X$.

Lemma 2. *Let $\mathcal{W} = \{W_1, \dots, W_h\}$ be a set of strings and let C be a block count table. Then the following holds:*

1. *if $C(X) = 0$ for all $X \in B$, then $D(\mathcal{W}, C) = \sum_{i=1}^h |W_i|$;*
2. *otherwise, if $C(X) < 0$ for some X , then $D(\mathcal{W}, C) = \infty$;*
3. *otherwise, if $\mathcal{W} = \emptyset$, then $D(\mathcal{W}, C) = \infty$;*
4. *otherwise, if $|W_h| < \min(F)$, then $D(\mathcal{W}, C) = D(\mathcal{W} \setminus \{W_h\}, C) + |W_h|$ (note that this includes the case where W_h is the empty string);*

5. otherwise, for $j \in \{0, 1, \dots, |W_h| - \min(F)\}$, define $\mathcal{W}^{(j)} = (\mathcal{W} \setminus \{W_h\}) \cup \{W_h[1..j]\}$.

Then $D(\mathcal{W}, C)$ is the minimum between $D(\mathcal{W} \setminus \{W_h\}, C) + |W_h|$ and the expression

$$\min_{0 \leq j \leq |W_h| - \min(F)} \left(\min_{X \in B: X \prec W_h[j+1..|W_h|]} (D(\mathcal{W}^{(j)}, C - X) + |W_h| - j - |X|) \right).$$

Proof. We will start by considering the first four cases. If all elements in C are zero, we must remove all markers from every string in $\mathcal{W}$ to obtain a count table with all zeroes, which leads to Case 1. If some element in C is negative, it is impossible to achieve that count table, and this leads to Case 2. From here on, we assume that all elements in C are non-negative, and at least one element is positive. If $\mathcal{W}$ is empty, it is not possible to achieve count table C with an empty set of strings, especially for those X with $C(X) > 0$, which leads to Case 3. If $|W_h| < \min(F)$, then W_h cannot contribute to any block, and we need to remove all its markers, which leads to Case 4.

Next, we consider Case 5. Let W_h and $\mathcal{W}^{(j)}$ be defined as in the statement. We will refer to the longer expression in Case 5 as the "long expression." Firstly, note that the long expression is well-defined, meaning that the two minimizations are not over empty sets. This is because we assume that $|W_h| \geq \min(F)$, and hence there exists an $X \in B$ that is a subsequence of $W_h[j+1..|W_h|]$ for any $j \leq |W_h| - \min(F)$.

Now, let $j^* \in \{0, 1, \dots, |W_h| - \min(F)\}$ and $X^* \in B$, with $X^* \prec W_h[j^*+1..|W_h|]$, such that j^* and X^* minimize the long expression. Let us define,

$$d = \min \begin{cases} D(\mathcal{W} \setminus \{W_h\}, C) + |W_h| \\ D(\mathcal{W}^{(j^*)}, C - X^*) + |W_h| - j^* - |X^*| \end{cases}$$

which is the value chosen in Case 5. Let M be a minimum-size set of markers to delete from $\mathcal{W}$ so that after deleting these markers, we obtain a C -splittable set of string. By

definition, we have $|M| = D(\mathcal{W}, C)$, and we need to prove that $|M|$ is both an upper bound and a lower bound on d .

Let's begin by proving that $|M| \leq d$. When $d = \infty$, this inequality is trivial. For finite d , we can obtain a C -splittable set of strings by deleting a maximum of d markers from $\mathcal{W}$. To see this, consider the following cases; If $d = D(\mathcal{W} \setminus W_h, C) + |W_h|$, we remove all markers from W_h and $D(\mathcal{W} \setminus W_h, C)$ markers from the remaining strings to achieve a C -splittable set of strings. Alternatively, if $d = D(W^{(j^*)}, C - X^*) + |W_h| - j^* - |X^*|$, we remove $|W_h| - j^* - |X^*|$ markers from the suffix of W_h to create X^* , a block in B , and then remove $D(W^{(j^*)}, C - X^*)$ markers from $W^{(j^*)}$ to create a $(C - X^*)$ -splittable string. Either way, we obtain a C -splittable set of strings by deleting d markers. Since M is optimal, it follows that $|M| \leq d$.

We now aim to prove that $|M| \geq d$. Let $\mathcal{W}'$ be the set of strings obtained by deleting each marker in M from $\mathcal{W}$. Suppose P' is an F -partition of $\mathcal{W}'$ that attains count table C . One possibility is that all markers of W_h are deleted by M . In this case, P is obtained solely with $\mathcal{W} \setminus W_h$ and we have $|M| = |W_h| + D(\mathcal{W} \setminus W_h, C) \geq d$.

We make the assumption that W_h contributes to at least one block. In this case, there exists a suffix $W_h[j' + 1..|W_h|]$ of length at least $\min(F)$ such that some markers of M are removed from it, resulting in a block X' that is included in the F -partition P' . It should be noted that j' falls between 0 and $|W_h| - \min(F)$, and $|W_h| - j' - |X'|$ markers must be removed from this suffix to form X' . Furthermore, the count table $C - X'$ is obtained by removing markers from $(\mathcal{W} \setminus \{W_h\}) \cup \{W_h[1..j']\} = W^{(j')}$. As a result, $D(\mathcal{W}, C) = |M| = D(W^{(j')}, C - X') + |W_h| - j' - |X'|$. The value of d is no greater than this value, since d is determined by minimizing over all possible values of j and X , including j' and X' . Hence, $|M| \geq d$, which concludes the proof. $\square$

We now need to outline the computational aspects of our approach. To compute the recurrence, we must perform computations on both $\mathcal{W}_1$ and $\mathcal{W}_2$. Let n be the maximum of

the sums of the lengths of all strings in $\mathcal{W}_1$ and $\mathcal{W}_2$ ($n = \max(\sum_{W \in \mathcal{W}_1} |W|, \sum_{W \in \mathcal{W}_2} |W|)$). If there exists a common F -partition P of $\mathcal{W}_1$ and $\mathcal{W}_2$, then no string in B occurs more than n times in P . We define $\mathbb{C}(n)$ as the set of all possible count tables with values that are at most n . In other words, $C \in \mathbb{C}(n)$ if and only if $C(X) \leq n$ for every $X \in B$. We can determine that $|\mathbb{C}(n)| = (n+1)^{|B|} \in \Theta(n^{|B|})$, which is a polynomial function of n , since $|B|$ is a constant.

To solve the strip recovery problem for $\mathcal{W}_1$ and $\mathcal{W}_2$, we need to find a count table C such that both $\mathcal{W}_1$ and $\mathcal{W}_2$ are C -splittable after a minimum number of marker deletions. Thus, we need to compute:

$$\min_{C \in \mathbb{C}(n)} (D(\mathcal{W}_1, C) + D(\mathcal{W}_2, C)).$$

Also, note that this idea generalizes trivially to the case where more than two sets of strings are given. The main algorithm is given below.

Theorem 3. *Algorithm 2.1 solves the GSR – F problem in time $O(n^{|F||\Sigma|^{\max(F)}})$.*

Proof. Assuming that `computeD` produces the correct values of D , it is easy to see the correctness of the algorithm. The algorithm computes $D(\mathcal{W}, C)$ in ascending order of the number of strings and the prefix of the last string. For a particular iteration with $\mathcal{W} = \{W_1, \dots, W_{h-1}, W_h[1..i]\}$ and $h \in [l]$, the computation of $D(\mathcal{W}, C)$ requires access to either $D(\mathcal{W} \setminus W_h, C)$, which has one fewer string, or some $D(\mathcal{W}^{(j)}, C - X)$, where $\mathcal{W}^{(j)}$ has the same number of strings as $\mathcal{W}$ but whose last string is shorter. As per the recurrence of Lemma 2, the order of computation allows us to assume that all values required for a specific $D(\mathcal{W}, C)$ have been calculated in previous iterations. Therefore, the algorithm is correct.

To determine the time complexity, we can focus on the time required to compute one $D(\mathcal{W}, C)$ entry. We can see that **`computeD`** function iterates over $n \cdot |\mathbb{C}(n)|$ combinations of $\mathcal{W}$ and C . The worst-case scenario is Case 5 of the recurrence, which requires the

```

1: Function getMinStripRecovery( $\mathcal{W}_1, \mathcal{W}_2$ )
2:   computeD( $\mathcal{W}_1$ )
3:   computeD( $\mathcal{W}_2$ )
4:   return  $\min_{C \in \mathbb{C}(n)} (D(\mathcal{W}_1, C) + D(\mathcal{W}_2, C))$ 
5: End Function

6: Function computeD( $\mathcal{W}_a = \{W_1, \dots, W_l\}$ )
7:   Let  $D$  be a map, initially empty
8:   For  $h = 1..l$ 
9:     For  $i = 1..|W_h|$ 
10:      Let  $\mathcal{W} = \{W_1, W_2, \dots, W_{h-1}, W_h[1..i]\}$ 
11:      For  $C \in \mathbb{C}(n)$ 
12:        Compute  $D(\mathcal{W}, C)$  using Lemma 2
13:      end
14:    end
15:  end
16: End Function

```

Algorithm 2.1:

Pseudocode for the computation of the dynamic programming.

enumeration of $O(n)$ values of j and $|B|$ values of X . For each j and X combination, it takes $O(n)$ time to check whether $X \prec W_h[j + 1..|W_h|]$. Thus, one $D(\mathcal{W}, C)$ computation takes $O(n^2 \cdot |B|) = O(n^2)$ time. As a result, the total time spent in $\text{compute}D(\mathcal{W}_1)$ and $\text{compute}D(\mathcal{W}_2)$ is $O(n^3 \cdot \mathbb{C}(n)) = O(n^{|B|+3})$. The search for the best C after these computations does not add any complexity, resulting in a running time of $O(n^{|B|+3}) = O(n^{|F||\Sigma|^{\max(F)+3}})$. $\square$

2.6 Fixed Alphabet with Unbounded Block Sizes Is NP-hard

Previously, we made the assumption that F was a fixed set of integers. However, we will now demonstrate that if F is a part of the input, then $XSR - F$ becomes NP-hard, even if the alphabet is constant. This indicates that for polynomial-time feasibility, F and Σ must remain fixed, unless $P = NP$.

To prove this, we utilize a reduction from the strongly NP-complete problem 3-Partition [43]. In 3-Partition, we are provided with an integer D and a set of integers $S = a_i | i = 1..3m$, where $D/4 < a_i < D/2$ for all i . The question is whether we can partition S into m subsets, each comprising three integers, such that the sum of each subset is exactly D . Due to 3-Partition being strongly NP-complete, each $a_i \in S$ is bounded by a polynomial in m .

For our $XSR - F$ instance, we construct the input strings A and B , as well as the input set F , in the following manner: The alphabets of A and B are both $\Sigma = \{0, 1, \#, \$\}$, and we define $F = \{a_1, a_2, \dots, a_{3m}, D + 1\}$. The strings A and B are defined as follows:

The fundamental idea behind this construction is that because F only contains the a_i 's and $D + 1$, any common F -partition must contain the 1..1 blocks of A . To address the 1..1 blocks of B , there must be a way to partition them into groups of three that achieve length D . We will now provide further details.

Theorem 4. *The XSR problem is NP-hard, even if the input consists of two strings on an*

$$\begin{aligned}
A &= \overbrace{\$ \dots \$}^{D+1} \overbrace{1 \dots 1}^{a_1} \overbrace{\$ \dots \$}^{D+1} \overbrace{1 \dots 1}^{a_2} \dots \overbrace{\$ \dots \$}^{D+1} \overbrace{1 \dots 1}^{a_{3m}} \overbrace{\# \dots \#}^{D+1} \overbrace{0 \dots 0}^{m(D+1)}, \\
B &= \overbrace{\# \dots \#}^{D+1} \overbrace{\$ \dots \$}^{3m(D+1)} \overbrace{(0 \dots 0}^{D+1} \overbrace{1 \dots 1)}^D)^m.
\end{aligned}$$

Figure 2.3: Strings A and B .

alphabet of size four.

Proof. First, we note that the construction above can be carried out in polynomial time since the values of the a_i 's (and therefore of D) are polynomial. We prove that a 3-partition instance (S, D) admits a 3-Partition if and only if the constructed A and B strings have a common F -partition.

Assuming (S, D) admits a 3-Partition $S_1, \dots, S_m$, we construct a common F -partition M of A and B as follows: In M , we add $3m$ copies of $\dots$ of length $D+1$, one copy of $\# \dots \#$ of length $D+1$, m copies of $0 \dots 0$ of length $D+1$, and for each $a_i \in S$, we add a $1 \dots 1$ substring of length a_i . Each string of M has its length in F , and it remains to show that they can be concatenated to obtain A or B . For A , this is easy to see as M is partitioned as shown in the figure above (plus the m copies of $0 \dots 0$ that result in the last 0 block of A). For B , the $\#$ -block and $\$$ -blocks are easily covered. We must form m copies of $0 \dots 01 \dots 1$ blocks. This can be done by taking, for each $S_t = a_i, a_j, a_k$ in the 3-Partition, one $0 \dots 0$ block of length $D+1$ followed by the $1 \dots 1$ blocks of length a_i, a_j , and a_k . Since these sum to D , this can be used to create the desired m blocks of the form $0 \dots 01 \dots 1$.

On the other hand, let us assume that A and B have a common F -partition M . It is important to note that the unique $\#$ -block in A is adjacent to the unique 0-block and the last 1-block, while in B , the $\#$ -block is not adjacent to a 0 or a 1. As a result, any substring in M that contains a $\#$ can only have $\#$ symbols. Additionally, the $\$$ -blocks in A are adjacent to the 1-blocks in A , but no $\$$ is adjacent to a 1 in B , so any substring in M

with a $\$$ can only contain $\$$ symbols. Furthermore, the 0-block in A is only adjacent to the $\#$ -block and not in B . Therefore, each substring in M with a 0 can only contain 0s, and any substring in M that contains a 1 can only have 1s.

Consider the $1 \dots 1$ block of length a_i in A . Since $D/4 < a_i < D/2$, this block must be in M . If this block were not in M , it would mean that the block was obtained by concatenating two or more blocks of 1s from M with a smaller length in F . This is not possible due to the bounds on a_i . Thus, M must contain a 1-block of length a_i for each $a_i \in S$. It is then easy to see that each of the 1-blocks in B must be partitioned into three blocks, with lengths a_i, a_j , and a_k , respectively, such that $a_i + a_j + a_k = D$. This partition corresponds to a solution for the 3-partition instance. $\square$

2.7 Conclusion

In this chapter, we explored several variations of F -Strip recovery problem. This problem can be viewed as a generalization of the maximum strip recovery (MSR) problem. In retrospect, we looked at the general F -Strip recovery problem (GSR-F) and the exact F -Strip recovery problem (XSR-F). We showed that the GSR-F problem can be solved in polynomial time for fixed F and alphabets. Furthermore, we explored the complexity of XSR-F under fixed F , fixed alphabet, and unbounded F . We showed that if all the elements in F are multiples of its minimum, then XSR-F can be solved in polynomial time. Consequently, we proved that if there exists some element in F such that it is not a multiple of $\min F$, then XSR-F is NP-hard. We used a clever reduction from the positive cubic XSAT problem to the XSR-F problem to prove the NP-hardness. Then we proved that even when we are restricted to instances in which $\mathcal{W}_1, \mathcal{W}_2$ both contain only one string each and each symbol occurs at most 6 times, the problem is still NP-hard. This demonstrates that the properties of set F directly affect the complexity of the XSR-F problem. Notably, XSR-F is NP-hard even if the set F only contains elements 2 and 3 ($F = \{2, 3\}$). Furthermore, we showed that

XSR-F problem under a fixed alphabet with unbounded F is also NP-hard using a reduction from the 3-partition problem. We believe this chapter helped to fill some gaps in existing literature about the complexity of variations of the MSR problem.

2.7.1 Notes

Part of this chapter has been published in Comparative Genomics: 19th International Conference (RECOMB-CG 2022) [68].

CHAPTER THREE

THE LONGEST LETTER-DUPPLICATED SUBSEQUENCE PROBLEM

3.1 Introduction

In biology, duplication is an important part of evolution. There are two major types of duplication: arbitrary segment duplications and tandem duplications. Arbitrary segment duplication means when you select a segment in a genome and paste it somewhere else in the genome, and tandem duplication (TD) is defined as a structure rearrangement whereby a segment of DNA is duplicated and inserted serially to the original segment [91] ($X \rightarrow XX$). It is known that segment duplications are frequent in cancer genomes [28, 74, 100]. Tandem duplications are common in different scenarios. It is known that tandem duplications of 3 nucleotide CAG is closely related to Huntington disease [82]. Whole-genome sequencing (WGS) data from tumors has revealed that massive rearrangements, as in the tandem duplicator phenotype, are a specific cancer phenotype [54]. Tandem duplications occur in some cancers [103], particularly in ovarian and breast cancer genomes. A subset of ovarian cancer samples shares a marked tandem duplication phenotype with triple-negative breast cancers [84]. In 1980, Szostak and Wu showed evidence that gene duplication is the main driving force behind evolution and that the majority of duplications are tandem [104]. In the first sequenced human genome, around 3% are in the form of tandem repeats [73].

Furthermore, tandem duplications were further studied in copying systems [37] and formal languages [13, 30, 110]. In 2020, Lafond et al. answered the open question of the complexity of computing the minimum tandem duplication distance between two sequences A and B (the minimum number of tandem duplications to convert A to B). The problem was shown to be NP-complete for the unbounded alphabet [71]. Later, Lafond et al. proved that the problem is NP-hard even when the $|\Sigma| \geq 4$, by encoding each letter in the unbounded

alphabet proof with square-free string over a new alphabet of size 4 (using the method from Leech's construction in [78]). This is important since it covers the most relevant case in biology, where $\Sigma = \{A, C, G, T\}$ for DNA sequences and $\Sigma = \{A, C, G, U\}$ for RNA sequences [71].

Motivated by the above applications (especially when some mutations occur after the duplications), new problem formulations related to duplications are proposed in this chapter. Given a sequence S of length n , a letter duplicated subsequence (*LDS*) of S is a subsequence of S in the form of $x_1^{d_1} x_2^{d_2} \cdots x_k^{d_k}$ with $x_i \in \Sigma$, where $x_j \neq x_{j+1}$ and $d_i \geq 2$ for all i in $[k]$ and j in $[k-1]$. (Each $x_i^{d_i}$ is called a *LDS block*.) Then we can define the problem of finding the longest letter-duplicated subsequence (*LLDS*) of S . We can obtain a simple linear-time algorithm for this. Let's consider the following example: Here, the original sequence is H and after some local mutations, insertions, and deletions, it becomes S . Now, when we calculate the longest-letter duplicated subsequence, we either get S_1 or S_2 which gives us a skeleton of the initial sequence. As an example,

$$H = AACACAGATGAT,$$

$$S = AACACGTCGAT,$$

$$S_1 = AACCGG,$$

$$S_2 = AACCTT.$$

A similar problem has been proposed recently called *longest run subsequence* problem, which was studied in [98], it differs from the longest-letter duplicated subsequence problem in that it asks for subsequence where each letter appears consecutively at most once in the solution and does not have to be repeated.

In this chapter, we will introduce variations of the LLSD problem, focusing on the

constrained and weighted cases. The constraint is to include all the letters in Σ in the final LDS solution. This helps to extract information about the maximum duplicated pattern that includes all the genes. Now, we can create two problems with this: feasibility testing (FT) which decides whether the given sequence S has a LDS that includes all letters in Σ and the problem of maximizing the LDS of S containing all the letters in the alphabet Σ (LLDS+). It would be interesting to find the complexity of these problems under different parameters, like the maximum number of occurrences d each letter can appear in the input sequence.

3.2 Preliminaries

Given a sequence S of length n , a letter-duplicated subsequence (LDS) of S is a subsequence of S in the form of $x_1^{d_1} x_2^{d_2} \cdots x_k^{d_k}$ with $x_i \in \Sigma$, where $x_j \neq x_{j+1}$ and $d_i \geq 2$ for all i in $[k]$ and j in $[k-1]$. (Each $x_i^{d_i}$ is called an LD-block.)

In this chapter, we focus on important variants around the fundamental LLDS problems, focusing on the constrained and weighted cases while exploring the complexity of these problems under the following constraint. The constraint is to demand that all letters in Σ appear in the resulting LDS. Using this constraint, we can define two main problems for LLDS. The first problem is feasibility testing (FT) which decides whether a given input sequence S has a LDS that contains all the letters in the alphabet Σ . The second problem is to find the maximum length LDS for a given sequence S which we denote LLDS+. We found out that the complexity of this problem relies on the parameter d which is the maximum number of times a letter can appear in S . When this parameter varies, we define each problem as $FT(d)$, $LLDS+(d)$ etc.

3.3 LLDS Problem

A subsequence S' of S is a letter-duplicated subsequence (LDS) of S if it is in the form of $x_1^{d_1}x_2^{d_2}x_3^{d_3}\cdots x_k^{d_k}$ where $x_i \in \Sigma, x_j \neq x_{j+1}$ and $d_i \geq 2$, for $i \in [k], j \in [k-1]$. $x_i^{d_i}$ a *letter-duplicated block* or *LD-Block*. An example is given as follows:

$$S = acgaacggtaccgt,$$

$$S_1 = aaattcc,$$

$$S_2 = ggttcc,$$

$$S_3 = ccc.$$

In the above example, S_1 , S_2 , and S_3 are all letter-duplicated subsequences of S . aaa , tt , and cc are LD-Blocks of S_1 , gg , tt , cc are LD-Blocks of S_2 and ccc is the LD-Block of S_3 . However, we are interested in the longest letter-duplicated subsequence. We solve the LLDS problem using a dynamic programming approach.

First, let's look at the following observation:

Observation 1. *Suppose there is an optimal LLDS solution for a given sequence S of length n , in the form of $x_1^{d_1}x_2^{d_2}\cdots x_k^{d_k}$. Then it is possible to decompose it into a generalized LD-subsequence $y_1^{e_1}y_2^{e_2}\cdots y_p^{e_p}$, where*

- $2 \leq e_i \leq 3$, for $i \in [p]$,
- $p \geq k$,
- y_j does not have to be different from y_{j+1} , for $j \in [p-1]$.

To prove this, we say any natural number $k \geq 2$ can be decomposed as $k = k_1 + k_2 + k_3 + \cdots + k_n$, such that $2 \leq k_i \leq 3$ for $1 \leq i \leq n$. Therefore, every d_i can be decomposed into

a sum of 2's and 3's. Then, using the given optimal solution and its decomposed sums of LD-Blocks, we could obtain the corresponding LD-subsequence by combining $y_i^{e_i} y_{i+1}^{e_{i+1}}$ when $y_i = y_{i+1}$.

Now let us design a dynamic programming algorithm for LLDS. Let $L(i)$ be the length of the optimal LLDS solution for $s[1 \cdots i]$. Then the recurrence relation is defined as follows:

$$\begin{aligned} L(0) &= 0, \\ L(1) &= 0, \\ L(i) &= \max \begin{cases} L(i-x-1) + 2 & x = \min\{x | S[i-x] = S[i]\}, x \in (0, i-1] \\ L(i-x) + 1 & x = \min\{x | S[i-x] = S[i]\}, x \in (0, i-1] \\ L(i-1) & \text{otherwise.} \end{cases} \end{aligned}$$

$L(i-x) + 1$ is the condition to handle the LD-Blocks of length 3 (by keeping the $S[i-x]$ for the next level computation). The reason why this step is important is that we would miss optimal solutions to the following sequences: $S = agfdaa$, where the optimal solution is aaa . The optimal solution is found in $L(n)$. For the running time for each $S[x]$ we need to find the closest $S[i]$ such that $S[x] = S[i]$. We can precompute this information and keep it in a data structure. With this change, the value of $L(n)$ could be found in linear time. Therefore, LLDS can be solved in linear time.

3.3.1 The Variants of LLDS

We will look into two variations of LLDS problem.

Definition 6. Constrained Longest Letter-Duplicated Subsequence (LLDS+)

Input: A sequence S with length n over an alphabet Σ and an integer ℓ .

Question: Does S contain a letter-duplicated subsequence S' with length at least ℓ such that all letters in Σ appear in S' ?

Definition 7. Feasibility Testing (FT)

Input: A sequence S with length n over an alphabet Σ .

Question: Does S contain a letter-duplicated subsequence S' such that all letters in Σ appear in S' ?

Definition 8. Weighted-LDS (W-LDS)

Input: A sequence S with length n over an alphabet Σ and a function $w_x(l)$ which is the weight of LD-block x^l , $x \in \Sigma, 2 \leq l \leq d$, where d is the maximum number of times a letter appears in S .

Question: Compute the maximum weight letter-duplicated string (weighted-LDS).

Note that feasibility testing for LLDS+ (FT) and LLDS+ are two different problems. If there is no feasible solution, then there is no LLDS+ solution, but if there is a feasible solution, then finding the longest letter-duplicated subsequence will still be non-trivial. Let d be the maximum number of times a letter in Σ can appear in S , then we represent that version of the LLDS+ and FT as **LLDS+(d)** and **FT(d)**.

It turns out that the decision versions of LLDS+(d) and FT(d) for LLDS+ are both NP-complete when $d \geq 4$, when $d = 3$, LLDS+(3) could be decided in polynomial-time, and we can come up with a factor-1.5 approximation algorithm for it in $O(n^2)$. We will also consider one more variation of LLDS called weighted-LDS, where the weight of an LD-block is an arbitrary positive function.

3.4 Hardness Result for LLDS+(d) and FT(d) When $d \geq 4$.

3.4.1 FT(4) Is NP-complete

For this reduction, we use a special version of the SAT problem called Monotone-(+, 2)-SAT problem. We will denote this problem as $(3^+, 1, 2^-)$ -SAT problem henceforth. The basic idea is to show $(3^+, 1, 2^-)$ -SAT problem is NP-complete by reducing the 3-coloring problem

and then reducing $(3^+, 1, 2^-)$ -SAT problem to the FT(4) problem. In the $(3^+, 1, 2^-)$ -SAT problem there are sets of 3-CNF clauses (F^+) and set of 2-CNF clauses (F^-). In 3-CNF clauses all three literals appear in their positive states, and each variable appear in exactly one of the 3-CNF clauses. 2-CNF clauses are defined using the same variables used in the 3-CNF clauses, however they appear in their negated format. Therefore, 2-CNF clauses contain negated variables. The $(3^+, 1, 2^-)$ -SAT problem asks whether there exists a truth assignment for $F^+ \wedge F^-$. The reduction for this problem was posted on the internet. First, we will prove the $(3^+, 1, 2^-)$ -SAT problem is NP-complete.

Definition 9. *$(3^+, 1, 2^-)$ -SAT problem: In this special case of SAT problem, we are given a set of 3-CNF clauses (F^+) and a set of 2-CNF clauses (F^-). In 3-CNF clauses, all three literals appear in their positive state, and each variable appears in exactly one of the 3-CNF clauses. 2-CNF clauses are defined using the same variables used in the 3-CNF clauses; however, they appear in their negated format. Therefore, 2-CNF clauses contain only negated variables.*

Theorem 5. *$(3^+, 1, 2^-)$ -SAT problem is NP-complete.*

Proof. The problem is clearly NP. Thus, let us move on to the reduction from 3-COLORING. In the 3-COLORING problem, we are given a graph $G = (V, E)$ and we need to assign one of 3 colors to each vertex $v \in V$ such that any adjacent vertices are given different colors.

For each vertex v , we create three boolean variables v_1, v_2, v_3 to denote 3-colors, which can be assigned to vertex v . Then we create the 3-CNF clause using these three variables: $(v_1 \vee v_2 \vee v_3)$. We do this for all the vertices in the graph G . The conjunction of all the 3-CNF clauses (F^+) is denoted as follows:

$$F^+ = \bigwedge_{v \in V} (v_1 \vee v_2 \vee v_3).$$

Now we move on to the 2-CNF clauses. We have two types of 2-CNF clauses. First, for each $v \in V$ we need to guarantee that only one color is selected for each vertex. Therefore, for type-I 2-CNF clauses, we create the following three 2-CNF clauses for each vertex, $v \in V$.

$$\text{for } 1 \leq i \neq j \leq 3, (\overline{v_i \wedge v_j}) = (\overline{v_i} \vee \overline{v_j}).$$

Note that we have three of these clauses for each vertex because we want to guarantee that only one of the v_1, v_2, v_3 variables is selected.

Then, for each edge $(u, v) \in E$, we create type-II 2-CNF clauses. For $i = 1, 2, 3$.

$$\overline{u_i \wedge v_i} = (\overline{u_i} \vee \overline{v_i}).$$

Note that for each edge $(u, v) \in E$ we create three of the above 2-CNF clauses. Let F^- be the conjunction of these 2-CNF clauses. Then $\phi = F^+ \wedge F^-$.

For the “Only-If” direction, obviously, if 3-COLORING has a solution, then for each vertex $v \in V$ only one of the colors is assigned, therefore we pick the variable that corresponds to that color, which will make the 3-CNF clause specific to that vertex true. Since only one of the colors is assigned to each vertex, type-I clauses are also true because only one of the three variables corresponding to vertex v , v_1, v_2, v_3 will be true at a given time. In the 3-COLORING solution, for each neighboring vertex, we have to pick different colors. Therefore, type-II clauses are also true.

For the “If” direction, suppose we have a solution for ϕ . First, let us look at the 2-CNF clauses. We know that for each vertex $v \in V$ we have three variables, and type-I 2-CNF clauses guarantee that only one of them can be true at a time; otherwise, at least one of the type-I clauses would be false. Then, because of the type-II clauses, truth assignment cannot have the variables corresponding to the same color be true at the same time. Therefore, it guarantees that variables corresponding to different colors are selected by the type-II clauses.

Finally, the 3-CNF clauses guarantee that at least one of the variables corresponding to each vertex is assigned true. Due to these reasons, if ϕ has a truth assignment, then the 3-COLORING instance also has a valid 3-COLORING. Hence, the theorem is proven. $\square$

Let us consider the example in the figure 3.1. First, the C^+ clause involving u is $(u_1 \vee u_2 \vee u_3)$, then the type-I 2-CNF formula involving w is $(\overline{u_1} \vee \overline{u_2}) \wedge (\overline{u_1} \vee \overline{u_3}) \wedge (\overline{u_2} \vee \overline{u_3})$, the the Type-II formula involving u is as follows:

$$\text{for edge } (u, w) : (\overline{u_1} \vee \overline{w_1}) \wedge (\overline{u_2} \vee \overline{w_2}) \wedge (\overline{u_3} \vee \overline{w_3}),$$

$$\text{for edge } (u, v) : (\overline{u_1} \vee \overline{v_1}) \wedge (\overline{u_2} \vee \overline{v_2}) \wedge (\overline{u_3} \vee \overline{v_3}),$$

$$\text{for edge } (u, z) : (\overline{u_1} \vee \overline{z_1}) \wedge (\overline{u_2} \vee \overline{z_2}) \wedge (\overline{u_3} \vee \overline{z_3}).$$

Then, we reduce $(3^+, 1, 2^-)$ -SAT problem to $FT(4)$. Given ϕ as the input over $3n$ variables $x_1, x_2, x_3, \dots, x_{3n}$, we label the 3-CNF clauses as $C_1^+, C_2^+, C_3^+, \dots, C_n^+$ and its 2-CNF clauses as $C_1^-, C_2^-, C_3^-, \dots, C_m^-$. Note that we label the vertices in the graph alphabetically. Therefore, $x_1 = u, x_2 = v, x_3 = w, x_4 = z$, etc. For each vertex x_i there are three variables defined in the reduction: $x_{i,1}, x_{i,2}, x_{i,3}$. We define $L(x_{i,j})$ where $j \in \{1, 2, 3\}$ as the sequence that contains type-1 2-CNF clauses containing $\overline{x_{i,j}}$, each of them repeated twice, followed by type-II CNF clauses, which are repeated twice as well. Each 2-CNF clause appears exactly 4 times in all of these lists. For example, $L(u_1) = L(x_{1,1})$ is the variable for the vertex u and color 1. As we can see, C_1^-, C_2^- are the type-1 CNF clauses that contain $x_{1,1}$ and C_4^-, C_5^-, C_6^- are the 2-CNF clauses that contain $x_{1,1}$. As an example,

$$L(u_1) = L(x_{1,1}) = C_1^- C_1^- \cdot C_2^- C_2^- \cdot C_4^- C_4^- \cdot C_5^- C_5^- \cdot C_6^- C_6^-,$$

$$L(u_3) = L(x_{1,3}) = C_2^- C_2^- \cdot C_3^- C_3^- \cdot C_{10}^- C_{10}^- \cdot C_{11}^- C_{11}^- \cdot C_{12}^- C_{12}^-.$$

Let C_i^+ be the clause $(x_{i,1} \vee x_{i,2} \vee x_{i,3})$, we construct $H_i = C_i^+ \cdot L(x_{i,1}) \cdot C_i^+ \cdot L(x_{i,2}) \cdot C_i^+ \cdot L(x_{i,3}) \cdot C_i^+$, Note that C_i^+ appears 4 times in H_i , and C_i^+ only appears in H_i . Then we construct the full sequence H :

$$H = g_0 g_0 \cdot H_1 \cdot g_1 g_1 \cdot H_2 \cdot g_2 g_2 \cdot H_3 \cdots g_{n-1} g_{n-1} \cdot H_n \cdot g_n g_n.$$

Each separator character $g_l (0 \leq l \leq n)$ appears exactly twice consecutively. Our claim is that if ϕ has a truth assignment, if and only if H induces a feasible LDS that contains all characters C_i^+ for $i = 1 \dots n$, C_j^- for $j = 1 \dots m$ and g_l where $l = 0 \dots n$.

Forward direction is straight forward, suppose we have a truth assignment for the ϕ , then for each x_i , at least one of the $(x_{i,1}, x_{i,2}, x_{i,3})$ is assigned TRUE. Let's say $x_{i,j}$ where $1 \leq j \leq 3$ is assigned TRUE, then we drop the $L(x_{i,j})$ in H_i and then delete the other two copies of the C_i^+ in H_i to form the $C_i^+ C_i^+$ LD-block. Since each 2-CNF copy appears 4 times in H and only one of the $x_{i,j}$ where $1 \leq j \leq 3$ is TRUE and since the same color is not assigned to neighboring vertices, at least two of them would appear as an LDS-block in a feasible LDS solution H' .

Now we look at the reverse direction.

Lemma 3. *If H admits a feasible solution, then in H_i exactly two non-empty subsequences of $L(x_{i,1}), L(x_{i,2})$ and $L(x_{i,3})$ appear in the feasible solution H' .*

Proof. Suppose we drop two of $L(x_{i,1}), L(x_{i,2})$ and $L(x_{i,3})$ and only one of them appears in the feasible LDS solution. For example, if $L(x_{i,1}), L(x_{i,2})$ are dropped, then type-I 2-CNF clauses involving $x_{i,1}$ and $x_{i,2}$ cannot be selected in the LDS solution. Note that type-I 2-CNF clauses involving $x_{i,1}$ and $x_{i,2}$ only appear in $L(x_{i,1})$ and $L(x_{i,2})$; therefore, it contradicts our initial assumption that H has a feasible solution.

Furthermore, we cannot pick all the $L(x_{i,1}), L(x_{i,2})$ and $L(x_{i,3})$ subsequences because, then, we cannot pick C_i^+ for the feasible LDS solution in H' . If we pick all the $L(x_{i,1}), L(x_{i,2})$

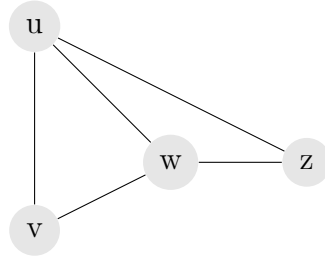


Figure 3.1: Example Graph for the $(3^+, 1, 2^-)$ -SAT problem reduction.

and $L(x_{i,3})$, none of the C_i^+ LDS blocks are feasible in H_i which contradicts our assumption that H has a feasible solution. $\square$

Now we need a way to identify which subsequence came from $L(x_{i,1})$, $L(x_{i,2})$ and $L(x_{i,3})$. We already stated that in each $L(x_{i,j})$, we represent the type-I 2-CNF clauses first. If $L(x_{i,2})$ is dropped, it is easier to find $L(x_{i,1})$ and $L(x_{i,3})$ because they are divided by C_i^+ . However, if we drop the $L(x_{i,1})$ or $L(x_{i,3})$ then after deleting the redundant C_i^+ , we could figure out the subsequences by looking at where type-I to type-II clauses change.

Therefore, if $C_i^+ C_i^+$ is selected by deleting $L(x_{i,j})$ then we assign $x_{i,j} \leftarrow TRUE$ and the other two variables to $FALSE$. This would give us a valid truth assignment for ϕ . This implies the following theorem:

Theorem 6. *FT(4) for LLDS+ is NP-Complete.*

Since FT(4) for LLDS+ is NP-complete, it is apparent that LLDS+(4) is also NP-hard.

Corollary 1. *The optimization version of LLDS+(4) is NP-hard.*

3.4.2 Inapproximability Results

The previous section's findings imply that the optimization version of LLDS+(d), where d is greater than or equal to 4, cannot be approximated in polynomial time (regardless of the approximation factor). This is because any such algorithm would need to produce a

feasible solution. One way to approximate LLDS+ is to develop a bicriteria approximation algorithm for LLDS+. A factor- (α, β) bicriteria approximation algorithm is a polynomial time algorithm that returns a solution of length at least $\frac{OPT}{\alpha}$ and includes at least $\frac{N}{\beta}$ letters, where $N = |\Sigma|$ and OPT is the optimal solution value of LLDS+. We prove that obtaining a bicriteria approximation algorithm for LLDS+ is no easier than approximating LLDS+ itself.

Theorem 7. *If LLDS+(d), $d \geq 4$, admitted a factor- $(\alpha, N^{1-\epsilon})$ bicriteria approximation for any $\epsilon < 1$, then LLDS+(d), $d \geq 6$, would also admit a factor- α approximations, where N is the alphabet size.*

Proof. Assuming the existence of a factor- $(\alpha, N^{1-\epsilon})$ bicriteria approximation algorithm $\mathcal{A}$, we can construct an instance S^* for LLDS+(4) as follows, where S is a sequence obtained from a $(3^+, 1, 2^-)$ -SAT problem; instance ϕ in the proof of Theorem 6. In addition to the set $\{C_i | i = 1..m\} \cup \{g_j | j = 1..n+1\}$ in the alphabet, we include a set of integers $\{1, 2, \dots, (m+n+1)^x - (m+n+1)\}$, where x is an integer to be determined. Therefore,

$$\Sigma = \{C_i | i = 1..m\} \cup \{g_j | j = 1..n+1\} \cup \{1, 2, \dots, (m+n+1)^x - (m+n+1)\}.$$

The sequence S^* is constructed as follows:

$$1 \cdot 2 \cdot ((m+n+1)^x - (m+n+1)) \cdot S \cdot ((m+n+1)^x - (m+n+1)) \cdot ((m+n+1)^x - (m+n+1) - 1) \cdots 2 \cdot 1.$$

It is evident that any bicriteria approximation for S^* would result in an approximate solution for S because any solution containing a number in $\{1, 2, \dots, (m+n+1)^x - (m+n+1)\}$ would produce a solution of size only 2.

Notice that $N = m + (n+1) + (m+n+1)^x - (m+n+1) = (m+n+1)^x$. In this case, the fraction of letters in Σ used to form such an approximate solution satisfies

$$\frac{m + (n + 1)}{(m + n + 1)^x} \leq \frac{1}{N^{1-\epsilon}}.$$

This implies that it is sufficient to choose $x \geq \lceil 2 - \epsilon \rceil = 2$. □

3.5 Polynomial Time algorithms for LLDS+(d) and FT(d) When $d = 3$

3.5.1 Solving Feasibility Testing When $d = 3$

We have established that LLDS+(d) is NP-complete for $d \geq 4$. In the following, we demonstrate that LLDS+(3) is solvable in polynomial time. Additionally, we refer to an LD-block of length j as a j -block for ease of expression. For the purpose of our discussion, we focus only on $j = 2, 3$.

Lemma 4. *Given a string S over Σ such that each letter in S appears at most 3 times, if a feasible solution for $FT(3)$ contains a 3-block then there is a feasible solution for $FT(3)$ that only uses 2-blocks.*

Proof. Suppose S is $S = \dots x^{(1)} \dots x^{(2)} \dots x^{(3)} \dots$, and $x^{(1)}x^{(2)}x^{(3)}$ is a 3-block in a feasible solution for $FT(3)$. Then we could replace $x^{(1)}x^{(1)}x^{(1)}$ with $x^{(1)}x^{(2)}$ or $x^{(2)}x^{(3)}$. The resultant solution would still be a feasible solution for LLDS+(3). □

Corollary 2. *Given a string S over Σ such that each letter in S appears at most 3 times, if there is an optimal solution, after renumbering, for any $i \in \Sigma$, $i^{(2)}$ must be in the optimal solution of LLDS+(3).*

The Corollary 2 is immediately evident from Lemma 4.

Lemma 4 implies that we can solve $FT(3)$ in $O(n)$ time as follows. We first re-number of the letters of S by their middle occurrence (from left to right) as $c_1, c_2, \dots, c_n$. Let $B_1(i) := c_i^{(1)}c_i^{(2)}$ and $B_2(i) := c_i^{(2)}c_i^{(3)}$. Let F be a feasible solution for S .

Let us start from $i = 1$, we put $F := B_1(i) = c_i^{(1)}c_i^{(2)}$ as the first 2-block of the current solution. Then, we loop through $i = 2$ to n as follows:

- If F overlaps $B_1(i)$ and $B_2(i)$, then output ‘no solution’ and exit.
- If F overlaps $B_1(i)$, then $F \leftarrow F \cdot B_2(i)$; otherwise, $F \leftarrow F \cdot B_1(j)$.

The returned solution F by the above algorithm is a feasible solution for FT(3) if the above algorithm does not exit with ‘no solution’. We can clearly see that the returned solution must include all the letters in the alphabet.

Theorem 8. *Let S be a string of length n . FT(3) of the string S can be decided in $O(n)$ time.*

Theorem 8 implies that LLDS+(3) has a factor-1.5 approximation, as any feasible solution for FT(3) would be a factor-1.5 approximation for LLDS+(3). However, the LLDS+(3) problem can be solved in polynomial time after checking that there exists a solution using FT(3) on S , in fact in linear time as well.

3.5.2 Optimization Version of LLDS+(3) When $d = 3$

We have developed an dynamic programming algorithm for LLDS+(3) that operates in polynomial time. Our approach leverages the fact that each letter in the input string appears exactly three times. This assumption is justified because if a letter appears only once, then a feasible solution cannot exist. Similarly, if a letter appears twice, both instances of that letter must be present in any valid solution, which allows us to eliminate any other occurrences of any letter that exist in between those two letters in S . Additionally, we can safely assume that a solution exists, as we can use the FT(3) method to verify its existence prior to applying the LLDS+(3) algorithm.

Recall that we re-numbered the letters in S by their middle occurrence (from left to right) as $c_1, c_2, \dots, c_n$. Following the previous lemma 4, any LLDS+(3) solution must have

a natural structure ordered by the middle occurrence $c_i^{(2)}$. We exploit this structure to come up with a linear dynamic programming solution for LLDS+(3).

For each character $c_i, 1 \leq i \leq n$, we define $B_1(i) := c_i^{(1)}c_i^{(2)}, B_2(i) := c_i^{(2)}c_i^{(3)}$, and $B_3(i) := c_i^{(1)}c_i^{(2)}c_i^{(3)}$. Then, we define the dynamic programming algorithm tables as follows. We define $D[i, j]$ as the maximum value of LDS for the sequence of letters $(c_1, c_2, \dots, c_i)$ where LD-block containing c_i is $B_j(i)$, for $1 \leq j \leq 3$; otherwise, if none of the $B_j(i)$ leads to a feasible solution, then $D[i, j] \leftarrow -\infty$. Similarly, we define another table $D[i]$ as the maximum value of LDS for sequence of letters $(c_1, c_2, \dots, c_i)$ as follows:

$$D[i] = \max_{j=1..3} D[i, j].$$

Then, we calculate the tables $D[-]$ and $D[-, -]$ in linear time as follows:

$$D[i, j] = \max_{1 \leq k \leq 3} \begin{cases} D[i-1, k] + |B_j(i)| & \text{if } B_k(i-1) \text{ does not overlap } B_j(i), \\ -\infty & \text{if } B_k(i-1) \text{ overlaps } B_j(i). \end{cases}$$

Recall that the value $|B_j(i)|$ is either 2 or 3. The optimal value for the final LLDS+(3) solution is stored in $D[n]$.

Theorem 9. *Given a string S of length n , where each letter appears exactly 3 times, the LLDS+(3) problem can be solved in $O(n)$ time.*

Proof. We first check whether there exists a feasible solution for FT(3) using the theorem 8. Then, we obtain an optimal solution for the LLDS+(3) solution by using the using the above dynamic programming algorithm. The above recurrence relation provides an optimal solution because the algorithm scans all (middle) letters from left to right and extends the previous solution using the best possible choice available. It also guarantees that the final solution includes all the letters in the alphabet. The linear running time is evident from the

fact that $D[i, j]$ can be calculated in constant time; hence, $D[i]$ can be calculated in constant time as well. Since we only have $O(n)$ number of cells to update, the total running time is $O(n)$. $\square$

We provide an example of the above-mentioned dynamic programming algorithm as follows: Let $S = 112132323$, then $D[1, 1] = 2$, $D[1, 2] = 2$, $D[1, 3] = 3$. After the update for $i = 2$, we have $D[2, 1] = 4$, $D[2, 2] = 5$, and $D[2, 3] = 5$. Then, after updating $i = 3$, we have $D[3, 1] = \infty$, $D[3, 2] = 6$, and $D[3, 3] = -\infty$. Therefore, the final value for $D[3]$ would be the maximum value of $D[3, 1]$, $D[3, 2]$ and $D[3, 3]$, which is 6.

3.6 Dynamic Programming Algorithm for Weighted-LDS

Let $S = S[1\dots n]$ be an input string, and let $w_x(l)$ denote the weight of an LD-block x^l , where $x \in \Sigma$, $2 \leq l \leq d$, and d is the maximum number of times a letter appears in S . The weight function $w_x(l)$ can be any positive function of x and l , and need not be increasing in l . For example, $w(aaa) = w_a(3) = 8$, and $w(aaaa) = w_a(4) = 5$.

Our objective is to compute the maximum weight letter-duplicated string (Weighted-LDS) using dynamic programming, given $w_x(l)$ for all $x \in \Sigma$ and l . We define $T(n)$ as the value of the optimal solution of $S[1\dots n]$ which contains the character $S[n]$. We also define $w[i, j]$ as the maximum weight LD-block $S[j]^l$ ($l \geq 2$) starting at position i and ending at position j . If such an LD-block does not exist, then $w[i, j] = 0$. Note that $S[j]^l$ does not necessarily have to contain $S[i]$ but it must contain $S[j]$. We can express the recurrence relation as follows:

$$T(0) = 0,$$

$$T(i) = \max_{S[y] \neq S[i]} \begin{cases} T(y) + w[y + 1, i] & \text{if } w[y + 1, i] > 0, \\ 0 & \text{otherwise.} \end{cases}$$

The final solution value is $\max_n T(n)$. This algorithm clearly takes $O(n^2)$ time, assuming $w[i, j]$ is given. We compute the table $w[-, -]$ next.

1. For each pair of l (bounded by d , the maximum number of times a letter appears in S) and letter x , compute

$$w'_x(l) = \max \begin{cases} w'_x(l-1) \\ w_x(l) \end{cases}$$

with $w'_x(1) = w_x(1)$. This can be done in $O(d|\Sigma|) = O(n^2)$.

2. Compute the number of occurrence of $S[j]$ in the range of $[i, j], N[i, j]$. Notice that $i \leq j$ and for the base case we have $S[0] = \emptyset$.

$$N(0, 0) = 0,$$

$$N(0, j) = N(0, k) + 1, \quad k = \max \begin{cases} \{y | s[y] = s[j], 1 \leq y < j\} \\ 0 \end{cases}$$

$$N(i, j) = \begin{cases} N(i-1, j), & \text{if } s[i-1] \neq s[j] \\ N(i-1, j) - 1, & \text{if } s[i-1] = s[j]. \end{cases}$$

This takes $O(n^2)$ time.

3. Finally, we compute,

$$w[i, j] = \begin{cases} w'_{s[j]}(N(i, j)), & \text{if } N(i, j) \geq 2 \\ 0, & \text{else.} \end{cases}$$

This step takes $O(n^2)$ as well.

Theorem 10. *Let S be a string of length n over an alphabet Σ , and d be the maximum number of times a letter appears in S . Given the weight function $w_x(l)$ for $x \in \Sigma$ and $l \leq d$, the maximum weight letter-duplicated subsequence (weighted-LDS) of S can be computed in $O(n^2)$ time.*

3.7 Conclusion

In this chapter, we considered the longest letter-duplicated subsequence problem and its variations. First, we demonstrated that the LLDS problem can be solved in polynomial time using a dynamic programming solution. Subsequently, we explored the constrained longest letter-duplicated subsequence (LLDS+(d)) problem, in which the final solution must encompass all the letters in the alphabet. We proved that for $d \geq 4$, the LLDS+(d) problem is NP-hard. Moreover, we proved that even the feasibility testing (FT(d)) of this problem is NP-hard, indicating the absence of an approximation. Furthermore, we demonstrated that we cannot even have a bi-criteria approximation for the LLDS+(d) problem. When $d \leq 3$, the LLDS+(d) problem becomes polynomially solvable, for which we provided a dynamic programming solution. We also provided a dynamic programming solution for the weighted-LDS problem, wherein we have an input function for the weight of an LD block. This could be useful in a biological context, especially when assigning different weights to specific genes in the input sequence.

We believe that the constrained longest-letter duplicated subsequence problem holds particular relevance in biological contexts, as it aims to extract a skeleton of the original sequence containing at least one copy of each gene. Notably, we provided a polynomial-time algorithm for the LLDS+(3) problem. In a biological context, a gene can appear more than three times in the genome, suggesting the need for heuristic algorithms to address the LLDS+(d) problem in future research.

In table 3.1, we provided an overview of the complexity of variations of the LLDS+(d)

Table 3.1: Summary of results on LLDS+ and FT

d	$LLDS+(d)$	$FT(d)$	Approximability of $LLDS+(d)$
$d \geq 4$	NP-hard	NP-complete	No approximation
$d = 3$	P	P	Not applicable

problem.

3.7.1 Notes

Part of this chapter has been published in 33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022) [72].

CHAPTER FOUR

THE LONGEST SUBSEQUENCE-REPEATED SUBSEQUENCE PROBLEM

4.1 Introduction

The task of discovering patterns within long sequences is a fundamental issue in string algorithms, as well as in the fields of combinatorial pattern matching and computational biology. This chapter focuses on identifying long patterns that occur on a global scale. Previous research has tackled this issue, including the challenge of determining the longest square subsequence within a given sequence S , which has a length of n . The objective is to locate the longest subsequence within S that appears twice. Kosowski's work in [64] found a solution to this problem with a time complexity of $O(n^2)$. It has been demonstrated that improving on this time complexity is not likely, as any algorithm with a complexity of $o(n^{2-\epsilon})$ would lead to a subquadratic limit for the longest common subsequence (LCS) problem. This is only feasible if the SETH conjecture is proven false, as demonstrated in [14].

As previously mentioned, duplication plays a crucial role in evolution in the field of biology. There are two primary forms of duplication: arbitrary segment duplications and tandem duplications. Arbitrary segment duplication involves the selection of a specific segment within a genome and its subsequent insertion in a different location. On the other hand, tandem duplication is a structural rearrangement that involves the duplication of a DNA segment and its insertion adjacent to the original segment in a serial manner ($X \rightarrow XX$) [91]. Studies have shown that segment duplications are commonly observed in cancer genomes [28, 74, 100]. Meanwhile, tandem duplications are frequently found in various scenarios. For example, the expansion of a three nucleotide CAG repeat through tandem duplications is closely linked to Huntington's disease [82]. Furthermore, the analysis

of whole-genome sequencing data from tumors has revealed that massive rearrangements, similar to the tandem duplicator phenotype, are a specific characteristic of cancer [54]. Tandem duplications have also been observed in certain cancers, such as ovarian and breast cancer genomes [84, 103]. Notably, a subset of ovarian cancer samples exhibits a distinct tandem duplication phenotype shared with triple-negative breast cancers [84]. In 1980, Szostak and Wu presented evidence suggesting that gene duplication, particularly in the form of tandem duplications, is the primary driving force behind evolution [104]. In fact, the first sequenced human genome revealed that approximately 3% of the genome consists of tandem repeats [73].

In addition to duplication, another driving force behind evolution is mutation. Suppose we have a genome $ACGT$ and we evolve this genome by doing several tandem duplications. Suppose we retrieve $S' = ACGT \cdot ACGT \cdot ACGT$, then we do a couple of more duplications on the first ACG in S' to retrieve $H = ACG \cdot ACG \cdot T \cdot ACGT \cdot ACGT$. Then we do some mutations on H such that the first A is changed to G the second A is changed to T and the third C is deleted. Then we would retrieve $H' = GCG \cdot TCG \cdot T \cdot AGT \cdot ACGT$. Then it becomes harder to figure out the tandem duplications that occurred in the latest sequence. Motivated by this application, Lai et al. proposed [72] a problem called *Longest Letter duplicated subsequence problem*. The longest letter-duplicated subsequence problem can be used to retrieve a skeleton of such genomes.

In this chapter, we propose a more general version of the longest-letter duplicated subsequence (LLDS) problem, which we denote as the *Longest-Subsequence Repeated Subsequence (LSRS) problem* of S . In this version of the problem, instead of finding letter-duplicated blocks, we try to extract subsequence-repeated blocks from the given sequence. In other words, each x_i is a subsequence instead of a letter. If we use the previous example H' , the LLDS of H' would be G^2T^4 while the LSRS would be $(CGT)^2(AGT)^2$ which implies that LSRS gives more information about the tandem duplications. This motivates us to

study LSRS and its related variants. Specifically, we are interested in the LSRS problem and its complexity when the maximum duplication number d is bound. This variation has a practical interest in bioinformatics, as in many genomes, whole genome duplication is a rare event, and the number of duplications is a small constant. For example, it was discovered that plants have undergone up to three rounds of whole genome duplications, resulting in a number of duplicates bounded by 8 [114].

4.2 Preliminaries

Let $\mathbb{N}$ be the set of natural numbers. For $q \in \mathbb{N}$, we use $[q]$ to represent the set of integers $\{1, 2, 3, \dots, q\}$ and $[i, j] = \{i, i+1, \dots, j\}$. Throughout this chapter, we will use Σ to denote a finite alphabet, and we will define the sequence S over Σ . $S[i]$ will be used to denote the i^{th} letter in S and $S[i \dots j]$ to denote the substring of S starting from i and ending at j indices. Using the run-length representations, S can be represented as $y_1^{d_1} y_2^{d_2} \dots y_q^{d_q}$ where $y_i \in \Sigma$, $y_j \neq y_{j+1}$ and $d_i \geq 1$ for all $i \in [q]$, $j \in [q-1]$. If a letter y appears multiple times in S , we could use $y^{(i)}$ to denote the i^{th} copy of it. A subsequence of S is a string obtained by deleting some letters in S . A square subsequence of S is one in the form of X^2 where X is a subsequence of S , a cubic subsequence of S is a sequence in the form of X^3 where X is a subsequence of S .

A subsequence S' of S is a subsequence repeated subsequence (SRS) of S if it is in the form of $x_1^{d_1} x_2^{d_2} \dots x_k^{d_k}$ where x_i is a subsequence of S and $x_j \neq x_{j+1}$ and $d_i \geq 2$, for all $i \in [k]$, $j \in [k-1]$. We name x_i a subsequence-repeated block (SR-block). For example, let $S = ATAGATGACGCGGCGCGT$, then $S_1 = ATA \cdot ATA \cdot CG \cdot CG \cdot CG \cdot CG$ and $S_2 = ATG \cdot ATG \cdot CG \cdot CG \cdot CG \cdot CG$ are solutions for the longest subsequence-repeated subsequence of S . Also, the longest square subsequence and cubic subsequence of S are $(ATAG)^2$ and $(AT)^3$ respectively.

4.3 LSRS Problem

A subsequence S' of S is a subsequence-repeated subsequence (SRS) of S if it is in the form of $x_1^{d_1} x_2^{d_2} x_3^{d_3} \dots x_k^{d_k}$ where x_i is a subsequence of S , $x_j \neq x_{j+1}$ and $d_i \geq 2$, for $i \in [k], j \in [k-1]$. $x_i^{d_i}$ a *subsequence-repeated block* or *SR-block*.

Assuming we have a polynomial time algorithm to solve the longest cubic subsequence problem, we can come up with a dynamic programming solution for LSRS. In fact, trivially, we can come up with a polynomial-time algorithm for the longest cubic subsequence problem by using the longest common subsequence solution for three input sequences. The idea is that we divide the input string into all the possible 3-partitions and run the longest common subsequence solution for three input sequences for every instance. Since there are $O(n^2)$ possible ways to divide the string into 3-partitions, we can solve the longest cubic subsequence problem in $O(n^5)$ time since the longest subsequence problem for three input sequences takes $O(n^3)$.

Observation 2. *Suppose there is an optimal LSRS solution for a given sequence S of length n , in the form of $x_1^{d_1} x_2^{d_2} \dots x_k^{d_k}$. Then it is possible to decompose it into a generalized SR-subsequence $y_1^{e_1} y_2^{e_2} \dots y_p^{e_p}$, where*

- $2 \leq e_i \leq 3$, for $i \in [p]$,
- $p \geq k$,
- y_j does not have to be different from y_{j+1} , for $j \in [p-1]$.

Proof. To prove this, we say any natural number $k \geq 2$ can be decomposed as $k = k_1 + k_2 + k_3 + \dots + k_n$, such that $2 \leq k_i \leq 3$ for $1 \leq i \leq n$. Therefore, every d_i can be decomposed into a sum of 2's and 3's. Then, using the given optimal solution and its decomposed sums of SR-Blocks, we could obtain the corresponding SR-subsequence by combining $y_i^{e_i} y_{i+1}^{e_{i+1}}$ when $y_i = y_{i+1}$. □

4.3.1 Solution for The LSRS Problem

Now let us design a dynamic programming algorithm for LSRS . Let $L(i)$ be the length of the optimal LSRS solution for $S[1 \cdots i]$. Let $LSquareS[i, j]$ be the longest square subsequence and $LCubicS[i, j]$ be the longest cubic subsequence of $S[i, j]$ respectively. Then the recurrence relation is defined as follows:

$$\begin{aligned} L(0) &= 0, \\ L(1) &= 0, \\ L(i) &= \max \begin{cases} L(j) + LSquareS[j + 1, i], & j < i - 1 \\ L(j) + LCubicS[j + 1, i], & j < i - 2 \end{cases} \end{aligned}$$

Computing the longest square subsequence of length n sequence takes $O(n^2)$ [64], since there are $O(n^2)$ cells in the dynamic programming table, it will take $O(n^4)$ to calculate all the longest square subsequences, and using the trivial algorithm for the longest cubic subsequence, it will take $O(n^7)$ time to calculate all the $O(n^2)$ cells. Once all the longest square subsequences and longest cubic subsequences are calculated, it will take quadratic time to calculate $L(n)$. Therefore, the time complexity is $O(n^7)$. One of the interesting areas is to further look at whether we could improve the Longest Common Subsequence algorithm to $O(n^4)$ for length n sequence. However, using the idea presented in Section 4.7 we can compute all the longest cubic subsequences of all the substrings in a given sequence S in $O(n^6)$ time (note that this is different from finding the longest cubic subsequence of a single string). Using this idea, we can solve LSRS in $O(n^6)$.

4.4 LSRS+ Problem

In this section, we focus on the following variations of the *LSRS* problem.

Definition 10. Constrained Longest Subsequence-Repeated Subsequence*(LSRS+ for short)***Input:** A sequence S with length n over an alphabet Σ and an integer ℓ .**Question:** Does S contain a subsequence-repeated subsequence S' with length at least ℓ such that all letters in Σ appear in S' ?**Definition 11. Feasibility Testing (FT for short)****Input:** A sequence S with length n over an alphabet Σ .**Question:** Does S contain a subsequence-repeated subsequence S'' such that all letters in Σ appear in S'' ?4.5 LSRS+(3) Is Polynomially Solvable

Our objective is to solve LSRS+(3), where we are given a sequence S of length n such that each letter occurs at least twice and at most three times. In fact, any optimal solution can be expressed as $x_1^{d_1} \cdots x_k^{d_k}$, where x_i is a subsequence of S and $d_i \in \{2, 3\}$ for $i \in [k]$. Throughout this subsection we assume that all letters in S appear at least twice and at most three times - if a letter appears only once in S then there is no solution for the corresponding LSRS+(3) instance. To accomplish this, we propose a dynamic programming approach. We define six tables: $S_2[i, j]$, $C_2[i, j]$, $S_3[i, j]$, $C_3[i, j]$, $L[i, j]$, and $C[i, j]$, where $1 \leq i < j \leq n$.

1. $C_2[i, j]$ is the set of letters that all appear at least twice in $S[i, j]$ and at least one letter has to appear exactly twice in $S[i, j]$; otherwise, $C_2[i, j]$ is empty.
2. $S_2[i, j]$ is the length of a longest square subsequence containing all the letters in $C_2[i, j]$. If such a local feasible solution does not exist, set $S_2[i, j] \leftarrow -1$; otherwise, we say that this local feasible solution is 2-feasible.
3. $C_3[i, j]$ is the set of letters that all appear three times in $S[i, j]$ if no letter in $S[i, j]$ appears exactly twice; otherwise $C_3[i, j]$ is empty.

4. $S_3[i, j]$ is the length of the longest cubic subsequence containing all the letters in $C_3[i, j]$; otherwise $S_3[i, j]$ is the length of a longest square subsequence containing all the letters in $C_3[i, j]$. If such a local feasible solution (cube or square) does not exist, set $S_3[i, j] \leftarrow -1$; otherwise, we say that this local feasible solution is 3-feasible or 2-feasible respectively (depending on whether the local solution is cubic or square).
5. $C[i, j]$ is the set of letters appearing at least twice in $S[i..j]$. The $C[i, j]$'s can be computed, each with a linear scan, in a total of $O(n^3)$ time. $C[i, j]$'s are only used to enforce the coverage condition.
6. $L[i, j]$ is the length of a feasible solution of $S[i, j]$ that covers all the letters in $C[i, j]$.

The initial values of $L[i, j]$, for $i < j$, can be set as follows:

$$L[i, j] = \begin{cases} S_3[i, j] & \text{If } S_3[i, j] > 0 \\ S_2[i, j] & \text{else If } S_2[i, j] > 0 \\ -1 & \text{Otherwise} \end{cases}$$

Then update the general case for $L[i, j]$ recursively as follows.

Note that the initialized solution might not be final. Hence, we need to update $L[i, j]$.

We update the general case for $L[i, j]$ recursively as follows. This is done bottom-up, ordered by the ascending length of $S[i, j]$.

$$L[i, j] = \begin{cases} \max \{ \max_{i < k < j} \{ L[i, k] + L[k + 1, j] \} \} & \text{If } L[i, k] > 0 \text{ and } L[k + 1, j] > 0 \\ & \text{and} \\ C[i, j] == C[i, k] \cup C[k + 1, j] & \\ -1 & \text{Otherwise} \end{cases}$$

The initial value of $C[i, j], i < j$ can be set as

$$C[i, j] = \begin{cases} C_3[i, j] & \text{If } S_3[i, j] > 0 \\ C_2[i, j] & \text{else if } S_2[i, j] > 0 \\ \emptyset & \text{Otherwise} \end{cases}$$

The $C[i, j] == C[i, k] \cup C[k + 1, j]$ condition is used to ensure that $L[i, j]$ is only updated when all the letters in $C[i, j]$ are covered. Then, the maximum of $L[i, k] + L[k + 1, j]$, if greater than the $L[i, j]$, replaces the previous $L[i, j]$ value.

An optimal solution is computed if $L[1, n] > 0$, and its solution value is stored in $L[1, n]$. Clearly, with an additional table, one can easily retrieve such an optimal solution, if exists.

Lemma 5. $C_2[i, j] \cap C_3[i, j] = \emptyset$.

Proof. This is obvious by definition. $C_2[i, j]$ is non-empty only when there is at least one letter appearing exactly twice in $S[i, j]$. Conversely, $C_3[i, j]$ is non-empty when there is no letter appearing exactly twice in $S[i, j]$. These two conditions are complementary, therefore their intersection is empty. $\square$

Regarding $S_3[i, j]$, the following lemma says that if a 3-feasible solution does not exist then any of those 2-feasible solutions could be stored.

Lemma 6. *If a 3-feasible solution for $S_3[i, j]$ does not exist, then any 2-feasible solution for $S[i, j]$ can be stored (without changing the optimal solution value).*

Proof. By definition, $C_3[i, j]$ contains all the letters in $S[i, j]$ which appear exactly three times; moreover, there is no letter x which appears exactly twice in $S[i, j]$. Hence, if a letter y appears exactly once in $S[i, j]$ it would never appear as a local feasible solution for $S_3[i, j]$.

Therefore, if a 3-feasible solution does not exist, by definition, we would consider only a 2-feasible solution for $S[i..j]$ which covers all the letters in $C_3[i, j]$. The length of such a 2-feasible solution is exactly $2 \cdot |C_3[i, j]|$. $\square$

Theorem 11. *Given a string S of length n , where each letter appears exactly three times, the problem of LSRS+(3) can be solved in $O(n^4)$ time.*

Proof. The cost is dominated by filling $S_3[i, j]$ and $S_2[i, j]$. Updating $L[i, j]$ takes $O(n^3)$ time. $S_3[-, -]$ has $O(n^2)$ cells, and the initial update of $S_3[i, j]$ cell can be done in $O(n)$ time. If there exists a longest cubic sequence that contains all the letters in $C_3[i, j]$, then it must be unique, and all the letters must only appear once in the repeated sequence. Therefore, we could divide the sequence into three segments of length $|C_3[i, j]|$ and check for the repeated sequences. If a cubic sequence exists, we can check whether it covers all the letters in $C_3[i, j]$ in linear time. If not, then we need to calculate the longest square subsequence of $S_3[i, j]$ and check whether it has the length $2 \cdot |C_3[i, j]|$. Calculating the longest square subsequence can be done in $O(n^2)$ using Kosowski's algorithm [64]. Therefore, in the worst case, the time taken to update the $S_3[i, j]$ cell is $O(n^2)$ and since $S_3[i, j]$ table has $O(n^2)$ cells, the total time would be $O(n^4)$. Similarly, updating the $S_2[i, j]$ table takes $O(n^4)$ time. $\square$

Clearly, LSRS+(3) has a solution (i.e., $L[1, n] > 0$) if and only if FT(3) has a feasible solution.

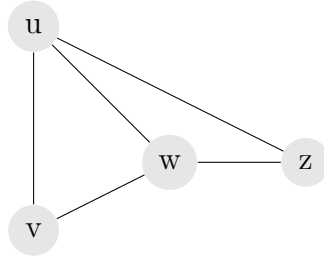


Figure 4.1: Example Graph for the $(3^+, 1, 2^-)$ -SAT problem reduction

4.6 FT(4) and LSRS+(4) Are NP-hard

In this section, we will demonstrate that LSRS+(4) is NP-complete through a reduction from $(3^+, 1, 2^-)$ -SAT problem. Firstly, we establish that $(3^+, 1, 2^-)$ -SAT problem is already known to be NP-complete, as shown in the previous chapter. Refer to the Theorem 5. We then proceed to construct a reduction from $(3^+, 1, 2^-)$ -SAT problem to LSRS+(4). In fact, we will reduce $(3^+, 1, 2^-)$ -SAT problem to FT(4) and then FT(4) to LSRS+(4) to complete the reduction.

Now, we will reduce $(3^+, 1, 2^-)$ -SAT problem to FT(4). We construct the input ϕ for $(3^+, 1, 2^-)$ -SAT problem directly from a 3-coloring instance. The resulting ϕ will contain $3n$ variables $x_1, x_2, \dots, x_{3n}$ and m 2-CNF clauses, with the 3-CNF clauses labeled as $C_1^+, C_2^+, \dots, C_n^+$ and the 2-CNF clauses labeled as $C_1^-, C_2^-, \dots, C_m^-$. The vertices in Figure 4.1 can be renamed alphabetically as $x_1, x_2, \dots, x_{3n}$, etc. It should be noted that the same C_i^+ and C_i^- clauses used in the proof of Theorem 5 are utilized in this reduction. The proof of Theorem 5 introduced two types of 2-CNF clauses, Type-1 and Type-2, as vertex color restriction and edge color restriction clauses, respectively.

For example, in Figure 4.1, the C_1^+ clause (the 3-clause for vertex u) would be,

$$C_1^+ = (u_1 \vee u_2 \vee u_3)$$

The type-1 2-CNF clauses for vertex u would be,

$$(\overline{u_1} \vee \overline{u_2}) \wedge (\overline{u_1} \vee \overline{u_3}) \wedge (\overline{u_2} \vee \overline{u_3}) = C_1^- \wedge C_2^- \wedge C_3^-$$

The Type-2 clauses for edge (u, v) would be,

$$(\overline{u_1} \vee \overline{v_1}) \wedge (\overline{u_2} \vee \overline{v_2}) \wedge (\overline{u_3} \vee \overline{v_3}) = C_4^- \wedge C_5^- \wedge C_6^-$$

The type-2 clauses for edge (u, w) would be,

$$(\overline{u_1} \vee \overline{w_1}) \wedge (\overline{u_2} \vee \overline{w_2}) \wedge (\overline{u_3} \vee \overline{w_3}) = C_7^- \wedge C_8^- \wedge C_9^-$$

The type-2 clauses for edge (u, z) would be,

$$(\overline{u_1} \vee \overline{z_1}) \wedge (\overline{u_2} \vee \overline{z_2}) \wedge (\overline{u_3} \vee \overline{z_3}) = C_{10}^- \wedge C_{11}^- \wedge C_{12}^-$$

Let $L(x_i)$ denote the set of type-1 2-CNF clauses that include $\overline{x_i}$, followed by the set of type-2 2-CNF clauses containing $\overline{x_i}$ repeated twice consecutively. In the example shown in Figure 4.1, $L(x_1) = L(u_1) = C_1^- C_1^- C_2^- C_2^- \cdot C_4^- C_4^- C_7^- C_7^- C_{10}^- C_{10}^-$. Additionally, three unique letters 1_i , 2_i , and 3_i are defined for each C_i^+ , and the alphabet used to create the final sequence H is constructed from these letters. Hence, the alphabet we use to construct the final sequence H is,

$$\Sigma = \{F_j^- | j \in [m]\} \cup \{1_i, 2_i, 3_i | i \in [n]\} \cup \{g_k, g'_k | k \in [n-1]\}.$$

The g_k, g'_k are used as separators and each appear twice. Then for each $C_i^+ = (x_{i,1} \vee$

$x_{i,2} \vee x_{i,3}$). We construct:

$$H_i = 2_i \cdot L(x_{i,1}) \cdot 1_i 2_i 1_i \cdot L(x_{i,2}) \cdot 2_i \cdot L(x_{i,3}) \cdot 3_i 2_i 3_i .$$

Each 2_i and all the 2-CNF clauses appear 4 times in H_i , while 1_i and 3_i appears twice. Finally, we construct the final sequence H as

$$H = H_1 \cdot g_1 g'_1 g_1 g'_1 \cdot H_2 \cdot g_2 g'_2 g_2 g'_2 \cdot H_3 \cdots g_{n-1} g'_{n-1} g_{n-1} g'_{n-1} \cdot H_n .$$

We claim that ϕ has a valid truth assignment if and only if H admits a feasible LSRS+(4) solution (the solution contains all the letters in the alphabet Σ).

The forward direction is pretty straight-forward. If there is a truth assignment, then only one of the variables $x_{i,1}, x_{i,2}, x_{i,3}$ for each i is assigned true. Then, for each $x_{i,j}, 1 \leq j \leq 3$, we delete $L(x_{i,j})$ in H_i . Then we delete some $1_i, 2_i, 3_i$ to obtain a feasible solution H'_i as follows:

1. If $j = 1$, we have $H'_i = 2_i 1_i 2_i 1_i \cdot L(x_{i,2}) L(x_{i,3}) \cdot 3_i 3_i$
2. If $j = 2$, then $H'_i = L(x_{i,1}) \cdot 1_i 2_i 1_i 2_i \cdot L(x_{i,3}) \cdot 3_i 3_i$
3. If $j = 3$, then $H'_i = L(x_{i,1}) \cdot 1_i 1_i \cdot L(x_{i,2}) \cdot 2_i 3_i 2_i 3_i$

Then, for the reverse direction, we show the following lemma:

Lemma 7. *If H admits a feasible (SRS) solution, then exactly two (non-empty) subsequences of $L(x_{i,1}), L(x_{i,2})$ and $L(x_{i,3})$ appear in the feasible solution H' (or, exactly one of the three is completely deleted from H).*

Proof. As a result of constructing type-1 2-CNF clauses on $x_{i,1}, x_{i,2}$, and $x_{i,3}$, if only one of the three lists, namely $L(x_{i,1}), L(x_{i,2})$, and $L(x_{i,3})$, which is a non-empty subsequence, appears

in a possible solution H' for the SRS problem, then the type-1 2-CNF clauses associated with $x_{i,2}$ and $x_{i,3}$ would be absent in H' , which contradicts the definition of a feasible solution.

However, the construction of H_i leads us to assert that leaving all three non-empty subsequences of the lists $L(x_{i,1}), L(x_{i,2})$, and $L(x_{i,3})$ in a feasible SRS solution H' is not possible (we only need to consider the portion obtained from H_i , say H'_i). If this were the case, we would consider five scenarios: (1) if $1_i 1_i$ and $3_i 3_i$ are substrings of H'_i ; (2) if $2_i X \cdot 2_i X$ is a substring of H'_i , with the first X being a substring in $L(x_{i,1})$ and the second X being a substring in $L(x_{i,2})$, (3) if $2_i X \cdot 2_i X$ is a substring of H'_i with the first X being a substring in $L(x_{i,2})$ and the second X being a substring in $L(x_{i,3})$, (4) if $X 2_i \cdot X 2_i$ is a substring of H'_i with the first X being a substring in $L(x_{i,1})$ and the second X being a substring in $L(x_{i,2})$, and (5) if $X 2_i \cdot X 2_i$ is a substring of H'_i with the first X being a substring in $L(x_{i,2})$ and the second X being a substring in $L(x_{i,3})$. We assert that all five of these cases are impossible:

1. Case 1: This is impossible as 2_i 's would not be present in a feasible solution.
2. Case 2-4: This is impossible as 1_i 's would not be present in a feasible solution.
3. Case 5: This is impossible as 3_i 's would not be present in a feasible solution.

□

Let $L'(x_{i,j})$ be a non-empty subsequence of $L(x_{i,j})$. With the above lemma, the reverse direction can be proved as follows.

1. If $H'_i = 2_i 1_i 2_i 1_i \cdot L'(x_{i,2}) \cdot L'(x_{i,3}) \cdot 3_i 3_i$, then assign $x_{i,1} \leftarrow TRUE$, $x_{i,2} \leftarrow FALSE$, $x_{i,3} \leftarrow FALSE$.
2. If $H'_i = L'(x_{i,1}) \cdot 1_i 2_i 1_i 2_i \cdot L'(x_{i,3}) 3_i 3_i$, then assign $x_{i,2} \leftarrow TRUE$, $x_{i,1} \leftarrow FALSE$, $x_{i,3} \leftarrow FALSE$.

3. If $H'_i = L'(x_{i,1}) \cdot 1_i 1_i \cdot L'(x_{i,2}) \cdot 2_i 3_i 2_i 3_i$, then assign $x_{i,3} \leftarrow TRUE$, $x_{i,1} \leftarrow FALSE$, $x_{i,2} \leftarrow FALSE$.

This assignment satisfies all the 2-CNF clauses $(\bar{x}_{i,j} \vee \bar{x}_{k,l})$ in ϕ . Each of these clauses must appear in either $L'(x_{i,j})$ or $L'(x_{k,l})$ in H' . At least one of $x_{i,j}$ and $x_{k,l}$ is assigned FALSE, making the clause TRUE. Thus, this proves the validity of this truth assignment for ϕ . This leads us to the following theorem:

Theorem 12. *FT(4) is NP-complete.*

Since FT(4) is NP-complete, the optimization problem LSRS+(4) is certainly NP-hard.

Corollary 3. *The optimization version of LSRS+(4) is NP-hard.*

Note that the above proof implies that the optimization version of LSRS+(4) does not admit any polynomial-time approximation (regardless of the approximation), as any such approximate solution would form a feasible solution for FT(4). In fact, using a similar argument as in [72] (Section 3.4.2), even finding a good bi-criteria for LSRS+(4) is not possible.

4.7 An $O(n^6)$ Algorithm for The Longest Cubic Subsequences of All Substrings in a Given Sequence

First, we define $LCS(S, b, c)$. We consider a sequence S of length n and integers b and c such that $0 < b < c < n$. The value $LCS(S, b, c)$ represents the length of the longest common subsequence between $S[1..b]$, $S[b+1..c]$, and $S[c+1..n]$. In the problem of finding the longest cubic subsequence, the objective is to determine $\max_{b,c} LCS(S, b, c)$. A match in S refers to a triple (i, j, k) where $i < j < k$ and $S[i] = S[j] = S[k]$. We say $(i, j, k) < (i', j', k')$ if $i < i'$, $j < j'$, and $k < k'$. Given integers b and c such that $0 < b < c < n$, $LCS(S, b, c)$ corresponds to the length of the longest sequence of matches $(i_1, j_1, k_1) < (i_2, j_2, k_2) < \dots < (i_l, j_l, k_l)$

where l is the length of the sequence, and each triple satisfies $i_h \in [1, b]$, $j_h \in [b + 1, c]$, and $k_h \in [c + 1, n]$.

To show that $LCS(S, b, c)$ corresponds to the shortest path between two specific vertices in the alignment graph of S against itself three times, we define the alignment digraph of S as follows: First, we construct a 3D grid graph with sides labeled $0, 1, \dots, n$, and edges oriented from the vertex labeled $(0, 0, 0)$ to the vertex labeled (n, n, n) . Next, we add a diagonal path of length two from $(i - 1, j - 1, k - 1)$ to (i, j, k) for each triple of positions (i, j, k) in S that match. Then, we consider the alignment graph of S against itself three times. In this graph, the vertices correspond to triples (i, j, k) , where i, j , and k are indices in S , and the edges correspond to matches between the characters in S .

Given this definition of the alignment digraph, we can observe that $LCS(S, b, c)$ corresponds to the length of the shortest path from $(0, b, c)$ to (b, c, n) in the alignment digraph. Specifically, the path consists of a sequence of diagonal edges that correspond to matches in S , where the i^{th} diagonal edge corresponds to the i^{th} match in the longest sequence of matches $(i_1, j_1, k_1) < (i_2, j_2, k_2) < \dots < (i_l, j_l, k_l)$ that satisfies the conditions described above. Thus, finding the longest cubic subsequence in S is equivalent to finding the shortest path from $(0, b, c)$ to (b, c, n) in the alignment digraph.

Formally, let

$$V(G) = \{v_{i,j,k} : i, j, k \in \{0, 1, \dots, n\}\} \cup \\ \{d_{i,j,k} : i < j < k \text{ and } S[i] = S[j] = S[k]\}.$$

As for $E(G)$, for each vertex $v_{i,j,k} \in V(G)$, add an edge from $v_{i,j,k}$ to the existing vertices in $\{v_{i+1,j,k}, v_{i,j+1,k}, v_{i,j,k+1}\}$. All the edges created in this step are called straight edges. Then, for each match (i, j, k) in S , add an edge from $v_{i-1,j-1,k-1}$ to $d_{i,j,k}$ and an edge from $d_{i,j,k}$ to $v_{i,j,k}$. We call these two diagonal edges and refer to the created path of length two as a

diagonal. When possible, we may write v_{ijk} instead of $v_{i,j,k}$.

It is important to note that $|V(G)|$ and $|E(G)|$ are both in $O(n^3)$ since each vertex has at most four incoming and outgoing edges. The diagonals have a length of two to ensure that the distance from any vertex v_{ijk} to another vertex q is no greater than the distance from $v_{i-1,j,k}$, $v_{i,j-1,k}$, or $v_{i,j,k-1}$ to q . This property is retained for all preceding vertices in the graph as well.

The purpose of G is shown below.

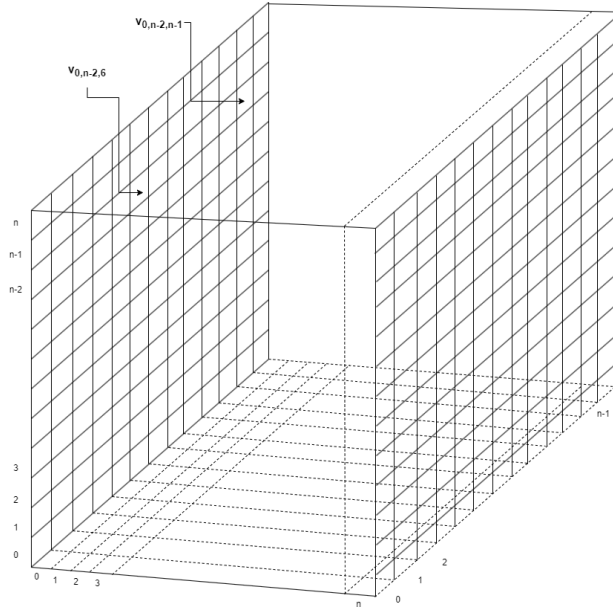


Figure 4.2: An illustration of the subgrid containing $v_{0,b,c}$ cells. (the y -axis has the b coordinates, the z -axis the c coordinates). We can think of the cell (a, b, c) in the 3D-grid as the vertex $v_{a,b,c}$.

Lemma 8. *For integers $0 < b < c < n$, there exists a common subsequence of length r between $S[1..b]$, $S[b+1..c]$, and $S[c+1..n]$ if and only if G contains a path of length $n - r$ from v_{0bc} to v_{bcn} .*

Proof. Suppose Q is a common subsequence of length r between $S[1..b]$, $S[b+1..c]$, and $S[c+1..n]$, and let $(i_1, j_1, k_1) < \dots < (i_r, j_r, k_r)$ be the set of matches that correspond to Q ,

ordered from left to right in the three substrings. In the graph G , there exist diagonal edges from v_{i_h-1, j_h-1, k_h-1} to v_{i_h, j_h, k_h} for every $h \in [r]$. Moreover, since the matches are ordered, we can traverse from v_{i_h, j_h, k_h} to $v_{i_{h+1}, j_{h+1}, k_{h+1}}$ with a path of length.

$$\begin{aligned} & i_{h+1} - 1 - i_h + j_{h+1} - 1 - j_h + k_{h+1} - 1 - k_h + 2 \\ &= i_{h+1} - i_h + j_{h+1} - j_h + k_{h+1} - k_h - 1 \end{aligned}$$

(To see this, go from v_{i_h, j_h, k_h} to $v_{i_{h+1}-1, j_{h+1}-1, k_{h+1}-1}$ using straight edges, and take the length two diagonal.)

Consider the path that starts at v_{0bc} and goes to v_{i_1-1, j_1-1, k_1-1} using straight edges, and then to v_{i_1, j_1, k_1} using the length two diagonal. From there, we visit the other v_{i_h, j_h, k_h} vertices in order using the aforementioned paths, then go from v_{i_r, j_r, k_r} to v_{bcn} using straight edges. This can be done with a path of length:

$$\begin{aligned} & i_1 + j_1 - b + k_1 - c - 1 + \\ & \sum_{h=1}^{r-1} (i_{h+1} - i_h + j_{h+1} - j_h + k_{h+1} - k_h - 1) + \\ & b - i_r + c - j_r + n - k_r \end{aligned}$$

After removing all the canceling terms, this becomes $-1 + \sum_{h=1}^{r-1} -1 + n = n - r$, as desired.

On the other hand, suppose there exists a path P of length $n - r$ from v_{0bc} to v_{bcn} . Note that a path containing only straight edges would require $b - 0 + c - b + n - c = n$ edges. However, each diagonal skips three straight edges but costs only two edges. Therefore, the length of P is n minus the number of diagonals used. This implies that P must use exactly r diagonals. Let $v_{i_1, j_1, k_1}, \dots, v_{i_r, j_r, k_r}$ be the endpoints of these diagonals in the order that they are visited, and let $(i_1, j_1, k_1), \dots, (i_r, j_r, k_r)$ be the set of matches corresponding to these

vertices. We claim that these matches form a set of r matches in a common subsequence of $S[1..b]$, $S[b+1..c]$, and $S[c+1..n]$.

Observe that due to the directions of the edges in G , we have $i_h \leq i_{h+1}$, $j_h \leq j_{h+1}$, $k_h \leq k_{h+1}$ for $h \in [r-1]$. Furthermore, since each diagonal corresponds to an increase of exactly 1 in each of the three coordinates, the above inequalities must be strict. Therefore, $(i_1, j_1, k_1) < \dots < (i_r, j_r, k_r)$.

We need to establish that every triple in the list is a match between characters in the substrings $S[1..b]$, $S[b+1..c]$, and $S[c+1..n]$. Let (i_h, j_h, k_h) be an arbitrary triple from the list. We aim to show that i_h belongs to the range $[1..b]$, j_h belongs to $[b+1..c]$, and k_h belongs to $[c+1..n]$.

Since no diagonal in G ends with a zero coordinate, we have $i_h \geq 1$. Moreover, if $i_h > b$, then visiting the vertex v_{i_h, j_h, k_h} would prevent us from returning to the starting vertex v_{bcn} , because of the directed edges in G . Therefore, we must have $i_h \leq b$. Similarly, we have $j_h \leq c$ and $k_h > c$, since otherwise, we would be unable to return to v_{bcn} .

Since the path P starts at v_{0bc} , we know that j_h must be greater than b because, after traversing any diagonal, the second coordinate must be $b+1$ or more. By the same reasoning, we must have $k_h > c$. Hence, we have shown that $1 \leq i_h \leq b$, $b+1 \leq j_h \leq c$, and $c+1 \leq k_h \leq n$, which implies that all the triples in the list correspond to matches between $S[1..b]$, $S[b+1..c]$, and $S[c+1..n]$.

Consequently, we can conclude that there exists a common subsequence of length r between $S[1..b]$, $S[b+1..c]$, and $S[c+1..n]$. $\square$

The proof presented in Lemma 8 shows that G captures all possible cubic subsequences of S . To determine if a common subsequence of length r exists, one approach is to try all possible combinations of b and c and, for each pair, compute the shortest path from vertex v_{0bc} to v_{bcn} . Unfortunately, this method requires examining $O(n^2)$ choices of b and c , and

each shortest path computation could take cubic time, leading to an overall time complexity of $O(n^5)$.

However, when it comes to finding the longest cubic subsequences of all substrings of a given sequence S , we can achieve this in $O(n^6)$ time. This may seem counterintuitive because calculating a cubic subsequence for one string takes $O(n^5)$ time, and since there are $O(n^2)$ possible substrings, we would expect to perform this calculation $O(n^2)$ times, resulting in a total time complexity of $O(n^7)$. However, the key is that these calculations share $O(n^3)$ source vertices.

Therefore, when we run a breadth-first search on a particular source vertex to calculate the distance to a target vertex, we can reuse the information calculated in that run for other substrings' longest cubic subsequence calculations. To summarize, to find the longest cubic subsequences of all substrings in a given sequence S using the alignment graph, for every $a < b < c < d \leq n$, the longest cubic subsequence for $S[a+1..d]$ is the minimum shortest distance from $v_{a,b,c}$ to $v_{b,c,d}$ across all pairs of b and c . By computing this for all pairs of a and d , where $0 \leq a < d \leq n$, we can find the longest cubic subsequences of all substrings in S .

When calculating the shortest distances from $v_{a,b,c}$ to a particular $v_{b,c,d}$ using breadth-first search, we are essentially computing the shortest distances from $v_{a,b,c}$ to $v_{b,c,d}$ for all $c < d \leq n$. Since there are $O(n^4)$ combinations for all a , b , c , and d , there are only $O(n^3)$ source vertices in these $O(n^4)$ combinations. Since each breadth-first search for vertex $v_{a,b,c}$ calculates distances for all vertices $v_{b,c,d}$ where $c < d \leq n$, we only need to run $O(n^3)$ breadth-first search queries in the alignment graph G . Since the alignment graph G contains $O(n^3)$ vertices and each breadth-first search takes $O(n^3)$ time, the total time to calculate all the necessary shortest paths is $O(n^6)$.

4.8 Conclusion

In this chapter, we investigated the longest subsequence-repeated subsequence (LSRS) problem. This problem can be viewed as a higher generalization of the LLDS problem investigated in Chapter 3. First, we demonstrated that the LSRS problem can be solved in polynomial time using a dynamic programming solution. Subsequently, we explored the constrained version of the LSRS problem (LSRS+(d)). In this variation, we require the solution to contain at least one occurrence of each letter in the alphabet. We showed that the LSRS+(d) problem is NP-hard for $d \geq 4$ using a reduction from $(3^+, 1, 2^-)$ -SAT problem. Notably, we showed that even the feasibility testing (FT(d)) version of the LSRS+(d) problem is NP-hard, implying the absence of a constant factor approximation. Furthermore, we showed that the LSRS+(3) problem is polynomially solvable using a dynamic programming solution. Notably, we used two sub-problems named longest square subsequence and longest cubic subsequence in the dynamic programming formulation. We provided a graph-based algorithm to calculate the longest square subsequence and longest cubic subsequence of all the substrings in a given sequence.

In biological contexts, tandem duplications may be accompanied by other types of rearrangements and mutations, such as deletions, substitutions, and insertions. Therefore, formulating a problem to recover the skeleton of a genome that has undergone tandem duplications along with other mutation events would be valuable. Thus, the LSRS formulation offers greater flexibility compared to the LLDS problem explored in Chapter 3. Furthermore, exploring the constrained version of LSRS is more suitable for biological contexts, as it aims to recover the original genome skeleton containing all the genes.

4.8.1 Open Problems

1. The current basic algorithm for the longest cubic subsequence problem has a time complexity of $O(n^5)$, which is not efficient enough. This is because we need to compute the longest common subsequence of three sequences for $O(n^2)$ possible splits, and each calculation takes $O(n^3)$ time. Hence, the total time complexity of the trivial algorithm is $O(n^5)$. The method used to compute longest cubic subsequence of all the substrings in S does not help reduce the time complexity of finding the longest cubic subsequence of a single string. Therefore, we need to improve the upper bounds of the longest cubic subsequence problem. Additionally, we should look into finding a better lower bound for the longest common subsequence. The current trivial lower bound is $O(n^3)$ because any algorithm that runs in $O(n^{3-\epsilon})$ would solve the longest common subsequence problem for three strings in $O(n^{3-\epsilon})$ time. We comment that recently Wang and Zhu gave a parameterized algorithm for the problem in $O(k^3 n^2)$ time, where k is a parameter at most n [108]. Therefore, when k is small, this problem can be solved in $o(n^5)$ time; but in the worst case it still runs in $O(n^5)$ time.

4.8.2 Notes

Part of this chapter has been published in 16th Annual International Conference on Combinatorial Optimization and Applications (COCO'A' 23) [67].

CHAPTER FIVE

DISTANCE MEASUREMENT FOR MITOCHONDRIAL DNA COMPARISON UNDER
 TDRL (TANDEM DUPLICATION RANDOM LOSS) MODEL FOR
 NON-PERMUTATION STRINGS

5.1 Introduction

Mitochondria are cellular organelles containing an extra-chromosomal genome that is distinct and separate from the nuclear genome. Responsible for the cell's energy production and other metabolic functions, mitochondria are inherited from the mother to offspring through the egg cell [35]. Typically, each organelle contains several identical copies of mitochondrial DNA (mtDNA) [1]. Functioning as a small circular chromosome within mitochondria, mtDNA serves as a significant molecular marker with notable characteristics, including high mutation and substitution rates, infrequent gene recombination, maternal transmission, a high copy number, and easy accessibility [52].

Due to these distinctive features, mtDNA is widely employed in phylogenetic analyses and phylogeographic studies [62, 81]. Additionally, mtDNA serves as a valuable tool for testing microevolutionary processes, exploring population genetic structure, and identifying cryptic species [113].

Typically, the length of mitochondrial DNA (mtDNA) ranges from 14,000 to 18,000 base pairs. For instance, human mtDNA is a double-stranded, circular molecule measuring 16,569 base pairs, and it encodes 37 genes, including two rRNAs, 22 tRNAs, and 13 polypeptides (ND1-6, ND4L, COI-COIII, ATPase6, ATPase8, and Cytb) [105]. Mammalian mtDNA exhibits remarkable organizational efficiency, with genes lacking introns and intergenic sequences, except for one regulatory region, being either absent or limited to a few bases. Both rRNA and tRNA molecules in mtDNA are unusually small [111].

Compared to the nuclear genome (nDNA), mtDNA is significantly smaller, making it more manageable for analysis. The reconstruction of evolutionary trees using mtDNA data is a common method for inferring relationships among living organisms. The widespread utilization of mtDNA is facilitated by the ease of obtaining new sequence data for a set of orthologous genes and the availability of numerous existing mtDNA sequences across various species. A comprehensive understanding of mitochondrial biology and its evolution is crucial for the robust application of mtDNA in inferring the evolutionary history of species and populations.

Mitochondrial genomes generally exhibit infrequent gene rearrangements, with identical or highly similar arrangements observed across numerous major animal groups and a greater number of differences between them. While most arrangements are comparable, a few cases showcase such significant differences between groups that they appear unrelated. Notably, these variations are also observed within specific major groups, such as nematodes [87, 88], mollusks [51, 112], and cnidarians [6, 7, 93].

A number of well-established evolutionary lineages are characterized by unique gene rearrangements (e.g., birds [34, 49, 65, 94], marsupials [56, 57, 89], echinoderms [3, 19, 31, 55, 101], the nematode family Rhabditidac [99], and the cnidarian group Octocorallia [93, 102]). This has encouraged optimism that mitochondrial gene arrangements may retain some signal of the ancient pattern of animal evolution. Some reports [9, 10, 101] have used the comparisons of these gene arrangements to resolve contentious evolutionary relationships.

It should be noted that successful comparisons of mitochondrial gene arrangements among plant [83], protist [46, 75, 76], and fungal [22, 90] have been published. With these successes and our greatly increased ability to sequence whole genomes, it is likely that comparisons of gene arrangements and other kinds of higher-order genomic structuring will become a common and useful source of phylogenetic information.

In [8], authors outlined a set of characteristics deemed ideal for inferring phylogeny.

Specifically, they proposed that the data set should be present in all studied taxa, unambiguously homologous, selectively neutral, sufficiently labile to generate robust signals across multiple branches, yet not so labile that subsequent changes would erode these signals. Authors in [8] mentioned that mtDNA exhibits some of these characteristics at exceptional levels. Several notable characteristics of mtDNA were highlighted in [8]. Firstly, mtDNA gene content is nearly invariant, with only minor exceptions. Secondly, the great number of potential arrangements makes convergence unlikely; therefore, gene arrangements are likely to be shared only as a result of common ancestry. Thirdly, all of the genes commonly found in animal mtDNA are homologous in the mtDNAs of plants, fungi, and/or protists, making their homology near certain.

A critique against the use of mtDNA for phylogenetic reconstruction is its maternal inheritance, which means it may not fully represent the nuclear inheritance of an organism [5]. Although mtDNA may not consistently reflect nuclear inheritance patterns, it does serve as a fundamental component of an organism's heritage and evolutionary history. It should be noted that although mtDNA may not be the only or definitive source of genetic information, it remains a valid source for understanding an organism's genetic makeup. Moreover, nuclear genes suffer from multiple technical drawbacks as well. Nuclear loci (nDNA) can be difficult to work with because of heterozygosity, low substitution rate, low copy number, and the frequent presence of paralogous loci with multiple copies [95]. Additionally, there is evidence that mtDNA and nDNA provide different inheritance patterns [95]; nevertheless, this is not necessarily uninformative because disagreement between these two data sets does not imply a false phylogenetic signal. In fact, when aiming to describe species boundaries, the incongruence observed among true trees (nDNA vs. mtDNA) should be considered and integrated [79].

Hence, to enhance the construction of phylogenetic trees, it is important to adopt an approach that incorporates both nuclear DNA (nDNA) and mitochondrial DNA

(mtDNA). Any incongruence observed between the datasets should be utilized to enrich the interpretation of the evolutionary history of the taxa under consideration.

There are several prevailing models for describing gene arrangements in mitochondrial DNA. **a)** Tandem duplication-random loss model (TDRL): part of the genome is duplicated, followed by random loss of the redundant copies; **b)** Tandem duplication and non-random loss (TDNL); **c)** Inter- or intra-mtDNA recombination; and Double replications and random loss (DRRL) model. Out of these models, the TDRL model is considered the most likely hypothesis for the gene arrangements in mtDNA. In this chapter, we introduce a distance metric under the Tandem Duplication and Random Loss (TDRL) model specifically designed for non-permutation strings, aiming to fill gaps in the exploration of distances between non-permutation strings.

5.2 Background

In [23], authors first introduced the theories surrounding the TDRL model on permutation strings. In this chapter, we try to improve this for non-permutation strings. One of the main differences in this model is that we do not consider TDs and deletions as atomic events but rather as two different events. This allows us to compare sequences of different lengths. Most of the time, after a tandem duplication, a duplicate gene is deleted from either the original or duplicated segment. However, this is not true for all the cases in a biological context. In fact, we can find scenarios where duplicate genes exist in the mitochondrial genome after TD occurs (i.e., *Boulengerula taitana* mitochondria [96], *Breviceps adspersus* mitochondrial, *Hyperolius marmoratus*, *Trichobatrachus robustus* [66], and *Aeluroscalabotes felinus* [85]). Therefore, we need a way to handle non-permutation strings in order to figure out the necessary TDs and deletions for phylogentic applications.

In our model, we adhere to the convention of defining a genome as a linear, unsigned string of integers ranging from 1 to n . We allow integers to appear more than once, essentially

transforming the string into a non-permutation. While it seems intuitively clear that the cost of a duplication should be some non-decreasing function of the length of the duplication, the precise functional form of this cost function is not evident. In this chapter, we employ two geometrically increasing cost functions for tandem duplications and deletions. We specify the cost of duplicating a segment of length k as α^k , where α is a constant parameter greater than or equal to 1, and the cost of deleting a segment of length k as β^k , where β is a constant parameter greater than or equal to 1 and $\beta \leq \alpha$. It is noteworthy that when $\alpha = 1$, the cost of duplicating a segment of length k is 1 regardless of the length of the segment.

We address the problem of determining the minimum-cost sequence of duplications and deletions needed to transform one sequence into another. In this chapter, we present a heuristic algorithm designed to minimize the overall cost of duplications and deletions required for this transformation.

Definition 12. Distance between two sequences using TDLR model under α, β parameters

Input: Sequences S and T over an alphabet Σ , along with two cost functions α^k and β^k where $\beta \leq \alpha$. We denote S as the starting sequence and T as the target sequence.

Goal: Find the minimum cost sequence of duplication and deletion steps required to transform the sequence S into T . In other words, suppose the j^{th} duplication step copies a segment of length k_j then cost of tandem duplications c_{td} is $\sum_{j=1}^n \alpha^{k_j}$ where n is the total number of duplication steps. Similarly, the cost of deletions c_d is $\sum_{i=1}^m \beta^{k_i}$. We aim to find the n such that $\sum_{j=1}^n \alpha^{k_j}$ is smallest and m such that $\sum_{i=1}^m \beta^{k_i}$ is the smallest.

By ensuring that $\beta \leq \alpha$, we establish that the cost of a deletion operation is either less than or equal to the cost of a duplication operation for a segment of the same length.

In this chapter, we present a heuristic algorithm designed to minimize the costs associated with tandem duplication and deletion operations. Initially, we delve into the

distinctive property of tandem duplication and deletion operations, exploring how we can leverage it to formulate a straightforward heuristic algorithm. This algorithm aims to identify a sequence of duplications and deletions that transforms one given sequence into another. Additionally, it is crucial to note that deletions alone cannot rectify the order of characters in the starting sequence. For instance, consider the sequences $S = 11234$ and $T = 432112$; in this case, a deletion operation alone cannot rearrange the order of occurrences of 1 and 4. Achieving this requires performing tandem duplication operations.

Next, let us introduce a special property of a tandem duplication and a deletion operation.

Lemma 9. *Given a string $S = s_1s_2s_3 \dots s_n$, copy of the segment $s_i s_{i+1} \dots s_j$ to the end of the S can be mimicked by a tandem duplication and a deletion.*

Proof. We can use the tandem duplication on the segment $s_i s_{i+1} \dots s_n$ and then delete part of the second copy.

$$S = s_1s_2 \dots s_i \dots s_j \dots s_n ,$$

$$S_1 = s_1s_2 \dots s_i \dots s_j \dots s_n \cdot s_i \dots s_j \underline{s_{j+1} \dots s_n} ,$$

$$S_2 = s_1s_2 \dots s_i \dots s_j \dots s_n s_i \dots s_j .$$

□

By leveraging this property, we can formulate a straightforward algorithm to identify a sequence involving tandem duplications and deletions. Essentially, the process involves identifying partitions within the target string using substrings from the original sequence. Subsequently, each partition can be appended to the end of the original string to construct the starting string.

We begin by presenting a finding from [40]. In this study, the authors proposed an approximation algorithm for two edit distance problems involving segment insertions and

deletions to transform a starting string into a target string. The first problem permits the copying of any segment from the starting sequence and inserting it anywhere, while the second problem allows the copying of a segment from the starting sequence or any intermediate sequence to create the target string. This result is particularly significant in our context because we can simulate a segment insert operation on a string using tandem duplication and deletion. Consequently, we can utilize the algorithm outlined in [40] to identify the necessary segments for insertion in order to transform the original string into the target string, employing tandem duplication and deletion to mimic the insertion operation.

The authors in [40] utilized a slightly modified Lempel-Ziv-77 algorithm that greedily partitions a string S into “phrases” (LZ phrases) from left to right, ensuring that each phrase $S[i : j]$ for $j > i$ must occur previously in S . To ensure the use of substrings from the original or intermediate strings, we can use $S\#T$ for the modified Lempel-Ziv compression, where $\#$ is a character not present in either S or T .

Let’s take a look at the following example:

$$S = 1234512532 \ ,$$

$$T = 12345345122532 \ ,$$

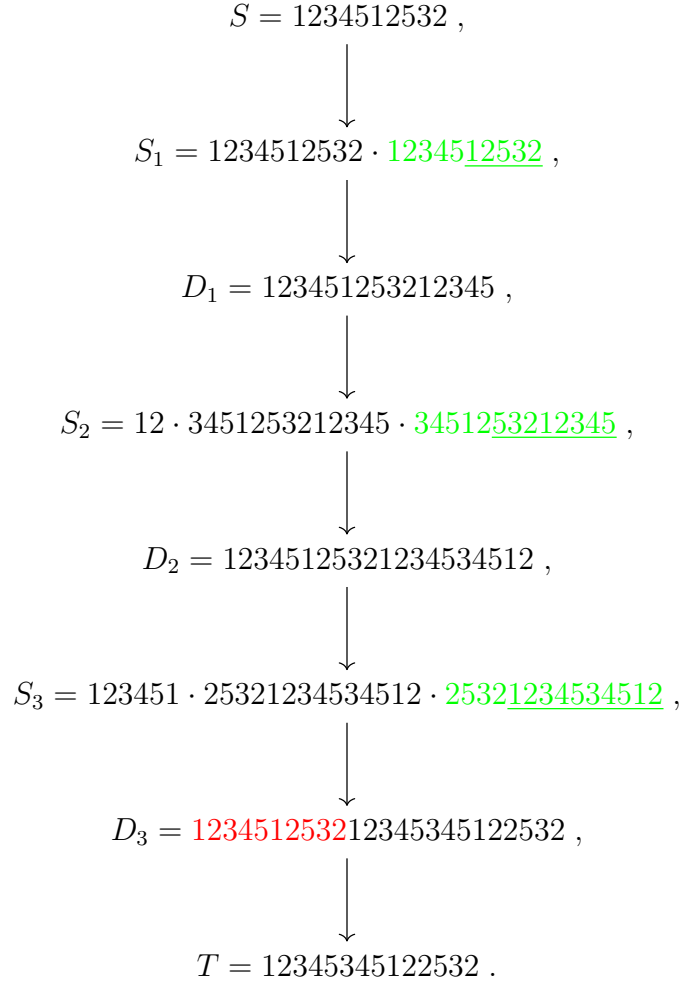
$$LZ(S) = \{1, 2, 3, 4, 5, 12, 5, 3, 2\} \ ,$$

$$LZ(S\#T) = \{1, 2, 3, 4, 5, 12, 5, 3, 2, 12345, 34512, 2532\}, \text{ and}$$

$$LZ(S\#T) - LZ(S) = \{12345, 34512, 2532\} \ .$$

In the example provided, we consider two strings, denoted as S and T , with S serving as the initial string. Using a slightly modified version of the $LZ-77$ algorithm as outlined in [40], we process the string $S\#T$ from left to right, identifying the LZ phrases. Subsequently, we

eliminate the LZ phrases obtained from processing the source string S alone. The remaining phrases constitute the decomposition of string T using substrings from S . Finally, each of these retained LZ phrases can be appended to the starting string through the application of tandem duplication and deletion operations.



The green color segments represent the duplicated copy for each tandem duplication, while the underlined segments represent the segment that is deleted right after the duplication operation. The red segment at the end symbolizes the deletion of the starting sequence from the intermediate string, which leads to the creation of the final target string.

This idea enables us to generate an approximation of tandem duplications and deletions.

The count of tandem duplications corresponds to the number of LZ phrases produced by the compression algorithm, while the number of deletions is the sum of LZ phrases plus one (accounting for the deletion of the starting sequence at the end). This approach provides a heuristic algorithm for estimating the distance using the TDRL model, potentially addressing challenges associated with duplicate genes.

Subsequently, we will explore improvements to refine this approach for a more accurate approximation of the TD cost and deletion cost. We will examine different parameter configurations for α and β . Initially, let's examine the parameter set where $\alpha = 1$ and $0 \leq \beta \leq \alpha$. In this configuration, the TD cost of a segment of length k is 1, irrespective of the length k . If $\beta = 0$, we can treat duplication and gene loss as a unified atomic event. If $\beta = 1$, we distinguish deletions as a distinct event, although with less than or equal significance compared to the duplication event.

5.3 Heuristic Algorithms

5.3.1 Heuristic Algorithm for $\alpha = 1, \beta \leq \alpha$

The idea behind this heuristic algorithm is to initially prioritize minimizing the number of tandem duplications required to transform the starting string into the target string. Subsequently, we aim to minimize the number of deletions needed based on the tandem duplications.

Theorem 13. *Given a parameter $\alpha = 1$ and two strings S and T where S is the original string, the minimum number of tandem duplication operations ($TD(S, T)$) needed to convert S to T' such that T is a subsequence in T' can be calculated as follows. Let us define $\rho_S(T)$ as the set of maximal increasing substrings in T relative to S and γ as follows:*

$$\gamma = \max_{c \in T} \left(\frac{OCC_T(c)}{OCC_S(c)} \right),$$

where $OCC_T(c)$ calculates the frequency of character c in sequence T . Consequently, γ represents the proportion of character c in the target string, specifically referring to the character with the highest ratio of occurrences in the target sequence compared to the initial sequence.

Then we define,

$$S' = S^{2^{\lceil \log_2(\gamma) \rceil}}, \text{ and}$$

$$TD(S, T) = \lceil \log_2(|\rho_{S'}(T)|) \rceil + \lceil \log_2(\gamma) \rceil .$$

For example,

$$S = 1234512532 ,$$

$$T = 1234534515253244 ,$$

$$\gamma = \max_{c \in T} \left(\frac{OCC_T(c)}{OCC_S(c)} \right) ,$$

$$\lceil \log_2(\gamma) \rceil = \lceil \log_2 \left(\frac{OCC_T(4)}{OCC_S(4)} \right) \rceil = \lceil \log_2 \left(\frac{4}{1} \right) \rceil = 2 , c = 4 ,$$

$$S' = ((1234512532) \cdot (1234512532)) \cdot (1234512532 \cdot 1234512532) ,$$

$$\rho_{S'}(T) = \{123453451525234, 4\}, \lceil \log_2(|\rho_{S'}(T)|) \rceil = 1 , \text{ and}$$

$$TD(S, T) = \lceil \log_2(|\rho_{S'}(T)|) \rceil + \lceil \log_2(\gamma) \rceil = \lceil \log_2(2) \rceil + \lceil \log_2(4) \rceil = 1 + 2 .$$

Then,

$$T' = ((\textcolor{red}{12345}125\textcolor{red}{32}) \cdot (123\textcolor{red}{45}125\textcolor{red}{32})) \cdot (1234\textcolor{red}{5}1\textcolor{red}{25}3\textcolor{red}{2} \cdot 123\textcolor{red}{4}512532) \cdot$$

$$(123\textcolor{red}{4}512532 \cdot 1234512532 \cdot 1234512532 \cdot 1234512532) .$$

Proof. When $\alpha = 1$, duplicating the entire string in each tandem duplication operation is

always optimal. The cost function being 1^k ensures that the cost of duplication remains independent of the size of the duplicated string. As deletion operations do not enable the correction of order changes between the starting and target strings, tandem duplications must be employed to address order changes.

First, we have to note that the $\lceil \log_2(\gamma) \rceil$ number of TDs are required to compensate for the extra characters in T . The reason for this minimum requirement becomes evident when considering the need to duplicate the extra characters. Since each TD allows to double the current number of occurrences of each character in the string, using WGD (*Whole Genome Duplication*), we can double the number of occurrences of each character at each step. Given that γ represents the maximum ratio of the occurrence of any character in T to its occurrence in S , $\lceil \log_2(\gamma) \rceil$ number of TDs would be enough to duplicate all the necessary number of occurrences for every character in T .

Once we perform the initial $\lceil \log_2(\gamma) \rceil$ WGDs, we transform the S into S' such that it possesses the necessary occurrences of each character to align with their respective counterparts. To determine the minimum number of TDs required to transform S' into T' , we associate the steps of radix sort with each duplication step. Identifying the maximal increasing substrings of T compared to S' allows us to label each character in the i^{th} maximal increasing substrings using the binary representation of i . It's important to note that in S' there are enough occurrences of each character present in T to establish a mapping between each character. Utilizing these labels, we can associate characters mapped with their respective counterparts in T using binary representation. While it is not mandatory for all characters in S' to be mapped to characters in T , it is essential for all characters in T to be mapped to characters in S' .

Proceeding with the characters in S' that are mapped in the order of appearance in S' and considering their assigned binary values in the same order as they appear, we can create a vector v containing these binary values. Then, we can utilize radix sort to sort these

binary values in v . We can map the k^{th} step of the radix sort, which organizes the k^{th} least significant bit of elements in v - to the k^{th} TD step that transforms S' into T' and vice versa. By mapping these steps to the steps of radix sort, we can establish that $\lceil \log_2(|\rho_{S'}(T)|) \rceil$ is the minimum number of TDs required to transform S' to T' such that T is a subsequence of T' .

For the above part, let us say $d(v)$ represents the number of steps needed to sort the vector v . To show that $d(v) = \lceil \log_2(|\rho_{S'}(T)|) \rceil$, we show that $d(v) \leq \lceil \log_2(|\rho_{S'}(T)|) \rceil$ and $d(v) \geq \lceil \log_2(|\rho_{S'}(T)|) \rceil$.

For the part of $d(v) \leq \lceil \log_2(|\rho_{S'}(T)|) \rceil$, we use induction to claim that after the first k steps, the maximal increasing substrings are sorted with respect to the k least significant bits of their label. When $k = 1$, this is trivial. For the inductive hypothesis, we assume that after k steps, the maximal increasing substrings are sorted with respect to the k least significant bits. Using the fact that radix sort is a stable sort, we can say that for the inductive step, elements in a maximal increasing substring are sorted with respect to the $k + 1$ least significant bit while maintaining the order of the k^{th} significant bit (using the stability property). Since the number of bits in binary representation is $\lceil \log_2(|\rho_{S'}(T)|) \rceil$, after $\lceil \log_2(|\rho_{S'}(T)|) \rceil$ steps, elements in the vector v are sorted with respect to the binary labels.

Then, to show that $d(v) \geq \lceil \log_2(|\rho_{S'}(T)|) \rceil$, we use the idea that, at each duplication step, we create at most two maximal increasing substrings from each maximal increasing substring of the previous string. The starting string S' has one maximal increasing substring, and T' would have $|\rho_{S'}(T)|$ substrings. Therefore, the number of duplication steps required to create T' is $\lceil \log_2(|\rho_{S'}(T)|) \rceil$.

This concludes that the minimum number of TDs required to fix the order changes is $\lceil \log_2(|\rho_{S'}(T)|) \rceil + \lceil \log_2(\gamma) \rceil$. □

Now that we have established a method to create a string T' such that T is a

subsequence, we can delete unnecessary blocks of strings from T' to create T . Let us now explore how we can extend the former Lempel-Ziv algorithm to find a solution that outputs the number of deletions as well.

Basically, we use the modified Lempel-Ziv algorithm introduced in [40] to find a partition of the target sequence T relative to T' . We calculate the $LZ(T' \# T) - LZ(T')$ which allows us to denote T using substrings of T' . Note that this compression algorithm tries to find the longest possible substring that exists in T' ; therefore, it minimizes the number of partitions of T . Now, we can count the number of deletions required to transform T' to T . Suppose the compression algorithm outputs k LZ phrases, then at most we need $k + 1$ deletions of blocks from T' to create T .

Next, we will prove a slight improvement in the notation of the maximal increasing substring calculation.

Theorem 14. *The number of minimum tandem duplications is in fact equivalent to the maximal increasing substrings of T compared to S , i.e.,*

$$TD(S, T) = \lceil \log_2(|\rho_{S'}(T)|) \rceil + \lceil \log_2(\gamma) \rceil = \lceil \log_2(|\rho_S(T)|) \rceil .$$

Proof. While not immediately evident, we can calculate the minimum number of tandem duplications needed to convert S to T' under parameters $\alpha = 1$ where T is a subsequence of T' for non-permutation strings, using $\lceil \log_2(|\rho_S(T)|) \rceil$, i.e.,

$$TD(S, T) = \lceil \log_2(|\rho_{S'}(T)|) \rceil + \lceil \log_2(\gamma) \rceil .$$

Let's consider the string S' , which consists of $2^{\lceil \log_2(\gamma) \rceil}$ copies of S . Therefore, we can write $\rho_{S'}(T)$ as $\frac{\rho_S(T)}{2^{\lceil \log_2(\gamma) \rceil}}$.

$$\begin{aligned}
TD(S, T) &= \lceil \log_2(|\frac{\rho_S(T)}{2^{\lceil \log_2(\gamma) \rceil}}|) \rceil + \lceil \log_2(\gamma) \rceil \\
&= \lceil \log_2(|\rho_S(T)|) \rceil - \lceil \log_2(2^{\lceil \log_2(\gamma) \rceil}) \rceil + \lceil \log_2(\gamma) \rceil \\
&= \lceil \log_2(|\rho_S(T)|) \rceil - \lceil \lceil \log_2(\gamma) \rceil \cdot \log_2(2) \rceil + \lceil \log_2(\gamma) \rceil \\
&= \lceil \log_2(|\rho_S(T)|) \rceil - \lceil \log_2(\gamma) \rceil + \lceil \log_2(\gamma) \rceil \\
&= \lceil \log_2(|\rho_S(T)|) \rceil
\end{aligned}$$

□

Lemma 10. *Given two non-permutation strings, the cost of transforming S to T under parameters $\alpha = 1$ and $\beta \leq (\alpha = 1)$, using tandem duplications and deletion operations can be calculated using Theorems 13,14 as follows:*

$$d(S, T) = (\lceil \log_2(|\rho_S(T)|) \rceil) + (LZ(T' \# T) - LZ(T') + 1) \cdot \beta.$$

Proof. Since the number of TDs ($\lceil \log_2(|\rho_S(T)|) \rceil$), and number of deletions are ($LZ(T' \# T) - LZ(T') + 1$), the total cost $d(S, T)$ would be,

$$d(S, T) = (\lceil \log_2(|\rho_S(T)|) \rceil) \cdot \alpha + (LZ(T' \# T) - LZ(T') + 1) \cdot \beta.$$

Since $\alpha = 1$,

$$d(S, T) = (\lceil \log_2(|\rho_S(T)|) \rceil) + (LZ(T' \# T) - LZ(T') + 1) \cdot \beta.$$

□

5.3.2 Heuristic Algorithm for $\alpha \geq 2$

Now, let's look into the heuristic algorithm designed for cost functions where $\alpha \geq 2$. It's important to note that duplicating the entire sequence may no longer be optimal for rectifying changes in the order of characters. This is due to the fact that the Tandem Duplication (TD) cost is α^k , and the cost increases exponentially with the size of the duplicated sequence. Consequently, utilizing whole genome duplication (WGD) as an approximation for TDs is no longer a favorable heuristic choice. Our proposed approach involves converting both S and T into stabilized permutation strings and then determining the bubble sort distance between them to approximate the number of TDs and deletions. To clarify, stabilized permutations are introduced to handle non-permutations. By converting these sequences into stabilized permutation sequences, we ensure equal length and an equal number of occurrences for each character.

The distance calculation between the transformed sequences involves two components: the number of operations required to convert the strings to stabilized permutations and the bubble sort distance between the stabilized permutation sequences. This combined approach provides a more effective means of estimating the overall distance. The significance of the bubble sort distance in this context lies in its ability to quantify the number of swaps of adjacent characters required to sort a permutation into the identity permutation. This is crucial because the act of swapping two adjacent characters can be simulated by employing one length-2 Tandem Duplication (TD) and two length-1 deletions.

Definition 13. *Given two sequences S, T that are non-permutation sequences, the corresponding stabilized permutation for S, T would be two sequences S', T' such that they are of equal length and each character in S', T' has equal occurrences in both sequences.*

For example, given two sequences $S = 1132145$ and $T = 13335245$, one such possible stabilized permutation sequences would be,

$$S' = 1133321455 ,$$

$$T' = 1113335245 .$$

In the subsection 5.3.2.1 we will provide an algorithm to transform two sequences into stabilized permutations.

Lemma 11. *Given a string $S = s_1 s_2 s_3 \dots s_i s_{i+1} \dots s_n$, we can swap the order of the i^{th} and $(i+1)^{th}$ characters in the string S using one length 2 TD and two length 1 deletions.*

Proof. We can easily do this by performing a tandem duplication on the substring $s_i s_{i+1}$ and then deleting the first occurrence of s_i and second occurrence of s_{i+1} .

$$S = s_1 s_2 s_3 \dots s_i s_{i+1} \dots s_n ,$$

$$S_1 = s_1 s_2 s_3 \dots s_i s_{i+1} \cdot s_i s_{i+1} \dots s_n ,$$

$$S_1 = s_1 s_2 s_3 \dots s_i s_{i+1} \cdot s_i s_{i+1} \dots s_n , \text{ and}$$

$$S_2 = s_1 s_2 s_3 \dots s_{i+1} s_i \dots s_n .$$

The green color segment represents the substring that is duplicated while red color segments represents the deletions. Final blue color segment represents the swapped characters. □

Corollary 4. *Given a sequence S and parameters α, β with cost functions α^k, β^k , we can swap two adjacent characters in S with one length 2 TD and two length 1 deletions that costs $\alpha^2 + 2 \cdot \beta$.*

α^2 is the cost for the duplication and $2 \cdot \beta$ is the cost for the deletions.

$c \in \Sigma$	$OCC_S(c)$	$OCC_T(c)$
1	3	3
2	5	4
3	2	2
4	1	3
5	1	4

Table 5.1: Occurrences of each character $c \in \Sigma$ in S and T .

5.3.2.1 Algorithm to Create Stabilized Permutations Given Two Strings S, T . Basically, we are given two non-permutation strings S and T defined on the alphabet Σ where each string contains at least one occurrence of each character $c \in \Sigma$. Then we go through each character $c \in \Sigma$ and check the relevant occurrences of c in S and T . If $OCC_S(c) = OCC_T(c)$, we leave it as it is. If $OCC_S(c) \neq OCC_T(c)$, then we define $\delta_c = |OCC_S(c) - OCC_T(c)|$, and we add δ_c occurrences of character c to the string that contains fewer occurrences. We add the extra characters next to the last occurrence of character c in those strings. For example,

$$S = 122321124325 ,$$

$$T = 5452321243215415 ,$$

$$S' = 12232112444325555 ,$$

$$T' = 54523212432215415 .$$

In the preceding example and as illustrated in Table 5.1, it is evident that character 4 occurs only once in S compared to its three occurrences in T . Consequently, we appended two additional instances of the character 4 after its last occurrence in S . Similarly, in the case

of T , which exhibits only four occurrences of character 2, we introduced an extra instance of 2 adjacent to the last occurrence of 2. These newly added characters are highlighted in green. It is worth noting that both S' and T' are of equal length, and each character maintains an identical frequency in both S and T .

Lemma 12. *The cost C for transforming non-permutation strings S and T to stabilized permutations S', T' can be calculated as follows:*

$$C = d(S, S') + d^*(T, T') ,$$

$$\delta_c = \begin{cases} 0 & \text{if } OCC_S(c) \geq OCC_T(c) \\ OCC_T(c) - OCC_S(c) & OCC_S(c) < OCC_T(c) , \end{cases}$$

$$d(S, S') = \sum_{c \in S} \delta_c \cdot \alpha^1 ,$$

$$\delta_c = \begin{cases} 0 & \text{if } OCC_T(c) \geq OCC_S(c) \\ OCC_S(c) - OCC_T(c) & OCC_T(c) < OCC_S(c) , \end{cases}$$

$$d^*(T, T') = \sum_{c \in T} \delta_c \cdot \beta^1 .$$

Essentially, the process involves determining the additional characters inserted into the string S and evaluating the cost of duplicating these extra characters using the cost of a length-1 tandem duplication. Conversely, in the case of the string T , the approach is reversed, as the transformation is from T' to T . Hence, the focus is on identifying the extra characters introduced into T' and calculating the cost of deletion for these characters. For example,

$$S = 122321124325 ,$$

$$T = 5452321243215415 ,$$

$$S' = 12232112444325555 ,$$

$$T' = 5452321243215415 ,$$

$$d(S, S') = 5 \cdot \alpha^1 ,$$

$$d^*(T, T') = 1 \cdot \beta^1 , \text{ and}$$

$$C = 5\alpha + \beta .$$

Next, we will look into the heuristics algorithm involving bubble sort distance for the $\alpha \geq 2$ parameter. We will prove why it is not a good choice for using WGD for this parameter in following theorem.

Theorem 15. *For two stabilized permutations with a parameter $\alpha \geq 2$, if provided with a sequence of arbitrary length tandem duplications (TDs) and deletions of any size, it is always feasible to formulate a sequence utilizing only TDs of length 2 and deletions of length 1 that incur no greater cost than the original sequence.*

Proof. Since these are stabilized permutations, we can replace each character with a unique character and create two proper permutations. Suppose we transform them into proper permutations using any mapping that we can find.

The idea for this proof is to show that the minimum cost of any k -length duplication and deletions associated with it can be simulated for lesser cost using a set of length-2 duplications and a set of length-1 deletion operations. Suppose we start with S' and perform a tandem duplication of k length followed by k character deletions (it could be of one whole block of length k or up to k different length 1 blocks individually) to create the string S'' . Note that since these are now permutations, we need to drop k characters after we do a k length duplication. Then, we can observe that, at most, there could be two maximal

increasing substrings in S'' compared to S' . Suppose P consists of the characters in the first maximal increasing substrings, and Q is the set of characters in the second maximal increasing substrings. Then, to calculate maximal bubble sort distance introduced between S' and S'' , we need to consider all the ordered pairs of P and Q .

Note that any two characters in P will not contribute to the order difference since they belong to one maximal increasing substring (respectively for Q as well). Only pairs of elements between P and Q can contribute to bubble sort distance. For example, in any (i, j) pair where $i \in Q$ and $j \in P$, if i appears before j in S'' , it will contribute to bubble sort distance since they are out of order. Therefore, the maximum bubble sort distance that can be introduced by a k length duplication is $|P| \cdot |Q|$. We can maximize this value by making $|P| = |Q| = \frac{k}{2}$ for even k and $|P| = \frac{k+1}{2}$ and $|Q| = \frac{k-1}{2}$ for odd k . Therefore, the maximum bubble sort distance is either $\frac{k^2}{4}$ or $\frac{(k+1)(k-1)}{4}$ for even and odd k , respectively.

Note that cost C for length k duplication and subsequent deletions will always lie in between $\alpha^k + \beta^k$ and $\alpha^k + k \cdot \beta$. The maximum cost will occur when we delete k -length block at once and the minimum cost will occur when we delete k length 1 characters individually.

$$\alpha^k + k \cdot \beta \leq C \leq \alpha^k + \beta^k .$$

Now we can consider the functions $f(k, \alpha, \beta) = \alpha^k + k \cdot \beta - (\frac{k^2}{4}\alpha^2 + \frac{k^2}{4} \cdot 2 \cdot \beta^1)$ for even k and $f(k, \alpha, \beta) = \alpha^k + k \cdot \beta - (\frac{(k-1)(k+1)}{4}\alpha^2 + \frac{(k-1)(k+1)}{4} \cdot 2 \cdot \beta^1)$ for odd k which represents the cost difference between choosing length k TDs vs. length 2 TDs (using swapping adjacent characters to sort the string). Note that under this model, we choose $\beta \leq \alpha$. We added β^k to represent k deletions that occur after k -duplication. Since for each adjacency swap we need to do two length 1 deletions, we added $2 \cdot \beta \frac{k^2}{4}$ and $2 \cdot \beta \frac{(k-1)(k+1)}{4}$ to the cost function of the length 2 duplication. When plotting for integers $k \geq 2$ and $\alpha \geq 2, \beta \leq \alpha$, we will see that the minimum value is attained at $k = 2$. This proves that, under the $\alpha \geq 2$ parameter, it

is always better to use adjacency swaps for transforming the starting sequence to the target sequence. $\square$

Theorem 16. *Given a parameter $\alpha \geq 2$ and two strings S and T where S is the original string, we can approximate the number of tandem duplication operations needed to convert S to T by calculating the bubble sort distance between S' and T' where S', T' are stabilized permutations,*

$$d(S, T) = d(S, S') + d_{BS}(S', T') \cdot (\alpha^2 + 2 \cdot \beta) + d^*(T, T') .$$

Note that $d(S, S')$ is the cost for transforming the starting sequence into a stabilized permutation, and $d^*(T, T')$ is the cost of transforming the stabilized permutation back to the target sequence (Lemma 12). $d_{BS}(S', T')$ is the bubble sort distance between S' and T' .

However, the distance metric under $\alpha = 1$ is not symmetric, while the distance metric for $\alpha \geq 2$ is symmetric. It is extremely difficult to create a symmetric distance metric for non-permutation strings under the TDRL model. Therefore, in order to make the first distance metrics symmetric, we offer the following resolution:.

Definition 14. *We define symmetric distance measurement $D(S, T)$ between two sequences under the TDRL model and with $\alpha = 1, \beta \leq \alpha$ parameters as follows,*

$$D(S, T) = \frac{d(S, T) + d(T, S)}{2} .$$

We take the average distance from S to T and T to S under these models. This metric can be used as a symmetric measurement for phylogenetic tree construction.

5.3.3 Experiments

5.3.3.1 Experiments on Real-World Data Sets. We implemented the heuristic algorithms in Java. These algorithms were run on a desktop with an Intel (R) Xeon (R) CPU, 3.70 GHz, and 64 GB of RAM. These algorithms were subjected to testing using 43 distinct species' mitochondrial gene order data, compared against the ancestral gene order of vertebrates as detailed in [85]. The sequences and gene annotations for these 43 species were obtained from the NCBI (<https://www.ncbi.nlm.nih.gov/>) database. The vertebrate ancestral gene order utilized for testing was sourced from the same reference [85].

For the $\alpha = 2$ heuristic algorithm, we calculated the bubble sort distance by modifying the merge sort algorithm. Specifically, during the merge step of the algorithm, we computed the inversions necessary to sort each sub-array. The inversions needed to calculate each sub-array can be computed in linear time with respect to the size of the sub-array. This allowed for the calculation of the bubble sort distance in $O(n \log n)$ time complexity. On the other hand, the $\alpha = 1$ heuristic algorithm operates in $O(n^2)$ time, as it necessitates the identification of maximal increasing substrings.

	$d(S, T)$				$d(T, S)$				
Species	MIS	TD	Dels	$d(S, T)$	MIS	TD	Dels	$d(T, S)$	$D(S, T)$
Clinocottus Analis	4	2	7	10	2	1	5	7	8.5
Boulengerula Taitana	2	1	3	5	1	0	3	4	4.5
Lampris Guttatus	1	0	3	4	-	-	-	0	4
Hapalogenys Analis	1	0	3	4	-	-	-	0	4
Trichiurus Japonicus	1	0	3	4	-	-	-	0	4
Rudarius Ercodes	1	0	3	4	2	1	4	6	5
Dallia Pectoralis	2	1	4	6	2	1	4	6	6
Normichthys Operosus	2	1	5	7	2	1	5	7	7

Ambassis	2	1	5	7	2	1	5	7	7
Gymnocephalus									
Aulorhynchus Flavidae	2	1	5	7	2	1	5	7	7
Trachyrhynchus Murrayi	2	1	5	7	2	1	5	7	7
Squalogadus Modificatus	2	1	5	7	2	1	5	7	7
Cetomurus Globiceps	2	1	5	7	2	1	5	7	7
Coelorhynchus Kishinouyei	2	1	5	7	2	1	5	7	7
Ventrifossa Garmani	2	1	5	7	2	1	5	7	7
Uroplatus Fimbriatus	1	0	3	4	-	-	-	0	4
Kurtus Gulliveri	2	1	5	7	2	1	5	7	7
Ponticola Kessleri	2	1	5	7	2	1	5	7	7
Chamaeleo Arabicus	3	2	7	10	3	2	7	10	10
Xenagama Taylora	3	2	7	10	3	2	7	10	10
Uromastix Benti	2	1	5	7	2	1	5	7	7
Phrynocephalus Putjatai	3	2	7	10	3	2	7	10	10
Pagothenia Borchgrevinki	2	1	5	7	2	1	5	7	7
Rhineura Floridana	2	1	5	7	2	1	5	7	7
Coloconger Cadenati	2	1	5	7	2	1	5	7	7
Ariosoma Shiroanago	2	1	5	7	2	1	5	7	7
Paraconger Notialis	2	1	5	7	2	1	5	7	7

Conger Japonicus	2	1	6	8	3	2	6	9	8.5
Heteroconger Hassi	2	1	5	7	2	1	5	7	7
Derichthys Serpentinus	2	1	5	7	2	1	5	7	7
Nessorhamphus Ingolfianus	2	1	5	7	2	1	5	7	7
Cynoponticus Ferox	2	1	5	7	2	1	5	7	7
Muraenesox Bagio	2	1	5	7				0	7
Facciolella Oxyrhyncha	2	1	5	7	2	1	5	7	7
Hoplunnis Punctata	2	1	5	7	2	1	5	7	7
Nettastoma Parviceps	2	1	5	7	2	1	5	7	7
Ophisurus Macrorhyn- chos	2	1	6	8	2	1	5	7	7.5
Chaenocephalus Acer- atus	2	1	5	7	2	1	5	7	7
Chionodraco Hamatus	2	1	5	7	2	1	5	7	7
Notothenia Coriiceps	2	1	5	7	2	1	5	7	7
Aneides Flavipuncta- tus	2	1	5	7	2	1	5	7	7
Stereochilus Margina- tus	2	1	5	7	2	1	5	7	7
Siphonops Annulatus	2	1	7	9	3	2	7	10	9.5

Table 5.2: The table presents the results obtained by running the heuristics algorithm with parameters $\alpha = 1$ and $\beta = 1$. It displays the distances from the vertebrate gene order to 43 different species, calculated using the heuristics algorithm.

Species	$d_{BS}(S', T')$	$d(S, S')$	$d^*(T', T)$	Distance
Clinocottus Analis	67	4	0	406
Boulengerula Taitana	0	2	0	2
Lampris Guttatus	0	0	1	1
Haplogenyis Analis	0	0	1	1
Trichiurus Japonicus	0	0	1	1
Rudarius Ercoodes	25	0	1	151
Dallia Pectoralis	2	0	0	12
Normichthys Operosus	1	0	0	6
Ambassis Gymnocephalus	1	0	0	6
Aulorhynchus Flavids	1	0	0	6
Trachyrincus Murrayi	2	0	0	12
Squalogadus Modificatus	2	0	0	12
Cetonurus Globiceps	3	0	0	18
Coelorinchus Kishinouyei	3	0	0	18
Ventrifossa Garmani	3	0	0	18
Uroplatus Fimbriatus	0	0	1	1
Kurtus Gulliveri	1	0	0	6
Ponticola Kessleri	1	0	0	6
Chamaeleo Arabicus	2	0	0	12
Xenagama Taylora	2	0	0	12
Uromastyx Benti	1	0	0	6
Phrynocephalus Putjatai	37	0	0	222
Pagothenia Borchgrevinki	6	0	0	36
Rhineura Floridana	6	0	0	36

Coloconger Cadenati	4	0	0	24
Ariosoma Shiroanago	4	0	0	24
Paraconger Notialis	4	0	0	24
Conger Japonicus	7	0	0	42
Heteroconger Hassi	4	0	0	24
Derichthys Serpentinus	4	0	0	24
Nessorhamphus Ingolfianus	4	0	0	24
Cynoponticus Ferox	4	0	0	24
Muraenesox Bagio	4	0	1	25
Facciolella Oxyrhyncha	4	0	0	24
Hoplunnis Punctata	4	0	0	24
Nettastoma Parviceps	4	0	0	24
Ophisurus Macrorhynchos	7	2	0	44
Chaenocephalus Aceratus	4	0	0	24
Chionodraco Hamatus	4	0	0	24
Notothenia Coriiceps	4	0	0	24
Aneides Flavipunctatus	4	0	0	24
Stereochilus Marginatus	4	0	0	24
Siphonops Annulatus	3	0	0	18

Table 5.3: Heuristics distance between vertebrate gene order and 44 species using the heuristics distance method, where $\alpha = 2$ and $\beta = 1$ parameters. The $d(S, S')$ column indicates the cost of transforming S to S' , while the $d^*(T', T)$ column indicates the cost of transforming T' to T .

5.4 Ancestral Gene Order Reconstruction Using The Feedback Arc Set Problem for Non-Permutation Strings.

5.4.1 Reducing Ancestral Gene Order Reconstruction Into a Feedback Arc Set Problem

For the construction of the ancestral gene order derived from two gene orders, we adopt a slightly modified graph-based solution outlined in [23]. Let's revisit Theorem 14, wherein we established that the minimum number of tandem duplications required to transform S into T is $\log_2(\rho_S(T))$. This can also be represented in terms of the number of descents in a graph. Consider non-permutation strings $S = s_1 s_2 \dots s_j \dots s_m$ and $T = t_1 t_2 t_3 \dots t_i \dots t_n$. Let us construct a graph G for S , where each character $c \in S$ corresponds to a vertex. For each (s_i, s_{i+1}) with $i \in [m - 1]$, we introduce a red edge e_i , forming a chain-like structure with $m - 1$ edges (refer to Figure 5.1 for an example). Next, we encode T into the graph G . To achieve this, it is imperative to establish a mapping between characters in S and T . This can be achieved by finding maximal increasing substrings of T relative to S and recording the characters mapped by the increasing substring. Subsequently, for each (t_i, t_{i+1}) with $i \in [n - 1]$, we add a blue edge to the graph G . Given the existence of a mapping for each character, we can precisely determine the vertices corresponding to the edge (t_i, t_{i+1}) . Let us call this modified graph $G_{S,T}$. A descent in T relative to S occurs when an edge $e_i = (t_i, t_{i+1})$ signifies that t_{i+1} precedes t_i in S (as per the established mapping). For example, let us consider the sequences $S = 12234$ and $T = 2133442$ in Figure 5.1.

If we look at Figure 5.1, we can observe that by using maximal increasing substrings, each character in T can be mapped to some character in S . The $G_{S,T}$ graph contains distinct red and blue edges for each corresponding adjacency in S and T . The descents in the graph $G_{S,T}$ in Figure 5.1 are edges $(2, 1), (3, 3), (4, 4), (4, 2)$ (relative to S). We can see that each of the mapped characters in these descents (edges) violates the order in S . Using this idea of descents, we can see that the number of descents relative to S (denoted as k) in this graph is

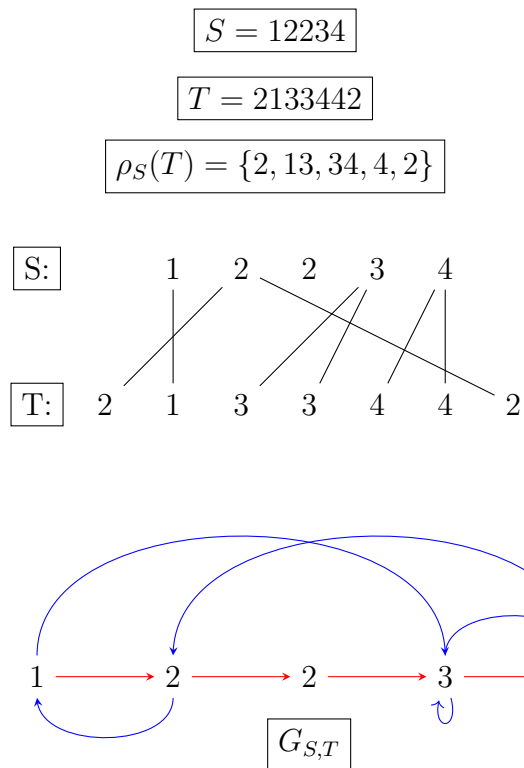


Figure 5.1: Example for the mapping between two strings S and T . Note that there can be some characters in S that are not mapped or mapped to more than one character in T .

$k = \rho_S(T) - 1$. Consequently, $TD(S, T) = \lceil \log_2(k + 1) \rceil$, where k is the number of descents in the graph.

Suppose we can find a sequence P such that the descents of S and T in the graph $G_{S,T}$ are k_1 and k_2 relative to P . If P minimizes the value $\lceil \log_2(k_1 + 1) \rceil + \lceil \log_2(k_2 + 1) \rceil$, we can use P as a possible ancestral gene order for S and T .

Therefore, our goal is to find a pair (b, r) that minimizes the $\lceil \log_2(k_1 + 1) \rceil + \lceil \log_2(k_2 + 1) \rceil$. Hence, we aim to find the set of blue edges and the set of red edges that transform the graph $G_{S,T}$ into a directed acyclic graph (DAG). Then, we can pick any permutation of vertices that is consistent with the topological sort of the remaining DAG, resulting in a string P that minimizes the tandem duplication distance to S and T . In Figure 5.2, we illustrate three possible directed acyclic graphs achievable by removing the minimum number of blue and red edges, based on the example presented in Figure 5.1. In this example, we could remove two edges and render the graph a DAG (note that we do not consider self-edges to be violating the acyclic constraint). It is important to note that there could be multiple solutions for the given directed graph, resulting in different possible ancestral gene orders. Furthermore, depending on the string that we choose to be the source sequence S , the resulting ancestral gene order may differ from other solutions (the roles of S and T could be interchanged). Additionally, we do not consider the self-edges to be violating the order of the vertices. This is because, when we can consider a sequence P that uses the consecutive repeating characters represented by the self-edges, self-edges would not be considered as descents in regards to S or T . Thus, they maintain the continuity of consecutive repeated characters in the final solution.

Figure 5.2 illustrates three possible DAGs that we can get by removing two edges from the $G_{S,T}$ in Figure 5.1. Let us consider the 5.2 (I) DAG. As we can see, the graph does not have any cycles except self-edges. Therefore, we can find a topological order in this graph. One possible order would be 212334. Note that when we reach a node with self-edges, we

exhaust the weight of the self-edges by repeating the character represented by that node before moving to the next node in the topological order. If we have multiple self-edges at a particular node, we can consolidate all the self-edges by replacing them with one self-edge with weight attached to it. The weight would signify the number of previous self-edges.

$$P = 2123344 ,$$

$$\rho_P(S) = \{12, 234\} ,$$

$$\rho_P(T) = \{213344, 2\} .$$

For the 5.2 (III), possible topological orders would be 2123344 and 1223344. For $P = 1223344$,

$$P = 1223344 ,$$

$$\rho_P(S) = \{12234\} ,$$

$$\rho_P(T) = \{2, 13344, 2\} ,$$

We can see that the sum of maximal increasing subsequences from P to S and T is 4 in both of these cases. Therefore, possible ancestral gene orders for S and T would be 2123344, 1223344.

Finding the set of edges that remove cycles in a DAG is called the feedback arc set problem. However, this problem is NP-complete. Given that real-world gene orders of mitochondrial genes typically comprise around 35 to 50 genes, we could use several approaches to solve this problem. These include using brute force, approximation algorithms [41], FPT-algorithms [24], or the MIP formulation [4] of the feedback arc set problem. It is important to note that we do not input the self-edges to the MIP instance of the implementation. However, when we output the ancestral gene order, we use the self-edges

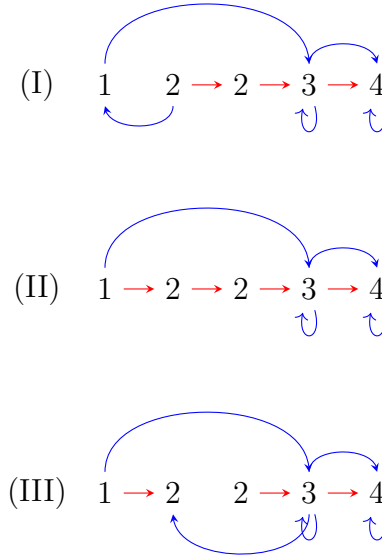


Figure 5.2: Three possible directed acyclic graphs achievable by removing the minimum number of blue and red edges, based on the example presented in Figure 5.1.

to output repeated consecutive characters.

5.4.2 Experiments

The feedback arc set solver was implemented utilizing the Mixed Integer Programming (MIP) formulation outlined in [4]. Google OR-Tools were used to solve the MIP formulation [92]. A concise overview of the MIP formulation is presented below.

The primary objective of the formulation is to determine the minimum cost ordering π^* of the nodes of $G = (V, E)$. Let $c_{i,j}$ denote the cost of the edge $(i, j) \in E$, and let $c_{i,j} = 0$ if $(i, j) \notin E$. Depending on the objective, we can make $c_{i,j}$ be 1 or the weight of the edge (i, j) in the graph. If we need to minimize the cardinality, we can make $c_{i,j} = 1$. Otherwise, $c_{i,j}$ is the weight associated with the edge. Furthermore, we define binary variables $y_{i,j}$ for all $i, j \in V, i \neq j$. We encode the following information in the binary variable $y_{i,j}$: let $y_{i,j} = 0$ if node i precedes node j in π , and let $y_{i,j} = 1$ otherwise. Any ordering π uniquely determines a corresponding y . We can use the following integer linear programming formulation:

$$\min_y \sum_{j=1}^n \left(\sum_{k=1}^{j-1} c_{k,j} y_{k,j} + \sum_{l=j+1}^n c_{l,j} (1 - y_{j,l}) \right)$$

subjects to

$$y_{i,j} + y_{j,k} - y_{i,k} \leq 1, \quad 1 \leq i < j < k \leq n$$

$$-y_{i,j} - y_{j,k} - y_{i,k} \leq 0, \quad 1 \leq i < j < k \leq n$$

$$y_{i,j} \in \{0, 1\} \quad 1 \leq i < j \leq n$$

Any y that correspond to the above formulation must corresponds to an ordering. Note that there are $O(n^2)$ binary variables and $O(n^3)$ constraints. During the experiments, if multiple directed edges were observed from node i to node j within the directed graph instance, we simplified these edges by substituting the multiple edges with a single directed edge. The weight of this consolidated edge is equal to the cumulative edge count of previous directed edges originating from node i to node j .

The ancestral gene order reconstruction algorithm was subjected to testing using 23 distinct species' mitochondrial gene order data. These specific species have been previously studied and documented in references [85] and [66]. From the pool of 23 species, we selected 18 pairs of mitochondrial gene order sets for further analysis using the ancestral gene order prediction algorithm. The sequences and gene annotations for these 23 species were sourced from the NCBI (<https://www.ncbi.nlm.nih.gov/>) database. In order to facilitate easier comprehension of the gene order, one-letter abbreviations of amino acids were used to annotate certain genes within the mitochondrial gene order. The summary of ancestor prediction tests is summarized in 3.1. Please refer to the figures mentioned in Table 3.1 for more information about the predicted gene order.

Test	Species	Elapsed time (ms)	TD(A,S)	TD(A,T)	TD(S,T)	Fig.
1	Siphonops Annulatus	452	1	1	2	5.3
	Trachyrincus Murrayi					
2	Siphonops Annulatus	738	1	1	2	5.4
	Ponticola Kessleri					
3	Hamaeleo Africanus	107	0	2	2	5.5
	Siphonops Annulatus					
4	Siphonops Annulatus	639	1	0	2	5.6
	Lampris Guttatus					
5	Rhineura Floridana	577	0	2	2	5.7
	Siphonops Annulatus					
6	Siphonops Annulatus	580	1	0	2	5.8
	Hapalogenys Analis					
7	Normichthys Operosus	96	0	2	2	5.9
	Pagothenia Borchgrevinki					
8	Dallia Pectoralis	131	0	1	1	5.10
	Muraenesox Bagio					
9	Dallia Pectoralis	116	0	2	2	5.11
	Squalogadus Modificatus					
10	Clinocottus Analis	90	0	2	2	5.12
	Aulorhynchus Flavidus					
11	Ventrifossa Garmani	111	0	2	2	5.13
	Normichthys Operosus					

Test	Species	Elapsed time (ms)	TD(A,S)	TD(A,T)	TD(S,T)	Fig.
12	Aulorhynchus Flavidae	94	0	2	2	5.14
	Ponticola Kessleri					
13	Rhineura Floridana	509	1	1	2	5.15
	Chamaeleo Africanus					
14	Hoplunnis Punctata	97	0	2	2	5.16
	Ventrifossa Garmani					
15	Hemisis Marmoratus	227	1	2	3	5.17
	Chamaeleo Arabicus					
16	Hemisis Marmoratus	276	0	2	2	5.18
	Hyperolius Marmoratus					
17	Hyperolius Marmoratus	269	1	2	3	5.19
	Trichobatrachus Robustus					
18	Hyperolius Marmoratus	284	1	2	3	5.20
	Breviceps Adpersus					

Table 5.4: This table contains a summary of the ancestor gene order prediction tests. TD(A,S) and TD(A,T) columns represent the minimum tandem duplication needed to transform the ancestor (A) into S and T . The elapsed time column contains the time taken by the algorithm to generate the ancestor sequence (ms). The TD(S,T) column represents the minimum number of tandem duplications required to transform S to T .

F	12S	V	16S	L	ND1	I	Q	M	ND2	A	C	W	N	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	L	ND5	ND6	E	CYTB	T	P	CR
Siphonops Annulatus																																					
F	12S	V	16S	L	ND1	I	Q	M	ND2	W	A	N	C	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	L	ND5	ND6	E	CYTB	T	P	CR
Predicted ancestor gene order																																					
F	12S	V	16S	ND1	I	L	Q	M	ND2	W	A	N	C	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	L	ND5	ND6	E	CYTB	T	P	CR
Trachyrincus Murrayi																																					

Figure 5.3: Ancestor gene order prediction between Siphonops Annulatus and Trachyrincus Murrayi.

F	12S	V	16S	L	ND1	I	Q	M	ND2	A	C	W	N	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	L	ND5	ND6	E	CYTB	T	P	CR
Siphonops Annulatus																																					
F	12S	V	16S	L	ND1	I	Q	M	ND2	W	A	N	C	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	L	ND5	ND6	E	CYTB	T	P	CR
Predicted ancestor gene order																																					
F	12S	V	16S	L	ND1	Q	I	M	ND2	W	A	N	C	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	L	ND5	ND6	E	CYTB	T	P	CR
Ponticola Kessleri																																					

Figure 5.4: Ancestor gene order prediction between Siphonops Annulatus and Ponticola Kessleri.

F	12S	V	16S	L	ND1	Q	I	M	ND2	W	A	N	C	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	L	ND5	ND6	E	CYTB	T	CR	P
Hamaleo Africanus																																					
F	12S	V	16S	L	ND1	Q	I	M	ND2	W	A	N	C	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	L	ND5	ND6	E	CYTB	T	CR	P
Predicted ancestor gene order																																					
F	12S	V	16S	L	ND1	I	Q	M	ND2	A	C	W	N	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	L	ND5	ND6	E	CYTB	T	P	CR
Siphonops Annulatus																																					

Figure 5.5: Ancestor gene order prediction between Hamaleo Africanus and Siphonops Annulatus.

F	12S	V	16S	L	ND1	I	Q	M	ND2	A	C	W	N	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	L	ND5	ND6	E	CYTB	T	P	CR
Siphonops Annulatus																																					
F	12S	V	16S	L	ND1	I	Q	M	ND2	W	A	N	C	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	L	ND5	ND6	E	CYTB	T	P	CR
Predicted ancestor gene order																																					
F	12S	V	16S	L	ND1	I	Q	M	ND2	W	A	N	C	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	L	ND5	ND6	E	CYTB	T	CR	
Lampris Guttatus																																					

Figure 5.6: Ancestor gene order prediction between Siphonops annulatus and Lampris Guttatus.

F	12S	V	16S	L	ND1	I	Q	M	ND2	W	A	N	C	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	L	ND5	CYTB	T	P	ND6	E	CR
Rhineura Floridana																																					
F	12S	V	16S	L	ND1	I	Q	M	ND2	W	A	N	C	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	L	ND5	CYTB	T	P	ND6	E	CR
Predicted ancestor gene order																																					
F	12S	V	16S	L	ND1	I	Q	M	ND2	A	C	W	N	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	L	ND5	ND6	E	CYTB	T	P	CR
Siphonops Annulatus																																					

Figure 5.7: Ancestor gene order prediction between *Rhineura Floridana* and *Siphonops Annulatus*.

F	12S	V	16S	L	ND1	I	Q	M	ND2	A	C	W	N	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	L	ND5	ND6	E	CYTB	T	P	CR
Siphonops Annulatus																																					
F	12S	V	16S	L	ND1	I	Q	M	ND2	W	A	N	C	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	L	ND5	ND6	E	CYTB	T	P	CR
Predicted ancestor gene order																																					
F	12S	V	16S	L	ND1	I	Q	M	ND2	W	A	N	C	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	L	ND5	ND6	E	CYTB	T	CR	
Hapalogenys Analis																																					

Figure 5.8: Ancestor gene order prediction between *Siphonops Annulatus* and *Haplophenys Analis*.

F	12S	V	16S	L	ND1	I	Q	M	ND2	W	A	N	C	Y	COX1	D	S	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	L	ND5	ND6	E	CYTB	T	P
Normichthys Operosus																																				
F	12S	V	16S	L	ND1	I	Q	M	ND2	W	A	N	C	Y	COX1	D	S	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	L	ND5	ND6	E	CYTB	T	P
Predicted ancestor gene order																																				
F	12S	V	16S	L	ND1	I	Q	M	ND2	W	A	N	C	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	L	ND5	CYTB	T	P	ND6	E
Pagothenia Borchgrevinki																																				

Figure 5.9: Ancestor gene order prediction between *Normichthys Operosus* and *Pagothenia Borchgrevinki*.

F	12S	V	16S	L	ND1	I	Q	M	ND2	W	A	N	C	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	L	ND5	ND6	E	CYTB	P	CR	T
Dallia Pectoralis																																					
F	12S	V	16S	L	ND1	I	Q	M	ND2	W	A	N	C	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	L	ND5	ND6	E	CYTB	P	CR	T
Predicted ancestor gene order																																					
F	12S	V	16S	L	ND1	I	Q	M	ND2	W	A	N	C	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	L	ND5	CYTB	T	ND6	E	CR	
Muraenesox Bagio																																					

Figure 5.10: Ancestor gene order prediction between *Dallia pectoralis* and *Muraenesox bagio*.

F	12S	V	16S	L	ND1	I	Q	M	ND2	W	A	N	C	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	L	ND5	ND6	E	CYTB
Dallia Pectoralis																																		
F	12S	V	16S	L	ND1	I	Q	M	ND2	W	A	N	C	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	L	ND5	ND6	E	CYTB
Predicted ancestor gene order																																		
F	12S	V	16S	ND1	I	L	Q	M	ND2	W	A	N	C	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	L	ND5	ND6	E	CYTB
Squalogadus Modificatus																																		

Figure 5.11: Ancestor gene order prediction between *Dallia Pectoralis* and *Squalogadus Modificatus*.

F	12S	V	16S	CR	P	L	ND1	I	Q	M	ND2	W	A	N	C	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	L	ND5	ND6	E	CYTB	T	CR	P
Clinocottus Analis																																							
F	12S	V	16S	CR	P	L	ND1	I	Q	M	ND2	W	A	N	C	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	L	ND5	ND6	E	CYTB	T	CR	P
Predicted ancestor gene order																																							
F	12S	V	16S	L	ND1	I	Q	M	ND2	W	A	N	C	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	S	H	L	ND5	ND6	E	CYTB	T	P	CR		
Aulorhynchus Flavidus																																							

Figure 5.12: Ancestor gene order prediction between *Clinocottus Analis* and *Aulorhynchus Flavidus*.

F	12S	V	16S	CR	P	L	ND1	I	Q	M	ND2	W	A	N	C	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	L	ND5	ND6	E	CYTB	T	CR	P
Ventrifossa Garmani																																							
F	12S	V	16S	CR	P	L	ND1	I	Q	M	ND2	W	A	N	C	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	L	ND5	ND6	E	CYTB	T	CR	P
Predicted ancestor gene order																																							
F	12S	V	16S	L	ND1	I	Q	M	ND2	W	A	N	C	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	S	H	L	ND5	ND6	E	CYTB	T	P	CR		
Normichthys Operosus																																							

Figure 5.13: Ancestor gene order prediction between *Ventrifossa Garmani* and *Normichthys Operosus*.

F	12S	V	16S	L	ND1	I	Q	M	ND2	W	A	N	C	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	S	H	L	ND5	ND6	E	CYTB	T	P	CR
Aulorhynchus Flavidus																																					
F	12S	V	16S	L	ND1	I	Q	M	ND2	W	A	N	C	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	S	H	L	ND5	ND6	E	CYTB	T	P	CR
Predicted ancestor gene order																																					
F	12S	V	16S	L	ND1	Q	I	M	ND2	W	A	N	C	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	L	ND5	ND6	E	CYTB	T	P	CR
Ponticola Kessleri																																					

Figure 5.14: Ancestor gene order prediction between *Aulorhynchus Flavidus* and *Ponticola Kessleri*.

F	12S	V	16S	L	ND1	I	Q	M	ND2	W	A	N	C	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	L	ND5	CYTB	T	P	ND6	E
Rhineura Floridana																																				
F	12S	V	16S	L	ND1	I	Q	M	ND2	W	A	N	C	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	L	ND5	ND6	E	CYTB	T	CR
Predicted ancestor gene order																																				
F	12S	V	16S	L	ND1	Q	I	M	ND2	W	A	N	C	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	L	ND5	ND6	E	CYTB	T	CR
Chamaeleo Africanus																																				

Figure 5.15: Ancestor gene order prediction between Rhineura Floridana and Chamaeleo Africanus.

F	12S	V	16S	L	ND1	I	Q	M	ND2	W	A	N	C	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	L	ND5	CYTB	T	ND6	E	P	CR
Hoplunnis Punctata																																					
F	12S	V	16S	L	ND1	I	Q	M	ND2	W	A	N	C	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	L	ND5	CYTB	T	ND6	E	P	CR
Predicted ancestor gene order																																					
F	12S	V	16S	L	ND1	I	Q	M	ND2	W	A	N	C	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	L	ND5	ND6	CYTB	T	P	E	CR
Ventrifossa Garmani																																					

Figure 5.16: Ancestor gene order prediction between Hoplunnis Punctata and Ventrifossa Garmani.

F	12S	V	16S	L	ND1	I	Q	M	ND2	W	N	OL	A	C	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	ND5	ND6	E	CYTB	P	L	T	CR
Hemisus Marmoratus																																						
F	12S	V	16S	L	ND1	I	Q	M	ND2	W	N	OL	A	C	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	L	ND5	ND6	E	CYTB	T	CR	P
Predicted ancestor gene order																																						
F	12S	V	16S	L	ND1	Q	I	M	ND2	W	A	N	C	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	L	ND5	ND6	E	CYTB	T	CR	P	
Chamaeleo Arabicus																																						

Figure 5.17: Ancestor gene order prediction between Hemisis Marmoratus and Chamaeleo Arabicus.

F	12S	V	16S	L	ND1	I	Q	M	ND2	W	N	OL	A	C	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	ND5	ND6	E	CYTB	P	L	T	CR			
Hemisis Marmoratus																																									
F	12S	V	16S	L	ND1	I	Q	M	ND2	W	N	OL	A	C	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	ND5	ND6	E	CYTB	P	L	T	CR			
Predicted ancestor gene order																																									
P	T	L	F	12S	V	16S	L	ND1	I	Q	M	ND2	CR	M	ND2	W	N	OL	A	C	Y	COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	ND5	ND6	E	CYTB	CR
Hyperolius Marmoratus																																									

Figure 5.18: Ancestor gene order prediction between Hemisis Marmoratus and Hyperolius Marmoratus.

COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	ND5	ND6	E	CYTB	CR	P	T	L	F	12S	V	16S	L	ND1	I	Q	M	ND2	CR	M	ND2	W	N	OL	A	C	Y					
Hyperolius Marmoratus																																														
COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	ND5	ND6	E	CYTB	CR	L	T	P	F	12S	V	16S	L	ND1	I	Q	M	ND2	CR	M	ND2	W	N	OL	A	C	Y					
Predicted ancestor gene order																																														
COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	ND5	CR	L	T	P	F	H	S	ND5	ND6	E	CYTB	CR	L	T	P	F	12S	V	16S	L	ND1	I	Q	M	ND2	W	N	OL	A	C	Y
Trichobatrachus Robustus																																														

Figure 5.19: Ancestor gene order prediction between Hyperolius Marmoratus and Trichobatrachus Robustus.

COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	ND5	ND6	E	CYTB	CR	P	T	L	F	12S	V	16S	L	ND1	I	Q	M	ND2	CR	M	ND2	W	N	OL	A	C	Y											
Hyperolius Marmoratus																																																				
COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	H	S	ND5	ND6	E	CYTB	T	L	CR	P	F	12S	V	16S	L	ND1	I	Q	M	ND2	CR	M	ND2	W	N	OL	A	C	Y											
Predicted ancestor gene order																																																				
COX1	S	D	COX2	K	ATP8	ATP6	COX3	G	ND3	R	ND4L	ND4	S	ND5	ND6	E	CYTB	T	P	F	12S	V	16S	L	T	P	F	12S	V	16S	H	S	P	F	12S	V	16S	L	ND1	I	Q	M	ND2	W	N	OL	W	N	OL	A	C	Y
Breviceps Adpersus																																																				

Figure 5.20: Ancestor gene order prediction between Hyperolius Marmoratus vs Breviceps Adpersus.

5.5 Conclusion

In this chapter, we explored a distance measurement for non-permutation strings under the tandem duplication random loss (TDRL) model with the objective of facilitating phylogenetic tree construction for mitochondrial gene order data. In the previous literature, distance measurements for the TDLR model did not consider non-permutation strings, as per our knowledge. Moreover, it considered tandem duplication and deletion operations as one atomic event. To address this gap, we introduced two parameters, α and β , serving as cost function parameters for duplicated blocks and deleted segments. We considered α^k and β^k as the costs associated with duplicating and deleting length k blocks, respectively. Our objective was to develop heuristic algorithms to calculate the minimum distance between two input sequences under the given parameter values.

For $\alpha \leq 1$ and $\alpha > 1$, we proposed two different heuristic algorithms to compute the TDRL distance between two input sequences. When $\alpha = 1$, the cost of duplication is independent of the length of the block. Thus, whole genome duplication (WGD) was used as a heuristic choice. Conversely, when $\alpha > 1$, WGD was no longer a good heuristic choice; thus, bubble sort distance was used as the heuristic choice. Finally, we introduced a heuristic algorithm to predict ancestor gene order between two input gene orders for non-permutation strings. We reduced predicting the ancestor gene to finding the feedback arc set of a graph instance.

We implemented TDRL distance heuristic algorithms and tested mitochondrial gene

order data from 43 species. We calculated the distance from these 43 species to the vertebrate gene order. The ancestral gene-order prediction algorithm was tested on 23 distinct species' mitochondrial gene-order data. The algorithms were implemented in Java and tested on a PC with an Intel (R) Xeon(R) CPU, 3.70GHz, and 64 GB of RAM. As the feedback arc set problem is NP-hard, we used a mixed integer programming (MIP) formulation and solved it using Google OR-Tool libraries.

As for future research directions, future research could explore extending distance measurements to circular gene order data. As mitochondrial genomes are circular, lacking a distinct starting point, it would be beneficial to extend the distance measurement algorithm and ancestral gene order prediction algorithm to consider all the possible alignments of two circular gene order data. Moreover, it would be useful to incorporate the signs of the genes into these distance measurements and the ancestral gene order prediction algorithm.

CHAPTER SIX

SUMMARY

In this dissertation, we have investigated four problems related to genome rearrangement. Chapter 2 presents several variants of the MSR (Maximal Strip Recovery) problem, termed the F -Strip Recovery problem. Part of this chapter has been published in the proceedings of the 19th International Conference on Comparative Genomics (RECOMB-CG 2022) [68]. Chapter 3 discusses the longest letter-duplicated subsequence problem, which is related to tandem duplications and aims to extract information about duplicated genome blocks. Part of this chapter has been published in the proceedings of the 33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022) [72]. Chapter 4 introduces the longest subsequence-repeated subsequence problem, which can be viewed as a higher generalization of the longest letter-duplicated subsequence problem. Part of this chapter has been published in the 16th Annual International Conference on Combinatorial Optimization and Applications (COCOA' 23) [67]. The final chapter focuses on distance measurements for non-permutation strings under the tandem duplication random loss (TDRL) model. We tested proposed heuristic algorithms on real-world data sets and published the results in this dissertation.

CHAPTER SEVEN

CONCLUSION

In this dissertation, we have investigated several problems related to genome rearrangements.

In chapter 2, we investigated variations of Maximal Strip Recovery (MSR) problem, which we call the *F-Strip Recovery problem*. Basically, this problem deals with partitioning a given set of strings into blocks of available sizes. These variations could be considered as higher generalization of MSR problem. We explored variations of this problem under different parameters and conditions such as allowed block sizes, bounded and unbounded alphabets and allowing markers to be deleted. We proved that Exact *F-Strip Recovery* problem is NP-hard for unbounded alphabets and unbounded F (block size list). Most importantly, we showed that it is challenging to partition two genomes into blocks of sizes from any fixed list unless we restrict ourselves to one size. Additionally, we provided a polynomial-time algorithm for the optimization version of GSR-F problem for fixed alphabets.

In chapter 3, we looked at the problem of longest letter-duplicated subsequence (LLDS). Essentially, given a sequence S , letter-duplicated subsequence is a subsequence in the form of $x_1^{d_1}x_2^{d_2}\dots x_k^{d_k}$ where $x_i \in \Sigma, x_j \neq x_{j+1}$ and $d_i \geq 2$ for all $i \in [k]$ and $j \in [k-1]$. Each $x_i^{d_i}$ block is called a LDS block. Finding the longest such subsequence for a given sequence is called longest letter-duplicated subsequence problem. We showed that LLDS problem can be solved in polynomial-time using a dynamic programming solution. Then we looked at a special variation of this problem called constrained longest letter-duplicated subsequence problem (LLDS+), wherein we require LLDS solution contain all the letters in the alphabet. We proved this version of this problem is NP-hard for parameter $d \geq 4$, using

a reduction from $(3^+, 1, 2^-)$ -SAT problem. In this case d is the maximum occurrence of any letter in the alphabet. Furthermore, we showed that when $d < 4$ LLDS+ is polynomially solvable. Essentially, we found the parameter that makes this problem difficult to solve. Additionally, we showed that obtaining a bi-criteria approximation for LLDS+(4) is as difficult as approximating LLDS+(4).

In chapter 4, we investigated the longest subsequence-repeated subsequence (LSRS) problem. This problem can be viewed as a higher generalization of LLDS problem that we investigated in chapter 3. The major difference of LSRS problem is that we aim to find the subsequence-repeated blocks in the given sequence instead of letter-duplicated blocks. In retrospect, given a sequence S , we are asked to find a subsequence-repeated subsequence in the form of $x_1^{d_1} x_1^{d_1} \cdots x_n^{d_n}$ where x_i is a subsequence of S and $x_j \neq x_{j+1}$ and $d_i \geq 2$, for $i \in [k], j \in [k-1]$. $x_i^{d_i}$ is called a subsequence-repeated block (*SR-block*). First we showed that finding the longest subsequence-repeated subsequence for a given input sequence S is polynomially solvable. We provided a dynamic programming solution for LSRS problem. Notably, we used two sub-problems in this dynamic programming formulation; the longest square subsequence problem and longest cubic subsequence problem. We demonstrated a graph based algorithm that allows us to calculate longest square subsequence and longest cubic subsequence of all substrings in a given sequence. We showed that longest cubic subsequences of all substrings can be calculated in $O(n^6)$ time. Next, we explored the constrained version of LSRS problem (LSRS+), and showed that this problem is NP-hard for parameter $d \geq 4$. Similarly, parameter d is the maximum allowed duplication of each letter in the given input sequence. The strings problems that we explored in chapter 3 and chapter 4 can be used to extract information about the genes that may have underwent tandem duplication events. As mentioned in chapter 4, finding minimum number of tandem duplication to transform a source sequence S to T is NP-hard [71], therefore, LLDS and LSRS formulations can be used to extract some information about genes that may have

underwent tandem duplications.

In this last chapter, we explored a distance measurement for non-permutation strings under tandem duplication random loss model (TDRL) for mitochondrion gene order data. Existing literature was based on the permutation strings, therefore we improved distance measurements calculations for non-permutation strings. This distance measurement could be used to generate phylogenetic trees for gene order data. We introduced two heuristics algorithms for calculating distance between non-permutation strings under TDRL model. These heuristics algorithms were implemented and tested on 23 set of real world data sets. Furthermore, we introduced an algorithm for predicting ancestral gene order of two given gene orders. We tested this algorithm on a pool of 23 species' mitochondrial gene order data. A possible direction for future work for this chapter would be to consider mitochondrial gene order data as a circular string and extend this proposed distance measurement for circular strings.

REFERENCES CITED

- [1] SMHL Anderson, MHL De Bruijn, AR Coulson, IC Eperon, F Sanger, and IG Young. Complete sequence of bovine mitochondrial dna conserved features of the mammalian mitochondrial genome. *Journal of molecular biology*, 156(4):683–717, 1982.
- [2] Yoann Anselmetti, Vincent Berry, Cédric Chauve, Annie Chateau, Éric Tannier, and Sèverine Bérard. Ancestral gene synteny reconstruction improves extant species scaffolding. *BMC Genomics*, 16:S11 – S11, 2015.
- [3] Shuichi Asakawa, Hyouta Himeno, K-I Miura, and Kimitsuna Watanabe. Nucleotide sequence and gene organization of the starfish asterina pectinifera mitochondrial genome. *Genetics*, 140(3):1047–1060, 1995.
- [4] Ali Baharev, Hermann Schichl, Arnold Neumaier, and TOBIAS Achterberg. An exact method for the minimum feedback arc set problem. *Journal of Experimental Algorithmics (JEA)*, 26:1–28, 2021.
- [5] J William O Ballard and Michael C Whitlock. The incomplete natural history of mitochondria. *Molecular ecology*, 13(4):729–744, 2004.
- [6] C. Timothy Beagley, Jane L. Macfarlane, Geneviève A. Pont-Kingdon, Ronald Okimoto, Norichika A. Okada, and David R. Wolstenholme. Mitochondrial genomes of anthozoa (cnidaria). In Ferdinando Palmieri, Sergio Papa, Cecilia Saccone, and Maria Nicola Gadaleta, editors, *Thirty Years of Progress in Mitochondrial Bioenergetics and Molecular Biology*, volume 5 of *Progress in Cell Research*, pages 149–153. Elsevier, 1995.
- [7] C Timothy Beagley, Ronald Okimoto, and David R Wolstenholme. The mitochondrial genome of the sea anemone metridium senile (cnidaria): introns, a paucity of trna genes, and a near-standard genetic code. *Genetics*, 148(3):1091–1108, 1998.
- [8] Jeffrey L Boore and Wesley M Brown. Big trees from little genomes: mitochondrial gene order as a phylogenetic tool. *Current opinion in genetics & development*, 8(6):668–674, 1998.
- [9] Jeffrey L Boore, Timothy M Collins, David Stanton, L Lynne Daehler, and Wesley M Brown. Deducing the pattern of arthropod phylogeny from mitochondrial dna rearrangements. *Nature*, 376(6536):163–165, 1995.
- [10] Jeffrey L Boore, Dennis V Lavrov, and Wesley M Brown. Gene translocation links insects and crustaceans. *Nature*, 392(6677):667–668, 1998.
- [11] Guillaume Bourque and Pavel A. Pevzner. Genome-scale evolution: reconstructing gene orders in the ancestral species. *Genome research*, 12 1:26–36, 2002.

- [12] Guillaume Bourque, Pavel A. Pevzner, and Glenn Tesler. Reconstructing the genomic architecture of ancestral mammals: lessons from human, mouse, and rat genomes. *Genome research*, 14 4:507–16, 2004.
- [13] Daniel P. Bovet and Stefano Varricchio. On the regularity of languages on a binary alphabet generated by copying systems. *Inf. Process. Lett.*, 44:119–123, 1992.
- [14] Karl Bringmann and Marvin Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. *2015 IEEE 56th Annual Symposium on Foundations of Computer Science*, pages 79–97, 2015.
- [15] Laurent Bulteau, Guillaume Fertin, and Irena Rusu. Maximal strip recovery problem with gaps: Hardness and approximation algorithms. *J. Discrete Algorithms*, 19:1–22, 2009.
- [16] Laurent Bulteau, Guillaume Fertin, and Irena Rusu. Sorting by transpositions is difficult. In *SIAM Journal on Discrete Mathematics*, 2010.
- [17] Laurent Bulteau and Christian Komusiewicz. Minimum common string partition parameterized by partition size is fixed-parameter tractable. In *ACM-SIAM Symposium on Discrete Algorithms*, 2013.
- [18] Laurent Bulteau and Mathias Weller. Parameterized algorithms in bioinformatics: An overview. *Algorithms*, 12:256, 2019.
- [19] Palmiro Cantatore, Marina Roberti, Guglielmo Rainaldi, Maria N Gadaleta, and Cecilia Saccone. The complete nucleotide sequence, gene organization, and genetic code of the mitochondrial genome of *paracentrotus lividus*. *Journal of Biological Chemistry*, 264(19):10965–10975, 1989.
- [20] Alberto Caprara. Sorting by reversals is difficult. In *Annual International Conference on Research in Computational Molecular Biology*, 1997.
- [21] Alberto Caprara. The reversal median problem. *INFORMS J. Comput.*, 15:93–113, 2003.
- [22] Barbara Cardazzo, Sonia Minuzzo, Geppo Sartori, Alessandro Grapputo, and Giovanna Carignani. Evolution of mitochondrial dna in yeast: gene order and structural organization of the mitochondrial genome of *saccharomyces uvarum*. *Current genetics*, 33:52–59, 1998.
- [23] Kamalika Chaudhuri, Kevin Chen, Radu Mihaescu, and Satish Rao. On the tandem duplication-random loss model of genome rearrangement. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 564–570, 2006.

- [24] Jianer Chen, Yang Liu, Songjian Lu, Barry O’sullivan, and Igor Razgon. A fixed-parameter algorithm for the directed feedback vertex set problem. In *Proceedings of the fortieth annual ACM symposium on Theory of computing*, pages 177–186, 2008.
- [25] Xin Chen, Jie Zheng, Zheng Fu, Peng Nan, Yang Zhong, Stefano Lonardi, and Tao Jiang. Assignment of orthologous genes via genome rearrangement. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2:302–315, 2005.
- [26] Zhixiang Chen, Bin Fu, Minghui Jiang, and Binhai Zhu. On recovering syntenic blocks from comparative maps. *Journal of Combinatorial Optimization*, 18:307–318, 2008.
- [27] Vicky Siu-Ngan Choi, Chunfang Zheng, Qian Zhu, and David Sankoff. Algorithms for the extraction of syntenic blocks from comparative maps. In *Workshop on Algorithms in Bioinformatics*, 2007.
- [28] Giovanni Ciriello, Martin Miller, Bulent Aksoy, Yasin Şenbabaoğlu, Nikolaus Schultz, and Chris Sander. Emerging landscape of oncogenic signatures across human cancers. *Nature genetics*, 45:1127–1133, 09 2013.
- [29] Peter Damaschke. Minimum common string partition parameterized. In *Workshop on Algorithms in Bioinformatics*, 2008.
- [30] Jürgen Dassow, Victor Mitrana, and Gheorghe Paun. On the regularity of duplication closure. *Bull. EATCS*, 69:133–136, 1999.
- [31] Carla De Giorgi, Antonia Martiradonna, Cecilia Lanave, and Cecilia Saccone. Complete sequence of the mitochondrial dna in the sea urchin *Paracentrotus lividus*: conserved features of the echinoid mitochondrial genome. *Molecular phylogenetics and evolution*, 5(2):323–332, 1996.
- [32] Mattéo Delabre, Nadia El-Mabrouk, Katharina T. Huber, Manuel Lafond, Vincent Moulton, Emmanuel Noutahi, and Miguel Sautie Castellanos. Evolution through segmental duplications and losses: a super-reconciliation approach. *Algorithms for Molecular Biology : AMB*, 15, 2020.
- [33] Romain Derelle, Hervé Philippe, and John K. Colbourne. Broccoli: combining phylogenetic and network analyses for orthology assignment. *Molecular biology and evolution*, 2020.
- [34] Paul Desjardins and Réjean Morais. Sequence and gene organization of the chicken mitochondrial genome: a novel gene order in higher vertebrates. *Journal of molecular biology*, 212(4):599–634, 1990.
- [35] Antonio Diez-Juan and Carlos Marín. Mitochondria and embryo viability. In Michael K. Skinner, editor, *Encyclopedia of Reproduction (Second Edition)*, pages 222–230. Academic Press, Oxford, second edition edition, 2018.

- [36] Guénola Drillon, Raphaël Champeimont, Francesco Oteri, Gilles Fischer, and Alessandra Carbone. Phylogenetic reconstruction based on syntenic block and gene adjacencies. *Molecular Biology and Evolution*, 37:2747 – 2762, 2019.
- [37] Andrzej Ehrenfeucht and Grzegorz Rozenberg. On regularity of languages generated by copying systems. *Discret. Appl. Math.*, 8:313–317, 1984.
- [38] Evan E. Eichler and David Sankoff. Structural dynamics of eukaryotic chromosome evolution. *Science*, 301:793 – 797, 2003.
- [39] Isaac Elias and Tzvika Hartman. A 1.375-approximation algorithm for sorting by transpositions. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 3, 2005.
- [40] Funda Ergun, S Muthukrishnan, and S Cenk Sahinalp. Comparing sequences with segment rearrangements. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 183–194. Springer, 2003.
- [41] Guy Even, Joseph Naor, Baruch Schieber, and Madhu Sudan. Approximating minimum feedback sets and multicuts in directed graphs. *Algorithmica*, 20:151–174, 1998.
- [42] Michal Ganczorz, Pawel Gawrychowski, Artur Jeż, and Tomasz Kociumaka. Edit distance with block operations. In *Embedded Systems and Applications*, 2018.
- [43] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.
- [44] Avraham Goldstein, Petra Kolman, and Jie Zheng. Minimum common string partition problem: Hardness and approximations. In *Electronic Journal of Combinatorics*, 2004.
- [45] Leo Goodstadt and Chris Paul Ponting. Phylogenetic reconstruction of orthology, paralogy, and conserved synteny for dog and human. *PLoS Computational Biology*, 2, 2006.
- [46] Michael W Gray, B Franz Lang, Robert Cedergren, G Brian Golding, Claude Lemieux, David Sankoff, Monique Turmel, Nicolas Brossard, Eric Delage, Tim G Littlejohn, et al. Genome structure and gene content in protist mitochondrial dnas. *Nucleic acids research*, 26(4):865–878, 1998.
- [47] Sridhar Hannenhalli and Pavel A. Pevzner. Transforming cabbage into turnip: polynomial algorithm for sorting signed permutations by reversals. *J. ACM*, 46:1–27, 1995.
- [48] Sridhar Hannenhalli and Pavel A. Pevzner. Transforming men into mice (polynomial algorithm for genomic distance problem). *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 581–592, 1995.

- [49] A Härlid, Axel Janke, and Ulfur Arnason. The mtdna sequence of the ostrich and the divergence between paleognathous and neognathous birds. *Molecular Biology and Evolution*, 14(7):754–761, 1997.
- [50] Tom Hartmann, Martin Middendorf, and Matthias Bernt. Genome rearrangement analysis: Cut and join genome rearrangements and gene cluster preserving approaches. *Methods in molecular biology*, 1704:261–289, 2018.
- [51] Richard J Hoffmann, JL Boore, and WM Brown. A novel mitochondrial genome organization for the blue mussel, *mytilus edulis*. *Genetics*, 131(2):397–412, 1992.
- [52] Mohammad Sajid Ali Howlader, Abhilash Nair, Sujith V Gopalan, and Juha Merilä. A new species of microhyla (anura: Microhylidae) from nilphamari, bangladesh. *PloS one*, 10(3):e0119825, 2015.
- [53] Shuai Hu, Wenjun Li, and Jianxin Wang. An improved kernel for the complementary maximal strip recovery problem. In *International Computing and Combinatorics Conference*, 2015.
- [54] Koichiro Inaki and Edison T. Liu. Structural mutations in cancer: mechanistic and functional insights. *Trends in Genetics*, 28(11):550–559, 2012.
- [55] Howard T Jacobs, David J Elliott, Veerabhadracharya B Math, and Andrew Farquharson. Nucleotide sequence and gene organization of sea urchin mitochondrial dna. *Journal of molecular biology*, 202(2):185–217, 1988.
- [56] Axel Janke, Gertraud Feldmaier-Fuchs, W Kelley Thomas, A Von Haeseler, and Svante Pääbo. The marsupial mitochondrial genome and the evolution of placental mammals. *Genetics*, 137(1):243–256, 1994.
- [57] Axel Janke, Xiufeng Xu, and Ulfur Arnason. The complete mitochondrial genome of the wallaroo (*macropus robustus*) and the phylogenetic relationship among monotremata, marsupialia, and eutheria. *Proceedings of the National Academy of Sciences*, 94(4):1276–1281, 1997.
- [58] Haitao Jiang, Jiong Guo, Daming Zhu, and Binhai Zhu. A 2-approximation algorithm for the complementary maximal strip recovery problem. In *Annual Symposium on Combinatorial Pattern Matching*, 2019.
- [59] Haitao Jiang, Zhong Li, Guohui Lin, Lusheng Wang, and Binhai Zhu. Exact and approximation algorithms for the complementary maximal strip recovery problem. *Journal of Combinatorial Optimization*, 23:493–506, 2012.
- [60] Haitao Jiang and Binhai Zhu. A linear kernel for the complementary maximal strip recovery problem. *J. Comput. Syst. Sci.*, 80:1350–1358, 2012.

- [61] Haitao Jiang, Binhai Zhu, Daming Zhu, and Hong Zhu. Minimum common string partition revisited. *Journal of Combinatorial Optimization*, 23:519–527, 2010.
- [62] Lichun Jiang, Li Zhao, Dongmei Cheng, Lilan Zhu, Min Zhang, Qiping Ruan, and Wei Chen. The complete mitochondrial genome sequence of the sichuan digging frog, *kaloula rugifera* (anura: Microhylidae) and its phylogenetic implications. *Gene*, 626:367–375, 2017.
- [63] Minghui Jiang. Inapproximability of maximal strip recovery. *Theor. Comput. Sci.*, 412:3759–3774, 2009.
- [64] Adrian Kosowski. An efficient algorithm for the longest tandem scattered subsequence problem. In *SPIRE*, 2004.
- [65] Yoshinori Kumazawa and Mutsumi Nishida. Variations in mitochondrial trna gene organization of reptiles as phylogenetic markers. *Molecular Biology and Evolution*, 12(5):759–772, 1995.
- [66] Atsushi Kurabayashi and Masayuki Sumida. Afrobatrachian mitochondrial genomes: genome reorganization, gene rearrangement mechanisms, and evolutionary trends of duplicated and rearranged genes. *BMC genomics*, 14:1–17, 2013.
- [67] Manuel Lafond, Wenfeng Lai, Adiesha Liyanage, and Binhai Zhu. The longest subsequence-repeated subsequence problem. In *International Conference on Combinatorial Optimization and Applications*, pages 446–458. Springer, 2023.
- [68] Manuel Lafond, Adiesha Liyanage, Binhai Zhu, and Peng Zou. The complexity of finding common partitions of genomes with predefined block sizes. In *Comparative Genomics: 19th International Conference, RECOMB-CG 2022, La Jolla, CA, USA, May 20–21, 2022, Proceedings*, pages 105–123. Springer, 2022.
- [69] Manuel Lafond, Magali Semeria, Krister M. Swenson, Éric Tannier, and Nadia El-Mabrouk. Gene tree correction guided by orthology. *BMC Bioinformatics*, 14:S5 – S5, 2013.
- [70] Manuel Lafond and Binhai Zhu. Permutation-constrained common string partitions with applications. In *String Processing and Information Retrieval: 28th International Symposium, SPIRE 2021, Lille, France, October 4–6, 2021, Proceedings 28*, pages 47–60. Springer, 2021.
- [71] Manuel Lafond, Binhai Zhu, and Peng Zou. Computing the tandem duplication distance is NP-hard. *SIAM J. Discret. Math.*, 36:64–91, 2022.
- [72] Wenfeng Lai, Adiesha Liyanage, Binhai Zhu, and Peng Zou. Beyond the longest letter-duplicated subsequence problem. In Hideo Bannai and Jan Holub, editors, *33rd Annual Symposium on Combinatorial Pattern Matching, CPM 2022, June 27–29, 2022*,

Prague, Czech Republic, volume 223 of *LIPICs*, pages 7:1–7:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

- [73] Eric Lander, Cong Chen, Lauren Linton, Bruce Birren, Chad Nusbaum, Michael Zody, Jennifer Baldwin, Keri Devon, Ken Dewar, Michael Doyle, William Fitzhugh, Roel Funke, Diane Gaige, Katrina Harris, Andrew Heaford, John Howland, Lisa Kann, Jessica Lehoczy, Rosie LeVine, and Lee Rowen. Initial sequencing and analysis of the human genome. *Nature*, 409, 02 2001.
- [74] Eric Lander and Matthew Meyerson. Integrated genomic analyses of ovarian carcinoma. *Nature*, 474, 10 2012.
- [75] B Franz Lang, Gertraud Burger, Charles J O’Kelly, Robert Cedergren, G Brian Golding, Claude Lemieux, David Sankoff, Monique Turmel, and Michael W Gray. An ancestral mitochondrial dna resembling a eubacterial genome in miniature. *Nature*, 387(6632):493–497, 1997.
- [76] C Leblanc, O Richard, B Kloareg, S Viehmann, K Zetsche, and C Boyen. Origin and evolution of mitochondria: what have we learnt from red algae? *Current Genetics*, 31:193–207, 1997.
- [77] Marcus Lechner, Maribel Hernández-Rosales, Daniel Doerr, Nicolas Wieseke, Annelyse Thévenin, Jens Stoye, Roland K. Hartmann, Sonja J. Prohaska, and Peter F. Stadler. Orthology detection combining clustering and synteny for very large datasets. *PLoS ONE*, 9, 2014.
- [78] John Leech. 2726. a problem on strings of beads. *The Mathematical Gazette*, 41(338):277–278, 1957.
- [79] Alan R Lemmon and Michel C Milinkovitch. The metapopulation genetic algorithm: An efficient solution for the problem of large phylogeny estimation. *Proceedings of the National Academy of Sciences*, 99(16):10516–10521, 2002.
- [80] Wenjun Li, Haiyan Liu, Jianxin Wang, Lingyun Xiang, and Yongjie Yang. An improved linear kernel for complementary maximal strip recovery: Simpler and smaller. *Theor. Comput. Sci.*, 786:55–66, 2019.
- [81] Yongmin Li, Huabin Zhang, Xiaoyou Wu, Dongwei Li, Peng Yan, and Xiaobing Wu. The complete mitochondrial genome of rhacophorus dennysi (anura: Rhacophoridae) with novel gene arrangements and its phylogenetic implications. *Pak. J. Zool.*, 2021.
- [82] Marcy MacDonald, Scott Strobel, Karen Draths, Jennifer Wales, and Peter Dervan. A novel gene containing a trinucleotide repeat that is expanded and unstable on huntington’s disease chromosomes. *Cell*, 72, 03 1993.
- [83] William Martin and Reinhold G Herrmann. Gene transfer from organelles to the nucleus: how much, what happens, and why? *Plant physiology*, 118(1):9–17, 1998.

- [84] David J. McBride, Dariush Etemadmoghadam, Susanna L. Cooke, Kathryn Alsop, J. George, Adam P. Butler, Juok Cho, Danushka Galappaththige, Chris D. Greenman, Karen D. Howarth, King Wai Lau, Charlotte K. Y. Ng, Keiran M. Raine, Jon W. Teague, David C. Wedge, Australian Ovarian Cancer Study Group, Xavier Caubit, Michael R. Stratton, James D. Brenton, Peter J. Campbell, P. Andrew Futreal, and David D. L. Bowtell. Tandem duplication of chromosomal segments is common in ovarian and breast cancer genomes. *The Journal of Pathology*, 227:446 – 455, 2012.
- [85] Paula Montaña-Lozano, Manuela Moreno-Carmona, Mauricio Ochoa-Capera, Natalia S Medina, Jeffrey L Boore, and Carlos F Prada. Comparative genomic analysis of vertebrate mitochondrial reveals a differential of rearrangements rate between taxonomic class. *Scientific reports*, 12(1):5479, 2022.
- [86] Cristopher Moore and John Michael Robson. Hard tiling problems with simple tiles. *Discrete & Computational Geometry*, 26:573–590, 2000.
- [87] Ronald Okimoto, Helen M Chamberlin, Jane L Macfarlane, and David R Wolstenholme. Repeated sequence sets in mitochondrial dna molecules of root knot nematodes (meloidogyne): nucleotide sequences, genome location and potential for host-race identification. *Nucleic Acids Research*, 19(7):1619–1626, 1991.
- [88] Ronald Okimoto, JL Macfarlane, DO Clary, and DR1204866 Wolstenholme. The mitochondrial genomes of two nematodes, *caenorhabditis elegans* and *ascaris suum*. *Genetics*, 130(3):471–498, 1992.
- [89] Svante Pääbo, W Kelley Thomas, Kristina M Whitfield, Yoshinori Kumazawa, and Allan C Wilson. Rearrangements of mitochondrial transfer rna genes in marsupials. *Journal of Molecular Evolution*, 33:426–430, 1991.
- [90] Bruno Paquin, Marie-Josée Laforest, Lise Forget, Ingeborg Roewer, Zhang Wang, Joyce Longcore, and B Franz Lang. The fungal mitochondrial genome project: evolution of fungal mitochondrial genomes and their gene expression. *Current genetics*, 31:380–395, 1997.
- [91] L. Penso-Dolfin, T. Wu, and C.D. Greenman. The combinatorics of tandem duplication. *Discrete Applied Mathematics*, 194:1–22, 2015.
- [92] Laurent Perron and Vincent Furnon. Or-tools.
- [93] Geneviève Pont-Kingdon, Norichika A Okada, Jane L Macfarlane, C Timothy Beagley, Cristi D Watkins-Sims, Thomas Cavalier-Smith, G Desmond Clark-Walker, and David R Wolstenholme. Mitochondrial dna of the coral sarcophyton glaucum contains a gene for a homologue of bacterial muts: a possible case of gene transfer from the nucleus to the mitochondrion. *Journal of molecular evolution*, 46:419–431, 1998.

- [94] Thomas W Quinn and Allan C Wilson. Sequence evolution in and around the mitochondrial control region in birds. *Journal of molecular evolution*, 37:417–425, 1993.
- [95] Daniel Rubinoff and Brenden S Holland. Between two extremes: mitochondrial dna is neither the panacea nor the nemesis of phylogenetic and taxonomic inference. *Systematic biology*, 54(6):952–961, 2005.
- [96] Diego San Mauro, David J Gower, Tim Massingham, Mark Wilkinson, Rafael Zardoya, and James A Cotton. Experimental design in caecilian systematics: phylogenetic information of mitochondrial genomes and nuclear rag1. *Systematic Biology*, 58(4):425–438, 2009.
- [97] David Sankoff. Edit distance for genome comparison based on non-local operations. In *Combinatorial Pattern Matching: Third Annual Symposium Tucson, Arizona, USA, April 29–May 1, 1992 Proceedings 3*, pages 121–135. Springer Berlin Heidelberg, 1992.
- [98] Sven Schrinner, Manisha Goel, Michael Wulfert, Philipp Spohr, Korbinian Schneeberger, and Gunnar W. Klau. The longest run subsequence problem. In *Workshop on Algorithms in Bioinformatics*, 2020.
- [99] Gerald S Shadel and David A Clayton. Mitochondrial dna maintenance in vertebrates. *Annual review of biochemistry*, 66(1):409–435, 1997.
- [100] Andrew Sharp, Devin Locke, Sean McGrath, Ze Cheng, Jeffrey Bailey, Rhea Vallente, Lisa Pertz, Royden Clark, Stuart Schwartz, Rick Segraves, Vanessa Oseroff, Daniel Pinkel, and Evan Eichler. Segmental duplications and copy-number variation in the human genome. *American journal of human genetics*, 77:78–88, 08 2005.
- [101] Michael J Smith, Allan Arndt, Sharon Gorski, and Elizabeth Fajber. The phylogeny of echinoderm classes based on mitochondrial gene arrangements. *Journal of Molecular Evolution*, 36:545–554, 1993.
- [102] Sérgio N Stampar, Michael B Broe, Jason Macrander, Adam M Reitzel, Mercer R Brugler, and Marymegan Daly. Linear mitochondrial genome in anthozoa (cnidaria): a case study in ceriantharia. *Scientific reports*, 9(1):6094, 2019.
- [103] Philip J. Stephens, David J. McBride, Meng-Lay Lin, Ignacio Varela, Erin D. Pleasance, Jared T. Simpson, Lucy A. Stebbings, Catherine Leroy, Sarah Edkins, Laura J. Mudie, Chris D. Greenman, Mingming Jia, Calli Latimer, Jon W. Teague, King Wai Lau, John Burton, Michael A. Quail, Harold P Swerdlow, Carol M. Churcher, Rachael Natrajan, Anieta M. Sieuwerts, John W. M. Martens, Daniel P. Silver, Anita Langerød, Hege Russnes, John A. Foekens, Jorge Sergio Reis-Filho, Laura J. van ’t Veer, Andrea L. Richardson, Anne-Lise Børreson-Dale, Peter J. Campbell, P. Andrew Futreal, and Michael R. Stratton. Complex landscapes of somatic rearrangement in human breast cancer genomes. *Nature*, 462:1005 – 1010, 2009.

- [104] Jack W. Szostak and Ray J. Wu. Unequal crossing over in the ribosomal dna of *saccharomyces cerevisiae*. *Nature*, 284:426–430, 1980.
- [105] Jan-Willem Taanman. The mitochondrial genome: structure, transcription, translation and replication. *Biochimica et Biophysica Acta (BBA)-Bioenergetics*, 1410(2):103–123, 1999.
- [106] Li-San Wang and Tandy J. Warnow. Distance-based genome rearrangement phylogeny. *Journal of Molecular Evolution*, 63:473–483, 2006.
- [107] Lusheng Wang and Binhai Zhu. On the tractability of maximal strip recovery. *Journal of computational biology : a journal of computational molecular cell biology*, 17 7:907–14, 2009.
- [108] Lusheng Wang and Binhai Zhu. Algorithms and hardness for the longest common subsequence of three strings and related problems. In *International Symposium on String Processing and Information Retrieval*, pages 367–380. Springer, 2023.
- [109] Geoffrey A. Watterson, Warren J. Ewens, Thomas E. Hall, and Adrienne Morgan. The chromosome inversion problem. *Journal of Theoretical Biology*, 99:1–7, 1982.
- [110] Ming wei Wang. On the irregularity of the duplication closure. *Bull. EATCS*, 70:162–163, 2000.
- [111] David R. Wolstenholme. Animal mitochondrial dna: Structure and evolution. volume 141 of *International Review of Cytology*, pages 173–216. Academic Press, 1992.
- [112] Nika Yamazaki, Rei Ueshima, Jonathan A Terrett, Shin-ichi Yokobori, Masayuki Kaifu, Ryoko Segawa, Takanori Kobayashi, Ken-ichi Numachi, Takuya Ueda, Kazuya Nishikawa, et al. Evolution of pulmonate gastropod mitochondrial genomes: comparisons of gene organizations of euhadra, cepaea and albinaria and implications of unusual trna secondary structures. *Genetics*, 145(3):749–758, 1997.
- [113] Jia-Yong Zhang, Le-Ping Zhang, Dan-Na Yu, Kenneth B Storey, and Rong-Quan Zheng. Complete mitochondrial genomes of *nanorana taihangnica* and *n. yunnanensis* (anura: Dicroglossidae) with novel gene arrangements and phylogenetic relationship of dicroglossidae. *BMC Evolutionary Biology*, 18(1):1–13, 2018.
- [114] Chunfang Zheng, P. Kerr Wall, James H. Leebens-Mack, Claude de Pamphilis, Victor A. Albert, and David Sankoff. Gene loss under neighborhood selection following whole genome duplication and the reconstruction of the ancestral populus genome. *Journal of bioinformatics and computational biology*, 7 3:499–520, 2009.
- [115] Chunfang Zheng, Qian Zhu, and David Sankoff. Removing noise and ambiguities from comparative maps in rearrangement analysis. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 4:515–522, 2007.