DUPLICATIONS AND DELETIONS IN GENOMES: THEORY AND APPLICATIONS

by

Peng Zou

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 2022

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Binhai Zhu for his patient guidance and encouragement to complete my PhD dissertation.

I also would like to thank all my committee members, Dr. Brittany Terese Fasy, Dr. David L. Millman and Dr. Sean Yaw for their helpful comments.

I also would like to express my gratitude to my family who has always supported and encouraged me.

TABLE OF CONTENTS

TABLE OF CONTENTS – CONTINUED

LIST OF TABLES

LIST OF TABLES – CONTINUED

Table Page

LIST OF FIGURES

LIST OF ALGORITHMS

ABSTRACT

In computational biology, duplications and deletions in genome rearrangements are important to understand an evolutionary process. In cancer genomics research, intra-tumor genetic heterogeneity is one of the central problems. Gene duplications and deletions are observed occurring rapidly in cancer during tumour formation. Hence, they are recognized as critical mutations of cancer evolution. Understanding these mutations are important to understand the origins of cancer cell diversity which could help with cancer prognostics as well as drug resistance explanation.

In this dissertation, first, we prove that the tandem duplication distance problem is NP-complete, even if $|\Sigma| \geq 4$, settling a 16-year old open problem. And we obtain some positive results by showing that if one of the input sequences, S, is exemplar, then one can decide if S can be transformed into T using at most k tandem duplications in time $2^{O(k^2)} + poly(n)$. Motivated by computing duplication patterns in sequences, a new fundamental problem called the longest letter-duplicated subsequence (LLDS) is investigated. We investigate several variants of this problem.

Due to fast mutations in cancer, genome rearrangements on copy number profiles are used more often than genome themselves. We explore the Minimum Copy Number Generation problem. We prove that it is NP-hard to even obtain a constant factor approximation. We also show that the corresponding parameterized version is $W[1]$-hard. These either improve the previous hardness result or solve an open problem. And we then give a polynomial algorithm for the Copy Number Profile Conforming problem.

Finally, we investigate the pattern matching with 1-reversal distance problem. With the known results on Longest Common Extension queries, one can design an $O(n+m)$ time algorithm for this problem. However, we find empirically that this algorithm is very slow for small m. We then design an algorithm based on the Karp-Rabin fingerprints which runs in an expected $O(nm)$ time. The algorithms are implemented and tested on real bacterial sequence dataset. The empirical results shows that the shorter the pattern length is (i.e., when $m < 200$), the more substrings with 1-reversal distance the bacterial sequences have.

CHAPTER ONE

INTRODUCTION

1.1 Motivation

During a biological evolution, genome rearrangements could occur in genomes in the form of duplications, deletions, insertions and reversals, which change the content of a genome or the order of the genes on a genome. In 1930s, Dobzhansky and Sturtevant [30] discovered 17 inversions/reversals between the arrangements of two Drosophila species. Also, it is well known that human and mice share very much the same genes, but the gene orders are different. With the advent of genome sequencing, genome analysis based on genome rearrangements has been done in various areas. For example, genome analysis researches have been done for the evolution of mitochondrial genomes of plants [9, 76, 77], fungi [16] and animals [45, 81]; chloroplast genomes [47, 56, 70]; genomes of lambdoid bacteriophages [19] and small viruses [48, 58]; and mammalian chromosomes [71, 72, 95].

It is now widely accepted that cancers arise from an accumulation of mutation events, where duplication is an important mutation event of evolution. There are two kinds of duplications: arbitrary segmental duplications (i.e., select a segment and paste it somewhere else) and tandem duplications (which is in the form of $AXB \rightarrow AXXB$, where X is any segment of the input sequence). It is known that the former duplications occur frequently in cancer genomes [26, 73, 88]. On the other hand, frequent tandem duplication and deletion events are observed occurring rapidly, which play an important role in tumour evolution and progression in many cancers. In fact, as early as in 1980, Szostak and Wu provided evidence that gene duplication is the main driving force behind evolution, and the majority

of duplications are tandem [89]. Tandem duplications are known to occur either at small scale at the nucleotide level, or at large scale at the genome level [20, 21, 22, 67, 88]. For instance, it is known that Huntington's disease is associated with the duplication of 3 nucleotides `CAG` [74], whereas, at the genome level, tandem duplications are known to involve multiple genes during cancer progression [75]. Furthermore, gene duplication is believed to be the main driving force behind evolution, and the majority of duplications affecting organisms are believed to be of the tandem type. As a result, around 3% of the human genome are formed of tandem repeats [89].

Independently, tandem duplications were also studied as early as in 1984 in *copying systems* [33]; as well as in formal languages [14, 29, 91]. In 2004, Leupold et al. posed a fundamental question regarding tandem duplications: what is the complexity to compute the tandem duplication distance between two sequences A and B (i.e., the minimum number of tandem duplications to convert A to B). In Chapter 2, this important fundamental question, the tandem duplication distance problem, is studied.

The newly tandem duplicated genes could have the same function as the original genes. However, it is also possible that these copied genes evolves with a new function by other mutation events, such as deletions [36]. The deletions could break the continuous tandem duplicated genes into several pieces. Identifying these functional genes could potentially help to understand the genome evolution and function. Motivated by the above applications, new problems related to duplications are proposed and studied in Chapter 3. Given a sequence S of length n, a letter-duplicated subsequence (LDS) of S is a subsequence of S in the form $x_1^{d_1}x_2^{d_2}\cdots x_k^{d_k}$ 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). Naturally, the problem of computing the longest letter-duplicated subsequence (LLDS) of S can be defined, and a simple linear time algorithm can be obtained. We then study important variants around the fundamental $LLDS$ problems, focusing on the constrained and weighted cases. In [85], S. Schrinner et al. studied another

variant of the LLDS problem, which has application in genome assembly.

It is known for some types of cancers, such as high-grade serous ovarian cancer (HGSOC), that heterogeneity is mainly acquired through genome rearrangements and endoreduplications, the replication of the genome without the usual mitosis reproduction cycle. These result in aberrant *copy number profiles* (CNPs), nonnegative integer vectors representing the numbers of genes occurring in a genome [73]. As we know, in the late stage of certain types of cancer, the genomes are progressing rapidly by segmental duplications and deletions, and hence obtaining the exact sequences of genome rearrangements become difficult. Instead, the number of copies of important segments can be predicted from expression analysis and carries important biological information. Therefore, significant research has recently been devoted to the analysis of genomic data represented as CNP's.

To understand how cancer progresses, an evolutionary tree is certainly desirable, and inferring such a tree based on these genomic data becomes a new problem. In [86], Schwarz et al. proposed a way to construct a phylogenetic tree directly from integer copy number profiles, the underlying problem being to convert a CNP into another one using the minimum number of duplication/deletions [87]. This was recently followed with several other distance measures between CNPs that can be used to reconstruct cancer phylogenies [28, 35, 82, 93, 94]. In [34], a more complex distance computation was used as a subroutine to compute an ancestor profile given a set of k profiles. The problem was shown to be NP-hard, though an ILP formulation was given. In fact, Chowdhury et al. considered copy number changes at different levels, from single gene, single chromosome to whole genome, to enhance the tumor phylogeny reconstruction [24]. In [79], another fundamental problem was proposed. The motivation is that in the early stages of cancer, when large number of endoreduplications are still rare, genome sequencing is still possible. However, in the later stage we might only be able to obtain cancer genomic data in the form of CNPs. This leads to the problem of comparing a sequenced genome with a genome with only copy-number information. In

Chapter 4, we investigate the *Minimum Copy Number Generation (MCNG)* problem, which was posed by Qingge et al. in 2018 [79].

Pattern matching is a very practical operation to identify some subsequences. In many applications like biology and communications, the occurrence of a copy of the pattern could be slightly altered by letter mutation and corruption. Therefore, the problem of pattern matching with k mismatches has been investigated rigorously as well. The most widely-used distance measure for the pattern matching with k mismatches problem is the Hamming distance. However, the genome rearrangement distance is more meaningful for genome sequences. As we know, the reversals are common operations on genomes. Computing the reversal distance between two unsigned genomes, possibly with letter/gene duplications, is NP-hard [25]. Motivated by the above applications, in Chapter 5, we consider the pattern matching problem under 1-reversal distance, where we want to list all substrings of input sequence which have a reversal distance at most 1 to the given pattern.

1.2 The Related Genome Rearrangement Events

In this section, we first describe the duplications, tandem duplications and deletions on a sequence and on a copy number profile.

1.2.1 Genome Rearrangements on a Sequence

We now describe several related rearrangement events on a sequence. A genome G is a string, i.e. a sequence of characters, all of which belong to some alphabet Σ. Given a genome $S = s_1, s_2, \ldots s_n$, a `reversal` takes a substring $S' = \underline{s_i s_{i+1} \ldots s_j}$ and reverses the order of all characters of S'. A `duplication` is to copy a substring $S' = \underline{s_i s_{i+1} \ldots s_j}$ and insert it before some letter s_k with $k \leq i$ or after some letter s_k with $k \geq j$. If S' is inserted right before s_i or right after s_j, the duplication is `tandem duplication`. A `deletion` is to delete S' from S. An example for all these rearrangement operations are shown in Table 1.1.

Source	Target	Operations
$\langle a, c, \underline{g, c, t}, a, g\rangle$	$\langle a, c, \underline{g, c, t}, a, \underline{g, c, t}, g\rangle$	duplication: $(3, 5, 6)$
$\langle a, c, \underline{g, c, t}, a, g\rangle$	$\langle a, c, \underline{g, c, t, g, c, t}, a, g\rangle$	tandem duplication: $(3, 5)$
$\langle a, c, \underline{g, c, t}, a, g\rangle$	$\langle a, c, \underline{t, c, g}, a, g\rangle$	reversal: $(3, 5)$
$\langle a, c, \underline{g, c, t}, a, g\rangle$	$\langle a, c, a, g\rangle$	deletion: $(3, 5)$

Table 1.1: An example of duplication, tandem duplication, reversal and deletion events on a sequence.

Copy Number Profile (Source)	Copy Number Profile (Target)	Operations
$\langle 2, \underline{3, 1, 2, 2}, 1\rangle$	$\langle 2, \underline{4, 2, 3, 3}, 1\rangle$	duplication: $\langle 3, 1, 2, 2\rangle$
$\langle 2, \underline{3, 2, 4, 2}, 1\rangle$	$\langle 2, \underline{2, 1, 3, 1}, 1\rangle$	deletion: $\langle 3, 2, 4, 2\rangle$

Table 1.2: An example for duplication and deletion events on a CNP

1.2.2 Duplication and Deletion on a Copy Number Profile

In [86], the evolution process was modeled as the genome rearrangements on a copy number profiles. A Copy Number Profile (or CNP) on Σ is a vector $\vec{c} = \langle c_1, \ldots, c_{|\Sigma|}\rangle$ that associates each character s_i of the alphabet with a non-negative integer $c_i \in \mathbb{N}$. Given a copy number profile $C = \langle c_1, c_2, \ldots, c_t\rangle$, a *duplication* is to change a substring of C, $\langle c_i, c_{i+1} \ldots c_j\rangle$, into $\langle c'_i, c'_{i+1}, c'_j\rangle$, such that $c'_l = c_l + 1$ if $c_l > 0$, for $l = i \ldots j$, but if $c_l = 0$, then $c' = 0$. A *deletion* changes a substring of C, $\langle c_i, c_{i+1} \ldots c_j\rangle$, into $\langle \max\{c_i - 1, 0\}, \max\{c_{i+1} - 1, 0\}, \max\{c_j - 1, 0\}\rangle$. An example is shown in Table 1.2.

CHAPTER TWO

THE TANDEM DUPLICATION DISTANCE PROBLEM

2.1 Introduction

Tandem duplications have received significant attention in the last decades, both in practice and theory. The combinatorial aspects of tandem duplications have been studied extensively by computational biologists [13, 40, 43, 64, 83], one question of interest being to reconstruct the evolution of a cluster of tandem repeats by duplications that could have given rise to the observed sequences. In parallel, various formal language communities [29, 69, 91] have investigated the expressive power of tandem duplications on strings.

From the latter perspective, a natural question arises: given a string S, what is the language that can be obtained starting from S and applying (any number of) tandem duplications, i.e., rules of the form $AXB \to AXXB$, where X can be any substring of S? This question was first asked in 1984 in the context of so-called `copying systems` [33]. Combined with results from [14], it was shown that this language is regular if S is on a binary alphabet, but not regular for larger alphabets. These results were rediscovered 15 years later in [29, 91]. In [69], it was shown that the membership, inclusion, and regularity testing problems on the language defined by S can all be decided in linear time (still on binary alphabets). In [51, 68, 69], similar problems are also considered on non-binary alphabets, when the length $|X|$ of duplicated strings is bounded by a constant. More recently, Cho et al. [23] introduced a tandem duplication system where the `depth` of a character, i.e., the number of "generations" it took to generate it, is considered. In [37, 52], the authors studied the `expressive power` of tandem duplications, a notion based on the subsequences that can be obtained from various types of copying mechanisms.

More directly related to our work, Alon et al. [2] recently investigated the minimum

number of duplications required to transform a string S into another string T. We call this number the `Tandem Duplication (TD) distance`. More specifically, the authors showed that on binary strings, the maximum TD distance between a square-free string S and a string T of length n is $\Theta(n)$. They also mentioned the unsolved algorithmic problem of computing the TD distance between S and T. In fact, in 2004 this question was posed in [69] (pp. 306, Open Problem 3) by Leupold et al. and has remained open ever since. We settle this open problem in this chapter first for an unbounded alphabet, then we extend the proof to a finite alphabet of size 4. Our technique is different from that used in [2], which only works for binary strings.

On the other hand, the TD distance is one of the many ways of comparing two genomes represented as strings in computational biology, other notable examples include breakpoint [44] and transpositions distance, the latter having recently been shown NP-hard in a celebrated paper of Bulteau et al. [17]. The TD distance has itself received special attention recently, owing to its role in cancer evolution [79].

2.2 Preliminaries

Let $[n]$ denote the set of integers $\{1, 2, \ldots, n\}$. Unless stated otherwise, all the strings in this chapter are on an alphabet denoted as Σ. If S_1 and S_1 are two strings, we usually denote their concatenation by S_1S_2. For a string S over an alphabet Σ, we write $\Sigma(S)$ for the subset of characters of Σ that have at least one occurrence in S. A string S is called *exemplar* if $|S| = |\Sigma(S)|$, i.e., each character presented in S occurs exactly once. A *substring* of S is a contiguous sequence of characters within S. A *prefix* (resp., *suffix*) is a substring that occurs at the beginning (resp., end) of S. A *subsequence* of S is a string that can be obtained by successively deleting characters from S.

A *tandem duplication* (TD) is an operation on string S that copies a substring X of S and inserts the copy after the occurrence of X in S. In other word, a TD transforms

$S = AXB$ into $AXXB$, where A, X and B are strings. Given another string T, we write $S \Rightarrow T$ if there exist strings A, B, X such that $S = AXB$ and $T = AXXB$. More generally, we write $S \Rightarrow_k T$ if there exist $S_1, \ldots S_{k-1}$ such that $S \Rightarrow S_1 \Rightarrow \ldots \Rightarrow S_{k-1} \Rightarrow T$. We also write $S \Rightarrow_* T$ if there exist some k such that $S \Rightarrow_k T$.

Definition 1. *The TD distance $dist_{TD}(S, T)$ between two strings S and T is the minimum value of k satisfying $S \Rightarrow_k T$. If $S \Rightarrow_* T$ does not hold, then $dist_{TD}(S, T) = \infty$.*

We use the term *distance* here to refer to the number of TD operations from a string S to another string T, but one may note that TD is not a metric in the formal sense. In particular, $dist_{TD}$ is not symmetric since duplications can only increase the length of a string.

A *square string* is a string of the form XX, i.e., a concatenation of two identical substrings. Given a string S, a *contraction* is the reverse of a tandem duplication. That is, it takes a square string XX contained in S and deletes one of the two copies of X. We write $T \rightarrowtail S$ if there exist strings A, B, X such that $T = AXXB$ and $S = AXB$. We also define $T \rightarrowtail_k S$ and $T \rightarrowtail_* S$ for contractions analogously as for TDs (note that $T \rightarrowtail_k S$ if and only if $S \Rightarrow_k T$ and $T \rightarrowtail_* S$ if and only if $S \Rightarrow_* T$). When there is no possible confusion, we sometimes write $T \rightarrowtail S$ instead $T \rightarrowtail_* S$.

We have the following problem.

Definition 2. *The k-*`Tandem Duplication (`*k*`-TD)` *problem:*
`Input:` *two strings S and T over the same alphabet Σ and an integer k.*
`Question:` *Is $dist_{TD}(S, T) \leq k$?*

In the `Exemplar-`*k*`-TD` variant of this problem, S is required to be exemplar. In either variant, we may call S the *source string* and T the *target string*. We will often use the fact that S and T form a `YES` instance if and only if T can be transformed into S by a sequence of at most k contractions. See Table. 2.1 for a simple example.

Sequence	Operations
$T = \langle a, c, \underline{g, g}, a, c, g\rangle$	contraction on $\langle g, g\rangle$
$T' = \langle \underline{a, c, g, a, c, g}\rangle$	contraction on $\langle a, c, g, a, c, g\rangle$
$S = \langle a, c, g\rangle$	

Table 2.1: An example for transforming sequence T to S by two contractions.

We recall that although we study the minimization problem here, it is unknown whether the question $S \Rightarrow_* T$ can be decided in polynomial time. Nonetheless, our NP-hardness reduction applies to 'promise' instances in which $S \Rightarrow_* T$ always holds.

2.3 The Cost-Effective Subgraph Problem

To facilitate the presentation of our hardness proof, we first make an intermediate reduction using the cost-effective subgraph problem, which we then reduce to the promise version of the exemplar-k-TD problem.

Suppose we are given a graph $G = (V, E)$ and an integer cost $c \in \mathbb{N}_{>0}$. For a subset $X \subseteq V$, let $E(X) = \{uv \in E : u, v \in X\}$ denote the edges inside of X. The *cost* of X is defined as

$$cost(X) = c \cdot (|E(G)| - |E(X)|) + |X| \cdot |E(X)|.$$

The formal definition of the cost-effective subgraph problem is defined as follows:

Definition 3. *The cost-effective subgraph problem: Given a graph G and an integer cost c, the question asks for a subset X of minimum cost.*

In the decision version of the problem, we are given an integer r and we want to know if there is a subset X whose cost is at most r. Observe that $X = \emptyset$ or $X = V$ are possible solutions.

The idea is that each edge "outside" of X costs c and each edge "inside" costs $|X|$. Therefore, we pay for each edge not included in X, but if X gets too large, we pay more for edges in X. We must therefore find a balance between the size of X and its number of edges. The connection with the TD problem can be roughly described as follows: in our reduction, we have many substrings that need to be deleted through contractions. We have to choose an initial set of contractions X and then, each substring have two ways to be contracted: one way required c contractions, and the other requires $|X|$.

An obvious feasible solution for a Cost-Effective Subgraph is to take $X = \emptyset$, which is of cost $c|E(G)|$. Another formulation of the problem could be whether there is a subset X of cost at most $c|E(G)| - p$, where p can be seen as a "profit" to maximize. Treating c and p as parameters, we show the NP-hardness and W[1]-hardness in parameters $c + p$ of the `Cost-Effective Subgraph` problem (we do not study the parameter r). Our reduction to the TD problem does not preserve W[1]-hardness and we only use the NP-hardness, but the W[1]-hardness might be of independent interest.

Before proceeding, we briefly argue the relevance of parameter c in the W[1]-hardness. If c is a fixed constant, then we may assume that any solution X satisfies $|X| \leq c$. This is because if $|X| > c$, every edge included in X costs more than c and putting $X = \emptyset$ yields a lower cost. Thus for fixed c, it suffices to brute-force every subset X of size at most c and we get a $n^{O(c)}$ time algorithm. Our W[1]-hardness shows that it is difficult to remove this exponential dependence between n and c.

Theorem 1. *The Cost-Effective Subgraph problem is NP-hard and W[1]-hard for parameter* $c + p$.

Proof. We reduce from CLIQUE. In this classic problem, we are given a graph G and an integer k, and must decide whether G contains a clique of size at least k, where a clique is a set of vertices in which every pair shares an edge. This problem is NP-hard [54] and also

W[1]-hard in parameter k [31]. We will assume that k is even (which does not alter either hardness results).

Let $(G,\ k)$ be a CLIQUE instance, letting $n := |V(G)|$ and $m := |E(G)|$. The graph in our Cost-Effective Subgraph instance is also G. We set the cost $c = 3k/2$, which is an integer since k is even, and set

$$r := c\left(m - \binom{k}{2}\right) + k\binom{k}{2} = cm + \binom{k}{2}(k-c) = cm - \frac{k}{2}\binom{k}{2}$$

we ask whether G admits a subgraph X satisfying $cost(X) \leq r$. We show that (G, k) is a YES instance to CLIQUE if and only if G contains a set $X \subseteq V(G)$ of cost at most r. This will prove both NP-hardness and W[1]-hardness in $c + p$ (noting that here $p = k/2\binom{k}{2}$).

The forward direction is easy to see. If G is a YES instance, it has a clique X of size exactly k. Since $|E(X)| = \binom{k}{2}$, the cost of X is precisely r.

Let us consider the converse direction. Assume that (G, k) is a NO instance of CLIQUE. Let $X \subseteq V(G)$ be any subset of vertices. We will show that $cost(X) > r$. There are 3 cases to consider depending on $|X|$.

Case 1: $|X| = k$. Since G is a NO instance, X is not a clique and thus $|E(X)| = \binom{k}{2} - h$, where $h > 0$. We have that $cost(X) = c(m-\binom{k}{2}+h)+k(\binom{k}{2}-h) = cm+\binom{k}{2}(k-c)+h(c-k) = r + h(c-k)$. Since $c > k$ and $h > 0$, the cost of X is strictly greater than r.

Case 2: $|X| = k + l$ for some $l > 0$. Denote $|E(X)| = \binom{k+l}{2} - h$, where $h \geq 0$ (actually, $h > 0$ but we do not bother). The cost of X is

$$\begin{aligned} cost(X) &= c\left(m - \binom{k+l}{2} + h\right) + (k+l)\left(\binom{k+l}{2} - h\right) \\ &= cm + \binom{k+l}{2}(k+l-c) + h(c-k-l) \\ &= cm + \binom{k+l}{2}(l-k/2) + h(k/2-l) \end{aligned}$$

Consider the difference

$$\begin{aligned} cost(X) - r &= \binom{k+l}{2}(l - k/2) - (-k/2)\binom{k}{2} + h(k/2 - l) \\ &= \frac{3kl^2}{4} - \frac{kl}{4} + \frac{l^3}{2} - \frac{l^2}{2} + h(k/2 - l) \end{aligned}$$

If $k/2 - l \geq 0$, then the difference is clearly above 0 regardless of h, and then $cost(X) > r$ as desired. Thus we may assume that $k/2 - l < 0$. In this case, we may assume that $h = \binom{k+l}{2}$, as this minimizes $cost(X)$. But in this case,

$$cost(X) = cm + \binom{k+l}{2}(l - k/2) + \binom{k+l}{2}(k/2 - l) = cm > r,$$

which concludes this case.

Case 3: $|X| = k - l$, with $l > 0$. If $k = l$, then $X = \emptyset$ and $cost(X) = cm > r$. So we assume $k > l$. Put $|E(X)| = \binom{k-l}{2} - h$, where $h \geq 0$. We have

$$\begin{aligned} cost(X) &= c\left(m - \binom{k-l}{2} + h\right) + (k - l)\left(\binom{k-l}{2} - h\right) \\ &= cm + \binom{k-l}{2}(k - l - c) + h(c - k + l) \\ &= cm + \binom{k-l}{2}(-k/2 - l) + h(k/2 + l) \end{aligned}$$

The difference with this cost and r is

$$\begin{aligned} cost(X) - r &= \binom{k-l}{2}(-k/2 - l) - (-k/2)\binom{k}{2} + h(k/2 + l) \\ &= \frac{3kl^2}{4} + \frac{kl}{4} - \frac{l^3}{2} - \frac{l^2}{2} + h(k/2 + l) \\ &> \frac{1}{4}(3l^3 + l^2) - \frac{1}{2}(l^3 + l^2) \geq 0 \end{aligned}$$

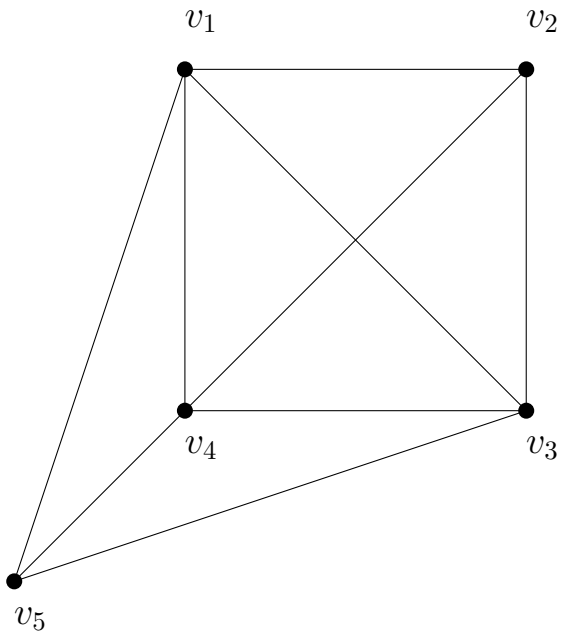

Figure 2.1: A clique instance (G, k), where $k = 4$.

the latter since $k > l \geq 1$. Again, it follows that $cost(X) > r$.

□

An example. In Figure 2.1, we are given a graph $G = \{v_1, v_2, v_3, v_4, v_5\}$ and an integer $k = 4$ for the clique problem. For the cost-effective subgraph problem, the $c = 3k/2 = 6$. The *cost* of set $X = \{v_1, v_2, v_3, v_4\}$ which is a clique in G is computed as follows:

$$cost(X) = c(9 - 6) + 4 \cdot 6 = 18 + 24 = 42.$$

And the *cost* of another set $\bar{X} = \{v_1, v_2, v_3, v_5\}$ which is not a clique in G is calculated as follows:

$$cost(\bar{X}) = c(9 - 5) + 4 \cdot 5 = 24 + 20 = 44.$$

In the above example, we show that the clique set achieves the lowest cost value.

2.4 The Tandem Duplication Distance Problem

2.4.1 NP-hardness of Exemplar-k-TD

In this subsection, we show the reduction form the *Cost-Effective Subgraph* problem in detail. Since the reduction is somewhat technical, we provide an overview of the techniques that we use. Let (G, c, r) be a *Cost-Effective Subgraph* instance where c is the cost and r the optimization value, and with vertices $V(G) = \{v_1, \ldots, v_n\}$. We construct strings S and T and argue on the number of contractions to go from T to S. We would like our source string to be $S = x_1x_2 \ldots x_n$, where each x_i is a distinct character that corresponds to vertex v_i. Let S' be obtained by doubling every x_i, i.e., $S' = x_1x_1x_2x_2 \ldots x_nx_n$. Our goal is to put $T = S'E_1E_2 \ldots E_m$, where each E_i is a substring gadget corresponding to edge $e_i \in E(G)$ that we must remove to go from T to S. Assuming that there is a sequence of contractions that transforms T into S, we make it so that we first want to contract some, but not necessarily all, of the doubled x_i's of S', resulting in another string S''. Let t be the number of x_i's contracted from S' to S''. For instance, we could have $S'' = x_1x_1x_2x_3x_3x_4x_5x_5$, where only x_2 and x_4 we contracted, and thus $t = 2$. The idea is that these contracted x_i's correspond to the vertices of a cost-effective subgraph. After T is transformed to $S''E_1 \ldots E_m$, we then force each E_i to use S'' to contract it. For $m = 3$, a contraction sequence that we would like to enforce would take the form

$$\underline{S'}E_1E_2E_3 \rightarrowtail \underline{S''E_1}E_2E_3 \rightarrowtail \underline{S''E_2}E_3 \rightarrowtail \underline{S''E_3} \rightarrowtail \underline{S''} \rightarrowtail S,$$

where we underline the substring affected by contractions at each step. We make it so that when contracting $S''E_iE_{i+1} \ldots E_m$ into $S''E_{i+1} \ldots E_m$, we have two options. Suppose that v_j, v_k are the endpoints of edge e_i. If, in S'', we had chosen to contract x_j and x_k, we can contract E_i using a sequence of t moves. Otherwise, we must contract E_i using another more

costly sequence of c moves. The total cost to eliminate the E_i gadgets is $c(m-e)+te$, where e is the number of edges that can be contracted using the first choice, i.e., for which both endpoints were chosen in S''.

Unfortunately, constructing S' and the E_i's to implement the above idea is not straightforward. The main difficulty lies in forcing an optimal solution to behave as we describe, i.e., enforcing going from S' to S'' first, enforcing the E_i's to use S'', and enforcing the two options to contract E_i with the desired costs. In particular, we must replace the x_i's by carefully constructed substring X_i. We must also repeat the sequence of E_i's a certain number p times. We now proceed with the technical details.

Theorem 2. *The Exemplar-k-TD problem is NP-complete, even if for the given string S and T, $S \Rightarrow_* T$ is guaranteed to hold.*

Proof. To see that the problem is in NP, note that $dist_{TD}(S,T) \leq |T|$ since each contraction from T to S removes a character. Thus a sequence of contractions can serve as a certificate, has polynomial size and is easy to verify.

For hardness, we reduce from the Cost-Effective Subgraph problem. Let (G,c,r) be an instance of Cost-Effective Subgraph problem, letting $n := |V(G)|$ and $m := |E(G)|$. Here c is the "outsider edge" cost and we ask whether there is subset $X \subseteq V(G)$ such that $c(m-|E(X)|)+|X||E(X)| \leq r$. We denote $V(G) = \{v_1,\ldots,v_n\}$ and $E(G) = \{e_1,\ldots,e_m\}$. The ordering of vertices and edges is arbitrary but remains fixed for the remainder of the proof. For convenience, we allow the edge indices to loop through 1 to m, and so we put $e_i = e_{i+lm}$ for any integer $l \geq 0$. Thus we may sometimes refer to an edge e_k with an index $k > m$, meaning that e_k is actually the edge $e_{((k-1) \bmod m)+1}$.

The construction. Let us first make an observation. If we take an exemplar string $X = x_1 \ldots x_l$ (i.e., a string in which no character occurs twice), we can double its characters and obtain a string $X' = x_1x_1 \ldots x_lx_l$. The length of X' is only twice that of X and

$dist_{TD}(X, X') = l$, i.e., going from X' to X requires l contractions. We will sometimes describe pairs of strings X and X' at distance l without explicitly describing X and X', but the reader can assume that X starts as an exemplar string of length l and we obtain X' by doubling each character, as above.

Now we show how to construct S and T. First let $d = m + 1$ and $p = m(n + m)^{10}$. The exact values of d and p are not crucial and will only refer to them when needed: for the most part, it is enough to think of d and p are simply "larger enough". Note however that p is a multiple of m. For later reference, the value of k we will use in the reduction is $k = p/m\dot{d}(r + nm) + 4cdn$.

Instead of doubling x_i's as in the intuition paragraph above, we will duplicate some characters d times. Moreover, we can't create a string T that behaves exactly as described above, but we will show that we can append p copies of carefully crafted substring to obtain the desired result. We need d and p to be high enough so that "enough" copies behave as we desire.

For each $i \in [n]$, define an exemplar string X_i of length d. Moreover, create enough characters so that no two X_i strings contain a character in common. Let X_i^d be a string satisfying $dist_{TD}(X_i, X_i^d) = d$.

Then for each $j \in \{0, 1, \ldots, 2p\}$, define an exemplar string B_j. Ensure that no B_j contains a character from an X_i string, and no two B_j's contain a common character. The B_j strings can consist of a single character, with the exception of B_0 and B_1 which are special. We assume that for B_0 and B_1, we have strings B_0^* and B_1^* such that

$$dist_{TD}(B_0, B_0^*) = dc + 2d - 2$$

$$dist_{TD}(B_1, B_1^*) = dn + 2d - 1$$

Again, this can be done using the doubling trick on exemplar strings. The B_j's are the

building blocks of larger strings. For each $q \in [2p]$, define

$$\mathcal{B}_q = B_q B_{q_1} \dots B_2 B_1 B_0 \qquad \mathcal{B}_q^0 = B_q B_{q_1} \dots B_2 B_1 B_0^*$$
$$\mathcal{B}_q^1 = B_q B_{q_1} \dots B_2 B_1^* B_0 \qquad \mathcal{B}_q^{01} = B_q B_{q_1} \dots B_2 B_1^* B_0^*$$

These strings are used as "blocks" and prevent certain contractions from happening. Note that $\mathcal{B}_q^0$ and $\mathcal{B}_q^1$ can be turned into $\mathcal{B}_q$ using $dc + 2d - 2$ contractions and $dn + 2d - 1$ contractions, respectively. Moreover, $\mathcal{B}_q^{01}$ can be turned into $\mathcal{B}_q^0$ using $dn+2d-1$ contractions and into $\mathcal{B}_q^1$ using $dc + 2d - 2$ contractions.

Also define the strings

$$\mathcal{X} = X_1 X_2 \dots X_n \qquad \mathcal{X}^d = X_1^d X_2^d \dots X_n^d$$

and for edge $e_q = v_i v_j$ with $q \in [p]$ whose endpoints are v_i and v_j, define

$$\mathcal{X}_{e_q} = X_1^d \dots X_{i-1}^d X_i^d X_{i+1}^d \dots X_{j-1}^d X_j^d X_{j+1}^d \dots X_n^d$$

Thus in $\mathcal{X}_{e_q}$, all X_k substrings are turned into X_k^d, except X_i and X_j.

Finally, define a new additional character Δ, which will be used to separate some of the components of our strings. We can now define S and T. We have

$$S = \mathcal{B}_{2p} \mathcal{X} \Delta = B_{2p} B_{2p-1} \dots B_2 B_1 B_0 X_1 X_2 \dots X_n \Delta$$

It follows from the definitions of $\mathcal{B}_{2p}$, $\mathcal{X}$ and Δ that S is exemplar. Now for $i \in [p]$, define

$$E_i := \mathcal{B}_i^{01} \mathcal{X}_{e_i} \Delta \mathcal{B}_{2p}^1 \mathcal{X} \Delta$$

which we will call the *edge gadget*. Define T as

$$\begin{aligned} T &= \mathcal{B}^0_{2p}\mathcal{X}^d\Delta\mathcal{B}^1_{2p}\mathcal{X}\Delta E_1E_2\dots E_p \\ &= \mathcal{B}^0_{2p}\mathcal{X}^d\Delta\mathcal{B}^1_{2p}\mathcal{X}\Delta[\mathcal{B}^{01}_1\mathcal{X}_{e_1}\Delta\mathcal{B}^1_{2p}\mathcal{X}\Delta][\mathcal{B}^{01}_2\mathcal{X}_{e_2}\Delta\mathcal{B}^1_{2p}\mathcal{X}\Delta]\dots[\mathcal{B}^{01}_p\mathcal{X}_{e_p}\Delta\mathcal{B}^1_{2p}\mathcal{X}\Delta] \end{aligned}$$

We add brackets for clarity only - they indicate the distinct E_i substrings, but the brackets are not actual characters of T. The idea is that T starts with $S' = \mathcal{B}^0_{2p}\mathcal{X}^d\Delta$, a modified S in which $\mathcal{B}_{2p}$ becomes $\mathcal{B}^0_{2p}$ and the X_i substrings are turned into X_i^d. This $\mathcal{X}^d$ substring serves as choice of vertices in our cost-effective subgraph problem. Each edge e_i has a "gadget substring" $E_i = \mathcal{B}^{01}_i\mathcal{X}_{e_i}\Delta\mathcal{B}^1_{2p}\mathcal{X}\Delta$. Since p is a multiple of m, the sequence of edge gadgets $E_1E_2\dots E_m$ is repeated p/m times. Our goal to go from T to S is to get rid of all these edge gadgets by contractions. Note that because a E_i gadget starts with $\mathcal{B}^{01}_i$ and the gadget E_{i+1} starts with $\mathcal{B}^{01}_{i+1}$, the substring E_{i+1} has a character that the substring E_i does not have.

The hardness proof. We now show that G admits a subset of vertices W of cost at most r if and only if T can be contracted to S using at most $p/m \cdot d(r + nm) + 4cdn$ contraction operations. We include the forward direction, which is the most instructive, in the main text.

($\Rightarrow$) Suppose that G has a subgraph W of cost at most r. Thus $c(m - |E(W)|) + |W||E(W)| \leq r$. To go from T to S, first consider an edge e_i that does not have both endpoints in W. We show how to get rid of the gadget substring E_i for e_i using $dn + dc$ contraction. Note that T contains the substring $\mathcal{B}^1_{2p}\mathcal{X}\Delta E_i = \mathcal{B}^1_{2p}\mathcal{X}\Delta[\mathcal{B}^{01}_i\mathcal{X}_{e_i}\Delta\mathcal{B}^1_{2p}\mathcal{X}\Delta]$, where brackets surround the E_i occurrence that we want to remove (note that here for $i > 1$, the prefix $\mathcal{B}^1_{2P}\mathcal{X}\Delta$ is the suffix of the previous E_{i-1} gadget, and for $i = 1$, it is the suffix of the starting block of T). We can first contract $\mathcal{B}^{01}_i$ to $\mathcal{B}^1_i$ using $dc + 2d - 2$ contractions, then contract $\mathcal{X}_{e_i}$ to $\mathcal{X}$ using $d(n-2)$ contractions. The result is the $\mathcal{B}^1_{2p}\mathcal{X}\Delta[\mathcal{B}^1_i\mathcal{X}\Delta\mathcal{B}^1_{2p}\mathcal{X}\Delta]$

substring, which becomes $\mathcal{B}^1_{2p}\mathcal{X}\Delta$ using two contractions (see below). This sum to $dc+2d-2+dn-2d+2 = dc+dn$ operations. More visually, the sequence of contractions works as follows (as before, brackets indicate the E_i substring and what remains of it, and the underlines are there to emphasize the substrings that participate in the contractions(s)):

$$
\begin{aligned}
&\mathcal{B}^1_{2p}\mathcal{X}\Delta[\mathcal{B}^{01}_i\mathcal{X}_{e_i}\Delta\mathcal{B}^1_{2p}\mathcal{X}\Delta] \\
\rightarrowtail &\mathcal{B}^1_{2p}\mathcal{X}\Delta[\mathcal{B}^1_i\mathcal{X}_{e_i}\Delta\mathcal{B}^1_{2p}\mathcal{X}\Delta] && (dc+2d-2 \text{ contractions}) \\
\rightarrowtail &\mathcal{B}^1_{2p}\mathcal{X}\Delta[\mathcal{B}^1_i\mathcal{X}\Delta\mathcal{B}^1_{2p}\mathcal{X}\Delta] && (d(n-2) \text{ contractions}) \\
= &B_{2p}B_{2p-1}\dots B_{i+1}B^1_i\mathcal{X}\Delta[\mathcal{B}^1_i\mathcal{X}\Delta\mathcal{B}^1_{2p}\mathcal{X}\Delta] \\
\rightarrowtail &B_{2p}B_{2p-1}\dots B_{i+1}B^1_i\mathcal{X}\Delta[\mathcal{B}^1_{2p}\mathcal{X}\Delta] && (1 \text{ contraction}) \\
= &\mathcal{B}^1_{2p}\mathcal{X}\Delta[\mathcal{B}^1_{2p}\mathcal{X}\Delta] \\
\rightarrowtail &\mathcal{B}^1_{2p}\mathcal{X}\Delta && (1 \text{ contraction})
\end{aligned}
$$

This sequence of $dn+dc$ contractions effectively removes the E_i substring gadget. Observe that after applying this sequence, it is still true that every remaining E_j gadget substring is preceded by $\mathcal{B}^1_{2p}\mathcal{X}\Delta$. We may therefore repeatedly apply this contraction sequence to every e_i not contained in W (including those e_i gadgets for which $i > m$). This procedure is thus applied to $p/m \cdot (m - |E(W)|)$ gadgets. We assume that we have done so, and that every e_i for which the E_i gadget substring remains is in W. Call the resulting string T'.

Now, let $\mathcal{X}_W$ be the substring obtained from $\mathcal{X}^d$ by contracting, for each $v_i \in W$, the string X^d_i to X_i. We assume that we have contracted the $\mathcal{X}^d$ substring of T' to $\mathcal{X}_W$, which uses $d|W|$ contractions (note that there is only one occurrence of $\mathcal{X}^d$ in T', namely right before the first Δ). Call T'' the resulting string. At this point, for every E_i substring gadget that remains, where E_i corresponds to edge $e_i = v_jv_k$, $\mathcal{X}_W$ contains the substrings X_j and X_k (instead of X^d_j and X^d_k).

Let i be the smallest integer for which the e_i substring gadget E_i is still in T'. This is the leftmost edge gadget still in T'', meaning that T'' has the prefix

$$\mathcal{B}^0_{2p}\mathcal{X}_W\Delta\mathcal{B}^1_{2p}\mathcal{X}\Delta[\mathcal{B}^{01}_i\mathcal{X}_{e_i}\Delta\mathcal{B}^1_{2p}\mathcal{X}\Delta]$$

where brackets indicate the E_i substring. To remove E_i, first contract $\mathcal{B}^{01}_i$ to $\mathcal{B}^0_i$, and contract $\mathcal{X}_{e_i}$ to X_W (This is possible since $e_i \subseteq W$). The result is $\mathcal{B}^0_{2p}\mathcal{X}_W\Delta\mathcal{B}^1_{2p}\mathcal{X}\Delta[\mathcal{B}^0_i\mathcal{X}_W\Delta\mathcal{B}^1_{2p}\mathcal{X}\Delta]$. One more contraction gets rid of the second half. This requires $dn+2d-1+d(|W|-2)+1 = dn + d|W|$ contractions. This procedure is applied to $p/m \cdot |E(W)|$ gadgets. To recap, the contraction sequence for E_i does as follows:

$$\begin{aligned}
&\mathcal{B}^0_{2p}\mathcal{X}_W\Delta\mathcal{B}^1_{2p}\mathcal{X}\Delta[\mathcal{B}^{01}_i\mathcal{X}_{e_i}\Delta\mathcal{B}^1_{2p}\mathcal{X}\Delta] && \\
\rightarrowtail &\mathcal{B}^0_{2p}\mathcal{X}_W\Delta\mathcal{B}^1_{2p}\mathcal{X}\Delta[\mathcal{B}^0_i\mathcal{X}_{e_i}\Delta\mathcal{B}^1_{2p}\mathcal{X}\Delta] && (dn + 2d - 1 \text{ contractions}) \\
\rightarrowtail &\mathcal{B}^0_{2p}\mathcal{X}_W\Delta\mathcal{B}^1_{2p}\mathcal{X}\Delta[\mathcal{B}^0_i\mathcal{X}_W\Delta\mathcal{B}^1_{2p}\mathcal{X}\Delta] && (d(|W| - 2) \text{ contractions}) \\
\rightarrowtail &\mathcal{B}^0_{2p}\mathcal{X}_W\Delta\mathcal{B}^1_{2p}\mathcal{X}\Delta && (1 \text{ contraction})
\end{aligned}$$

After we repeat this for every E_i, all that remains is the string $\mathcal{B}^0_{2p}\mathcal{X}_W\Delta\mathcal{B}^1_{2p}\mathcal{X}\Delta$. We contract $\mathcal{X}_W$ to $\mathcal{X}$ using $d(n - |W|)$ contractions (in total, going from $\mathcal{X}^d$ to $\mathcal{X}$ required dn moves). Then contract $\mathcal{B}^0_{2p}$ and $\mathcal{B}^1_{2p}$ to $\mathcal{B}_{2p}$ using $dc+2d-2+dn+2d-1 = d(c+n+4)-3$ contractions. one more contraction of the second half of the string yields S. The summary of the number

of contractions made is

$$\begin{aligned}
&\frac{p}{m} \cdot (m - |E(W)|) \cdot (dc + dn) + \frac{p}{m} \cdot |E(W)| \cdot (dn + d|W|) + dn + d(c + n + 4) - 3 \\
\leq &\frac{p}{m} \cdot (m - |E(W)|) \cdot (dc + dn) + \frac{p}{m} \cdot |E(W)| \cdot (dn + d|W|) + 4cdn \\
= &\frac{p}{m} \cdot d \cdot (c + n)(m - |E(W)|) + \frac{p}{m} \cdot d \cdot (n + |W|)|E(W)| + 4cdn \\
= &\frac{p}{m} \cdot d \cdot [c(m - |E(W)|) + |W||E(W)| + nm] + 4cdn \\
\leq &\frac{p}{m} \cdot d(r + nm) + 4cdn
\end{aligned}$$

as desired.

($\Leftarrow$) This direction of the proof is proved by several separate claims. The challenging part is to show that each E_i substring must get removed separately in this sequence, and that "most" of them incur a cost of either $dn + dt - 2$ or $dn + dc - 2$ for some t (this "most" is the reason that we need a large p).

Suppose that T can be turned into S using α contractions, where $\alpha \leq p/m \cdot d(r + nm) + 4cdn$. Let $C_1, \ldots, C_\alpha$ be a corresponding sequence of contractions. Here, each C_i contraction is given by a pair of positions ranging over both copies of the contracted substring. The idea is to show that, for some integer t, many of the E_i substrings are removed after t of the X_i^d substrings from X^d have been contracted to X_i. This set of t X_i's corresponds to the vertices of a cost-effective subgraph. The main components of the proof are to show that each E_i must be removed, no two E_i's are affected by the same contraction, and most (though perhaps not all) E_i require either $dn + dt$ or $dc + dn - 1$ contractions.

Denote $T(l)$ as the string obtained from T after applying the first l contractions $C_1, \ldots, C_l$ in the sequence, with $T(0) = T$ and $T(\alpha) = S$. A *block* of $T(l)$ is a substring P of $T(l)$ whose last character is Δ, that has only one occurrence of Δ and that is a maximal string with this property (hence in $T(l)$, the first character of P is either preceded by Δ or

is the start of $T(l)$). For instance, each E_i substring is made of 2 blocks.

We need a (conceptual) mapping from the character of $T(l)$ to choose of T. We assume that each character of T is distinguishable, i.e., each character has a unique identifier associated to it (we do not define it explicitly, but for instance each character can be labeled by its position in T). When contracting a substring DD from $T(l)$ to $T(l+1)$, we assume that the characters of the second half are deleted. That is, if $T(l) = LDDR$ and $T(l+1) = LDR$, only the characters from the first, leftmost D substring remain. Therefore when going from $T(l)$ to $T(l+1)$, some characters might change position but they keep the same identifier. Thus each character of $T(l)$ corresponds to a distinct character in T, namely the one with the same identifier. When we say that a character x from $T(l)$ *belongs* to a substring P of T, we mean that x corresponds to a character of P in T under this mapping.

For a substring P of T, we say that P is *removed* in $T(l)$ if $T(l)$ has no characters that belong to P. We say that P is removed if there is some $T(l)$, with $1 \leq l \leq \alpha$, in which P is removed.

Claim 1. *Each E_i substring must be removed in $T(\alpha)$.*

Proof. Consider the first, leftmost block $\mathcal{B}^0_{2p}\mathcal{X}^d\Delta$ of T, and recall that all of its characters are distinct. Observe that for any $T(l)$ and any symbol $s \in \Sigma$, $T(l)$ has an occurrence of s that belongs to this block (as there is no way to completely remove all occurrences of a symbol from the first block $\mathcal{B}^0_{2p}\mathcal{X}^d\Delta$, by our way of deleting the rightmost copy in contractions). Since $\Sigma(E_i) \subseteq \Sigma(\mathcal{B}^0_{2p}\mathcal{X}^d\Delta)$, this means that if E_i is not removed, the last string $T(\alpha)$ in the sequence has at least two occurrences of some character in Σ. Because S is exemplar, this contradicts that $T(\alpha) = S$. □

Notice that in T, $E_i = \mathcal{B}^{01}_i\mathcal{X}_{e_i}\Delta\mathcal{B}^1_{2p}\mathcal{X}\Delta$ has two blocks. We write $E' = \mathcal{B}^{01}_i\mathcal{X}_{e_i}\Delta$ to denote the first block of E_i. We let $E'_i(l)$ be the substring of $T(l)$ formed by all the characters that belong to E'_i, noting that $E'_i(l)$ is possibly the empty string or a subsequence of E'_i. For

$a \in \{0, 1, 01\}$, a block $BX\Delta$ is called a $\mathcal{B}^a\mathcal{X}\Delta$-block if $BX\Delta$ is a subsequence of $\mathcal{B}_i^a\mathcal{X}^d\Delta$ and $\Sigma(BX\Delta) = \Sigma(\mathcal{B}_i^a\mathcal{X}^d\Delta)$. In other words, $BX\Delta$ has all the symbols that occur in $\mathcal{B}_i^a\mathcal{X}^d\Delta$ in the same order, although the number of occurrences of a symbol might differ. A string obtained by concatenating an arbitrary number of $\mathcal{B}_{2p}^1\mathcal{X}\Delta$-blocks is called a $\mathcal{B}_{2p}^1\mathcal{X}\Delta$-cluster, which could possibly be the empty string. Using notation borrowed from regular languages, we write $(\mathcal{B}_{2p}^1\mathcal{X}\Delta)^*$ to denote a $\mathcal{B}_{2p}^1\mathcal{X}\Delta$-cluster.

Claim 2. *For any l, $T(l)$ has the form*

$$BX\Delta(\mathcal{B}_{2p}^1\mathcal{X}\Delta)^*E'_{i_1}(l)(\mathcal{B}_{2p}^1\mathcal{X}\Delta)^*E'_{i_2}(l)(\mathcal{B}_{2p}^1\mathcal{X}\Delta)^*\ldots E'_{i_h}(l)(\mathcal{B}_{2p}^1\mathcal{X}\Delta)^*,$$

where

- *$BX\Delta$ is a $\mathcal{B}_{2p}^0\mathcal{X}\Delta$-block*
- *$1 \leq i_1 < i_2 < \ldots < i_h \leq p$*
- *each $(\mathcal{B}_{2p}^1\mathcal{X}\Delta)^*$ is a $\mathcal{B}_{2p}^1\mathcal{X}\Delta$-cluster, and*
- *for each $j \in \{i_1, \ldots, i_h\}$, $E'_j(l)$ is a $\mathcal{B}_j^{01}\mathcal{X}\Delta$-block.*

Proof. Notice that the statement is true for $l = 0$, since T has the required form. Assume the claim is false and let l be the smallest integer for which $T(l)$ is a counter-example to the claim. Thus we may assume that $T(l-1)$ has the same form as in the claim statement. Let D be the string that was contracted from $T(l-1)$ to $T(l)$, so that $T(l-1)$ contained DD as a substring, and the second D substring gets removed from $T(l-1)$. We shall refer to the first D as the left D and the second as the right D. If D does not contain a Δ character, then DD is entirely contained in a single block. Contracting DD cannot remove all occurrences of a symbol nor change their order, and thus the above form must be preserved (every $\mathcal{B}_i^a\mathcal{X}\Delta$-block will remain a $\mathcal{B}_i^a\mathcal{X}\Delta$-block). Assume instead that the last character of D is Δ. Then

$DD = D'\Delta D'\Delta$ for some string D', and removing the second $D'\Delta$ half only removes entire blocks of $T(l-1)$. As this block cannot be $BX\Delta$ and since each $E'_{i_j}(l-1)$ is itself a block, this preserves the form of the claim.

Therefore, we may assume that the last character of D is not Δ, but that D has at least one Δ character. Observe that no character from the $BX\Delta$-block can get removed by such a contraction, since the left half of DD is kept. It follows that the first condition of the claim is preserved after contracting DD. It is easy to see that the second condition is also preserved. For the other two conditions, we have four cases to consider depending on where the right half of DD, i.e., the removed substring, is located in $T(l-1)$.

1. The leftmost character removed belongs to a $E'_j(l-1)$ substring of $T(l-1)$. In this case, because D contains a Δ, the right half of DD must contain the Δ of $E'_j(l-1)$. Let b be the first character of $E'_j(l-1)$, which is the first character of $\mathcal{B}_j^{01}$ since $E'_j(l-1)$ is a $\mathcal{B}_j^{01}\mathcal{X}\Delta$-block, by assumption. We treat b as a uniquely identifiable character in $T(l-1)$. Note that this b is preceded by Δ in $T(l-1)$. There are two subcases: either this b is the leftmost removed character or not. In the first case, $D = bD'$ for some D', which we illustrate as follows (we add brackets around the two copies of D, and underline the removed half):

$$T(l-1) = T'[bD'][\underline{bD'}]T'',$$

for some strings T' and T''. Here the second b is the one from $E'_j(l-1)$. Since it is preceded by Δ in $T(l-1)$, this implies that D' (and thus D) ends with a Δ. But we are assuming that D does not end with Δ. Therefore we know that b is not the leftmost character removed from $T(l-1)$.

It follows that the b character belongs to the left half of DD. This case can be illustrated

as follows:

$$T(l-1) = T'[D'bD''][\underline{D'bD''}]T''$$

where $D = D'bD''$ for some strings D' and D''. Here, the first b is the first character from $E'_j(l-1)$. We can argue that D' is not empty: if D' is empty, then D'' contains Δ because we are assuming that D contains Δ. But the first b illustrated above is in $E'_j(l-1)$, and this implies that the left $D'bD''$ ends after $E'_j(l-1)$, contradicting that the leftmost deleted character is inside $E'_j(l-1)$.

Since D' is not empty, it follows that D must contain Δb as a substring. But there is only one occurrence of Δb in $T(l-1)$, as $E'_j(l-1)$ is the only block that starts with b. Therefore, $T(l-1)$ cannot contain DD as a substring, a contradiction.

2. The rightmost character removed is in some $E'_j(l-1)$ substring. Again, if we put b as the first character of $E'_j(l-1)$, this means that the removed D contains Δb as a substring (if not, D cannot contain a Δ), which has only one occurrence. We get the same contradiction.

3. The leftmost and rightmost characters that get removed belong to distinct $(\mathcal{B}^1_{2p}\mathcal{X}\Delta)$-clusters, implying the existence of at least one $E'_j(l-1)$ in between. The same type of Δb substring argument applies, since the removed D contains the first character of $E'_j(l-1)$ and its preceding Δ.

4. The leftmost and rightmost characters that get removed belong to the same $(\mathcal{B}^1_{2p}\mathcal{X}\Delta)$-cluster. In this case, it is not hard to verify that the result is yet another $(\mathcal{B}^1_{2p}\mathcal{X}\Delta)$-cluster, which preserves the desired form.

The cases above cover every possibility: we have covered the cases where the removed substring begins or ends in a $E'_j(l-1)$, and the cases where both its extremities end in

a cluster. This proves the claim. □

We will say that a contraction C_l affects $E'_i(l)$ if at least one character of $E'_i(l)$ is in the substring corresponding to C_l. Recall that C_l spans over both copies of the contracted substring, and so $E'_i(l)$ could be affected by C_l even if none of its characters gets removed.

Claim 3. *For any l, the contraction C_l from $T(l)$ to $T(l+1)$ does not affect two distinct $E'_i(l)$ and $E'_j(l)$ substrings of $T(l)$.*

Proof. Suppose the claim is false, and let $T(l)[a_1..a_2]$ be the substring of $T(l)$ affected by the contraction, where $T(l)[a_1..a_2] = DD$ for some string D. Assume that $T(l)[a_1..a_2]$ contains characters from both $E'_i(l)$ and $E'_j(l)$, where $i < j$. Let b_i, b_j be the first characters of $E'_i(l)$ and $E'_j(l)$, respectively, which are the first character of $\mathcal{B}_i^{01}$ and $\mathcal{B}_j^{01}$ by Claim 2. Then $T(l)[a_1..a_2]$ must contain the substring Δb_j, since $E'_j(l)$ occurs later than $E'_i(l)$ in $T(l)$. Since Δb_j occurs only once in $T(l)$ as argued in the previous claim, Δb_j cannot be a substring of D. This is only possible if D starts with b_j (and consequently ends with Δ). Now, since $E'_i(l)$ does not contain b_j, $T(l)[a_1..a_2]$ cannot start with a suffix of $E'_i(l)$. Yet some characters of $E'_i(l)$ are in $T[a_1..a_2]$, implying that the substring Δb_i is in $T(l)$. Again, this substring occurs only once in $T(l)$, and thus D must start with b_i and end with Δ. But this is impossible since $b_i \neq b_j$. □

Notice that T has one occurrence of the $\mathcal{X}^d = X_1^d \dots X_n^d$ substring. We will therefore refer to the $\mathcal{X}^d$ substring of T without ambiguity. For $i \in [n]$, we let $X_i(l)$ denote the substring of $T(l)$ formed by all the characters that belong to the X_i^d substring of $\mathcal{X}^d$. We will say that X_i is *activated* in $T(l)$ if $X_i(l) = X_i$. Intuitively speaking, X_i is activated in $T(l)$ if it has undergone d contractions to turn it from X_i^d into X_i.

Claim 4. *Let $i \in [p]$, and suppose that E'_i is not removed in $T(l-1)$ but is removed in $T(l)$. Let t be the number of X_i's that were activated in $T(l-1)$. Suppose that v_{i_1} and v_{i_2} are the*

two endpoints of edge e_i.

Then the number of contractions that have affected E'_i is at least $dc + dn - 1$ if X_{i_1} or X_{i_2} is not activated in $T(l-1)$, or at least $\min\{dt + dn, dc + dn - 1\}$ if X_{i_1} and X_{i_2} are both activated in $T(l-1)$.

Proof. By Claim 2, in $T(l-1)$, $E'_i(l-1)$ belongs to a $\mathcal{B}_i^{01}\mathcal{X}\Delta$-block. As $E'_i(l-1)$ gets removed completely after the l-th contraction of some substring DD, it follows that D must contain a substring that is equal to $E'_i(l-1)$. The second D of the DD square certainly contains the $E'_i(l-1)$ substring that gets removed, but consider the copy of $E'_i(l-1)$ in the first D of the DD square. That is, we can represent the contraction as

$$T'[D_1\hat{E}'_i(l-1)D_2][\underline{D_1E'_i(l-1)D_2}]T''$$

where $D = D_1E'_i(l-1)D_2$ and $\hat{E}'_i(l-1)$ is a substring equal to $E'_i(l-1)$. Since $E'_i(l-1)$ is a block, this $\hat{E}'_i(l-1)$ copy is a substring of a (possibly larger) block. By Claim 3, there are only two such possible blocks: either it is $BX\Delta$, which is the $\mathcal{B}_{2p}^0\mathcal{X}\Delta$-block at the start of $T(l-1)$, or it is a $\mathcal{B}_{2p}^1\mathcal{X}\Delta$-block from a cluster preceding $E'_i(l-1)$. We analyze these two cases, which will prove the two cases of the claim.

Suppose that $\hat{E}'_i(l-1)$ is located in the first block $BX\Delta$ of $T(l-1)$. Note that since $\mathcal{X}_{e_i}$ contains X_{i_1} and X_{i_2} in their contracted form (as supposed to $X_{i_1}^d$ or $X_{i_2}^d$), X_{i_1} and X_{i_2} must be activated in $T(l-1)$ for the DD contraction to be possible. Moreover for $E'_i(l-1)$ to be equal to a substring of $BX\Delta$, every other X_j with $j \neq i_1, i_2$ that is activated must be contracted in $E'_i(l-1)$ (i.e., E'_i contains X_j^d, but must contain X_j in $E'_i(l-1)$). This requires at least $d(t-2)$ contractions. Moreover, B contains the B_1 substring, whereas E'_i contains B_1^*. There must have been at least $dn+2d-1$ affecting the $\mathcal{B}_i^{01}$ substring of E'_i. Counting the contraction removing $E'_i(l-1)$, this implies the existence of $d(t-2)+dn+2d-1+1 = dn+dt$ contractions affecting E'_i.

If instead $\hat{E}'_i(l-1)$ was located in a $\mathcal{B}^1_{2p}\mathcal{X}\Delta$-block, call this block P, then it suffices to note that P contains B_0 as a substring whereas E'_i contains B^*_0. Counting the contraction that removes $E'_i(l-1)$, it follows that at least $dc+2d-1$ contractions must have affected E'_i. □

The above shows that there are two types of contractions that can remove E'_i from $T(l)$. Either it uses the $BX\Delta$ substring at the start of $T(l-1)$, or it uses a block from a $\mathcal{B}^1_{2p}\mathcal{X}\Delta$-cluster. We will call the E'_i's that get removed in the first manner Type 1, and those that get removed in the second manner Type 2.

We would like to show that every Type 1 E'_j gets removed with the same set of activated X_i's, but it might not be the case. Rather, our next goal is to show that "many" E'_j's of Type 1 use the same activated X_i's. For $k \in [p]$, denote by $act(E'_k)$ the set of activated X_i's when E'_k gets removed (i.e., when E'_k is not removed from $T(l-1)$ but is removed from $T(l)$). Let us partition $[p]$ into intervals of integers $P_a = [1+am..m+am]$, where $a \in \{0,\ldots,p/m-1\}$. We say that interval P_a is *homogeneous* if, for each $i,j \in P_a$ such that E'_i and E'_j are of Type 1, $act(E'_i) = act(E'_j)$. In other words, P_a is homogeneous if all the Type 1 E'_i substrings corresponding to those in P_a are removed with the same set of activated X_i's.

Claim 5. *There are at least $p/m-2n$ homogeneous intervals.*

Proof. Observe that once an X_i is activated, it remains so for the rest of the contraction sequence. Since there are n of the X_i's, there are only $n+1$ possible values for $act(E'_k)$ (counting the case when none of them are activated). There are p/m intervals, and it follows that at most $n+1 \leq 2n$ of them are not homogeneous. □

We can now go on with the final elements of the proof. Define $cost(E'_i)$ as the number of contractions that affect E'_i. Let $P_{a_1},\ldots,P_{a_h}$ be the set of homogeneous intervals, $h \geq p/m-2n$. Choose the P_a interval among those whose sum of corresponding E'_i costs is

minimized - in other words choose P_a such that

$$\sum_{a \in P_a} cost(E_i') = \min_{j \in [h]} \sum_{i \in P_{a_j}} cost(E_i')$$

By Claim 3, no two E_i''s share their cost, and by the minimality of P_a the total number of contractions is at least

$$(\frac{p}{m} - 2n) \sum_{i \in P_a} cost(E_i')$$

We will only bother with these contractions and we make no assumption on the non-homogeneous intervals. Assume that there is at least one $i \in P_a$ such that E_i' is of Type 1. Then by Claim 4, $cost(E_i')$ is either at least $\min\{dc+dn-1, dt+dn\}$ where $t = |act(E_i')|$, or $cost(E_i')$ is at least $dc+dn-1$. If $dt+dn \geq dc+dn-1$, we may assume that E_i' is of Type 2 since removing E_i' using Type 2 contractions will not increase its cost. We will therefore assume that if there is at least one E_i' of Type 1 in P_a. then $dt+dn < dc+dn-1$ and thus $cost(E_i') \geq dt+dn$.

Now, choose any i in P_a such that E_i' is of Type 1, and let W be the set of vertices of G corresponding to those in $act(E_i')$. That is, $v_j \in W$ if and only if X_j is activated when E_i' gets removed. If there does not exist and E_i' of Type 1 to choose, then define $W = \emptyset$. Denote $|W| = t$ and $|E(W)| = s$. We claim that W is subgraph of G satisfying $c(m-s)+ts \leq r$.

Assume $c(m-s)+ts > r$ (otherwise, we are done). As we are dealing with integers, this means $c(m-s)+ts \geq r+1$. We will derive a contradiction on the assumed number of contractions. For any E_i' where $i \in P_a$, by Claim 4, either e_i is not in W and $cost(E_i') \geq dc+dn-1$, or e_i is in W and $cost(E_i') \geq dt+dn$. Note that we needed to choose P_a to be homogeneous to guarantee that every Type 1 E_i' uses the same value of t in the cost $dt+dn$.

It follows that the total number of contractions is at least

$$
\begin{aligned}
&(\frac{p}{m}-2n)\sum_{i\in P_a} cost(E_i')\\
\geq&(\frac{p}{m}-2n)[(m-s)(dc+dn-1)+s(dt+dn)]\\
=&(\frac{p}{m}-2n)[d((c+n)(m-s)+s(t+n))-m+s]\\
=&(\frac{p}{m}-2n)[d(c(m-s)+st)+d(nm-ns+ns)-m+s]\\
=&(\frac{p}{m}-2n)\cdot d\cdot[c(m-s)+st+nm]+(\frac{p}{m}-2n)(s-m)\\
\geq&(\frac{p}{m}-2n)\cdot d\cdot[r+1+nm]+(\frac{p}{m}-2n)(s-m)\\
=&(\frac{p}{m}-2n)\cdot d\cdot[r+nm]+(\frac{p}{m}-2n)(d+s-m)\\
=&\frac{p}{m}\cdot d(r+nm)-2dn(r+nm)+(\frac{p}{m}-2n)(d+s-m)
\end{aligned}
$$

Now if d and p are large enough, the above is strictly greater $p/m\cdot d(r+nm)+4cdn$, leading to a contradiction. Our chosen values $d=m+1$ and $p=(n+m)^{10}$ easily verify this. We have therefore shown that W has the desired cost. This concludes the proof. □

An example. In Figure 2.2, we show a graph G with $W=\{v_2,v_3,v_4\}$. The sequence T containing E_1 and E_2 is as follows. We set $d=4$ (certainly it can be larger, e.g., 9), $\mathcal{X}=X_1X_2X_3X_4X_5X_6$, where $X_i=x_{i1}x_{i2}x_{i3}x_{i4}$. And we set $X_i^4=x_{i1}x_{i1}x_{i2}x_{i2}x_{i3}x_{i3}x_{i4}x_{i4}$ and $\mathcal{X}^4=X_1^4X_2^4X_3^4X_4^4X_5^4X_6^4$. For $e_i=(v_j,v_k)$, $\mathcal{X}_{e_i}=X_1^4\cdots X_{j-1}^4X_jX_{j+1}^4\cdots X_{k-1}^4X_kX_{k+1}^4\cdots X_n^4$.

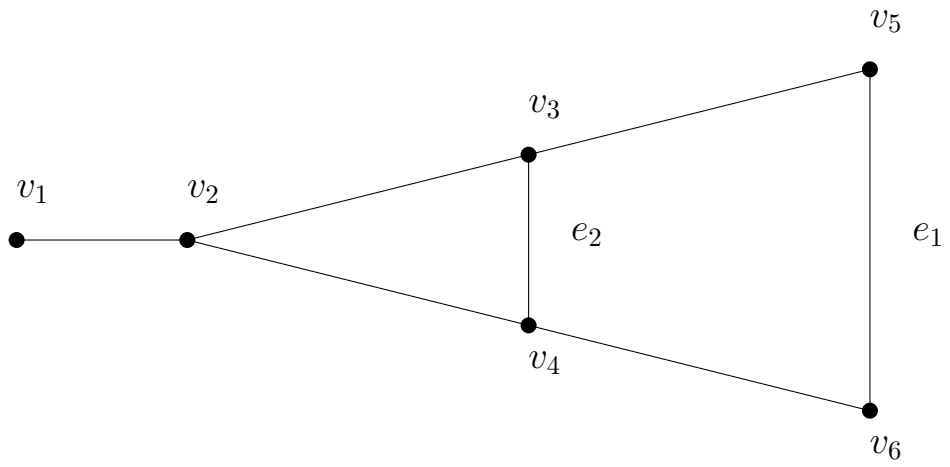

Figure 2.2: A graph G with $W = \{v_2, v_3, v_4\}$. In the constructed gadgets, we only show those for $e_1 = (v_5, v_6)$ and $e_2 = (v_3, v_4)$. For clarity, we assume $d = 4$.

$$
\begin{aligned}
T =& B_{2p}B_{2p-1}\cdots B_2B_1B_0^*X_1^4X_2^4X_3^4X_4^4X_5^4X_6^4\Delta && //\mathcal{B}_{2p}^0\mathcal{X}^d\Delta \\
& B_{2p}B_{2p-1}\cdots B_2B_1^*B_0X_1X_2X_3X_4X_5X_6\Delta && //\mathcal{B}_{2p}^1\mathcal{X}\Delta \\
& B_1^*B_0^*X_1^4X_2^4X_3^4X_4^4X_5X_6\Delta && //\mathcal{B}_1^{01}\mathcal{X}_{e_1}\Delta \\
& B_{2p}B_{2p-1}\cdots B_2B_1^*B_0X_1X_2X_3X_4X_5X_6\Delta && //\mathcal{B}_{2p}^1\mathcal{X}\Delta \\
& B_2B_1^*B_0^*X_1^4X_2^4X_3X_4X_5^4X_6^4\Delta && //\mathcal{B}_2^{01}\mathcal{X}_{e_2}\Delta \\
& B_{2p}B_{2p-1}\cdots B_2B_1^*B_0X_1X_2X_3X_4X_5X_6\Delta && //\mathcal{B}_{2p}^1\mathcal{X}\Delta \\
& \cdots\cdots && \text{omitted.}
\end{aligned}
$$

In the forward direction, as $e_1 \notin E(W)$, E_1 is contracted as follows:

$$
\begin{aligned}
& B_{2p}B_{2p-1}\cdots B_2B_1^*B_0X_1X_2X_3X_4X_5X_6\Delta[B_1^*B_0^*X_1^4X_2^4X_3^4X_4^4X_5X_6\Delta\mathcal{B}_{2p}^1\mathcal{X}\Delta] \\
\Rightarrow & B_{2p}B_{2p-1}\cdots B_2B_1^*B_0X_1X_2X_3X_4X_5X_6\Delta[B_1^*B_0X_1^4X_2^4X_3^4X_4^4X_5X_6\Delta\mathcal{B}_{2p}^1\mathcal{X}\Delta] \\
\Rightarrow & B_{2p}B_{2p-1}\cdots B_2B_1^*B_0X_1X_2X_3X_4X_5X_6\Delta[B_1^*B_0X_1X_2X_3X_4X_5X_6\Delta\mathcal{B}_{2p}^1\mathcal{X}\Delta] \\
\Rightarrow & B_{2p}B_{2p-1}\cdots B_2B_1^*B_0X_1X_2X_3X_4X_5X_6\Delta[\mathcal{B}_{2p}^1\mathcal{X}\Delta] \\
\Rightarrow & B_{2p}B_{2p-1}\cdots B_2B_1^*B_0X_1X_2X_3X_4X_5X_6\Delta \\
= & \mathcal{B}_{2p}^1\mathcal{X}\Delta
\end{aligned}
$$

Similarly, as $e_2 \in E(W)$, E_2 is contracted as follow.

$$
\begin{aligned}
&\mathcal{B}^0_{2p} X_1^4 X_2^4 X_3^4 X_4^4 X_5^4 X_6^4 \Delta B_{2p} B_{2p-1} \cdots B_2 B_1^* B_0 \mathcal{X} \Delta [B_2 B_1^* B_0^* X_1^4 X_2^4 X_3 X_4 X_5^4 X_6^4 \Delta \mathcal{B}^1_{2p} \mathcal{X} \Delta] \\
=&\mathcal{B}^0_{2p} \mathcal{X}^4 \Delta \mathcal{B}^1_{2p} \mathcal{X} \Delta [\mathcal{B}_2^{01} \mathcal{X}_{e_2} \Delta \mathcal{B}^1_{2p} \mathcal{X} \Delta] \\
\to&\mathcal{B}^0_{2p} X_1^4 X_2 X_3 X_4 X_5^4 X_6^4 \Delta B_{2p} B_{2p-1} \cdots B_2 B_1^* B_0 \mathcal{X} \Delta [B_2 B_1^* B_0^* X_1^4 X_2^4 X_3 X_4 X_5^4 X_6^4 \Delta \mathcal{B}^1_{2p} \mathcal{X} \Delta] \\
=&\mathcal{B}^0_{2p} \mathcal{X}_W \Delta \mathcal{B}^1_{2p} \mathcal{X} \Delta [\mathcal{B}_2^{01} \mathcal{X}_{e_2} \Delta \mathcal{B}^1_{2p} \mathcal{X} \Delta] \\
\to&\mathcal{B}^0_{2p} X_1^4 X_2 X_3 X_4 X_5^4 X_6^4 \Delta B_{2p} B_{2p-1} \cdots B_2 B_1^* B_0 \mathcal{X} \Delta [B_2 B_1 B_0^* X_1^4 X_2^4 X_3 X_4 X_5^4 X_6^4 \Delta \mathcal{B}^1_{2p} \mathcal{X} \Delta] \\
=&\mathcal{B}^0_{2p} \mathcal{X}_W \Delta \mathcal{B}^1_{2p} \mathcal{X} \Delta [\mathcal{B}_2^{0} \mathcal{X}_{e_2} \Delta \mathcal{B}^1_{2p} \mathcal{X} \Delta] \\
\to&\mathcal{B}^0_{2p} X_1^4 X_2 X_3 X_4 X_5^4 X_6^4 \Delta B_{2p} B_{2p-1} \cdots B_2 B_1^* B_0 \mathcal{X} \Delta [B_2 B_1 B_0^* X_1^4 X_2 X_3 X_4 X_5^4 X_6^4 \Delta \mathcal{B}^1_{2p} \mathcal{X} \Delta] \\
=&\mathcal{B}^0_{2p} \mathcal{X}_W \Delta \mathcal{B}^1_{2p} \mathcal{X} \Delta [\mathcal{B}_2^{0} \mathcal{B}_W \Delta \mathcal{B}^1_{2p} \mathcal{X} \Delta] \\
\to&B_{2p} B_{2p-1} \cdots B_2 B_1 B_0^* X_1^4 X_2 X_3 X_4 X_5^4 X_6^4 \Delta \cdot B_{2p} B_{2p-1} \cdots B_2 B_1^* B_0 X_1 X_2 X_3 X_4 X_5 X_6 \Delta \\
=&\mathcal{B}^0_{2p} \mathcal{X}_W \Delta \mathcal{B}^1_{2p} \mathcal{X} \Delta \\
\to&\cdots \text{contract other } E_i \text{ in } E(W) \\
\to&B_{2p} B_{2p-1} \cdots B_2 B_1 B_0^* X_1 X_2 X_3 X_4 X_5 X_6 \Delta \cdot B_{2p} B_{2p-1} \cdots B_2 B_1^* B_0 X_1 X_2 X_3 X_4 X_5 X_6 \Delta \\
=&\mathcal{B}^0_{2p} \mathcal{X} \Delta \mathcal{B}^1_{2p} \mathcal{X} \Delta \\
=&S.
\end{aligned}
$$

Note that in the reverse direction, we do not necessarily know the optimal way to contract E_k (e.g., when E_k is not in a homogeneous interval). For instance, the optimal

solution could contract E_1 as follows:

$$
\begin{aligned}
&\mathcal{B}^0_{2p}X^4_1X^4_2X^4_3X^4_4X^4_5X^4_6\Delta B_{2p}B_{2p-1}\cdots B_2B^*_1B_0\mathcal{X}\Delta[B^*_1B^*_0X^4_1X^4_2X^4_3X^4_4X_5X_6\Delta\mathcal{B}^1_{2p}\mathcal{X}\Delta]\\
=&\mathcal{B}^0_{2p}\mathcal{X}^4\Delta\mathcal{B}^1_{2p}\mathcal{X}\Delta[\mathcal{B}^{01}_1\mathcal{X}_{e_1}\Delta\mathcal{B}^1_{2p}\mathcal{X}\Delta]\\
\rightarrow&\mathcal{B}^0_{2p}X^4_1X^4_2X^4_3X^4_4X^4_5X^4_6\Delta B_{2p}B_{2p-1}\cdots B_2B^*_1B_0\mathcal{X}\Delta[B_1B^*_0X^4_1X^4_2X^4_3X^4_4X_5X_6\Delta\mathcal{B}^1_{2p}\mathcal{X}\Delta]\\
=&\mathcal{B}^0_{2p}\mathcal{X}^4\Delta\mathcal{B}^1_{2p}\mathcal{X}\Delta[\mathcal{B}^0_1\mathcal{X}_{e_1}\Delta\mathcal{B}^1_{2p}\mathcal{X}\Delta]\\
\rightarrow&\mathcal{B}^0_{2p}X^4_1X^4_2X^4_3X^4_4X_5X_6\Delta B_{2p}B_{2p-1}\cdots B_2B^*_1B_0\mathcal{X}\Delta[B_1B^*_0X^4_1X^4_2X^4_3X^4_4X_5X_6\Delta\mathcal{B}^1_{2p}\mathcal{X}\Delta]\\
=&\mathcal{B}^0_{2p}\mathcal{X}_{e_1}\Delta\mathcal{B}^1_{2p}\mathcal{X}\Delta[\mathcal{B}^0_1\mathcal{X}_{e_1}\Delta\mathcal{B}^1_{2p}\mathcal{X}\Delta]\\
\rightarrow&\mathcal{B}^0_{2p}X^4_1X^4_2X^4_3X^4_4X_5X_6\Delta B_{2p}B_{2p-1}\cdots B_2B^*_1B_0X_1X_2X_3X_4X_5X_6\Delta\\
=&\mathcal{B}^0_{2p}\mathcal{X}_{e_1}\Delta\mathcal{B}^1_{2p}\mathcal{X}\Delta
\end{aligned}
$$

Of course, in this way the $\mathcal{X}^d$ after $\mathcal{B}^0_{2p}$ will lose contents (i.e., some X^d_i will be contracted into X_i). However, the above contractions can only last in at most n homogeneous intervals.

In the next subsection, we show that the hardness result holds even when the alphabet is of size four (fitting that the DNA and RNA sequences are over the set of nucleotides {A,C,G,T} and {A,C,G,U}, respectively).

2.4.2 NP-hardness with Alphabet of Size Four

We reduce the minimum tandem duplication problem on arbitrary alphabet to the minimum tandem duplication problem on alphabets of size four. We show that we can encode each character into a string on alphabet of size four and preserve the hardness. The main difficulty is to show that no contraction occurs inside the encoded strings, which requires a specific form of square-free strings.

Lemma 1. *Let m be an integer. Then there exists an algorithm that takes time polynomial in m that outputs pairs of square-free strings $(A_1, B_1), \ldots, (A_m, B_m)$ over alphabet $\{a, b, c\}$ such that for each $i \in \{1, \ldots, m\}$:*

1. $100i - 3 \leq |A_i| \leq 100i$ *and* $|B_i| = 1000m - |A_i|$*;*

2. $A_i[1] = a$

3. $B_i[1] = b$ *and* $B_i[|B_i|] = b$*.*

Proof. In [66] , Leech describes the following process to generate an infinite square-free string. Define $w_0 = a$, and for $j > 0$, recursively obtain w_j from w_{j-1} by replacing each character with a specified substring of length 13 (see paper for details). Each w_j is square-free, and for our purposes we may apply this process and stop when the condition $|w_j| > 2000m$ is met for the first time (note that this takes polynomial time).

Assume that the first a occurrence of w_j is at position p, and for $k \geq 1$, define $A(k) = w_j[p..p+k-1]$. Thus each $A(k)$ starts with a and has length k (and we will choose a subset of these to prove the lemma).

Similarly, assume that the first b of w_j is at position q, and for each $k \geq 1$, define $B' = w_j[q..q+k-1]$. Let $r \geq q+k-1$ be the first position at or after $q+k-1$ such that $w_j[r] = b$. Define $B(k) = w_j[q..r]$. Note that $r \leq q+k+2$, as otherwise $w_j[q-1..q+k+2]$ would be a binary string of length 4 and would therefore contain a square. It follows that $k \leq |B(k)| \leq k+3$. Moreover, each $B(k)$ starts and ends with a b.

To conclude the lemma, for each $i \in \{1, \ldots, m\}$, put $B_i = B(1000m - 100i)$ and $A_i = A(1000m - |B_i|)$. We have $1000m - 100i \leq B_i \leq 1000m - 100i + 3$, and, since $|A_i| = 1000m - |B_i|$, we have $100i - 3 \leq |A_i| \leq 100i$. The other conditions of lemma follow from our construction. □

We can now proceed to our reduction.

Theorem 3. *The TD problem is NP-hard for any* $|\Sigma| \geq 4$*, where* Σ *is the alphabet of the input strings.*

Proof. Our reduction is from the TD problem with unbounded alphabet size, which is NP-hard owing to Theorem 2. We show that TD problem is NP-hard when $|\Sigma| = 4$, which clearly implies hardness for any $|\Sigma| \geq 4$ since one can add a new dummy character at the end of S and T if $|\Sigma|$ is smaller than the desired value. Let (S, T, k) be an instance of the TD problem, and let $\Sigma = \{s_1, \ldots, s_m\}$ be the alphabet of S and T with $m = |\Sigma|$.

Let $(A_1, B_1), \ldots, (A_m, B_m)$ be pairs of ternary square-free strings that satisfy the conditions of Lemma 10, i.e., each $|A_i|$ is between $100i - 3$ and $100i$, $|B_i| = 1000m - |A_i|$, with A_i starting with a, and B_i starting and ending with b. By the lemma, these can be obtained in polynomial time. Assume that the alphabet of the A_i and B_i strings is $\{a, b, c\}$, and let x be another symbol.

We encode each symbol $s_i \in \Sigma$ on alphabet $\{a, b, c, x, \}$ be defining

$$h(s_i) = xcxA_ixB_i$$

Note that each $h(s_i)$ string has the same length $1000m + 4$. Also note that each A_i has a distinct length, and thus that each B_i also has a distinct length. This implies that each A_i is matched with a distinct B_i determined by its length (and vice-versa). Moreover, since A_i and B_i are square-free, we can argue that $h(s_i)$ is also square-free. Indeed, no square can start at the first x of $h(s_i)$ since xc only occurs once in $h(s_i)$; no square can start at the second character c since the only possible occurrence of cs is at the end of A_ix, and A_i has lengths at least 97; no square can start at the second x since xa occurs only once. Therefore, a square would need to start in A_i and contain the x after it, which is not possible since there is only one such x.

We then encode S and T into S' and T', respectively, by replacing each character with

their encoding. That is, put

$$S' = h(S[1])h(S[2])\dots h(S[|S|]) \qquad T' = h(T[1])h(T[2])\dots h(T[|T|])$$

Observe that in both strings, all occurrences of $xcxa$ mark the start of an encoded symbols (since all A_i strings start with a), and occurrences of xb mark a separation point between some A_i and B_i. In what follows, we say that a string Y *encodes* X if $Y = h(X[1])\dots h(X[|X|])$.

We show that T can be transformed into S using k contractions if and only if T' can be turned into S' using k contractions.

($\Rightarrow$) Assume that T can be transformed into S using k contractions. Let $T = T_0, T_1, \dots, T_k = S$ be the sequence of strings encountered during these contractions. We define a sequence of contractions from T' to S' in which $T' = T'_0, T'_1, \dots, T'_k = S'$ are the substrings encountered during this sequence, such that each T'_i encodes T_i. This is true when $i = 0$. Assume inductively that T'_{i-1} encodes T_{i-1}, and that the i-th contraction affects some substring RR of T_{i-1} i.e., $T_{i-1} = ARRB \rightarrowtail ARB$. Then $T'_{i-1} = A'R'R'B'$, where A' encodes A, R' encodes R and B' encodes B. We can apply $A'R'R'B' \rightarrowtail A'R'B'$ and obtain T'_i that encodes T_i. It follows that $T'_k = S'$, as desired.

($\Leftarrow$) Assume that there is a sequence of k contractions $D'_1, \dots, D'_k$ transforming T' into S' (assume that the D'_i's contain the start and end index of the contraction). Let T'_i be the string obtained after applying the i-th contraction in this sequence, with $T'_0 = T'$. We construct a sequence of contractions $D_1, \dots, D_k$ transforming T into S. To achieve this, we construct the D_i contractions by maintaining the following invariant: by defining T_i as the string obtained after applying $D_1, \dots, D_i$ on T, then T'_i encodes T_i.

This is true for $i = 0$ since $T'_0 = T'$ encodes $T_0 = T$. Now assume inductively that T'_{i-1} is obtained by encoding each character of T_{i-1}.

Assume that D'_i contracts a string RR of T'_{i-1}. Since T'_{i-1} is a concatenation of $h(s_j)$

strings, the first character of the RR substring must be in some $h(s_j)$ substring of T'_{i-1}. We consider four possible cases.

Case 1. The first character of RR is in the xcs substring at the start of $h(s_j) = xcxA_jxB_j$. Thus we may write $xcx = C_1C_2$ such that C_2 is a prefix of RR (note that C_1 is possibly empty, but not C_2). It is easy to check R must contains at least a prefix of A_j. This implies that R contains xa as a substring. i.e., $R = C_2aY$ for some Y.

The contraction $T'_{i-1} \rightarrowtail T'_i$ has the form

$$T'_{i-1} = PC_1[C_2aY][C_2aY]Q \rightarrowtail PC_1[C_2aY]Q = T'_i$$

for some strings P, Q. Because C_2a contains xa, it must mark the start of an encoding, meaning that Y has C_1 as a suffix. Writing $Y = Y'C_1$, we get

$$T'_{i-1} = PC_1[C_2aY'C_1][C_2aY'C_1]Q \rightarrowtail PC_1[C_2aY'C_1]Q = T'_i$$

We can replace D'_i with the following contraction:

$$T'_{i-1} = P[C_1C_2aY'][C_1C_2aY']C_1Q \rightarrowtail P[C_1C_2aY']C_1Q = T'_i$$

Notice that P encodes some prefix $\hat{P}$ of T_{i-1} (P is possibly empty). The C_1C_2aY' substring that follows also encodes some substring Y of T_{i-1} and, since all encodings have the same length, the second C_1C_2aY' copy that follows encodes the same substring $\hat{Y}$. This implies that Q, if not empty, starts with xcx and encodes a suffix $\hat{Q}$ of T_{i-1}. Therefore, $T_{i-1} = \hat{P}\hat{Y}\hat{Y}\hat{Q}$, and it becomes easy to see that if D_i contracts $\hat{Y}\hat{Y}$, T'_i encodes T_i.

Case 2. The first character of RR is in the A_j substring of $h(s_j)$. Write $A_j = A_j^1A_j^2$ such that A_j^2 is a prefix of RR. Because A_j is square-free, RR cannot be entirely contained in A_j. Thus R contains the x after A_j. This x cannot be the last character of R, as otherwise RR

would be fully contained in $h(s_j)$ since $|B_j| > |A_j|$, which is again not possible since $h(s_j)$ is square-free. Thus the first occurrence of x in R is followed by b, implying that the second R copy must start in the A_l portion of some $h(s_l)$ encoding (since this is the only way to have xb as a substring before xa or xc).

Hence, we may assume that $R = A_j^2 x B_j x Y$ for some non-empty Y. The contraction has the form

$$PxcxA_j^1[A_j^2xB_jxY][A_j^2xB_jxY]Q \rightarrowtail PxcxA_j^1[A_j^2xB_jxY]Q$$

for some strings P and Q. Since each B_j is associated with a unique A_j, Y must have $xcxA_j^1$ as a suffix. We may write $Y = Y'xcxA_j^1$ and

$$T'_{i-1} = PxcxA_j^1[A_j^2xB_jxY'xcxA_j^1][A_j^2xB_jxY'xcxA_j^1]Q \rightarrowtail PxcxA_j^1[A_j^2xB_jxY'xcxA_j^1]Q = T'_i$$

It becomes easy to see that D'_i can be replaced by the contraction

$$T'_{i-1} = P[xcxA_j^1A_j^2xB_jxY'][xcxA_j^1A_j^2xB_jxY']xcxA_j^1Q \rightarrowtail P[xcxA_j^1A_j^2xB_jxY']xcxA_j^1Q = T'_i$$

and we can apply Case 1.

Case 3. The first character of RR is the x occurrence in $h(x_j)$ between A_j and B_j. Thus R starts with xb, implying that the second R copy also starts with the x occurrence between some A_l and B_l. The contraction $T'_{i-1} \rightarrowtail T'_i$ has the form

$$T'_{i-1} = PxcxA_j[xB_jxY][xB_jxY]Q \rightarrowtail PxcxA_j[xB_jxY]Q = T'_i$$

for some strings P, Y, Q. Because B_j is associated only with A_j in the encodings, we may

write $Y = Y'xcxA_j$. We get

$$T'_{i-1} = PxcxA_j[xB_jxY'xcxA_j][xB_jxY'xcxA_j]Q \rightarrowtail PxcxA_j[xB_jxY'xcxA_j]Q = T'_i$$

We can replace D'_i with an alternate contraction as follows:

$$T'_{i-1} = P[xcxA_jxB_jxY'][xcxA_jxB_jxY']xcxA_jQ \rightarrowtail P[xcxA_jxB_jxY']xcxA_jQ = T'_i$$

and, once again, we can apply Case 1.

Case 4. The first character of RR is in the B_j substring of $h(s_j)$. Write $B_j = B_j^1B_j^2$ such that B_j^2 is a prefix of R. We argue that the first R copy cannot be entirely in B_j^2. Otherwise, the second R copy would contain characters outside of B_j^2 since B_j is square-free. That is, the second R copy would contain an x but not the first, which is impossible.

Thus the first R copy contains the x after B_j^2. We cannot have $R = B_j^2x$. Indeed, B_j^2x is followed by cx, and because $R = B_j^2x$, the only possible contraction is $RR = B_j^2xcx = cxcx$. This is not possible, since B_j^2 ends with a b.

Therefore, R contains the x after B_j^2, and also the c after it. The only occurrences of xc are at the start of encodings, so the contraction has the form

$$T'_{i-1} = P[B_j^2xcY][B_j^2xcY]Q \rightarrowtail P[B_j^2xcY]Q = T'_i$$

Now, the second R copy starts with a suffix of some B_l, and it happens that B_j and B_l have the same suffix B_j^2. We can write $B_l = B_l^1B_j^2$ such that Y has $A_lxB_l^1$ as a suffix. Write $Y = Y'A_lxB_l^1$ so that

$$T'_{i-1} = P[B_j^2xcY'A_lxB_l^1][B_j^2xcY'A_lxB_l^1]Q \rightarrowtail P[B_j^2xcY'A_lxB_l^1]Q = T'_i$$

Recall that A_l is uniquely matched with $B_l = B_l^1 B_j^2$ in the encodings, so Q must have B_j^2 as a prefix. Write $Q = B_j^2 Q'$ so that

$$T'_{i-1} = P[B_j^2 xcY' A_l x B_l^1][B_j^2 xcY' A_l x B_l^1] B_j^2 Q' \rightarrowtail P[B_j^2 xcY' A_l x B_l^1] B_j^2 Q' = T'_i$$

We can replace this contraction with

$$T'_{i-1} = P B_j^2 [xcY' A_l x B_l^1 B_j^2][xcY' A_l x B_l^1 B_j^2] Q' \rightarrowtail P B_j^2 [xcY' A_l x B_l^1 B_j^2] Q' = T'_i$$

We can once again apply Case 1.

We have thus shown how to construct a sequence of contractions that transforms T into a string T_k such that T'_k encodes T_k. Since T'_k encodes S', it follows that $T_k = S$, which concludes the proof. □

2.5 An FPT Algorithm for the Exemplar Problem

Let us recall that in the `Exemplar-`k`-TD` problem, we receive a string S in which every character is different, a string T, and an integer k. We want to know whether S can be transformed into T using at most k TDs. In this section, we show that `Exemplar-`k`-TD` can be solved in time $2^{O(k^2)} + poly(n)$ by obtaining a kernel of size $O(k2^k)$ (here n is the length of T).

We first note that there is a very simple, brute-force algorithm to solve the k-TD problem, which is the variant of the TD problem with parameter k, the number of TDs to turn S into T (including `Exemplar-`k`-TD` as a particular case). This only established membership in the XP class, but it is useful to evaluate the complexity of our kernelization later on.

Proposition 1. *The k-TD problem can be solved in time $O(n^{2k})$, where n is the size of the*

target string.

Proof. Let (S, T) be a given instance of k-TD. Consider the branching algorithm that, starting from T, tries to contract every substring of from XX in T and recurses on each resulting substring, decrementing k by 1 each time (the branching stops when S is obtained or when k reaches 0 without attaining S). We obtain a search tree of depth at most k and degree at most n^2, and thus it has $O(n^{2k})$ nodes. Visiting the internal nodes of this search tree only requires enumerating $O(n^2)$ substrings, which form the set of children of the node. Hence, there is no added computation cost to consider when visiting a node. □

From now on, we assume that we have an **Exemplar-k-TD** instance (S, T), and so that S is exemplar.

Let x and y be two consecutive characters in S (i.e., xy is a substring of S). We say that xy is (S, T)-stable if in T, every occurrence of x in T is followed by y and every occurrence of y is preceded by x. An (S, T)-stable substring $X = x_1 \dots x_l$, where $l \geq 2$, is a substring of S such that $x_i x_{i+1}$ is (S, T)-stable for every $i \in [l-1]$. We also define a string with a single character x_i to be a (S, T)-stable substring (provided x_i appears in S and T). If any substring of S that strictly contains X is not an (S, T)-stable substring, then X is called a maximal (S, T)-stable substring. Note that these definitions are independent of S and T, and so the same definitions apply for (S, T)-stability, for any strings X and Y.

We show that every maximal (S, T)-stable substring can be replaced by a single character, and that if T can be obtained from S using at most k tandem duplications, then this leaves strings of bounded size.

We first show that, roughly speaking, stability is maintained by all tandem duplications when going from S to T.

Lemma 2. *Suppose that $dist_{TD}(S, T) = k$ and let X be an (S, T)-stable substring. Let $S = S_0, S_1, \dots, S_k = T$ be any minimum sequence of strings transforming S to T by tandem*

duplications. Then X is (S, S_i)-stable for every $i \in [k]$.

Proof. Assume the lemma is false, and let S_i be the first of $S_1, \ldots, S_k$ that does not verify the statement. Then there are two characters x, y belonging to X such that xy is (S, T)-stable, but xy is not (S, S_i)-stable.

We claim that, under our assumption, xy is not (S, S_j)-stable for any $j \in \{1, \ldots, k\}$. As this includes $S_k = T$, this will contradict that xy is (S, T)-stable. We do this by induction — as a base case, xy is not (S, S_i)-stable so this is true for $j = i$. Assume that xy is not (S, S_{j-1})-stable, where $i < j \leq k$. Let D be the duplication transforming S_{j-1} to S_j (here $D = (a, b)$ contains the start and end positions of the substring of S_{j-1} to duplicate).

suppose first that xy is not (S, S_{j-1})-stable because S_{j-1} has an occurrence of x that is not followed by y. Thus S_{j-1} has an occurrence of x, say at position p_x, followed by $z \neq y$. If we assume that xy is (S, S_j)-stable, then a y character must have appeared after this x from S_{j-1} to S_j. Changing the character next to this x is only possible if the last character duplicated by D is the x at position p_x and the first character of D is a y. In other words, denoting $S_{j-1} = A_1 y A_2 x z A_3$ for appropriate A_1, A_2, A_3 substrings, the D duplication must do the following

$$A_1 \underline{y A_2 x} z A_3 \Rightarrow A_1 \underline{y A_2 x y A_2 x} z A_3$$

as otherwise, the character next to the above occurrence of x will remain z. But then, there is still an occurrence of x followed by z, and it follows that xy cannot be (S, S_j)-stable.

So suppose instead that xy is not (S, S_{j-1})-stable because S_{j-1} has an occurrence of y preceded by $z \neq x$. Assume the character preceding this y has changed in S_j and has become x. But one can verify that this is impossible. For completeness, we present each possible case: either D includes both z and y, includes one of them or none. These cases are represented below, and each one of them leads to an occurrence of y still preceded by z

(the left-hand side represents S_{j-1} and the right-hand side represents S_j, and the A_i's are the relevant substrings in each case):

$$\text{Include both: } A_1\underline{A_2zyA_3}A_4 \Rightarrow A_1\underline{A_2zyA_3A_2zyA_3}A_4$$

$$\text{Include } z \text{ only: } A_1\underline{A_2zyA_3} \Rightarrow A_1\underline{A_2zA_2zyA_3}$$

$$\text{Include } y \text{ only: } A_1zy\underline{A_2}A_3 \Rightarrow A_1zy\underline{A_2yA_2}A_3$$

$$\text{Include none: } A_1\underline{A_2}zyA_3 \Rightarrow A_1\underline{A_2A_2zyA_3} \text{ or } A_1zy\underline{A_2}A_3 \Rightarrow A_1zy\underline{A_2A_2}A_3$$

We have therefore shown that xy cannot be (S, S_j)-stable, and therefore not (S, T)-stable, which concludes the proof. □

Let S' be a substring obtained from S by tandem duplications, and let $X := S'[a..b]$ be the substring of S' at position from a to b. Suppose that we apply a duplication $D = (c, d)$, which copies the substring $S'[c..d]$. Then we say that D *cuts* X if one of the following holds:

- $a < c \leq b$ and $b < d$, in which case we say that D cuts X to the right;
- $c < a$ and $a \leq d < b$, in which case we say that D cuts X to the left;
- $(a, b) \neq (c, d)$ and $a \leq c < d \leq b$, in which case D cuts X inside;

In other words, if we write $X = X_1X_2$ and $S' = UVX_1X_2WY$, cutting to the right takes the form $UVX_1\underline{X_2W}Y \Rightarrow UBX_1\underline{X_2WX_2W}Y$. Cutting to the left takes the from $U\underline{VX_1}X_2WY \Rightarrow U\underline{VX_1VX_1}X_2WY$. Rewriting $X = X_1X_2X_3$ and $S' = UX_1X_2X_3V$, cutting inside takes the form $UX_1\underline{X_2}X_3V \Rightarrow UX_1\underline{X_2X_2}X_3V$. Note that if D does not cut any occurrence of a maximal (S, S')-stable substring X and S'' is obtained by applying D on S', then X is (S, S'')-stable.

The next lemma shows that we can assume that maximal stable substrings never get cut, and thus always get duplicated together. The idea is that any duplication that cuts an X_j can be replaced by an equivalent duplication that doesn't.

Lemma 3. *Suppose that* $dist_{TD}(S,T) = k$*, and let* $X_1, \ldots, X_l$ *be the set of maximal* (S,T)*-stable substrings. Then there exists a sequence of tandem duplications* $D_1, \ldots, D_k$ *transforming* S *into* T *such that no occurrence of an* X_j *gets cut by a* D_i*.*

In other word, for all $i \in [k]$ *and all* $j \in [l]$*,* D_i *does not cut any occurrence of* X_j *in the string obtained by applying* $D_1, \ldots D_{i-1}$ *to* S*.*

Proof. Let $D_1^*, \ldots, D_k^*$ be a sequence of tandem duplications transforming S into T, and for $i \in [k]$, let S_i be the string obtained by applying the first i duplications. Put $S_0 := S$. We show that any S_i, $i \in [k]$, can be obtained from S_{i-1} by a duplication D_i that does not cut any X_j occurrence in S_{j-1}, $j \in [l]$. This proves the lemma, since $D_1, \ldots, D_k$ will form the desired sequence of duplications.

Fix $i \in [k]$, and assume that D_i^* cuts some of the X_j's. We note that since S is exemplar, the X_j's have pairwise distinct characters. Hence D_i^* can cut at most two occurrences of a maximal (S, S_i)-stable substrings, at most one to the left and at most one to the right (if an X_j is cut inside, only one string can get cut). Also, by Lemma 1, we know that every X_j substring is (S, S_i)-stable. We have four cases to consider:

- D_i^* cuts some X_j inside. Write $X_j = X_j^1 X_j^2 X_j^3$, where at least one of X_j^1 or X_j^2 is non-empty, and $S_{i-1} = A X_j^1 X_j^2 X_j^3 B$. This results in

$$S_{i-1} = A X_j^1 \underline{X_j^2} X_j^3 B \Rightarrow A X_j^1 \underline{X_j^2 X_j^2} X_j^3 B = S_i$$

  Since characters from X_j^1, X_j^2 and X_j^3 are pairwise disjoint, X_j cannot be (S, S_i)-stable, a contradiction of Lemma 1.

- D_i^* cuts some X_j to the right, but no other string to the left. Then we may write X_j and S_{i-1}, respectively, as $X_j = X_j^1 X_j^2$ and $S_{i-1} = A X_j^1 X_j^2 BC$ such that D copies the

substring $X_j^2 B$. This gives

$$S_{i-1} = AX_j^1 \underline{X_j^2 BC} \Rightarrow AX_j^1 \underline{X_j^2 BX_j^2 BC} = S_i$$

But by Lemma 1, X_j is (S, S_i)-stable. Since S_i has BX_j^2 as a substring, this must mean that $B = \hat{B}X_j^1$ for some substring $\hat{B}$ (note that we use the fact that X_j^2 has distinct characters, and thus that the occurrence of X_j^1 must be entirely in B). Therefore $S_{i-1} = AX_j^1 X_j^2 \hat{B} X_j^1 C$. Since X_j is also (S, S_{i-1})-stable, this in turn implies that $C = X_j^2 \hat{C}$ for some substring $\hat{C}$, and in fact we get

$$S_{i-1} = AX_j^1 \underline{X_j^2 \hat{B} X_j^1} X_j^2 \hat{C} \Rightarrow AX_j^1 \underline{X_j^2 \hat{B} X_j^1 X_j^2 \hat{B} X_j^1} X_j^2 \hat{C} = S_i$$

We can replace D_i^* by a duplication that copies $X_j^1 X_j^2 \hat{B}$, i.e.,

$$S_{i-1} = A\underline{X_j^1 X_j^2 \hat{B}} X_j^1 X_j^2 \hat{C} \Rightarrow A\underline{X_j^1 X_j^2 \hat{B} X_j^1 X_j^2 \hat{B}} X_j^1 X_j^2 \hat{C} = S_i$$

Since this duplication starts with X_j and copies itself right before another occurrence of X_j, it is clear that it does not cut any maximal (S, T)-stable substring, as desired.

- D_i^* cuts some X_j to the left, but cuts no string to the right. Then we may write

$$S_{i-1} = A\underline{BX_j^1} X_j^2 C \Rightarrow A\underline{BX_j^1 BX_j^1} X_j^2 C = S_i$$

Similarly as in the previous case, since X_j is (S, S_i)-stable, we must have $B = X_j^2 \hat{B}$. We are led to deduce that $A = \hat{A}X_j^1$. Therefore we have

$$S_{i-1} = \hat{A}X_j^1 \underline{X_j^2 \hat{B} X_j^1} X_j^2 C \Rightarrow \hat{A}X_j^1 \underline{X_j^2 \hat{B} X_j^1 X_j^2 \hat{B} X_j^1} X_j^2 C = S_i$$

As before, we could instead duplicate the substring $X_j^1 X_j^2 \hat{B}$ occuring right after $\hat{A}$.

- D_i^* cuts some X_j to the left and some X_h to the right. Note that $X_j = X_h$ is possible, which we will in fact show to hold. We may write $X_j = X_j^1 X_j^2$ and $X_h = X_h^1 X_h^2$ such that we get

$$S_{i-1} = AX_j^1 \underline{X_j^2 B X_h^1} X_h^2 C \Rightarrow AX_j^1 \underline{X_j^2 B X_h^1 X_j^2 B X_h^1} X_h^2 C = S_i$$

Now, X_j is (S, S_i)-stable and S_i contains $X_h^1 X_j^2$ as a substring. It follows that the last character of X_h^1 must be the last character of X_j^1 (applying the (S, X_j)-stability argument on the $X_h^1 X_j^2$ substring). In other word, X_h and X_j have a character in common. Since S is exemplar, the set of maximal (S, T)-stable strings $X_1, \ldots, X_l$ have pairwise disjoint sets of characters and partition S into substrings. We deduce that $X_j = X_h$, as we predicted.

We now want to show that $X_h^1 = X_j^1$. Note that both X_j^1 and X_h^1 are prefixes of X_j (for X_h^1, this is because $X_h = X_j$). Moreover, as argued the last character of X_h^1 is also the last character of X_j^1. These two observations establish that $X_h^1 = X_j^1$ (and therefore $X_h^2 = X_j^2$). This allows us to rewrite S_i and S_{i+1} as

$$S_{i-1} = AX_j^1 \underline{X_j^2 B X_j^1} X_j^2 C \Rightarrow AX_j^1 \underline{X_j^2 B X_j^1 X_j^2 B X_j^1} X_j^2 C = S_i$$

It becomes clear that we can duplicate the $X_j^1 X_j^2 B$ substring after A in S_{i-1} to obtain S_i. This duplication does not cut any maximal (S, T)-stable substring.

We have thus shown that if D_i^* cuts some occurrence of one or more of the X_j's then D_i^* can be replaced by a duplication D_i that yields the same string S_i as D_i^*. The only case remaining is when D_i^* does not cut any X_j. In that case, we set $D_i = D_i^*$. This shows that we can find the claimed sequence $D_1, \ldots, D_k$ in which no X_j ever gets cut. $\square$

The above implies that we may replace each maximal (S,T)-stable substring X of S and T by a single character, since we may assume that characters of X are always duplicated together (assuming, of course, that S is exemplar). It only remains to show that the resulting strings are small enough. The proof of the following lemma has a very simple intuition. First, S has exactly 1 maximal (S,S)-stable substring. Each time we apply a duplication, we "break" at most 2 stable substrings, which creates 2 new ones. So if we apply k duplications, there are at most $2k+1$ such substrings in the end.

Lemma 4. *If $dist_{TD}(S,T) \leq k$, then there are at most $2k+1$ maximal (S,T)-stable substrings.*

Proof. Let $S = S_0, S_1, \ldots, S_k = T$ be any minimum sequence of strings transforming S to T by tandem duplications. We show by induction that, for each $i \in \{0,1,\ldots,k\}$, the number of maximal (S,S_i)-stable substrings is at most $2i+1$. For $i=0$, there is only one maximal (S,S)-stable substring, namely S itself. Now assume that there are at most $2(i-1)+1 = 2i-1$ maximal (S,S_{i-1})-stable substrings. Let $\mathcal{X} = \{X_1,\ldots,X_l\}$ be the set of these substrings, $l \leq 2i-1$. We then know that S_{i-1} can be written as a concatenation of X_j's from $\mathcal{X}$ (with possible repetitions). The duplication D transforming S_{i-1} to S_i copies some of these X_j's entirely, except at most two X_j's at the ends which it may copy partially (i.e., D cuts at most two substrings from $\mathcal{X}$). In other words, the substring duplicated by D can be written as $X_j^2 X_{a_1} X_{a_2} \ldots X_{a_r} X_h^1$, where $X_j = X_j^1 X_j^2$ and $X_h = X_h^1 X_h^2$ from some $j,h \in [l]$ (and $X_{a_1},\ldots,X_{a_r} \in \mathcal{X}$). Going further, S_{i-1} and S_i can be written, using appropriate substrings A,B,C that are concatenation of elements of $\mathcal{X}$, as

$$S_{i-1} = AX_j^1 \underline{X_j^2 B X_h^1} X_h^2 C \Rightarrow AX_j^1 \underline{X_j^2 B X_h^1 X_j^2 B X_h^1} X_h^2 C = S_i$$

Now, any $X_r \in \mathcal{X} \setminus \{X_j, X_h\}$ is (S,S_i)-stable. Moreover, X_j^1, X_j^2, X_h^1 and X_h^2 are also (S,S_i)-stable. This shows that the number of maximal (S,S_i)-stable substrings is at most

$2i - 1 - 2 + 4 = 2i + 1$, as desired. □

We can now transform an instance (S, T) of **Exemplar-k-TD** to a kernel, an equivalent instance (S', T') of size depending only on k.

Theorem 4. *An instance (S, T) of **Exemplar-k-TD** admits a kernel (S', T') in which $|S'| \leq 2k + 1$ and $|T'| \leq (2k + 1)2^k$.*

Proof. Let S', T' be obtained from an instance (S, T) by replacing each maximal (S, T)-stable substring by a distinct character. We first prove that (S', T') is indeed a kernel by establishing its equivalence with (S, T). Clearly if (S', T') can be solved using at most k duplications, then the same applies to (S, T). By Lemma 2, the converse also holds: if (S, T) can be solved with at most k duplications, we may assume that these duplications never cut a maximal (S, T)-stable substring, and so these duplications can be applies on (S', T').

Then by Lemma 3, we know that S' has at most $2k+1$ characters. If $dist_{TD}(S', T') \leq k$, then each duplication can at most double the size of the previous string. Therefore, T' must have size at most $(2k + 1)2^k$. □

The kernelization can be performed in polynomial time, as one only needs to identify maximal (S, T)-stable substrings and contract them (we do not bother with the exact complexity for now). Running the brute-force algorithm from Proposition 1 yields the following.

Corollary 1. *The exemplar k-tandem duplication problem can be solved in time $O(((2k + 1)2^k)^{2k} + poly(n)) = 2^{O(k^2)} + poly(n)$, where n is the size of the input.*

2.6 Conclusion

In this chapter, we solve the k-Tandem Duplication problem posed by Leupold et al. in 2004 and show that computing the TD distance from a string S to a string T is NP-hard.

We show that this result holds even if S is *exemplar*, i.e., if each character of S is distinct. Exemplar string are commonly studied in computational biology [80], since they represent genomes that existed prior to duplication events. We note that simply deciding if S can be transformed into T by a sequence of TDs still has unknown complexity. In our case, we show that the hardness of minimizing TDs holds on instances in which such a sequence is guaranteed to exist.

As demonstrated by the transpositions distance in [17], obtaining NP-hardness results for string distances can sometimes be an involving task. Our hardness reduction is also quite technical, and one of the tools we develop for it is a new problem we call the *Cost-Effective Subgraph*. In this problem, we are given a graph $G = (V, E)$ with a cost c, and we must choose a subset X of V. Each edge with both endpoints in X has a cost of $|X|$, every other edge costs c, and the goal is to find a subset X of minimum cost. We show that this problem is W[1]-hard for parameter $p + c$, where p is the cost that we can save below the upper bound $c|E(G)|^3$. The problem enforces optimizing the tradeoff between covering many edges versus having a large subset of high cost, which might be applicable to other problems. In our case, it captures the main difficulty in computing TD distances. Then, we proved this problem is still NP-hard, even if $|\Sigma| \geq 4$ by encoding each letter in the unbounded alphabet with a square-free string over a new alphabet of size 4. Finally, we obtain positive results by showing that if S is exemplar, then one can decide if S can transformed into T using at most k duplications in time $2^{O(k^2)} + poly(n)$. The result is obtained through an exponential size kernel.

2.6.1 Note

The results in this chapter have been first presented at STACS 2020 [61], and then as a journal paper in SIAM Journal on Discrete Mathematics [62].

CHAPTER THREE

THE LONGEST LETTER-DUPLICATED SUBSEQUENCE PROBLEM

3.1 Introduction

Given a sequence S of length n, a letter-duplicated subsequence (LDS) of S is a subsequence of S in the form $x_1^{d_1}x_2^{d_2}\cdots x_k^{d_k}$ 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.) Naturally, the problem of computing the longest letter-duplicated subsequence (LLDS) of S can be defined, and a simple linear time algorithm can be obtained.

In this chapter, we focus on important variants around the fundamental LLDS problem, focusing on the constrained and weighted cases. The constraint is to demand that all letters in Σ appear in a resulting LDS, which simulates that in a genome with duplicated genes, how to compute the maximum duplicated pattern while including all the genes. Then we have two problems: feasibility testing (FT for short, which decides whether an LDS of S containing all letters in Σ exists) and the problem of maximizing the length of a resulting LDS where all letters in the alphabet appear, which we call LLDS+. It turns out that the status of these two problems change quite a bit when d, the maximum number a letter can appear in S, varies. We denote the corresponding problems as $FT(d)$ and $LLDS+(d)$ respectively. Let $|S| = n$, we summarize our main results in this paper as follows:

1. We show that when $d \geq 6$, both $FT(d)$ and (the decision version of) $LLDS+(d)$ are NP-complete, which implies that $LLDS+(d)$ does not have a polynomial-time approximation algorithm when $d \geq 6$.

2. We show that when $d = 3$, $FT(d)$ is decidable in $O(n^2)$ time, which implies that $LLDS+(3)$ admits a factor-1.5 approximation. With an increasing running time, we

could improve the factor to $1.5 - O(\frac{1}{n})$.

3. When a weight of an LD-block is any positive function (i.e., it does not even have to grow with its length), we present a non-trivial $O(n^2)$ time dynamic programming algorithm for this Weighted-LDS problem.

In the literature, the only known related work is to compute the longest *square* subsequence of an input sequence S, for which Kosowski gave an $O(n^2)$ time algorithm [59]. In this chapter, we briefly mention two extra variations of the LLDS problem, where in the solution, i.e., a subsequence of S in the form of $x_1^{d_1}x_2^{d_2}\cdots x_k^{d_k}$, each x_i is either a substring or a subsequence of S. Then, what Kosowski considered is the more restricted version of the latter, i.e., $x_1^{d_1}x_2^{d_1}$, with $x_1 = x_2$ and $d_1 = d_2$.

3.2 Preliminaries

Let $\mathbb{N}$ be the set of natural numbers. For $q \in \mathbb{N}$, we use $[q]$ to represent the set $\{1, 2, ..., q\}$. Throughout this chapter, a sequence S is over a finite alphabet Σ. We use $S[i]$ to denote the i-th letter in S and $S[i..j]$ to denote the substring of S starting and ending with indices i and j respectively. (Sometimes we also use $(S[i], S[j])$ as an interval representing the substring $S[i..j]$.) With the standard run-length representation, S can be represented as $y_1^{a_1}y_2^{a_2}\cdots y_q^{a_q}$, with $y_i \in \Sigma, y_j \neq y_{j+1}$ and $a_j \geq 1$, for $i \in [q], j \in [q-1]$. If a letter a appears multiple times in S, we could use $a^{(i)}$ to denote the i-th copy of it (reading from left to right). Finally, a *subsequence* of S is a string obtained by deleting a set of letters from S.

3.3 The LLDS Problem

A subsequence S' of S is a letter-duplicated subsequence (LDS) of S if it is in the form of $x_1^{d_1}x_2^{d_2}\cdots x_k^{d_k}$, with $x_i \in \Sigma, x_j \neq x_{j+1}$ and $d_i \geq 2$, for $i \in [k], j \in [k-1]$. We call each $x_i^{d_i}$ in S' a *letter-duplicated block* (LD-block, for short). For instance, let $S = abcacabcb$, then

$S_1 = aaabb$, $S_2 = ccbb$ and $S_3 = ccc$ are all letter-duplicated subsequences of S, where aaa and bb in S_1, cc and bb in S_2, and ccc in S_3 all form the corresponding LD-blocks. Certainly, we are interested in the longest ones — which gives us the longest letter-duplicated subsequence (LLDS) problem.

As a warm-up, we solve this problem by dynamic programming. We first have the following observation.

Observation 1. *Suppose that there is an optimal LLDS solution for a given sequence S of length n, in the form of $x_1^{d_1} x_2^{d_2} \dots x_k^{d_k}$. Then it is possible to decompose it into a generalized LD-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* $j \in [p-1]$.

The proof is straightforward: For any natural number $\ell \geq 2$, we can decompose it as $\ell = \ell_1 + \ell_2 + \ldots + \ell_z \geq 2$, such that $2 \leq \ell_j \leq 3$ for $1 \leq j \leq z$. Consequently, for every $d_i > 3$, we could decompose it into a sum of 2's and 3's. Then, clearly, given a generalized LD-subsequence, we could easily obtain the corresponding LD-subsequence by combining $y_i^{e_i} y_{i+1}^{e_{i+1}}$ when $y_i = y_{i+1}$.

We now design a dynamic programming algorithm for LLDS. Let $L(i)$ be the length of

the optimal LLDS solution for $S[1..i]$. The recurrence for $L(i)$ is as follows.

$$L(0) = 0,$$

$$L(1) = 0,$$

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

Note that the step involving $L(i-x)+1$ is essentially a way to handle a generalized LD-subsequence of length 3 (by keeping $S[i-x]$ for the next level computation) and cannot be omitted following the above observation. For instance, if $S = dabcdd$ then without that step we would miss the optimal solution ddd.

The value of the optimal LLDS solution for S can be found in $L(n)$. For the running time, for each $S[x]$ we just need to scan S to find the closest $S[i]$ such that $S[x] = S[i]$. With this information, the table L can be filled in linear time. With a simple augmentation, the actual sequence corresponding to $L(n)$ can also be found in linear time. Hence LLDS can be solved in $O(n)$ time.

3.4 The *LLDS+* Problem

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

Definition 4. Constrained Longest Letter-Duplicated Subsequence
(LLDS+ for short)

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 5. Feasibility Testing *(FT for short)*

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''?*

For LLDS+ we are really interested in the optimization version, i.e., to maximize ℓ. Note that, though looking similar, FT and the decision version of LLDS+ are different: if there is no feasible solution for FT, certainly there is no solution for LLDS+; but even if there is a feasible solution for FT, computing an optimal solution for LLDS+ could still be non-trivial.

Finally, let d be the maximum number of times a letter in Σ appears in S. Then, we can represent the corresponding versions for LLDS+ and FT as $LLDS+(d)$ and $FT(d)$ respectively.

It turns out that (the decision version of) $LLDS+(d)$ and $FT(d)$ are both NP-complete when $d \geq 6$, while when $d = 3$ the status varies: $FT(3)$ can be decided in $O(n^2)$ time, which immediately implies that $LLDS+(3)$ has a factor-1.5 approximation. (If we are willing to increase the running time, still polynomial but higher than $O(n^2)$, with a simple twist we could improve the approximation factor for $LLDS+(3)$ to $1.5-O(\frac{1}{n})$.) We present the details in the next subsection. In Section 3.5, we consider an extra version of LLDS, Weighted-LDS, where the weight of an LD-block is an arbitrary positive function.

3.4.1 Hardness for LLDS+(d) and FT(d) when $d \geq 6$

We first try to prove the NP-completeness of the (decision version of) LLDS+(d), when $d \geq 6$. In fact, we need a very special version of SAT, which is possibly the simplest version of SAT remaining NP-complete.

Given a 3SAT formula ϕ, which is a conjunction of m disjunctive clauses (over n variable x_i's), each clause F_j containing exactly 3 literals (i.e., in the form of x_i or $\bar{x}_i$), the problem

is to find whether there is a satisfiable truth assignment for ϕ.

Definition 6. $(\leq 2, 1, \leq 3)$-*SAT: this is a special case of SAT where each variable* x_i *appears at most twice and* $\bar{x}_i$ *appears exactly once in* ϕ*; moreover, each clause contains either two or three literals (which is called 2-clause and 3-clause henceforth).*

Lemma 5. $(\leq 2, 1, \leq 3)$-*SAT is NP-complete.*

Proof. We modify the proof by Tovey [90]. Given a 3-SAT formula ϕ, without loss of generality, assume that each variable x_i and its complement $\bar{x}_i$ appears in (different clauses of) ϕ. We convert ϕ to ϕ' in the form of $(\leq 2, 1 \leq 3)$-SAT as follows.

- if both x_i and $\bar{x}_i$ appears once in ϕ, do nothing.
- if x_i appears twice and $\bar{x}_i$ appears once in ϕ, do nothing.
- if $\bar{x}_i$ appears twice and x_i appears once in ϕ, replace $\bar{x}_i$ with a new variable z and replace x_i by $\bar{z}$.
- Otherwise, if the total number of literals of x_i (i.e., x_i and $\bar{x}_i$) is $k \geq 4$ then introduce k variables $y_{i,1}, y_{i,2}, \cdots, y_{i,k}$ replacing the k literals of x_i respectively. Moreover, let $z_{i,j}$ be $y_{i,j}$ if the j-th literal of x_i is x_i and let $z_{i,j}$ be $\bar{y}_{i,j}$ if the j-th literal of x_i is $\bar{x}_i$. Finally, add k 2-clauses as $(z_{i,j} \vee \bar{z}_{i,j+1})$ for $j = 1..k-1$ and $(z_{i,k} \vee \bar{z}_{i,1})$. (Note that it always holds that $\bar{\bar{z}} = z$.)

Following [90], when $k \geq 4$, the 2-clauses added will force all $z_{i,j}$'s to have all `True` values or all `False` values. (The only difference between our construction and Tovey's is that all literals appearing at least 4 times in the original clauses in ϕ are replaced by positive variables in the form of $y_{i,j}$'s; the negated literal $\bar{y}_{i,j}$ could only occur in the newly created 2-clauses — exactly once for each $y_{i,j}$. On the other hand, each $y_{i,j}$ occur twice — once in the original 3-clauses of ϕ and once in the newly created 2-clauses.) It is obvious to see that

ϕ is satisfiable if and only if ϕ' is satisfiable. The transformation obviously takes $O(|\phi|)$ time. Hence the lemma is proven. □

We comment that $(\leq 2, 1 \leq 3)$-SAT, while seemingly similar to SAT3W (each clause has at most 3 literals and each clause has at most one negated variable [84]), is in fact different from it. (Following the Dichotomy Theorem for SAT by Schaefer [84], SAT3W is in P.) The difference is that in ϕ' we could even have a clause containing 3 negated variables.

Now let ϕ be an instance of $(\leq 2, 1, \leq 3)$-SAT where either x_i and $\bar{x}_i$ appears once in ϕ (we call such an x_i a *(1,1)-variable*), or x_i appears twice and $\bar{x}_i$ appears once in ϕ (we call such an x_i a *(2,1)-variable*), for $i = 1..n$. Without loss of generality, we assume $\phi = F_1 \wedge F_2 \wedge \cdots \wedge F_m$ and there are n variables $x_1, x_2, \cdots, x_n$; moreover, we assume that F_j cannot contain x_i and $\bar{x}_i$ at the same time. Given F_j we say F_jF_j forms a *2-duplicated clause-string*.

Given a *(2,1)-sequence* $T = ACABCB$ over $\{A, B, C\}$, where A, B and C all appear twice, it is easy to verify that the maximal LD-subsequences of T are $AABB$ or CC. Similarly, given a *(1,1)-sequence* $T = ACCA$ over $\{A, C\}$, where A and C both appear twice, it is easy to see that the maximal (longest) LD-subsequences of T are AA or CC.

For each (1,1)-variable x_i, i.e., x_i and $\bar{x}_i$ appears once in ϕ, say x_i in F_j and $\bar{x}_i$ in F_k, we define L_i as a (1,1)-sequence: $F_jF_kF_kF_j$. For each (2,1)-variable x_i, i.e., x_i appears twice and $\bar{x}_i$ appears once in ϕ, say x_i in F_j and F_k, and $\bar{x}_i$ in F_ℓ, we define L_i as a (2,1)-sequence: $F_jF_\ell F_jF_kF_\ell F_k$.

Now we proceed to construct the sequence S from an $(\leq 2, 1, \leq 3)$-SAT instance ϕ.

$$S = g_1g_1L_1g_2g_2 \cdots g_ig_iL_i \cdots g_{n-1}g_{n-1}L_{n-1}g_ng_nL_ng_{n+1}g_{n+1}.$$

We claim the following: ϕ is satisfiable if and only if LLDS+ has a solution of length at least $2(n+1) + 4K_1 + 2K_2 + 2J$, where K_1, K_2 are the number of (2,1)-variables in ϕ which

are assigned `True` and `False` respectively and J is the number of (1,1)-variables in ϕ.

Proof. "Only-if part": Suppose that ϕ is satisfiable. If a (1,1)-variable x_i is assigned `True`, to have a solution for LLDS+, in L_i we select the 2-duplicated clause-string F_jF_j; likewise, if x_i is assigned `False` we select F_kF_k instead. Similarly, if a (2,1)-variable x_i is assigned `True`, to have a solution for LLDS+, in L_i we select two 2-duplicated clause-strings $F_jF_jF_kF_k$; likewise, if x_i is assigned `False` we select $F_\ell F_\ell$. Since g_ig_i only occurs once in S and T, we must include them in the solution. Clearly we have a solution for LLDS+ with length $2(n+1)+4K_1+2K_2+2J$.

"If part": If LLDS+ has a solution of length at least $2(n+1)+4K_1+2K_2+2J$, by definition, it must contain all g_ig_i's. To find the truth assignment, we look at the contents between g_ig_i and $g_{i+1}g_{i+1}$ in the solution as well as in S (i.e., L_i). If x_i is a (1,1)-variable, $L_i = F_jF_kF_kF_j$ and in the solution F_jF_j is kept then we assign $x_i \leftarrow$ `True`; otherwise, we assign $x_i \leftarrow$ `False`. If x_i is a (2,1)-variable, $L_i = F_jF_\ell F_jF_kF_\ell F_k$ and in the solution either $F_jF_jF_kF_k$, F_jF_j or F_kF_k is kept then we assign $x_i \leftarrow$ `True`. (When F_jF_j or F_kF_k is kept, then the LLDS+ solution could be longer by augmenting this sub-solution to $F_jF_jF_kF_k$.) If in the solution $F_\ell F_\ell$ is kept instead then we assign $x_i \leftarrow$ `False`. Since all clauses must appear in a solution of LLDS+, clearly ϕ is satisfied. □

We comment that $2(n+1)+4K_1+2K_2+2J = 2(n+1)+2K_1+2n = 4n+2+2K_1$, as $K_1+K_2+J = n$. But the former makes our arguments more clear. This reduction obviously takes $O(m+n)$ time. Note that each 3-clause F_j appears 6 times in S and each 2-clause F_ℓ appears 4 times in S respectively, while each $g_k, k \in [n+1]$, appears twice in S. Since we could arbitrarily add an LD-block u^j, with $u \notin \Sigma$ and $j \geq 6$, at the end of S, we have the following theorem.

Theorem 5. *The decision version of LLDS+(d) is NP-complete for $d \geq 6$.*

We next present an example for this proof.

Example. Let $\phi = F_1 \wedge F_2 \wedge F_3 \wedge F_4 \wedge F_5 = (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (x_2 \vee \bar{x}_3 \vee x_4) \wedge (x_1 \vee x_4 \vee x_5) \wedge (\bar{x}_4 \vee \bar{x}_5)$. Then

$$S = g_1 g_1 F_1 F_2 F_1 F_4 F_2 F_4 \cdot g_2 g_2 F_1 F_2 F_1 F_3 F_2 F_3 \cdot g_3 g_3 F_1 F_3 F_1 F_2 F_3 F_2$$

$$\cdot g_4 g_4 F_3 F_5 F_3 F_4 F_5 F_4 \cdot g_5 g_5 F_4 F_5 F_5 F_4 \cdot g_6 g_6,$$

Corresponding to the truth assignment, $x_1, x_4 =$ `True` and $x_2, x_3, x_5 =$ `False`, we have

$$S' = g_1 g_1 F_1 F_1 F_4 F_4 \cdot g_2 g_2 F_2 F_2 \cdot g_3 g_3 F_3 F_3 \cdot g_4 g_4 F_3 F_3 F_4 F_4 \cdot g_5 g_5 F_5 F_5 \cdot g_6 g_6,$$

which is of length $2(5+1) + 4 \times K_1 + 2 \times K_2 + 2 \times 1 = 12 + 4 \times 2 + 2 \times 2 + 2 = 26$.

The above theorem implies the following corollary.

Corollary 2. *FT(d) is NP-complete for $d \geq 6$.*

Proof. The reduction remains the same. We just need to augment the proof in the reverse direction. Suppose there is a feasible solution S'' for S for the feasibility testing problem. Again, all $g_i g_i$'s must be in S''. We now look at the contents between $g_i g_i$ and $g_{i+1} g_{i+1}$ in S (i.e., L_i) and S''. Corresponding to L_i, if in S'' we have an empty string between $g_i g_i$ and $g_{i+1} g_{i+1}$, then we can assign x_i either as `True` or `False`. If $L_i = F_j F_k F_k F_j$, i.e., x_i is a (1,1)-variable, and $F_j F_j$ is kept in S'' then we assign $x_i \leftarrow$ `True`; otherwise, we assign $x_i \leftarrow$ `False`. If $L_i = F_j F_\ell F_j F_k F_\ell F_k$, i.e., x_i is a (2,1)-variable, and either $F_j F_j F_k F_k$, $F_j F_j$ or $F_k F_k$ is kept in S'' then we assign $x_i \leftarrow$ `True`. If in the solution $F_\ell F_\ell$ is kept instead then we assign $x_i \leftarrow$ `False`. By definition, all clauses must appear in S'' (solution of FT), clearly ϕ is satisfied. It is clear that FT belongs to NP as a solution can be easily checked in polynomial time. □

The above corollary essentially implies that the optimization version of *LLDS+(d)*,

$d \geq 6$, does not admit any polynomial-time approximation algorithm (regardless of the approximation factor), since any such approximation would have to return a feasible solution. A natural direction to approach LLDS+ is to design a bicriteria approximation for LLDS+, where a factor-(α, β) bicriteria approximation algorithm is a polynomial-time algorithm which returns a solution of length at least OPT/α and includes at least N/β letters, where $N = |\Sigma|$ and OPT is the optimal solution value of LLDS+. We show that obtaining a bicriteria approximation algorithm for LLDS+ is no easier than approximating LLDS+ itself.

Corollary 3. *If* LLDS+$(d), d \geq 6$*, 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 *approximation, where* N *is the alphabet size.*

Proof. Suppose that a factor-$(\alpha, N^{1-\epsilon})$ bicriteria approximation algorithm $\mathcal{A}$ exists. We construct an instance S^* for *LLDS+*(6) as follows. (Recall that S is the sequence we constructed from an $(\leq 2, 1 \leq 3)$-SAT instance ϕ in the proof of Theorem 1.) In addition to $\{F_i | i = 1..m\} \cup \{g_j | j = 1..n+1\}$ in the alphabet, we use a set of integers $\{1, 2, ..., (m+n+1)^x - (m+n+1)\}$, where x is some integer to be determined. Hence,

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

We now construct S^* as

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

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

Clearly, any bicriteria approximation for S^* would return an approximate solution for S as including any number in $\{1, 2, ..., (m+n+1)^x - (m+n+1)\}$ would result in a solution of size only 2.

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

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

which means it suffices to choose $x \geq \lceil 2-\epsilon \rceil = 2$. □

3.4.2 Solving the Feasibility Testing Version for $d = 3$

For the Feasibility Testing Version, as mentioned earlier, Corollary 2 implies that the problem is NP-complete when $d \geq 6$. We next show that if $d = 3$, then the problem can be decided in polynomial time.

Lemma 6. *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 $FT(3)$ which only uses 2-blocks.*

Proof. Suppose that $S = \cdots a^{(1)} \cdots a^{(2)} \cdots a^{(3)} \cdots$, and $a^{(1)}a^{(2)}a^{(3)}$ is a 3-block in a feasible solution for $FT(3)$. (Recall that the superscript only indicates the appearance order of letter a.) Then we could replace $a^{(1)}a^{(2)}a^{(3)}$ by either $a^{(1)}a^{(2)}$ or $a^{(2)}a^{(3)}$. The resulting solution is still a feasible solution for $FT(3)$. □

Lemma 6 implies that the $FT(3)$ problem can be solved using 2-SAT. For each letter a, we denote the interval $(a^{(1)}, a^{(2)})$ as a variable v_a, and we denote $(a^{(2)}, a^{(3)})$ as $\bar{v}_a$. (Clearly one cannot select $a^{(1)}a^{(2)}$ and $a^{(2)}a^{(3)}$ as 2-blocks at the same time.) Then, if another interval $(b^{(1)}, b^{(2)})$ overlaps the interval $(a^{(1)}, a^{(2)})$, we have a 2-SAT clause $\overline{v_a \wedge v_b} = (\bar{v}_a \vee \bar{v}_b)$. Forming a 2-SAT instance ϕ" for all such overlapping intervals and it is clear that we can decide whether ϕ" is satisfiable in $O(n^2)$ time (as we could have $O(n^2)$ pairs of overlapping intervals).

Theorem 6. *Let S be a string of length n. Whether $FT(3)$ has a solution can be decided in $O(n^2)$ time.*

The theorem immediately 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). In the following, we extend this trivial observation to have a factor-$(1.5 - O(\frac{1}{n}))$ approximation for LLDS+(3).

Corollary 4. *Let S be a string of length n such that each letter appears at most 3 times in S. Then LLDS+(3) admits a polynomial-time approximation algorithm with a factor of $1.5 - O(\frac{1}{n})$ if a feasible solution exists.*

Proof. First fix some constant (positive integer) D ($D < |\Sigma|$). Then for $t = 1$ to D, we enumerate all the sets which contains letters appearing exactly 3 times in S. For a fixed t, let such a set be $F_t = \{a_1, a_2, ..., a_t\}$. We put the 3-blocks $a_i^{(1)}a_i^{(2)}a_i^{(3)}$, $i = 1..t$, in the solution. (If two such 3-blocks overlap, then we immediately stop to try a different set F_t'; and if all valid sets of size t have been tried, we increment t to $t + 1$.) The substrings of S, between $a_i^{(1)}$ and $a_i^{(2)}$, and $a_i^{(2)}$ and $a_i^{(3)}$, will then be deleted. Finally, for the remaining letters we use 2-SAT to test whether all together, with the 3-blocks, they form a feasible solution (note that $a_i^{(1)}a_i^{(2)}a_i^{(3)}$ will serve as an obstacle and no valid interval for 2-SAT should contain it), this can be checked in $O(n^2)$ time following Theorem 6. Clearly, with this algorithm, either we compute the optimal solution with at most D 3-blocks, or we obtain an approximate solution of value $2|\Sigma| + D$. Since OPT is at most $3|\Sigma|$, the approximation factor is

$$\frac{3|\Sigma|}{2|\Sigma| + D} = 1.5 - O(\frac{1}{|\Sigma|}),$$

which is $1.5 - O(\frac{1}{n})$, because $|\Sigma|$ is at least $\lceil n/3 \rceil$. The running time of the algorithm is $O(\binom{|\Sigma|}{D} \cdot O(n^2)) = O(n^{D+2})$, which is polynomial as long as D is a constant. □

In the next section, we show that if the LD-blocks are arbitrarily positively weighted, then the problem can be solved in $O(n^2)$ time. Note that the $O(n)$ time algorithm for the

LLDS problem assumes that the weight of any LD-block is its length, which has the property that $\ell(s) = \ell(s_1) + \ell(s_2)$, where $s = s_1 s_2$, s_1 and s_2 are LD-blocks on the same letter x, and $\ell(s)$ is the length of s (or the total number of letters of x in s_1 and s_2).

3.5 The Weighted-LDS Problem

Given the input string $S = S[1...n]$, let $w_x(\ell)$ be the weight of LD-block $x^\ell, x \in \Sigma, 2 \leq \ell \leq d$, where d is the maximum number of times a letter appears in S. Here, the weight can be thought of as a positive function of x and ℓ and it does not even have to be increasing on ℓ. For example, it could be that $w(aaa) = w_a(3) = 8, w(aaaa) = w_a(4) = 5$. Given $w_x(\ell)$ for all $x \in \Sigma$ and ℓ, we aim to compute the maximum weight letter-duplicated string (Weighted-LDS) using dynamic programming.

Define $T(n)$ as the value of the optimal solution of $S[1...n]$ which contains the character $S[n]$. Define $w[i, j]$ as the maximum weight LD-block $S[j]^\ell$ ($\ell \geq 2$) starting at position i and ending at position j; if such an LD-block does not exist, then $w[i, j] = 0$. Notice that $S[j]^\ell$ does not necessarily have to contain $S[i]$ but it must contain $S[j]$. We have the following recurrence relation.

$$T(0) = 0,$$

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

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

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

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

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 $S[0] = \emptyset$.

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

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

and,

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

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

3. Finally, we compute

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

Table 3.1: Input table for $w_x(\ell)$, with $S = ababbaca$ and $d = 4$.

$x\backslash\ell$	1	2	3	4
a	5	10	20	15
b	4	16	8	3
c	1	3	5	7

Table 3.2: Table $w'_x(\ell)$, with $S = ababbaca$ and $d = 4$.

$x\backslash\ell$	1	2	3	4
a	5	10	20	20
b	4	16	16	16
c	1	3	5	7

This step also takes $O(n^2)$ time. We thus have the following theorem.

Theorem 7. *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 $w_x(\ell)$ for $x \in \Sigma$ and $\ell \leq d$, the maximum weight letter-duplicated subsequence (Weighted-LDS) of S can be computed in $O(n^2)$ time.*

We can run a simple example as follows. Let $S = ababbaca$. Suppose the table $w_x(\ell)$ is given as Table 3.1 At the first step, $w'_x(\ell)$ is the maximum weight of a LD-block made with x and of length at most ℓ. The corresponding table $w'_x(\ell)$ can be computed as Table 3.2. At the end of the second step, we have Table 3.3 computed. From Table 3.3, the table $w[-,-]$ can be easily computed and we omit the details. For instance, $w[1,-] = [0, 0, 10, 16, 16, 20, 0, 20]$. With that, the optimal solution value can be computed as $T(8) = 36$, which corresponds to the optimal solution $aabbaa$.

Table 3.3: Part of the table $N[i, j]$, with $S = ababbaca$ and $d = 4$.

$i\backslash j$	1	2	3	4	5	6	7	8
8	0	0	0	0	0	0	0	1
...	...	...	...	...	...	...	...	...
3	0	0	1	1	2	2	1	3
2	0	1	1	2	3	2	1	3
1	1	1	2	2	3	3	1	4

Table 3.4: Summary of results on LLDS+ and FT, the ? indicates that the problem is still open.

d	$LLDS+(d)$	$FT(d)$	Approximability of $LLDS+(d)$
$d \geq 6$	NP-hard	NP-complete	No approximation
$d = 3$	?	P	1.5-$O(\frac{1}{n})$
$d = 4, 5$	?	?	?

3.6 Conclusion

We consider the constrained longest letter-duplicated subsequence (LLDS+) and the corresponding feasibility testing (FT) problems , where all letters in the alphabet must occur in the solutions. We parameterize the problems with d, which is the maximum number of times a letter appears in the input sequence. For convenience, we summarize the results one more time in the following table. Obviously, we have many open problems.

We also consider the weighted version (without the 'full-appearance' constraint), for which we give a non-trivial $O(n^2)$ time dynamic programming solution.

If we stick with the 'full-appearance' constraint, one direction is to consider two additional variants of the problem where the solutions must be a subsequence of S, in

the form of $x_1^{d_1}x_2^{d_2}\cdots x_k^{d_k}$ with x_i being a substring (resp. subsequence) of S with length at least 2, $x_j \neq x_{j+1}$ and $d_i \geq 2$ for all i in $[k]$ and j in $[k-1]$. Intuitively, for many cases these variants could better capture the duplicated patterns in S. At this point, the NP-completeness results (similar to Theorem 5 and Corollary 2) would still hold with minor modifications to the proofs. (This reduction is still from $(\leq 2, 1, \leq 3)$-SAT and is additionally based on the following fact: given a *(2,1)-sequence* $T = ABCCAB$ over $\{A, B, C\}$, where A, B and C all appear twice, the corresponding maximal 'substring-duplicated-subsequences' or 'subsequence-duplicated-subsequences' of T are $ABAB$ or CC.) But whether these extensions allow us to design good approximation algorithms needs further study. Note that, without the 'full-appearance' constraint, when x_i is a subsequence of S, the problem is a generalization of Kosowski's longest square subsequence problem [59] and can certainly be solved in polynomial time.

3.6.1 Note

The results in this chapter will be presented at CPM 2022 [63].

CHAPTER FOUR

GENOMIC PROBLEMS INVOLVING COPY NUMBER PROFILES

4.1 Introduction

Given a genome G represented as a string and a copy number profile $\vec{c}$, the *Minimum Copy Number Generation* (MCNG) problem asks for the minimum number of deletions and duplications needed to transform G into any genome in which each character occurs as many times as specified by $\vec{c}$. Qingge et al. [79] proved that the problem is NP-hard when the duplications are restricted to be tandem and posed several open questions: (1) Is the problem NP-hard when the duplications are arbitrary and/or deletions are allowed? (2) Does the problem admit a decent approximation? (3) Is the problem fixed-parameter tractable (FPT)? In this chapter, we answer all these three open questions. We show that MCNG is NP-hard to approximate within any constant factor, and that it is W[1]-hard when parameterized by the solution size. The inapproximability follows from a new general-purpose lemma on set-cover reductions that require an exact cover in one direction, but not the other. The W[1]-hardness uses a new set-cover variant in which every optimal solution is an exact cover. These set-cover extensions can make reductions easier, and may be of independent interest.

We also consider a new fundamental problem called *Copy Number Profile Conforming* (CNPC), which is defined as follows. Given two CNP's $\vec{c_1}$ and $\vec{c_2}$, compute two strings/genomes S_1 and S_2 with $cnp(S_1) = \vec{c_1}$ and $cnp(S_2) = \vec{c_2}$ such that the distance between S_1 and S_2, $d(S_1, S_2)$, is minimized. The distance $d(S_1, S_2)$ could be general, which means it could be any genome rearrangement distance (such as reversal, transposition, and tandem duplication, etc). We make the first step by showing that if $d(S_1, S_2)$ is measured by the breakpoint distance then the problem is polynomially solvable.

4.2 Preliminaries

A genome G is a string, i.e., a sequence of characters, all of which belong to Σ (the characters of G can be interpreted as genes or segments - in this chapter we assume the latter, i.e., Σ is a set of segments). We use genome and string interchangeably in this chapter, when the context is clear. A *substring* of G is a sequence of contiguous characters that occur in G, and a *subsequence* is a string that can be obtained from G by deleting some characters. We write $G[p]$ to denote the character at position p of G (the first position being 1), and we write $G[i..j]$ for the substring of G from position i to j, inclusively. For $s \in \Sigma$, we write $G - s$ to denote the subsequence of G obtained by removing all occurrences of s.

We represent an alphabet as an ordered list $\Sigma = (s_1, s_2, \ldots, s_m)$ of distinct characters. Slightly abusing notation, we may write $s \in \Sigma$ if s is a member of this list. We write $n_s(G)$ to denote the number of occurrences of $s \in \Sigma$ in a genome G. A *Copy-Number Profile* (or CNP) on Σ is a vector $\vec{c} = \langle c_1, \ldots, c_{|\Sigma|} \rangle$ that associates each character s_i of the alphabet with a non-negative integer $c_i \in \mathbb{N}$; formally,

$$cnp(G) = \langle n_{s_1}(G), n_{s_2}(G), \ldots, n_{s_m}(G) \rangle.$$

We may write $\vec{c}(s)$ to denote the number associated with $s \in \Sigma$ in $\vec{c}$. We write $\vec{c} - s$ to denote the CNP obtained from $\vec{c}$ by setting $\vec{c}(s) = 0$. As an example, if $\Sigma = (a, b, c)$ and $G = abbcbbcca$, then $cnp(G) = \langle 2, 4, 3 \rangle$ and $\vec{c}(a) = 2$.

Deletions and duplications on strings

We now describe the two string events of *deletion* and *duplication*. Both are illustrated in Table 4.1

Given a genome G, a *deletion* on G takes a substring of G and removes it. Deletions are denote by a pair (i, j) of the positions of the substring to remove. Applying deletion

Sequence	Operations
$G_1 = abbc \cdot \sout{cab} \cdot cab$	$del(5, 7)$
$G_2 = a \cdot \underline{bbcc} \cdot ab$	$dup(2, 5, 6)$
$G_3 = abbcca \cdot \underline{bbcc} \cdot b$	

Table 4.1: Three strings (or toy genomes), G_1, G_2 and G_3. From G_1 to G_2, a deletion is applied to $G_1[5..7]$. From G_2 to G_3, a duplication is applied to $G_2[2..5]$, with the copy inserted after position 6.

(i, j) to G transforms G into $G[1..i-1]G[j+1..n]$.

A *duplication* on G takes a substring of G, copies it and inserts the copy anywhere in G, except inside the copies substring. A duplication is defined by a triple (i, j, p) where $G[i..j]$ is the string to duplicate and $p \in \{0, 1, \ldots, i-1, j, \ldots, n\}$ is the position after which we insert (inserting after 0 prepends the copied substring to G). Applying duplication (i, j, p) to G transforms G into $G[1..p]G[i..j]G[p+1..n]$.

An *event* is either a deletion of a duplication. If G is a genome and e is an event, we write $G\langle e\rangle$ to denote the genome obtained by applying e on G. Given a sequence $E = (e_1, \ldots, e_k)$ of events, we define $G\langle E\rangle = G\langle e_1\rangle G\langle e_2\rangle \ldots G\langle e_k\rangle$ as the genome obtained by successively applying the events of E to G. We may also write $G\langle e_1, \ldots, e_k\rangle$ instead of $G\langle(e_1, \ldots, e_k)\rangle$.

The most natural application of the above events is to compare genomes.

Definition 7. *Let G and G' be two strings over alphabet Σ. The Genome-to-Genome distance between G and G', denoted $d_{GG}(G, G')$, is the size of the smallest sequence of events E satisfying $G\langle E\rangle = G'$.*

Note that d_{GG} has recently been considered in [46]. We also define a distance between a genome G and a CNP $\vec{c}$, which is the minimum number of events to apply to G to obtain a genome with CNP $\vec{c}$.

Definition 8. *Let G be a genome and $\vec{c}$ be a CNP, both over alphabet Σ. The Genome-to-CNP distance between G and $\vec{c}$, denote $d_{GCNP}(G, \vec{c})$, is the size of the smallest sequence of events E satisfying $cnp(G\langle E\rangle) = \vec{c}$.*

The above definition leads to the following problem, which was first studied in [79].
The *Minimum Copy Number Generation (MCNG)* problem:
`Instance:` a genome G and a CNP $\vec{c}$ over alphabet Σ.
`Task:` compute $d_{GCNP}(G, \vec{c})$.

Qingge et al. proved that the MCNG problem is NP-hard when all the duplications are restricted to be tandem [79]. In the next section, we prove that this problem is not only NP-hard, but also NP-hard to approximate within any constant factor.

4.3 Hardness of Approximation for MCNG

In this section, we show that the d_{GCNP} distance is hard to approximate within any constant factor. This result actually holds if only deletions on G are allowed. This restriction makes the proof significantly simpler, so we first analyze the deletions-only case. We then extend this result to deletions and duplications.

Both proofs rely on a reduction from SET-COVER. Recall that in SET-COVER, we are given a collection of sets $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$ over universe $U = \{u_1, u_2, \ldots, u_m\} = \cup_{S_i \in \mathcal{S}} S_i$, and we are asked to find a set cover of $\mathcal{S}$ having minimum cardinality (a set cover of $\mathcal{S}$ is a subset $\mathcal{S}^* \subseteq \mathcal{S}$ such that $\cup_{S \in \mathcal{S}^*} S = U$). If $\mathcal{S}'$ is a set cover in which no two sets intersect, then $\mathcal{S}'$ is called an *exact cover*.

There is one interesting feature (or constraint) of our reduction g, which transforms a SET-COVER instance $\mathcal{S}$ into a MCNG instance $g(\mathcal{S})$. A set cover $\mathcal{S}^*$ only works on $g(\mathcal{S})$ if $\mathcal{S}^*$ is actually an exact cover, and a solution for $g(\mathcal{S})$ can be turned into a set cover for $\mathcal{S}^*$ that is not necessarily exact. Thus we are unable to reduce directly from either SET-

COVER nor its exact version. We provide a general-purpose lemma for such situations, and our reductions serve as an example of its usefulness.

The proof relies on a result on t-SET-COVER, the special case of SET-COVER in which every given set contains at most t elements. It is known that for any constant $t \geq 3$, the t-SET-COVER problem is hard to approximate within a factor $\ln t - c \ln \ln t$ for some constant c not depending on t [94].

Lemma 7. *Let $\mathcal{B}$ be a minimization problem, and let g be a function that transforms any SET-COVER instance $\mathcal{S}$ into a instance $g(\mathcal{S})$ of $\mathcal{B}$ in polynomial time. Assume that both the following statements hold:*

- *any exact cover $\mathcal{S}^*$ of $\mathcal{S}$ of cardinality at most k can be transformed in polynomial time into a solution of value at most k for $g(\mathcal{S})$;*
- *any solution of value at most k for $g(\mathcal{S})$ can be transformed in polynomial time into a set cover of $\mathcal{S}$ of cardinality at most k.*

Then unless $P = NP$, there is no constant factor approximation algorithm for $\mathcal{B}$.

Proof. Suppose for contradiction that $\mathcal{B}$ admits a factor b approximation for some constant b. Choose any constant t such that t-SET-COVER is hard to approximate within factor $\ln t - c \ln \ln t$, and such that $b < \ln t - c \ln \ln t$. Note that t might be exponentially larger that b, but is still a constant.

Now, let $\mathcal{S}$ be an instance of t-SET-COVER over the universe $U = \{u_1, \ldots, u_m\}$. Consider the intermediate reduction g' that transforms $\mathcal{S}$ into another t-SET-COVER instance $g'(\mathcal{S}) = \{S' \subseteq S : S \in \mathcal{S}\}$. Since t is a constant, $g(\mathcal{S})$ has $O(|\mathcal{S}|)$ sets and this can be carried out in polynomial time.

Now define $S' = g'(\mathcal{S})$ and consider the instance $B = g(\mathcal{S}') = g(g'(\mathcal{S}))$. By the assumptions of the lemma, a solution for B of value k yields a set cover $\mathcal{S}^*$ for S'. Clearly, $\mathcal{S}^*$

can be transformed into a set cover for instance $\mathcal{S}$: for each $S' \in \mathcal{S}^*$, there exists $S \in \mathcal{S}$ such that $S' \subseteq S$, so we get a set cover for $\mathcal{S}$ by adding this corresponding superset for each $S \in \mathcal{S}^*$. Thus B yields a set cover of $\mathcal{S}$ with at most k sets.

In the other direction, consider a set cover $\mathcal{S}^* = \{S_1, \ldots, S_k\}$ of $\mathcal{S}$ with k sets. This easily translates into an *exact* cover of $\mathcal{S}'$ with k sets by taking the collection

$$\{S_1, S_2 \setminus S_1, S_3 \setminus (S_1 \cup S_2), \ldots, S_k \setminus \bigcup_{i=1}^{k-1} S_i\}.$$

By the assumptions of the lemma, this exact cover can then be transformed into a solution of value at most k for instance B.

Therefore, $\mathcal{S}$ has a set cover of cardinality at most k if and only if B has a solution of value at most k. By this correspondence, a factor b approximation for $\mathcal{B}$ would provide a factor $b < \ln t - c \ln \ln t$ approximation for t-SET-COVER. □

4.3.1 Constructing Genomes and CNPs from SET-COVER Instances

All of our hardness results rely on Lemma 7. we need to provide a reduction from SET-COVER to MCNG and prove that both assumptions of the lemma are satisfied.

This reduction is the same for deletions-only and deletions-and-duplications. Given $\mathcal{S}$ and U, we construct a genome G and a CNP $\vec{c}$ as follows (an example is illustrated in Figure 4.1). The alphabet is $\Sigma = \Sigma_{\mathcal{S}} \cup \Sigma_U$, where $\Sigma_{\mathcal{S}} := \{\langle \beta_{S_i} \rangle : S_i \in \mathcal{S}\}$ and $\Sigma_U := \{\alpha_{u_i} : u_i \in U\}$. Thus, there is one character for each set of $\mathcal{S}$ and each element of U. Here, each $\langle \beta_{S_i} \rangle$ is a character that serves as a separator between characters to delete. For a set $S_i \in \mathcal{S}$, define the string $q(S_i)$ as any string that contains each character of $\{a_u : u \in S_i\}$ exactly once (in any fixed order, say by their indices). We put

$$G = \langle \beta_{S_1} \rangle q(S_1) \langle \beta_{S_2} \rangle q(S_2) \ldots \langle \beta_{S_n} \rangle q(S_n),$$

i.e., G is the concatenation of the strings $\langle\beta_{S_i}q(S_i)$. As for the CNP $\vec{c}$, put

- $\vec{c}(\langle\beta_{S_i}\rangle) = 1$ for each $S_i \in \mathcal{S}$;
- $\vec{c}(\alpha_u) = f(u) - 1$ for each $u \in U$, where $f(u) = |\{S_i \in \mathcal{S} : u \in S_i\}|$ is the number of sets from $\mathcal{S}$ that contain u.

$$S_1 = \{1,2,3\} \quad S_2 = \{1,3,4\} \quad S_3 = \{2,3,5\}$$
$$G = \langle\beta_{S_1}\rangle\alpha_1\alpha_2\alpha_3\langle\beta_{S_2}\rangle\alpha_1\alpha_3\alpha_4\langle\beta_{S_3}\rangle\alpha_2\alpha_3\alpha_5$$
$$\vec{c}(\alpha_1) = \vec{c}(\alpha_2) = 1 \quad \vec{c}(\alpha_3) = 2 \quad \vec{c}(\alpha_4) = \vec{c}(\alpha_5) = 0$$

Figure 4.1: An example of our construction, with $\mathcal{S} = \{S_1, S_2, S_3\}$ and $U = \{1,2,3,4,5\}$

.

Notice that in G, each $\langle\beta_S\rangle$ already has the correct copy-number, whereas each α_u needs exactly one less copy. Our goal is thus to reduce the number of each α_u by 1. This concludes the construction of **MCNG** instances from SET-COVER instances. We now focus on the hardness of the deletions-only cases.

4.3.2 Warmup: the Deletions-only Case

Suppose that we are given a set cover instance $\mathcal{S}$ and U, and let G and $\vec{c}$ be the genome and CNP, respectively, as constructed above.

Lemma 8. *Given an exact cover $\mathcal{S}^*$ for $\mathcal{S}$ of cardinality k, one can obtain a sequence of k deletions transforming G into a genome with CNP $\vec{c}$.*

Proof. Denote $\mathcal{S}^* = \{S_{i_1}, \ldots S_{i_k}\}$. Consider the sequence of k deletions that deletes the substrings $q(S_{i_1}), \ldots, q(S_{i_k})$ (i.e., the sequence first deletes the substring $q(S_{i_1})$, then deletes $q(S_{i_2})$, and so on until $q(S_{i_k})$ is deleted). Since $S_{i_1}, \ldots, S_{i_k}$ is an exact cover, this sequence

removes exactly one copy of each $\alpha_u \in \Sigma_U$ and does not affect the $\langle\beta_S\rangle$ characters. Thus the k deletions transform G into a genome with the desired CNP $\vec{c}$. □

Lemma 9. *Given a sequence of k deletions transforming G into a genome with CNP $\vec{c}$, one can obtain a set cover for $\mathcal{S}$ of cardinality at most k.*

Proof. Suppose that the deletion events $E = (e_1, \ldots, e_k)$ transform G into a genome G^* with CNP $\vec{c}$. Note that no e_i deletion is allowed to delete a set-character $\langle\beta_{S_i}\rangle \in \Sigma_{\mathcal{S}}$, as there is only one occurrence of $\langle\beta_{S_i}\rangle$ in G and $\vec{c}(\langle\beta_{S_i}\rangle) = 1$. Thus all deletions remove only α_u characters. In other words, each e_j in E either deletes a substring of G between some $\langle\beta_{S_i}\rangle$ and $\langle\beta_{S_{i+1}}\rangle$ with $1 \leq i < n$, or e_j deletes a substring after $\langle\beta_{S_n}\rangle$. Moreover, exactly one of each α_u occurrences gets deleted from G.

Call $\langle\beta_{S_i}\rangle \in \Sigma_{\mathcal{S}}$ affected if there is some event of E that deletes at least one character between $\langle\beta_{S_i}\rangle$ and $\langle\beta_{S_{i+1}}\rangle$ with $1 \leq i < n$, and call $\langle\beta_{S_n}\rangle$ affected if some event of E deletes characters after $\langle\beta_{S_n}\rangle$. Let $\mathcal{S}^* := \{S_i \in \mathcal{S} : \langle\beta_{S_i}\rangle \text{is affected}\}$. Then $|\mathcal{S}^*| \leq k$, since each deletion affects at most one $\langle\beta_{S_i}\rangle$ and there are k deletion events. Moreover, $\mathcal{S}^*$ must be a set cover, because each $\alpha_u \in \Sigma_U$ has at least one occurrence that gets deleted and thus at least one set containing u that is included in $\mathcal{S}^*$. This concludes the proof. □

We have shown that all the assumptions required by Lemma 7 are satisfied. The inapproximability follows.

Theorem 8. *Assuming $P \neq NP$, there is no polynomial-time constant factor approximation algorithm for MCNG when only deletions are allowed.*

We mention without proof that the reduction can be adaptable to the duplication-only case, by putting $\vec{c}(\alpha_u) = f(u) + 1$ for each $u \in U$.

4.3.3 The Real Deal: Deletions and Duplications

We now consider both deletions and duplications. The reduction uses the same construction as in Subsection 4.3.1. Thus we assume that we have a SET-COVER instance $\mathcal{S}$ over U, and a corresponding instance of MCNG with genome G and CNP $\vec{c}$. In that case, we observe that Lemma 8 still holds whether we allow deletion only, or both deletions and duplications. Thus we only need to show that the second assumption of Lemma 7 holds.

Unfortunately, this is not as simple as in the deletions-only case. The problem is that duplications may copy α_u and $\langle\beta_{S_i}\rangle$ occurrences, and we lose control over what gets deleted, and over what $\langle\beta_{S_i}\rangle$ each α_u corresponds to (in particular, $\langle\beta_{S_i}\rangle$ might now get deleted, which did not occur in the deletions-only case).

Nevertheless, the analogous result can be shown. That is, using the above reduction, our goal is to show that, given a sequence of k events (deletions and duplications) transforming G into a genome with CNP $\vec{c}$, one can obtain a set cover for $\mathcal{S}$ of cardinality at most k.

We need new notation and intermediate results beforehand. Let $E = (e_1, \ldots, e_k)$ be a sequence of events transforming genome G into another genome G'. We would like to distinguish each position of G in order to know which specific character of G is at the origin of a character of G'.

We augment each individual character of G with a unique identifier, which is its position in G. That is, let $G = g_1 g_2 \cdots g_n$, define a new alphabet $\hat{\Sigma} = (g_1^1, g_2^2, \cdots, g_n^n)$ and define the genome $\hat{G} = (g_1^1, g_2^2, \cdots, g_n^n)$. Here, two characters g_i and g_j may be identical, but g_i^i and g_j^j are two distinct characters. We call $\hat{\Sigma}$ the augmented alphabet and $\hat{G}$ the augmented genome of G. For instance if $G = aabcb$ and $\Sigma = (a, b, c)$, then $\hat{\Sigma} = (a^1, a^2, b^3, c^4, b^5)$ and $\hat{G} = a^1a^2b^3c^4b^5$.

Since G and $\hat{G}$ have the same length, we may apply the sequence E on $\hat{G}$, resulting in a genome $\hat{G}' := \hat{G}\langle E\rangle$ on alphabet $\hat{\Sigma}$. Now $\hat{G}'$ may contain some characters of $\hat{\Sigma}$ multiple times owing to duplications, but if we remove the superscript identifier from the characters

of $\hat{G}'$, we obtain G'. The idea is that the identifiers on the characters of $\hat{G}'$ tell us precisely where each character of $\hat{G}'$ "comes from" in $\hat{G}$ (and thus G).

Definition 9. *Let G and G' be genomes and let E an event sequence such that $G' = G\langle E\rangle$. Let $\hat{G}$ be the augmented genome of G and let $\hat{G}[i] = g^i$ be the character at position i.*

If there is at least one occurrence of g^i in $\hat{G}\langle E\rangle$, then position i is called important with respect to E. Otherwise, position i is called unimportant with respect to E.

Roughly speaking, position i is unimportant if it eventually get deleted, and any character that was copied from position i from a duplication also gets deleted, as well as a copy of this copy, and so on - in other words, position i has no "descendant" in G' when applying E.

First, we prove a few general properties that are useful. Recall that $G - s$ removes all occurrences of s from G, and $\vec{c} - s$ puts $\vec{c}(s) = 0$.

Proposition 2. *Let G be a genome over alphabet Σ, let $\vec{c}$ be a CNP and let $s \in \Sigma$. Then $d_{GCNP}(G - s, \vec{c} - s) \leq d_{GCNP}(G, \vec{c})$.*

The next technical lemma states that if a genome alternates between positions to keep and positions to delete n times, then we need n events to remove the unimportant ones.

Lemma 10. *Let $\Sigma = X \cup Y$ be an alphabet defined by two disjoint sets $X = \{x_1, \ldots, x_n\}$ and Y. Let $G = Y_0 x_1 Y_1 x_2 Y_2 \ldots x_n Y_n$ be a genome on Σ, where for all $i \in [n]$, Y_i is a non-empty string over alphabet Y and Y_0 is possibly empty string on alphabet Y. Moreover let $\vec{c}$ be a CNP such that $\vec{c}(x_i) = 1$ for all $x_i \in X$ and $\vec{c}(y) = 0$ for all $y \in Y$. Then $d_{GCNP}(G, \vec{c}) \geq n$, with equality when Y_0 is empty.*

We may now prove the second assumption of Lemma 7.

Lemma 11. *Let $\mathcal{S}$ be a SET-COVER instance, and let G and $\vec{c}$ be the corresponding MCNG instance. Given a sequence of k events (deletions and duplications) transforming G into a genome with CNP $\vec{c}$. one can obtain a set cover for $\mathcal{S}$ of cardinality at most k.*

Proof. Suppose that the events $E = (e_1, \ldots, e_k)$ transform G into a genome G^* with CNP $\vec{c}$. We construct a set cover for $\mathcal{S}$ of cardinality k. For a position p with $G[p] = \alpha_u \in \Sigma_U$, define $pred(p)$ as the first $\Sigma_{\mathcal{S}}$ character to the left of position p. To be precise, if p' is the largest integer satisfying $G[p'] \in \Sigma_{\mathcal{S}}$ and $p' < p$, then $pred(p) = G[p']$. Note that since $G[1] = \langle \beta_{S_1} \rangle$, $pred(p)$ is well-defined. Notice that by construction, if $G[p] = \alpha_u$ and $\langle \beta_S \rangle = pred(p)$, then $u \in S$. The set of $pred(p)$ of unimportant positions p will correspond to our set cover, which we now prove by separate claims.

Claim 6. *For each $u \in U$, there is at least one position p of G such that $G[p] = \alpha_u$ and such that p is unimportant w.r.t. E.*

Proof. If we assume this is not the case, then each of the $f(u)$ positions p of G having $G[p] = \alpha_u$ has a descendant in G^*, implying that G^* has at least $f(u)$ copies of α_u and thereby contradicting that G^* complies with $\vec{c}(\alpha_u) = f(u) - 1$. □

Recall that $U = \{u_i, \ldots, u_m\}$. Given that the claim holds, let $P = \{p_1, \ldots, p_m\}$ be any set of positions of G such that for each $i \in [m]$, $G[p_i] = \alpha_{u_i}$ and p_i is unimportant w.r.t. E (choosing arbitrarily if there are multiple choices for p_i). Define $\Sigma_P = \{pred(p_i) : p_i \in P\}$ and $\mathcal{S}^* = \{S_i \in \mathcal{S} : \langle \beta_{S_i} \rangle \in \Sigma_P\}$.

Claim 7. *$\mathcal{S}^*$ is a set cover.*

Proof. For each $u_i \in U$, there is an unimportant position $p_i \in P$ such that $G[p_i] = \alpha_{u_i}$. Moreover, $pred(p_i)$ is some character $\langle \beta_S \rangle$ such that $\langle \beta_S \rangle \in \Sigma_P$ and such that $u_i \in S$. Since $S \in \mathcal{S}^*$, it follows that each u_i is covered. □

It remains to show that $\mathcal{S}^*$ has at most k sets. Denote $P' = P \cup \{p : G[p] \in \Sigma_P\}$. Let $\tilde{G}$ be the subsequence of G obtained by keeping only positions in P' (i.e., if we denote $P' = \{p'_1, \ldots, p'_l\}$ with $p'_1 < p'_2 < \ldots < p'_l$, then $\tilde{G} = G[p'_1]G[p'_2] \ldots G[p'_l]$). furthermore, define the CNP $\vec{c}_0$ such that $\vec{c}_0(\langle \beta_{S_i} \rangle) = 1$ for all $\langle \beta_{S_i} \rangle \in \Sigma_P$, $\vec{c}_0(\langle \beta_{S_i} \rangle) = 0$ for all $\langle \beta_{S_i} \rangle \in \Sigma_{\mathcal{S}} \setminus \Sigma_P$,

and $\vec{c}_0(\alpha_u) = 0$ for all $\alpha_u \in \Sigma_U$. Note that $\tilde{G}$ has the form $\langle\beta_{S_{i_1}}\rangle D_1\langle\beta_{S_{i_2}}\rangle D_2 \ldots \langle\beta_{S_{i_r}}\rangle D_r$ for some r, where the D_i's are substrings over alphabet Σ_U. This is form of Lemma 10.

Claim 8. $d_{GCNP}(\tilde{G}, \vec{c}_0) \leq k$.

Proof. Let G' be the genome obtained by replacing every position p of G by some dummy character λ, except for the positions of P' (thus if we remove all the λ occurrences we obtain $\tilde{G}$). Since G and G' have the same length, we can apply the E events on G'. Let $G'' := G'\langle E\rangle$, and let l be the number of occurrences of λ in G''. Recall that p' contains only positions p such that $G[p] \in \Sigma_P$, or such that p is unimportant w.r.t. E and $G[p] \in \Sigma_U$. It follows that if a position q is important w.r.t. E, then $G'[q] = \Sigma_P \cup \{\lambda\}$. Moreover, for any $\langle\beta_S\rangle \in \Sigma_P$, G'' has as many occurrences of $\langle\beta_S\rangle$ as in $G\langle E\rangle$. In other words, G'' has one occurrence of each $\langle\beta_S\rangle \in \Sigma_P$ and the rest is filled with λ.

Let $\vec{c}_1$ be the CNP satisfying $\vec{c}(\lambda) = l$, $\vec{c}_1(\langle\beta_{S_i}\rangle) = \vec{c}_0(\langle\beta_{S_i}\rangle) = 1$ for every $\langle\beta_{S_i}\rangle \in \Sigma_P$, and $\vec{c}_1(x) = 0$ for any other character x. Then clearly, $\vec{c}_1 = cnp(G'')$, which implies $d_{GCNP}(G', \vec{c}_1) \leq k$ since E transforms G' to G''. Moreover by Proposition 2, $d_{GCNP}(G' - \lambda, \vec{c}_1 - \lambda) \leq d_{GCNP}(G', \vec{c}_1) \leq k$. The claim follows from the observation that $\tilde{G} = G' - \lambda$ and $\vec{c}_0 = \vec{c}_1 - \lambda$. □

Observe that $\tilde{G}$ and $\vec{c}_0$ have the required form for Lemma 10 (with $|\Sigma_P|$ important positions), and so $d_{GCNP}(\tilde{G}, \vec{c}_0) \leq |\Sigma_P|$. It follows from Claim 8 that $k \geq d_{GCNP}(\tilde{G}, \vec{c}_0) \geq |\Sigma_P| = |\mathcal{S}^*|$. We thus have a set cover $\mathcal{S}^*$ for $\mathcal{S}$ of cardinality at most k, completing the proof. □

We arrive to out main inapproximability result, which again follows from Lemma 7.

Theorem 9. *Assuming $P \neq NP$, there is no polynomial-time constant factor approximation algorithm MCNG.*

In the next section, we prove that the MCNG problem, parameterized by the solution size, is W[1]-hard. This answers another open question is [79]. We refer readers for more details on FPT and $W[1]$-hardness to the book by Downey and Fellows [32].

4.4 W[1]-hardness for MCNG

Since SET-COVER is W[2]-hard, naturally we would like to use the above reduction to prove the W[2]-hardness of MCNG. However, the fact that we use t-SET-COVER with constant t in the proof of Lemma 7 is crucial, the t-SET-COVER is in FPT. On the other hand, the property that is really needed in the instance of this proof, and in our MCNG reduction, is that we can transform any set cover instance into an exact cover. We capture this intuition and show that SET-COVER instances that have this property are W[1]-hard to solve.

An instance of SET-COVER-with-EXACT-COVER, or SET-COVER-EC for short, is a pair $I = (\mathcal{S}, k)$ where k is an integer and $\mathcal{S}$ is a collection of sets forming a universe U. In this problem, we require that $\mathcal{S}$ satisfies the property that any set cover for $\mathcal{S}$ of size at most k is also an exact cover. We are asked whether there exists a set cover for $\mathcal{S}$ of size at most k (in which case this set cover is also an exact cover).

Lemma 12. *The SET-COVER-EC problem is W[1]-hard for parameter k.*

Proof. We show W[1]-hardness using the techniques introduced by Fellows et al. which is coined as MULTICOLORED-CLIQUE [38]. In the MULTICOLORED-CLIQUE problem, we are given a graph G, an integer k and a coloring $c : V(G) \to [k]$ such that no two vertices of the same color share an edge. We are asked whether G contains a clique of k vertices (noting that such a clique must have a vertex of each color). This problem is W[1]-hard w.r.t. k.

Given an instance (G, k, c) of MULTICOLORED-CLIQUE, we construct an instance $I = (\mathcal{S}, k')$ of SET-COVER-EC. We put $k' = k + \binom{k}{2}$. For $i \in [k]$, let $V_i = \{v \in V(G) : c(v) = i\}$ and for each pair $i < j \in [k]$, let $E_{ij} = \{uv \in E(G) : u \in V_i, v \in V_j\}$. The universe U of the SET-COVER-EC instance has one element for each color i, one element for each pair $\{i, j\}$ of distinct colors, and two elements for each edge, one for each direction of the edge. That is,

$$U = [k] \cup \binom{[k]}{2} \cup \{(u, v) \in V(G) \times V(G) : uv \in E(G)\}$$

Thus $|U| = k + \binom{[k]}{2} + 2|E(G)|$. For two colors $i < j \in [k]$, we will denote $U_{ij} = \{(u, v), (v, u) : u \in V_i, v \in V_j, uv \in E_{ij}\}$, i.e., we include in U_{ij} both elements corresponding to each $uv \in E_{ij}$. Now, for each color class $i \in [k]$ and each vertex $u \in V_i$, add to $\mathcal{S}$ the set

$$S_u = i \cup \{(u, v) : v \in N(u)\}$$

where $N(u)$ is the set of neighbors of u in G. Then for each $i < j \in [k]$, and for each edge $uv \in E_{ij}$, add to $\mathcal{S}$ the set

$$S_{uv} = \{\{i, j\}\} \cup \{(x, y) \in U_{ij} : x \notin \{u, v\}\}$$

The idea is that S_{uv} can cover every element of U_{ij}, except those ordered pairs whose first element is u or v. Then if we do decide to include S_{uv} in a set cover, it turns out that we will need to include S_u and S_v to cover these missing ordered pairs. See Figure 4.2 for an example. For instance if we include S_{u_2,v_3} in a cover, the uncovered (u_2, v_3) and (v_3, u_2) can be covered with S_{u_2} and S_{v_3}. We show that G has a multicolored clique of size k if and only if $\mathcal{S}$ admits a set cover of size k'. Note that we have not shown yet that $(\mathcal{S}, k')$ is an instance of SET-COVER-EC, i.e., that any set cover of size at most k' is also an exact cover. This

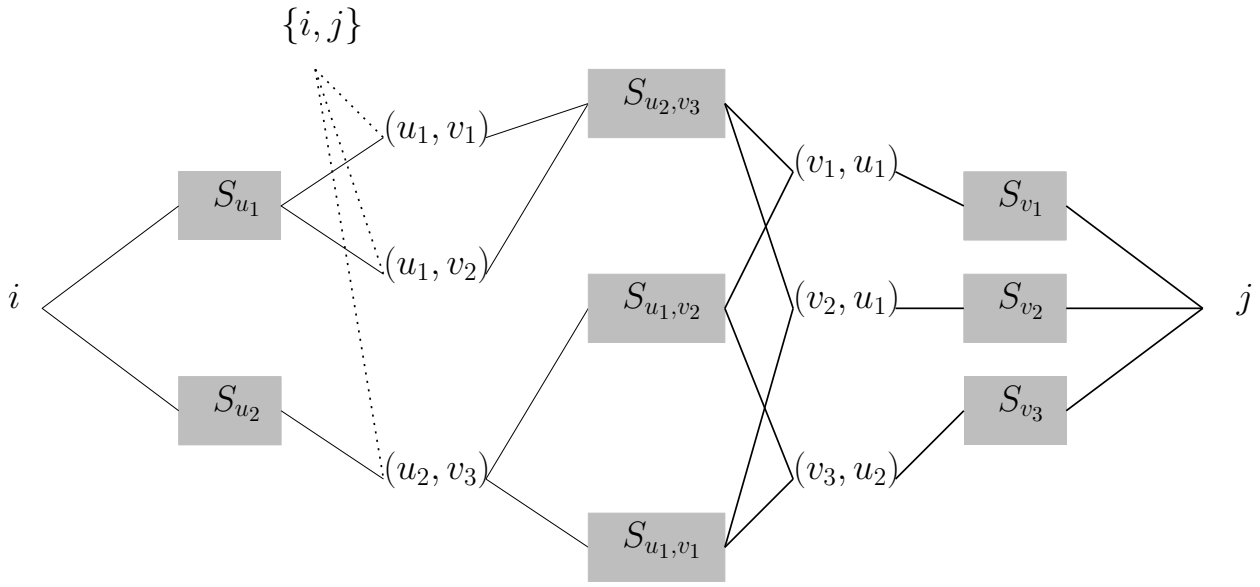

Figure 4.2: A graphical example of the constructed sets for the U_{ij} elements of a graph (not shown) with $E_{ij} = \{u_1v_1, U_1v_2, u_2v_3\}$, where the u_l's are in V_i and the v_l's in V_j (sets have a gray background, edges represent containment, the $\{i, j\}$ lines are dotted only for better visualization).

will be a later part of the proof.

First suppose that G has a multi-colored clique $C = \{v_1, \ldots, v_k\}$, where $v_i \in V_i$ for each $i \in [k]$. Consider the collection

$$\mathcal{S}^* = \{S_{v_1}, \ldots, S_{v_k}\} \cup \{S_{v_iv_j} : v_i, v_j \in C, 1 \leq i < j \leq k\},$$

the cardinality of $\mathcal{S}^*$ is $k + \binom{k}{2} = k'$. Each element $i \in U \cap [k]$ is covered since we include a set S_{v_i} for each color. Each element $\{i, j\} \in U \cap \binom{[k]}{2}$ is covered since we include a set $S_{v_iv_j}$ for each color pair i, j with $i < j$. Consider an element $(X_i, X_j) \in U \cap (V(G) \times V(G))$, where $x_i \in V_i$ and $y_i \in V_j$. Note that either $i < j$ or $j < i$ is possible, and that $v_iv_j \in E(G)$. If $x_i \notin \{v_i, v_j\}$, then $S_{v_iv_j}$ covers (x_i, y_j). If $x_i = v_i$, then S_{v_i} covers (x_i, v_j) and if $x_i = v_j$, then S_{v_j} covers (x_i, v_j). Thus $\mathcal{S}^*$ is a set cover, and is of size at most k'.

For the converse direction, suppose that $\mathcal{S}^*$ is a set cover for $\mathcal{S}$ of size at most $k' = k + \binom{k}{2}$.

Note that to cover the elements of $U \cap [k]$, $\mathcal{S}^*$ must have at least one set S_u such that $u \in V_i$ for each color class $i \in [k]$. Moreover, to cover the elements of $U \cap \binom{[k]}{2}$, $\mathcal{S}^*$ must have at least one set S_{uv} such that $u \in V_i$, $v \in V_j$ for each $i, j \in [k]$ pair. We deduce that $\mathcal{S}^*$ has exactly $k + \binom{k}{2}$ sets. Hence for color $i \in [k]$, there is *exactly* one set S_u in $\mathcal{S}^*$ for which $u \in V_i$, and for each $\{i, j\}$ pair, there is *exactly* one S_{uv} set in $\mathcal{S}^*$ for which $u \in V_i$, $v \in V_j$.

We claim that $C = \{u : S_u \in \mathcal{S}^*\}$ is a multi-colored clique. We already know that C contains one vertex of each color. Now, suppose that some $u, v \in C$ do not share an edge, where $u \in V_i$, $v \in V_j$ and $i < j$. Let S_{xy} be the set of $\mathcal{S}^*$ that covers $\{i, j\}$, with $x \in V_i$, $y \in V_j$. Since uv is not an edge but xy is, we know that $u \neq x$ or $v \neq y$ (or both). Moreover, S_{xy} does not cover the (x, y) and (y, x) elements of U_{ij}, and we know that at least one of these is not covered by S_u nor S_v (if $u \neq x$, then none covers (x, y), if $v \neq y$, then none covers (y, x)). But $(x, y) \in U_{ij}$, and S_u, S_v and S_{xy} are the only sets of $\mathcal{S}^*$ that have elements of U_{ij}, contradicting that $\mathcal{S}^*$ is a set cover. This shows that C is a multi-colored clique.

It remains to show that $\mathcal{S}^*$ is an exact cover. Observe that no two distinct S_u and S_v sets in $\mathcal{S}^*$ can intersect because u and v must be of a different color, and no two distinct S_{uv} and S_{xy} sets in $\mathcal{S}^*$ can intersect because $\{u, v\}$ and $\{x, y\}$ must be from two different color pairs. Suppose that $S_u, S_{xy} \in \mathcal{S}^*$ do intersect, and say that $x \in V_i$, $y \in V_j$ and $i < j$. Then all elements in $S_u \cap S_{xy}$ are of the form (u, v) for some v. Choose any such (u, v). If u is of color i, then $u \neq x$ since otherwise by construction S_{xy} could not contain (u, v). But when $u \neq x$, no set of $\mathcal{S}^*$ covers the element (x, y) (it is not S_u nor S_{xy}, the only two possibilities). If u is of color j, then $u \neq y$ since again S_{xy} could not contain (u, v). In this case, no set of $\mathcal{S}^*$ covers (y, x). We reach a contradiction and deduce that $\mathcal{S}^*$ is an exact cover. □

It is now almost immediate that MCNG is W[1]-hard with respect to the natural parameter, namely the number of events to transform a genome G into a genome with a given profile $\vec{c}$.

Theorem 10. *The MCNG problem is W[1]-hard.*

We do not know whether SET-COVER-EC or MCNG are also in W[1], i.e., whether they are W[1]-complete. We have finished presenting the negative results on MCNG. An immediate question is whether we could obtain positive result on a related problem. In the next section, we present positive result for an interesting variation of MCNG.

4.5 The Copy Number Profile Conforming Problem

We define the more general *Copy Number Profile Conforming* (CNPC) problem as follows:

Definition 10. *Give two CNP's* $\vec{c}_1 = \langle u_1, u_2, \ldots, u_n \rangle$ *and* $\vec{c}_2 = \langle v_1, v_2, \ldots, v_n \rangle$*, with* $u_i, v_i \geq 0$ *and* $u_i, v_i \in \mathbb{N}$*, the CNPC problem asks to compute two strings* S_1 *and* S_2 *with* $cnp(S_1) = \vec{c}_1$ *and* $cnp(S_2) = \vec{c}_2$ *such that the distance between* S_1 *and* S_2*,* $d(S_1, S_2)$*, is minimized.*

Let $\sum_i u_i = m_1$, $\sum_i v_i = m_2$, we assume that m_1 and m_2 are bounded by a polynomial of n. (This assumption is needed as the solution of our algorithm could be of size $\max\{m_1, n_2\}$.) We simply say $\vec{c}_1$, $\vec{c}_2$ are polynomially bounded. Note that $d(S_1, S_2)$ is very general distance measure, i.e., it could be any genome rearrangement distance (like reversal, transposition, and tandem duplication, etc, or their combination, e.g. tandem duplication + deletion). In this section, we use the breakpoint distance and the adjacency number. Our definitions fo these notions are adapted from Angibard et al. [7] and Jiang et al. [53], which generalize the corresponding concepts on permutations [92].

Given two sequences $A = a_1 a_2 \ldots a_n$ and $B = b_1 b_2 \ldots b_m$, if $\{a_i, a_{i+1}\} = \{b_j, b_{j+1}\}$ we say that $a_i a_{i+1}$ and $b_j b_{j+1}$ are matched to each other (in the graph theory terminology, they are an edge). Consider a maximum cardinality matching between length 2 substrings of A and B. A matched pair is called an *adjacency*, and an unmatched pair is called a *breakpoint* in A and B respectively. Then, the multiset of 2-substrings of A (resp. B) that belong

$$
\begin{aligned}
sequence\ A &= \langle\ a\ c\ b\ d\ c\ b\ \rangle \\
2\ substrings\ of\ A &= \langle ac\ cb\ bd\ dc\ cb\ \rangle \\
sequence\ B &= \langle\ a\ b\ c\ a\ b\ c\ d\ \rangle \\
2\ substrings\ of\ B &= \langle ab\ bc\ ca\ ab\ bc\ cd\ \rangle \\
matched\ pairs &: (cb \leftrightarrow bc), (dc \leftrightarrow cd), (cb \leftrightarrow bc) \\
a(A,B) &= \{bc, bc, cd\} \\
b_A(A,B) &= \{ac, bd\} \\
b_B(A,B) &= \{ab, da, ab, cd\}
\end{aligned}
$$

Figure 4.3: Example for adjacency and breakpoint definitions, with $d_b(A,B) = 2$ and $d_b(B,A) = 4$.

to a breakpoint is denoted as $b_A(A,B)$ (resp. $d_B(A,B)$) and the corresponding number is $d_b(A,B)$ (resp. $d_b(B,A)$), and the number of common adjacencies between A and B is denoted as $a(A,B)$. Note that $d_b(A,B)$, $d_b(B,A)$ and $a(A,B)$ do not depend on a particular choice of maximum matching. We illustrate the above definitions in Fig. 4.3 .

Coming back to our problem, we define $d(S_1,S_2) = d_b(S_1,S_2) + d_b(S_2,S_1)$. From the definitions we have

$$d_b(S_1,S_2) + d_b(S_2,S_1) + 2 \cdot a(S_1,S_2) = (m_1 - 1) + (m_2 - 1),$$

or,

$$d_b(S_1,S_2) + d_b(S_2,S_1) = m_1 + m_2 - 2 \cdot a(S_1,S_2) - 2.$$

Hence, the problem is really to maximize $a(S_1, S_2)$.

Definition 11. *Given n-dimensional vectors $\vec{u} = \langle u_1, u_2, \ldots, u_n \rangle$ and $\vec{w} = \langle w_1, w_2, \ldots, w_n \rangle$, with $u_i, w_i \geq 0$, and $u_i, w_i \in \mathbb{N}$, we say $\vec{w}$ is a sub-vector of $\vec{u}$ if $w_i \leq u_i$ for $i = 1, \ldots, n$, also denote this relation as $\vec{w} \leq \vec{u}$.*

Henceforth, we simply call $\vec{u}, \vec{w}$ integer vectors, with the understanding that no item in a vector is negative.

Definition 12. *Given two n-dimensional integer vectors $\vec{u} = \langle v_1, u_2, \ldots, u_n \rangle$ and $\vec{v} = \langle v_1, v_2, \ldots, v_n \rangle$, we say $\vec{w}$ is a common sub-vector of $\vec{u}$ and $\vec{v}$ if $\vec{w}$ is a sub-vector of $\vec{u}$ and $\vec{w}$ is also a sub-vector of $\vec{v}$ (i.e., $\vec{w} \leq \vec{u}$ and $\vec{w} \leq \vec{v}$). Finally, $\vec{w}$ is the maximum common sub-vector of $\vec{u}$ and $\vec{v}$ if there is no common sub-vector $\vec{w}' \neq \vec{w}$ of $\vec{u}$ and $\vec{v}$ which satisfies $\vec{w} \leq \vec{w}' \leq \vec{u}$ or $\vec{w} \leq \vec{w}' \leq \vec{v}$.*

An example is illustrated as follows. We have $\vec{u} = \langle 3, 2, 1, 0, 5 \rangle$, $\vec{v} = \langle 2, 1, 3, 1, 4 \rangle$, $w' = \langle 2, 1, 0, 0, 3 \rangle$ and $\vec{w} = \langle 2, 1, 1, 0, 4 \rangle$. Both $\vec{w}$ and $\vec{w}'$ are common sub-vectors for $\vec{u}$ and $\vec{v}$, $\vec{w}'$ is not the maximum common sub-vector of $\vec{u}$ and $\vec{v}$ (since $\vec{w}' \leq \vec{w}$) while $\vec{w}$ is.

Given a CNP $\vec{u} = \langle u_1, u_2, \ldots, u_n \rangle$ and alphabet $\Sigma = (x_1, x_2, \ldots, x_n)$, for $i \in \{1, 2, \ldots, n\}$, we use $S(\vec{u})$ to denote the multiset of letters (genes) corresponding to $\vec{u}$; more precisely, u_i denotes the number of x's in $S(\vec{u})$. Similarly, given a multiset of letters Z, we use $s(Z)$ to denote a string where all the letters in Z appear exactly once (counting multiplicity; i.e., $|Z| = |s(Z)|$). $s(Z)$ is similarly defined when Z is a CNP. We present **Algorithm 4.1**.

Let $\Sigma = \{a, b, c, d, e\}$. Also let $\vec{c}_1 = \{2, 2, 2, 4, 1\}$ and $\vec{c}_2 = \langle 4, 4, 1, 1, 1 \rangle$. We walk through the algorithm using this input as follows.

1. The maximum common sub-vector $\vec{v}$ of $\vec{c}_1$ and $\vec{c}_2$ is $\vec{v} = \langle 2, 2, 1, 1, 1 \rangle$.

2. Compute $S(\vec{v}) = \{a, a, b, b, c, d, e\}$, $S(\vec{c}_1) = \{a, a, b, b, c, c, d, d, d, d, e\}$ and $S(\vec{c}_2) = \{a, a, a, a, b, b, b, b, c, d, e\}$. Compute $X = \{c, d, d, d\}$ and $Y = \{a, a, b, b\}$.

3. Identify d and a such that $d \in S(\vec{v})$ and $a \in S(\vec{v})$, and $d \in X$ while $a \in Y$.

4. Compute $s(\vec{v}) = dabbcea$, $s_1 = dabbcea \cdot d$ and $s_2 = a \cdot dabbcea$.

5. Insert elements in $X - \{d\} = \{c, d, d\}$ arbitrarily at the right end of s_1 to obtain S_1, and insert all the elements in $Y - \{a\} = \{a, b, b\}$ at the right end of s_2 to obtain S_2.

6. Return $S_1 = dabbcea \cdot d \cdot cdd$ and $S_2 = a \cdot dabbcea \cdot abb$.

Theorem 11. *Let $\vec{c}_1$, $\vec{c}_2$ be polynomially bounded. The number of common adjacencies generated by Algorithm 4.1 is optimal with a value either n^* or $n^* - 1$, where $n^* = \sum_{i=1}^{n} v_i$ with the maximum common sub-vector of $\vec{c}_1$ and $\vec{c}_2$ being $\vec{v} = \langle v_1, v_2, \ldots, v_n \rangle$.*

Proof. First, note that if $\vec{v}$ is a 0-vector (or $S(\vec{v}) = \emptyset$) then there will not be any adjacency in S_1 and S_2. Henceforth we discuss $S(\vec{v}) \neq \emptyset$.

Notice that a common adjacency between S_1 and S_2 must come from two letters which are both in $S(\vec{v})$. That naturally gives us $n^* - 1$ adjacencies, where $n^* = |S(\vec{v})|$, which can be done by using the letters in $S(\vec{v})$ to form two arbitrary strings S_1 and S_2 (for which $s(\vec{v})$ is a common substring). If $\{x, y\}$ can be found such that $x, y \in S(\vec{v})$ and $x \neq y$, and one of them is in X (say $x \in X$), and the other is in Y (say $y \in Y$), then, obviously we could obtain $s_1 = s(\vec{v}) \circ x$ and $s_2 = y \circ s(\vec{v})$ which are substrings of S_1 and S_2 respectively. Clearly, there are $n^* = |S(\vec{v})|$ adjacencies between s_1 and s_2 (and also S_1 and S_2).

To see that this is optimal, first suppose that no $\{x, y\}$ pair as above can be found. This can only occur when there are no two components $i < j$ in $\vec{c}_1 = \langle c_{1,1}, \ldots, c_{1,i}, \ldots, c_{1,j}, \ldots, c_{1,n} \rangle$, $\vec{c}_2 = \langle c_{2,1}, \ldots, c_{2,i}, \ldots, c_{2,j}, \ldots, c_{2,n} \rangle$, and in the maximum common sub-vector $\vec{v} = \langle v_1, \ldots, v_i, \ldots, v_j, \ldots, v_n \rangle$ of $\vec{c}_1$ and $\vec{c}_2$ which satisfy that $\min\{c_{1,i}, c_{2,i}\} = v_i \neq 0$ and $\max\{c_{1,i}, c_{2,i}\} \neq v_i$, and $\min\{c_{1,j}, c_{2,j}\} = v_j \neq 0$ and $\max\{c_{1,j}, c_{2,j}\} \neq v_j$. If this condition holds, then all the components i in $s(\vec{c}_1 - \vec{v})$ and $s(\vec{c}_2 - \vec{v})$, i.e., $c_{1,i} - v_i$ and $c_{2,i} - v_i$, have the property that at least one of the two is zero

and $v_i = 0$. Therefore, except for the letters corresponding to $\vec{v}$, no other adjacency can be formed. As any string with CNP $\vec{v}$ has n^* characters, at most $n^* - 1$ adjacencies can be formed. If an $\{x, y\}$ pair can be found, let $b \in \Sigma$, and let v_b be the minimum copy-number of b in $\vec{c}_1$ or $\vec{c}_2$, i.e., $v_b = \min\{c_{1,b}, c_{2,b}\}$. Assume this minimum occurs in $\vec{c}_1$, w.l.o.g. There can be at most $2v_b$ adjacencies involving b in $\vec{c}_1$, and thus at most $2v_b$ adjacencies in common involving v_b. Summing over every $b \in \Sigma$, the sum of common adjacencies, counted for each character individually, is at most $\sum_{b\in\Sigma} 2v_b = 2n^*$. Since each adjacency is counted twice in this sum, the number of common adjacencies is at most n^*. □

Note that if we only want the breakpoint distance between S_1 and S_2, then the polynomial boundness condition of $\vec{c}_1$ and $\vec{c}_2$ can be withdrawn as we can decide whether $\{x, y\}$ exists by searching directly in the CNPs (vectors).

4.6 Conclusion

In this chapter, we answered two recent open questions regarding the computational complexity of the Minimum Copy Number Generation problem. Our technique could be used for other optimization problems where the solution involves Set Cover whose solution must also be an exact cover. We also present a polynomial time algorithm for the Copy Number Profile Conforming (CNPC) problem when the distance is the classical breakpoint distance. The breakpoint distance is static, and we leave open the question for solving or approximating CNPC with dynamic rearrangement distance such as reversal, duplication + deletion, etc.

4.6.1 Note

The results in this chapter have been presented at CPM 2020 [60].

Algorithm 4.1: CNPC algorithm

1: Compute the maximum common sub-vector $\vec{v}$ of $\vec{c}_1$ and $\vec{c}_2$.

2: Given the gene alphabet Σ, compute $S(\vec{v})$, $S(\vec{c}_1)$ and $S(\vec{c}_2)$. Let $X = S(\vec{c}_1) - S(\vec{v})$ and $Y = S(\vec{c}_2) - S(\vec{v})$.

3: if $S(\vec{v}) = \emptyset$, then return two arbitrary strings $s(\vec{c}_1)$ and $s(\vec{c}_2)$ as S_1 and S_2, exit; otherwise, continue.

4: Find $\{x, y\}$, $x, y \in \Sigma$ and $x \neq y$, such that $x \in S(\vec{v})$ and $y \in S(\vec{v})$, and exactly one of x, y is in X (say $x \in X$), and the other is in Y (say $y \in Y$). If such an $\{x, y\}$ cannot be found then return two strings S_1 and S_2 by concatenating letters in X and Y arbitrarily at the ends of $s(\vec{v})$ respectively, exit; otherwise, continue.

5: Compute an arbitrary sequence $s(\vec{v})$ with the constraint that the first letter is x and the last letter is y. Then obtain $s_1 = s(\vec{v}) \circ x$ and $s_2 = y \circ s(\vec{v})$ ($\circ$ denotes concatenation).

6: Finally, insert all the elements in $X - \{x\}$ arbitrarily at the two ends of s_1 to obtain S_1, and insert all the elements in $Y - \{y\}$ arbitrarily at the two ends of s_2 to obtain S_2.

7: **Return** S_1 and S_2.

CHAPTER FIVE

PATTERN MATCHING UNDER THE 1-REVERSAL DISTANCE

5.1 Introduction

The pattern matching problem is one of the well-studied classical problems in computer science. Given a text sequence T of length n, a pattern P of length m, the problem is to compute all substrings in T which are exactly the same as P. Over the last four decades, lots of algorithms have been developed. The standard text book solution is the Knuth-Morris-Pratt algorithm [57] which runs in linear time. In many practical cases, the Boyer-Moore algorithm avoids reading each character of the input text to achieve a sublinear algorithm [15], hence the algorithm has been implemented in the emacs editor and the Unix system (even though it runs in $O(nm)$ time in the worst case).

In many applications like biology and communications, the occurrence of a copy of the pattern in T could be slightly altered, for example, letter mutation and corruption. Therefore, the problem of pattern matching with k mismatches has been investigated rigorously as well. The problem is formally defined as follows: Given a text sequence T of length n, a pattern sequence P of length m, and an integer k, the problem is to find all substrings in the text sequence with at most k mismatches to the pattern sequence. The most widely-used measure for the pattern matching with k mismatches problem is the Hamming distance, which is the number of the locations where two strings have different letters when aligned together (see Fig. 5.1 for an example). (Note that the case $k = 0$ is exactly the pattern matching problem.)

Landau and Vishkin first introduced the k-mismatch pattern matching problem in 1985 [65], and they presented two algorithms running in $O(m^2 + k^2 n)$ and $O(k(m \log m + n))$ respectively. Galil and Giancarlo subsequently improved the bound to $O(nk)$ [39]. Then,

$$S_1 = A\ \boldsymbol{C}\ \boldsymbol{G}\ C\ \boldsymbol{C}\ A\ T\ \boldsymbol{G}\ C$$

$$S_2 = A\ \boldsymbol{G}\ \boldsymbol{C}\ C\ \boldsymbol{G}\ A\ T\ \boldsymbol{T}\ C$$

Figure 5.1: An example for two strings S_1 and S_2 with Hamming distance 4.

Abrahamson showed that the problem can be solved in time $O(n\sqrt{m \log m})$, where k is subsumed in the running time [1]. In [6], Amir, Lewenstein and Porat presented an improved algorithm, which runs in time $O(n\sqrt{k \log k})$. There has been a series of research on the streaming version of the problem, which is not quite related to this paper and is not reviewed further, but interested readers are referred to [27, 41].

In this chapter, we consider the pattern matching problem under the 1-reversal distance, where we want to list all substrings of T which has a reversal distance at most 1 to the pattern P. The major motivation if from computational biology where reversals are common operations in genome rearrangements. (Computing the reversal distance between two unsigned genomes, possibly with letter/gene duplications is NP-hard [25]; in fact, the problem remains NP-hard when the unsigned genomes are permutations [18] and constant-factor approximations are know for the latter case [10, 41].) To the best of our knowledge, this problem has never been studied. The closest related works on pattern matching are in references [4, 5], where Amir et al. studied the *Pattern Matching with Swaps* problem, where the problem is to find all *swapped versions* of the given pattern, where a swap operation exchanges two neighboring letters and each letter can participate in at most one swap. Clearly in general a reversal cannot be implemented with these kinds of swaps (unless the reversal operation is applied on a substring of length 2). On the other hand, we note that in computational biology, there has been some research on pattern matching with address

errors (rearrangement distances) [3], though reversal distance was not directly mentioned in the paper. In fact, using the ℓ_1 distance (i.e., the sum of difference of address changes for all the letters in P and in T), the algorithm in [3] can only find the shortest reversal of a segment of P in a potential match in T. In addition, there has been some research on searching tandem duplications in T [12].

The pattern matching under the 1-reversal distance can be solved in $O(n + m)$ time using the known Longest Common Extension (LCE) queries. However, with initial empirical results, it seems that such a solution is very slow when m is small. We then focus on practical solutions for this problem. We design an expected $O(mn)$ time algorithm using the Karp-Rabin fingerprints for the pattern matching problem under 1-reversal distance. We implemented this pattern matching algorithm and applied it to the bacterial E. coli genome sequences. The empirical findings indicates that for $m \in [4, 14]$ there are many more substrings with 1-reversal distance to the pattern compared with with 0-reversal distance (i.e., identical with the pattern).

5.2 Preliminaries

In this section, we present the relevant definitions and notions. A (DNA) sequence is represented by a string over $\Sigma = \{A, C, G, T\}$. Given a string S, we denote by $S[i]$ the i-th symbol of S. We denote by $S[i, j]$ the substring $S[i, j] = s_i s_{i+1} \dots s_j$ of string $S = s_1 s_2 \dots s_i s_{i+1} \dots s_j \dots s_n$. (Note that if S is actually implemented as an array in C++, the first index would be 0.)

Longest Common Extension (LCE) queries The longest common extension (LCE) problem is defined as: Given a string S, for any pair of index (i, j), return the longest common substring of S starting at position i and j. It has been used to solve various string problems starting in 1980s, and is a textbook problem [42]. Ilie et al. and Barton et al.

discussed practical implementation of a solution which uses suffix array, inverse suffix array, longest common prefix array and range minimum queries [11, 49, 50]. This be the basis of a solution we compare with.

Karp-Rabin fingerprints In [55], Karp and Rabin introduced the fingerprint method to represent a string as a number. The fingerprint can be computed in linear time and updated in constant time. It has been widely used.

The Karp-Rabin fingerprint of a string $S = s_1 s_2 \cdots s_m$ is defined as

$$H(S) = \sum_{i=1}^{m} s_i r^i \bmod p,$$

where r is randomly chosen from $\mathbb{F}_p$, where $\mathbb{F}_p = \{b \mid b \leq p \text{ and } b \text{ is prime}\}$. Given two strings S_1 and S_2, if $S_1 = S_2$, then obviously we have $H(S_1) = H(S_2)$. In [78], Benny Porat and Ely Porat proved the following lemma.

Lemma 13. *[78] Given two different strings S_1 and S_2 of length $m \leq n$, and a random r chosen from $\mathbb{F}_p$ where p is a prime number with $p \in \Theta(N^4)$, the probability that $H(S_1) = H(S_2)$ is less than $\frac{1}{n^3}$.*

In [78], n was set to be $n < N$ to make the probability that $H(S_1) = H(S_2)$ less than $\frac{1}{n^3}$. Note that the set $\mathbb{F}_p$ can be generated by the sieve algorithm of Atkin and Bernstein, which runs in $O(N/\log\log N)$ time [8]. It is also known in that the Karp-Rabin fingerprint can be updated in $O(1)$ time, meaning the fingerprint of $s_2 s_3 ... s_i$ and $s_1 s_2 s_3 ... s_{i+1}$ can both be computed from the fingerprint of $s_1 s_2 s_3 ... s_i$ in $O(1)$ time. (This can be done as follows: Let $h_i = s_1 r + s_2 r^2 + \cdots + s_i r^i \mod p$. Then $\frac{h_i - s_1 r}{r} = s_2 r^{2-1} + s_3 r^{3-1} + \cdots + s_i r^{i-1} \mod p = s_2 r + s_3 r^2 + \cdots + s_i r^{i-1} \mod p$, which is exactly the fingerprint of $s_2 s_3 ... s_i$. Similarly, $h_i + s_{i+1} r^{i+1} = ((s_1 r + s_2 r^2 + s_3 r^3 + \cdots + s_i r^i) + s_{i+1} r^{i+1})) \mod p$, which is exactly $s_1 r + s_2 r^2 + s_3 r^3 + \cdots + s_{i+1} r^{i+1} \mod p$, or the fingerprint of $s_1 s_2 s_3 ... s_{i+1}$.) Hence, in our fingerprint

calculation, once the fingerprint of the first m-substring in the text T is computed in $O(m)$ time, the subsequent fingerprints, for all the remaining m-substrings, can each be updated and computed in $O(1)$ times.

We next show that if $S = s_1 s_2 s_3 ... s_m$ is cut into blocks/segments of length exactly x $(x < m)$, $B_1, B_2, ..., B_{\lceil m/x \rceil}$, except possibly for the last one, and let $h(B_i) = s_{(i-1)x+1} r + s_{(i-1)x+2} r^2 + \cdots + s_{(i-1)x+x} r^x \mod p$. We have the following lemma.

Lemma 14. *The fingerprint of $S' = s_2 s_3 ... s_m s_{m+1}$ can be obtained $h(B_i)$'s in $O(m/x)$ time.*

Proof. By definition, the fingerprints of the blocks in S is $h(B_1), h(B_2), \cdots, h(B_{\lceil m/x \rceil}$. Then the fingerprint for S' is

$$\{h(B_1)/r + h(B_2) r^{x-1} + \cdots + h(B_i) r^{(i-1)x-1} + \cdots + h(B_{\lceil m/x \rceil}) r^{(\lceil m/x \rceil - 1)x-1}\}$$
$$+ s_{m+1} r^{m+1} - s_1 \mod p,$$

which takes $O(m/x)$ time. □

Reversal operation We give the formal definition and notion for the reversal operation as follows.

Definition 13. *(Reversal Operation) A reversal is an operation that reverses the order of a substring. More formally, the reversal operation $\rho(i, j)$ transforms the string S to S' as follows:*

$$S = s_1 \cdots s_{i-1} \underline{s_i s_{i+1} \cdots s_j} \; s_{j+1} \ldots s_n$$
$$S' = s_1 \cdots s_{i-1} \underline{s_j \cdots s_{i+1} s_i} \; s_{j+1} \ldots s_n$$

Note that, given two strings S and T, if a reversal operation $\rho(i, j)$ transforms S to T, the front parts and the end parts of S and T are exactly same. In addition, while there may

be more than one possible reversal operations which can transform S to T, we are interested an optimal (or minimal) reversal operation which transforms S into T.

Definition 14. *(Optimal 1-Reversal Operation) A reversal $\rho(i,j)$ is said to be an optimal 1-reversal operation, if the reversal $\rho(i,j)$ can transform a sequence S to a sequence T, and there is no other reversal operation $\rho(k,\ell)$ exists, where $i < k$ or $j > \ell$.*

From the definition of optimal 1-reversal operations, we have the following lemma.

Lemma 15. *Given two sequences S and T, if an optimal 1-reversal operation $\rho(i,j)$ can transform S to T, then $S[i] \neq S[j]$, $S[i] \neq T[i]$ and $S[j] \neq T[j]$.*

Proof. Assume that there exists a reversal operation $\rho(k,\ell)$ which can transform S to T, where we have $S[k] = T[k]$. By the definition of a reversal operation, $S[k] = T[\ell]$ and $S[\ell] = T[k]$, hence we have $S[k] = S[\ell] = T[k] = T[\ell]$. Then there must exist a reversal operation $\rho(m+1, n-1)$ which transform S into T. Therefore, if $\rho(i,j)$ is an optimal reversal operation, then we have $S[i] \neq S[j]$, $S[i] \neq T[i]$ and $S[j] \neq T[j]$. □

Definition 15. *(Reversal Distance) Given two strings S and T with the same content (i.e., multiset of letters), the Reversal Distance $d_R(S,T)$ is the minimum number of reversal operations that transforms S to T.*

Then we give the formal definition of the problem we investigate in this chapter as follows:

Definition 16. *The Pattern Matching with 1-reversal Distance problem:*

`Input:` *a text sequence $T = t_1 t_2 \dots t_n$ of length n and a pattern sequence $P = p_1 p_2 \dots p_m$ of length m over $\Sigma = \{A, C, G, T\}$, where $m < n$*

`Question:` *find all substrings from the text sequence T of length m which have at most 1-reversal distance to the pattern P.*

5.3 The Algorithms

In this section, we present two algorithms, running in $O(n+m)$ and $O(mn)$ time respectively, for the pattern matching with 1-reversal distance problem. Given a string S, let $\bar{S}$ be its reversal.

5.3.1 LCE-based solution

The first algorithm is based on LCE queries. We construct three sequences $T_1 = P\#T$, $T_2 = \bar{P}\#\bar{T}$ and $T_3 = P\#\bar{T}$, where $\#$ is a new character, and we build the data structure in $(m+n)$ time/space for LCE queries using the results in [11]. Suppose a match between P and a segment of T is between $P = P[1..m]$ and $T[k,l]$ where $\bar{P}[i',j'] = T[i,j]$ and $k-l+1 = m$. This can be checked as follows: an LCE query of $(1, m+1+k)$ in T_1, which will return length ℓ_1 (and locate i' and i when there is a match); an LCE query of $(1, m+n-l+2)$ in T_2, which will return length ℓ_2 (and locate index $m-j'+1$ and $m+n-j+2$, hence j' and j, when there is a match); and finally we use another LCE query $(i', m+n-j+2)$ in T_3, which will return length ℓ_3. Finally, we check whether $\ell_1+\ell_2+\ell_3 = m$, if so, a match is confirmed. This obviously take $O(1)$ time. Since we have to search all possible $n-m+1$ matches, this search takes $O(n-m+1)$ time. Therefore, we have the following theorem.

Theorem 12. *Pattern matching under the 1-reversal distance problem can be solved in $O(m+n)$ time, where m is the length of the pattern and n is the length of the text.*

Obviously, the above result is theoretically optimal. However, based on our empirical testing, the algorithm is quite slow even for small m. Hence, we are interested in designing a practical algorithm which is fast for practical datasets. We will make use of the Karp-Rabin fingerprints in this case.

5.3.2 Fingerprint-based solution

Given a text sequence T of length n and a pattern sequence P of length m, we denote by T^k the m-substring of T starting at index k, i.e., $T^k = T[k, k+m-1]$. In the preprocessing step, for the fingerprint computation, we convert the sequence over the original $\Sigma = \{A, C, G, T\}$ to the sequence over $\Sigma' = \{1, 2, 3, 4\}$. Moreover, we compute $\mathbb{F}_p$ as a set of prime numbers using the prime sieves algorithm by Atkin and Bernstein [8]. Then, r is randomly chosen from the $\mathbb{F}_p$.

Next, we show how to determine if $d_R(P, T^k) \leq 1$, where $k = 0, \cdots, n-m-1$. Recall that, by Lemma 15 if $d_R(P, T^k) = 1$ then we must locate the optimal 1-reversal operation between the pattern P and the substring T^k. As our goal is to reduce the running time by using the Karp-Rabin fingerprint, we preprocess the text T by cutting it into substrings of length x (note that the last ones might be of lengths strictly less than x). Since in practice when building the fingerprints for T, we do not know the exact length of m in advance, x and m might not be exactly the same, even if we prepare the fingerprints for several different x's for T. Hence, we use two lists P_ℓ and T_ℓ to represent these two lists composed of x-substrings respectively. (Note that the last ones might have a length less than x.)

Then, we compute the corresponding Karp-Rabin fingerprint lists F_P and F_T for P_ℓ and T_ℓ respectively. To find the first index i where T^k and P differ, i.e., where $T^k[i] \neq P[i]$, we compare the corresponding pairs of the fingerprint numbers in the two lists F_P and F_T sequentially. (Note that this comparison for the fingerprint of T^k, $F_T[T^k]$, might involve a sliding window computation of the fingerprints which takes $O(m/x)$ time when the previous ones are given — as shown in Lemma 14.) At any step, if these two numbers are different (starting with the x-length block containing $T^k[i]$), then by Lemma 13 and 14, with a high probability (i.e., at least $1 - \frac{1}{n^3}$), we can find the index i from the corresponding sequences in the two lists P_ℓ and T_ℓ in $O(m/x)$ time. (If the fingerprints of T^k and P are the same, we will still check if the corresponding two sequences of two lists P_ℓ and T_ℓ, i.e., P and T^k, are

$T_\ell =$ | "3321" | "1344" | "2314" | "4342" | "231" |

$F_T =$ | 57 | 22 | 31 | 92 | 66 |

$T_\ell[3, 13] =$ | "1" | "1344" | "2314" | "43" |

$F_T[3, 13] =$ | 1 | 22 | 31 | 23 |

$T_\ell[T^3] =$ | "1134" | "4231" | "443" |

$F_T[T^3] =$ | 68 | 61 | 22 |

$P_\ell =$ | "1134" | "4241" | "343" |

$F_P =$ | 68 | 66 | 98 |

Figure 5.2: An example for computing the fingerprint list

the same. If these two corresponding sequences P and T^K are different, we can still find the index i in a brute force way. Note that, from Lemma 13, the probability of such an event is at most $\frac{1}{n^3}$.) An example of this process is shown in Fig. 5.2. In this example, we have $n = 19, m = 11, x = 4, r = 5$, and the prime $p = 101$. The text $T =$ 3321 1344 2314 4342 231 and pattern $P =$ 1134 4241 343. We focus on $T^3 = T[3, 13]$ — here the index of an array in C++ starts with 0. Note that once $F_T[T^3]$ is sequentially computed (by updating the previous ones), we will find that i should occur in the second block as the fingerprint is 61, which is different from the corresponding one in P. Then $i = 6$ is computed by searching the corresponding substrings in the corresponding blocks of T^3 and P.

We can construct the lists and the corresponding fingerprint lists in reverse order (i.e., from right to left) to find the last index j such that $T^k[j] \neq P[j]$. The pseudocode for

locating i and j is shown in Algorithm 5.2. Note that when we search j all the x-substrings in T and P do not have to be aligned from right to left.

Next, we need to determine whether $T^k[i,j]$ is a reversal of $P[i,j]$. Recall that $\bar{S}$ is the reversal of a given sequence S. For this process, we also compute the fingerprint list $\bar{F}_P$ for the reversed pattern sequence list $\bar{P}_\ell$.

After the indices i and j have been computed, we first retrieve the part $\bar{P}_\ell[i,j]$ of the sequence list $\bar{P}_\ell$ for the reversed substring $\bar{P}[i,j]$. In addition, we retrieve the partial lists $\bar{F}_P[i,j]$, $T_\ell[i,j]$ and $F_T[i,j]$. Next, if all the fingerprints in $\bar{F}_P[i,j]$ and $F_T[i,j]$ (i.e., $F_T[T^k]$ after alignment) are the same and all sequences of $\bar{P}_\ell[i,j]$ and $T_\ell[i,j]$ (i.e., $T_\ell[T^k]$ after alignment) are same, then $d_r(P,F^k)=1$.

The last step of the algorithm is to update the lists F_T and T_ℓ for checking the next substring in T. The pseudocode of the algorithm is shown in Algorithm 5.3. For comparison purpose, in the algorithm, we count both the substrings with 0-reversal distance (i.e, identical matchings) and the substrings with-1 reversal distance.

Theorem 13. *Assume that $\mathbb{F}_p$ is given and r is already randomly selected from $\mathbb{F}_p$, the pattern matching problem under the 1-reversal distance can be solved in expected $O(mn)$ time.*

Proof. In the preprocessing step of the algorithm, converting the input DNA sequences (text and pattern) over $\Sigma=\{A,C,G,T\}$ to the sequences over $\Sigma'=\{1,2,3,4\}$ can be done in $O(n+m)$ time. The computation of the fingerprint for P can be done in $O(m)$ time, regardless of the selection of x.

Following Lemma 14, the computation of the corresponding fingerprint for each of T^k can be done in $O(m/x+x)$ time — after T^1 is computed in $O(m)$ time. Therefore, in each iteration, the indices i and j (representing the starting and ending index of a substring in T where the 1-reversal occurs) can be computed using fingerprints in $O(m/x+x)$ time.

Algorithm 5.2: The boundary location algorithm

```
1: function LOCATION(F_P, F_T, P_ℓ, T_ℓ)
2:     i ← −1, j ← −1
3:     for index_1 = 0 to |F_P| − 1 do
4:         if F_P[index_1] == F_T[index_1] then
5:             if P_ℓ[index_1] ≠ T_ℓ[index_1] then
6:                 for index_2 = 0 to |P_ℓ[index_1]| − 1 do
7:                     if P_ℓ[index_1][index_2] ≠ T_ℓ[index_1][index_2] then
8:                         i ← index_1 ∗ |P_ℓ[index_1]| + index_2;
9:                         break;
10:                    end if
11:                end for
12:            end if
13:        else
14:            for index_2 = 0 to |P_ℓ[index_1]| − 1 do
15:                if P_ℓ[index_1][index_2] ≠ T_ℓ[index_1][index_2] then
16:                    i ← index_1 ∗ |P_ℓ[index_1]| + index_2;
17:                    break;
18:                end if
19:            end for
20:        end if
21:    end for
```

```
22:     for index_1 = |T_l| − 1 to 0 do
23:         if F_P[index_1] == F_T[index_1] then
24:             if P_ℓ[index_1] ≠ T_ℓ[index_1] then
25:                 for index_2 = |P_ℓ[index_1]| − 1 to 0 do
26:                     if P_ℓ[index_1][index_2] ≠ T_ℓ[index_1][index_2] then
27:                         j ← (|P_ℓ| − index_1) ∗ |P_ℓ[index_1]| + index_2;
28:                         break;
29:                     end if
30:                 end for
31:             end if
32:         else
33:             for index_2 = |P_ℓ[index_1]| − 1 to 0 do
34:                 if P_ℓ[index_1][index_2] ≠ T_ℓ[index_1][index_2] then
35:                     j ← (|P_ℓ| − index_1) ∗ |P_ℓ[index_1]| + index_2;
36:                     break;
37:                 end if
38:             end for
39:         end if
40:     end for
41: end function
```

Algorithm 5.3: The 1-reversal checking algorithm

function REVERSAL(sequence T of length n, sequence P of length m ,t)
  $T, P \leftarrow$ convert T and P to strings over $\Sigma = \{1, 2, 3, 4\}$;
  Compute $\mathbb{F}_p$ using the Atkin-Bernstein algorithm;
  Randomly select r is from $\mathbb{F}_p$;
  $k \leftarrow 0$, $counter0 \leftarrow 0$, $counter1 \leftarrow 0$;
  $P_\ell, \bar{P}_\ell, T_\ell[T^k] \leftarrow$ divide $P, \bar{P}$ and T^k into substring with length $\lceil\sqrt{m}\rceil$;
  $F_P \leftarrow fingerprint\_computation(P_\ell)$;
  $\bar{F}_P \leftarrow fingerprint\_computation(\bar{P}_\ell)$;
  $F_T[T^k] \leftarrow fingerprint\_computation(T_\ell[T^k])$;
  while $k \leq n - m - 1$ **do**
    $i, j \leftarrow locate(F_P, F_T, P_\ell, T_\ell)$;
    if i, j exists **then**
      if $\bar{F}_P[i, j] == F_T[T^k][i, j]$ **then**
        if $\bar{P}_\ell[i, j] == T_\ell[T^k]l[i, j]$ **then**
          $counter1 \leftarrow counter1 + 1$;
        end if
      end if
    else
      $counter0 \leftarrow counter0 + 1$;
    end if
    $k \leftarrow k + 1$;
    Update $F_T[T^k], T^k$;
  end while
end function

Determining whether the reversal parts of two lists in T^k and P are equal can be done in $O(m/x + x)$ time using fingerprints. Since the fingerprint method is a randomized one, we still need to check whether a target T^k matches P letter-wise when $H(T^k) = H(P)$ (following Lemma 13, the probability is at most $\frac{1}{n^3}$). Therefore, the expected running time of each query step is at most

$$1 \times O(m + m/x + x) + \frac{1}{n^3} \times O(m) = O(m + m/x + x) = O(m).$$

Since we have $n - m + 1$ number of m-substrings in T, the total expected running time of the algorithm is $(n - m + 1) \times O(m) = O(mn)$. □

We show some empirical results in the next section. It would be interesting to know the difference when $x = m$ and $x = \sqrt{m}$; certainly whether we have to check $T^k = P$ when $H(T^k) = H(P)$.

5.4 Empirical Results

We implemented this algorithm in C++. The algorithm was run on a laptop with Dual-Core Intel i5 CPU, 2.6 GHz, 8 GB RAM. We tested our algorithm on the"Escherichia coli strain: FORC_028 Genome sequencing" (E. coli for short), with length of 5,704,396, which is downloaded from `https://www.ncbi.nlm.nih.gov/bioproject/PRJNA294502`.

We first searched for a longest m-substring of E. coli where it matches another substring with a reversal distance one. Our code runs for more than 10 days and we found the following substring in E. coli with length 22: `TAACACGCTGGCCCTGTACGCG`, the start position is 105. The only match is a substring in E. coli starting at position 825122: `TAAC CCCGGTCGCATGTACGCG`. Although we are unable to identify any biological meaning of these two substrings, the search indicates that m is usually small.

Next we make some empirical comparisons. In all the comparisons, we fix $N = mn$, hence $p \in \Theta(N)$. (Note that Lemma 13 needs a much larger p, which is not practical using our computer.) We mostly focus on comparing the search time on different m (even though for E. coli m is meaningful only when it is small). For larger m we try to make use of synthetic data by modifying the E. coli sequence. From now one, we either use a prefix of the E. coli sequence of length 1M, or a synthetic sequence obtained from this prefix (with the same length).

5.4.1 With and without letter checking

Given n and m, we cut the first n letters in each of the sequences as the text T and the first m letters as the pattern P. In the implementation of the algorithm, we count the substrings with 0-reversal distance and the substrings with 1-reversal distance separately. The running time of the query algorithm is the average over 10 tries, represented as *Time* in Table 5.1.

Since skipping the actual check $T^k = P$ when $H(T^k) = H(P)$ might potentially improve the running time, in Table 5.2, we use the same setting to report the running time as well as the computed counts: *counter0* and *counter 1*. It can be seen that in Table 5.2, we have false positive cells (marked red, meaning some string which is different from the pattern but share the same fingerprint). In other words, while skipping the letter-wise check can slightly improve the running time, it can occasionally report wrong answers.

The empirical results show that the E. coli sequences contains more substrings with 1-reversal distance than the substrings with 0-reversal distance. The shorter the pattern length is, the more substrings with 1-reversal distance E. coli has.

5.4.2 Comparison with LCE-based solution

In Table 5.3, we show the running time using the LCE-based solution. (The code was downloaded from [11].) It can be seen that when m is in the range of [4,13], the fingerprint-

Table 5.1: The average query time over 10 tries when $x = m$ and when there is a letter-wise check after the fingerprints are found to be match.. The correct answer was obtained using a brute-force method for *counter1*.

$\|P\|$	$\|T\|$	$counter0$	$counter1$	Time (milliseconds)	Correct answer
4	1000000	2620	18261	1787	18261
5	1000000	482	8578	2012	8578
6	1000000	141	2750	2271	2750
7	1000000	29	801	2483	801
8	1000000	6	261	2664	261
9	1000000	1	89	2871	89
10	1000000	1	42	3148	42
11	1000000	1	16	3375	16
12	1000000	0	4	3659	0
13	1000000	0	1	3784	1
14	1000000	0	0	4074	0
15	1000000	0	0	4329	0
16	1000000	0	0	4510	0

Table 5.2: The average query time over 10 tries when $x = m$ and when there is no letter-wise check after the fingerprints are found to be match. The correct answer was obtained using a brute-force method for *counter1*. The underlined (red) cells indicate the appearance of false-positive cases.

$\|P\|$	$\|T\|$	$counter0$	$counter1$	Time (milliseconds)	Correct answer
4	1000000	2620	18261	1756	18261
5	1000000	482	8578	1979	8578
6	1000000	141	2750	2203	2750
7	1000000	29	801	2447	801
8	1000000	6	261	2643	261
9	1000000	1	89	2835	89
10	1000000	1.1	42	3092	42
11	1000000	1	16.1	3330	16
12	1000000	0	4	3490	4
13	1000000	0	1.1	3764	1
14	1000000	0	0	4016	0
15	1000000	0	0.1	4269	0
16	1000000	0	0.1	4478	0

Table 5.3: The query time using the longest common extension algorithm (not counting preprocessing time). All the settings are the same as in Table 5.1.

$\|P\|$	$\|T\|$	LCE time without counting preprocessing (milliseconds)
4	1000000	58578
5	1000000	57849
6	1000000	55656
7	1000000	55836
8	1000000	52292
9	1000000	54649
10	1000000	55351
11	1000000	55189
12	1000000	54163
13	1000000	55078
14	1000000	54673
15	1000000	53814
16	1000000	53262

based solution is 14-32 times faster than the LCE-based solution.

Since Table 5.1 shows a stable running time pattern, we tried to increase the pattern size artificially. We randomly generate a pattern of length m, and randomly reverse a segment of it and paste this on the text T, again randomly. Step by step, 100 such segments were pasted on T. We then ran the algorithms as in Table 5.1 (i.e., $x = m$ and a letter-wise check is applied whenever the fingerprints are found to match). The running time is shown in Table 5.4. It can be seen that only after $m > 200$ the running times of the fingerprint-based algorithm are starting to be slower than the LCE-based solution.

Table 5.4: The running time and correct counts using a simulated dataset with 100 patterns, $x = m$ and letter-wise checking is applied when the fingerprints match.

$\|P\|$	$\|T\|$	$counter0$	$counter1$	Time (milliseconds)	Correct answer
50	1000000	0	100	13786	100
100	1000000	0	100	25303	100
150	1000000	0	100	36980	100
200	1000000	0	100	48406	100
250	1000000	0	100	60323	100
300	1000000	0	100	72290	100
350	1000000	0	100	83466	100
400	1000000	0	100	95807	100
500	1000000	0	100	119369	100
600	1000000	0	100	141435	100
700	1000000	0	100	165368	100
800	1000000	0	100	191787	100

5.4.3 With and without the x-cuts

In our time analysis, x was set as a parameter. It would be interesting to to know whether choosing a different value of x would make much difference. In Table 5.5 and Table 5.6, we use $x = \lceil \sqrt{m} \rceil$. The other setting of Table 5.5 (resp. Table 5.6) is exactly the same as in Table 5.1 (resp. Table 5.4). For convenience, we also include the running time without letter check (when a match in fingerprints is found).

It can be seen that the running time when x-cuts are used, the running time does not improve until when m reaches about 100, but the improvement is small even when $m = 800$. In this case, it takes 191787ms to run for the case without x-cuts in m (or $x = m$), while it takes 164061ms with x-cut. Again, using no letter check does not improve much in terms of running time (nonetheless it could still give false-positive answers). We also note that the running time with x-cuts might not be stable (e.g., when $m = 40,000$), this is possibly due to that the offsets of cutting are quite different for different values of m.

5.5 Conclusion

In this chapter, we consider the pattern matching under the 1-reversal distance problem. Given a text T of length n and a pattern P of length m, we design an $O(m+n)$ time algorithm using LCE and an $O(mn)$ time randomized algorithm using the Karp-Rabin fingerprints. The algorithms are implemented in C++ and tested on a segment of the "Escherichia coli strain: FORC_028 Genome sequencing" of length 1M. The empirical results show the following: (1) m is only biologically meaningful when it is small (roughly 4-22); in fact, in this range the shorter the pattern length is, the more substrings with 1-reversal distance to the pattern; (2) The running time of the LCE solution is stable and is much slower than the fingerprint algorithm until m reaches roughly 200; when $m > 200$ the LCE solution starts to outperform the fingerprint algorithm (this was tested on simulated data obtained from the

Table 5.5: The average query time over 10 tries, $x = \sqrt{m}$. The pattern and text are the same as in Table 5.1.

$\|P\|$	$\|T\|$	Time with letter check	Time without letter check
4	1000000	2364	2343
5	1000000	2947	2937
6	1000000	2940	2934
7	1000000	4584	3986
8	1000000	4053	3988
9	1000000	3705	3693
10	1000000	4846	4728
11	1000000	4613	4580
12	1000000	5046	5008
13	1000000	6081	5956
14	1000000	6159	6138
15	1000000	5813	5766
16	1000000	5532	5436

Table 5.6: The average query time over 10 tries, $x = \sqrt{m}$. The pattern and text are the same as in Table 5.4.

$\|P\|$	$\|T\|$	Time with letter check	Time without letter check
50	1000000	16986	16890
100	1000000	13873	13769
150	1000000	35776	35545
200	1000000	53373	52042
250	1000000	49828	49820
300	1000000	56228	56046
350	1000000	80145	79678
400	1000000	36071	35922
500	1000000	92173	91992
600	1000000	64825	64787
700	1000000	104279	102936
800	1000000	164061	163564

E. coli segment); (3) The fingerprint algorithm is randomized, so the same fingerprint does not always imply a match, but our empirical results show that additional letter check (to confirm a match) does not significantly increase the searching time; (4) Cutting T (and P) into segments of length roughly $\sqrt{m}$ will make the search faster only when $m > 100$.

An interesting problem is when there are additional errors in a potential match in T (say Hamming distance k) after the reversal of a segment of the pattern is performed. The current methods, whether LCE-based or fingerprint-based, do not seem to work directly. We are currently working along this direction.

CHAPTER SIX

CONCLUSION AND FUTURE WORK

In this dissertation, we have investigated several fundamental problems and some applications related to duplications and deletions in genome rearrangements.

The first problem we investigated is the tandem duplication distance problem: what is the complexity to compute the tandem duplication distance between two sequences A and B. For proving the hardness result, we first introduced the cost-effective subgraph problem. We proved that the cost-effective subgraph problem is NP-hard and $W[1]$-hard for parameter $c+p$. Then, we reduced this problem to the promise version of the exemplar-k-TD problem and proved that the exemplar-k-TD problem is NP-hard. Subsequently, we proved this problem is still NP-hard, even if $|\Sigma| \geq 4$ by encoding each letter in the unbounded alphabet with a square-free string over a new alphabet of size 4. Finally, we designed an FPT algorithm for the exemplar version of this problem. Some open questions were answered in this work, and still many of them deserve investigation as follows: (1) If $|\Sigma| = 3$, is the TD problem still NP-hard? (2) What is the complexity to decide whether S can be converted to T only by tandem duplications?

Second, different variants of the longest letter-duplicated subsequence (LLDS) problem were studied. Given a sequence S of length n, a letter-duplicated subsequence (LDS) of S is a subsequence of S in form $x_1^{d_1}x_2^{d_2}\cdots x_k^{d_k}$ 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). The natural question is to compute the longest letter-duplicated subsequence (LLDS). For the LLDS problem without any constraint, we gave a linear time dynamic programming algorithm. Then, we proved that a constrained version LLDS problem is NP-hard. In this version, the alphabet Σ is unbounded, each letter appears in S at least 6 times and all the letters in Σ must appear

in the solution. We also showed that when each letter appears at most 3 times, then the problem admits a factor $1.5-O(\frac{1}{n})$ approximation. Finally, we presented a non-trivial $O(n^2)$ time dynamic programming algorithm for the weighted-LDS problem, where the weight of each LD-block is any positive function (i.e., it does not even have to grow with its length).

Third, we investigated the minimum copy number generation (MCNG) problem. Given a genome G represented as a string and a copy number profile $\vec{c}$, the minimum copy number generation (MCNG) problem asks for the minimum number of deletions and duplications needed to transform G into any genome in which each character occurs as many times as specified by $\vec{c}$. We showed that MCNG is NP-hard to approximate within any constant factor, and that it is W[1]-hard when parameterized by the solution size. We also presented a polynomial time algorithm for the copy number profile conforming (CNPC) problem when the distance is the classical breakpoint distance.

Finally, we considered the pattern matching under the 1-reversal distance problem. For this problem, we designed an $O(m + n)$ time algorithm using LCE and an $O(mn)$ time randomized algorithm using the Karp-Rabin fingerprints. These algorithms were implemented and tested on a segment of the "Escherichia coli strain: FORC_028 Genome Sequencing" of length 1M. The empirical results show the following: (1) the shorter the pattern length is, the more substrings with 1-reversal distance to the pattern; (2) when the pattern length is short, the LCE solution is much slower than the fingerprint algorithm. When the pattern length is bigger than 200, the LCE solution starts to outperform the fingerprint algorithm. A possible direction for future work would be using a different distance, e.g., transposition, etc.

REFERENCES CITED

[1] Karl Abrahamson. Generalized string matching. *SIAM Journal on Computing*, 16(6):1039–1051, 1987.

[2] Noga Alon, Jehoshua Bruck, Farzad Farnoud Hassanzadeh, and Siddharth Jain. Duplication distance to the root for binary sequences. *IEEE Transactions on Information Theory*, 63(12):7793–7803, 2017.

[3] Amihood Amir, Yonatan Aumann, Gary Benson, Avivit Levy, Ohad Lipsky, Ely Porat, Steven Skiena, and Uzi Vishne. Pattern matching with address errors: rearrangement distances. *Journal of Computer and System Sciences*, 75(6):359–370, 2009.

[4] Amihood Amir, Yonatan Aumann, Gad M. Landau, Moshe Lewenstein, and Noa Lewenstein. Pattern matching with swaps. *Journal of Algorithms*, 37(2):247–266, 2000.

[5] Amihood Amir, Richard Cole, Ramesh Hariharan, Moshe Lewenstein, and Ely Porat. Overlap matching. *Information and Computation*, 181(1):57–74, 2003.

[6] Amihood Amir, Moshe Lewenstein, and Ely Porat. Faster algorithms for string matching with k mismatches. *Journal of Algorithms*, 50(2):257–275, 2004.

[7] Sébastien Angibaud, Guillaume Fertin, and Irena Rusu. On the approximability of comparing genomes with duplicates. In *International Workshop on Algorithms and Computation*, pages 34–45. Springer, 2008.

[8] Arthur Atkin and Daniel Bernstein. Prime sieves using binary quadratic forms. *Mathematics of Computation*, 73(246):1023–1030, 2004.

[9] Anne Atlan and Denis Couvet. A model simulating the dynamics of plant mitochondrial genomes. *Genetics*, 135(1):213–222, 1993.

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

[11] Carl Barton, Costas S. Iliopoulos, and Solon P. Pissis. Fast algorithms for approximate circular string matching. *Algorithms for Molecular Biology*, 9(1):1–10, 2014.

[12] Gary Benson. An algorithm for finding tandem repeats of unspecified pattern size. In *Proceedings of the Second Annual International Conference on Computational Molecular Biology*, pages 20–29, 1998.

[13] Gary Benson and Lan Dong. Reconstructing the duplication history of a tandem repeat. In *ISMB*, pages 44–53. AAAI, 1999.

[14] Daniel P. Bovet and Stefano Varricchio. On the regularity of languages on a binary alphabet generated by copying systems. *Information Processing Letters*, 44(3):119–123, 1992.

[15] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[16] Thomas D. Bruns and Jeffrey D. Palmer. Evolution of mushroom mitochondrial DNA: Suillus and related genera. *Journal of Molecular Evolution*, 28(4):349–362, 1989.

[17] Laurent Bulteau, Guillaume Fertin, and Irena Rusu. Sorting by transpositions is difficult. *SIAM Journal on Discrete Mathematics*, 26(3):1148–1180, 2012.

[18] Alberto Caprara. Sorting permutations by reversals and Eulerian cycle decompositions. *SIAM Journal on Discrete Mathematics*, 12(1):91–110, 1999.

[19] S. Casjens, G. Hatfull, and R. Hendrix. Evolution of dsDNA tailed-bacteriophage genomes. In *Seminars in Virology*, volume 3, pages 383–383. HARCOURT BRACE JOVANOVICH, 1992.

[20] Brian Charlesworth, Paul Sniegowski, and Wolfgang Stephan. The evolutionary dynamics of repetitive DNA in eukaryotes. *Nature*, 371(6494):215–220, 1994.

[21] 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.

[22] Zhi-Zhong Chen, Lusheng Wang, and Zhanyong Wang. Approximation algorithms for reconstructing the duplication history of tandem repeats. *Algorithmica*, 54(4):501–529, 2009.

[23] Da-Jung Cho, Yo-Sub Han, and Hwee Kim. Bound-decreasing duplication system. *Theoretical Computer Science*, 793:152–168, 2019.

[24] Salim Akhter Chowdhury, Stanley E. Shackney, Kerstin Heselmeyer-Haddad, Thomas Ried, Alejandro A. Schäffer, and Russell Schwartz. Algorithms to model single gene, single chromosome, and whole genome copy number changes jointly in tumor phylogenetics. *PLoS Computational Biology*, 10(7):e1003740, 2014.

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

[26] Giovanni Ciriello, Martin L. Miller, Bülent Arman Aksoy, Yasin Senbabaoglu, Nikolaus Schultz, and Chris Sander. Emerging landscape of oncogenic signatures across human cancers. *Nature Genetics*, 45(10):1127–1133, 2013.

[27] Raphaël Clifford, Tomasz Kociumaka, and Ely Porat. The streaming k-mismatch problem. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1106–1125. SIAM, 2019.

[28] Garance Cordonnier and Manuel Lafond. Comparing copy-number profiles under multi-copy amplifications and deletions. *BMC Genomics*, 21(2):1–12, 2020.

[29] Jürgen Dassow, Victor Mitrana, and Gheorghe Paun. On the regularity of duplication closure. *Bull. EATCS*, 69:133–136, 1999.

[30] Th Dobzhansky and Alfred H. Sturtevant. Inversions in the chromosomes of drosophila pseudoobscura. *Genetics*, 23(1):28, 1938.

[31] Rod G. Downey and Michael R. Fellows. Fixed-parameter tractability and completeness II: On completeness for W[1]. *Theoretical Computer Science*, 141(1-2):109–131, 1995.

[32] Rodney G. Downey and Michael Ralph Fellows. *Parameterized complexity*. Springer Science & Business Media, 2012.

[33] Andrzej Ehrenfeucht and Grzegorz Rozenberg. On regularity of languages generated by copying systems. *Discrete Applied Mathematics*, 8(3):313–317, 1984.

[34] Mohammed El-Kebir, Benjamin J. Raphael, Ron Shamir, Roded Sharan, Simone Zaccaria, Meirav Zehavi, and Ron Zeira. Copy-number evolution problems: complexity and algorithms. In *International Workshop on Algorithms in Bioinformatics*, pages 137–149. Springer, 2016.

[35] Mohammed El-Kebir, Benjamin J. Raphael, Ron Shamir, Roded Sharan, Simone Zaccaria, Meirav Zehavi, and Ron Zeira. Complexity and algorithms for copy-number evolution problems. *Algorithms for Molecular Biology*, 12(1):1–11, 2017.

[36] Chuanzhu Fan, Ying Chen, and Manyuan Long. Recurrent tandem gene duplication gave rise to functionally divergent genes in drosophila. *Molecular Biology and Evolution*, 25(7):1451–1458, 2008.

[37] Farzad Farnoud, Moshe Schwartz, and Jehoshua Bruck. The capacity of string-duplication systems. *IEEE Transactions on Information Theory*, 62(2):811–824, 2015.

[38] Michael R. Fellows, Danny Hermelin, Frances Rosamond, and Stéphane Vialette. On the parameterized complexity of multiple-interval graph problems. *Theoretical Computer Science*, 410(1):53–61, 2009.

[39] Zvi Galil and Raffaele Giancarlo. Improved string matching with k mismatches. *ACM SIGACT News*, 17(4):52–54, 1986.

[40] Olivier Gascuel, Michael D. Hendy, Alain Jean-Marie, and Robert McLachlan. The combinatorics of tandem duplication trees. *Systematic Biology*, 52(1):110–118, 2003.

[41] Shay Golan, Tomasz Kociumaka, Tsvi Kopelowitz, and Ely Porat. The streaming k-mismatch problem: Tradeoffs between space and total time. In *31st Annual Symposium on Combinatorial Pattern Matching*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[42] Dan Gusfield. Algorithms on stings, trees, and sequences: Computer science and computational biology. *ACM Sigact News*, 28(4):41–60, 1997.

[43] Dan Gusfield and Jens Stoye. Linear time algorithms for finding and representing all the tandem repeats in a string. *Journal of Computer and System Sciences*, 69(4):525–546, 2004.

[44] Sridhar Hannenhalli and Pavel A. Pevzner. Transforming men into mice (polynomial algorithm for genomic distance problem). In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 581–592. IEEE, 1995.

[45] Richard J. Hoffmann, J.L. Boore, and W.M. Brown. A novel mitochondrial genome organization for the blue mussel, mytilus edulis. *Genetics*, 131(2):397–412, 1992.

[46] Patrick Holloway, Krister Swenson, David Ardell, and Nadia El-Mabrouk. Ancestral genome organization: an alignment approach. *Journal of Computational Biology*, 20(4):280–295, 2013.

[47] Sara B. Hoot and Jeffrey D. Palmer. Structural rearrangements, including parallel inversions, within the chloroplast genome of anemone and related genera. *Journal of Molecular Evolution*, 38(3):274–281, 1994.

[48] R. Hull. Genome organization of retroviruses and retroelements: evolutionary considerations and implications. In *Seminars in Virology*, volume 3, pages 373–373. Harcourt Brace Jovanovich, 1992.

[49] Lucian Ilie, Gonzalo Navarro, and Liviu Tinta. The longest common extension problem revisited and applications to approximate string searching. *Journal of Discrete Algorithms*, 8(4):418–428, 2010.

[50] Lucian Ilie and Liviu Tinta. Practical algorithms for the longest common extension problem. In *International Symposium on String Processing and Information Retrieval*, pages 302–309. Springer, 2009.

[51] Masami Ito, Peter Leupold, and Kayoko Shikishima-Tsuji. Closure of language classes under bounded duplication. In *International Conference on Developments in Language Theory*, pages 238–247. Springer, 2006.

[52] Siddharth Jain, Farzad Farnoud Hassanzadeh, and Jehoshua Bruck. Capacity and expressiveness of genomic tandem duplication. *IEEE Transactions on Information Theory*, 63(10):6129–6138, 2017.

[53] Haitao Jiang, Chunfang Zheng, David Sankoff, and Binhai Zhu. Scaffold filling under the breakpoint and related distances. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 9(4):1220–1229, 2012.

[54] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Springer, 1972.

[55] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[56] E.B. Knox, S.R. Downie, and J.D. Palmer. Chloroplast genome rearrangements and the evolution of giant lobelias from herbaceous ancestors. *Molecular Biology and Evolution*, 10(2):414–430, 1993.

[57] Donald E. Knuth, James H. Morris, Jr, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[58] Eugene V. Koonin, Valerian V. Dolja, and T. Jack Morris. Evolution and taxonomy of positive-strand RNA viruses: implications of comparative analysis of amino acid sequences. *Critical Reviews in Biochemistry and Molecular Biology*, 28(5):375–430, 1993.

[59] Adrian Kosowski. An efficient algorithm for the longest tandem scattered subsequence problem. In *International Symposium on String Processing and Information Retrieval*, pages 93–100. Springer, 2004.

[60] Manuel Lafond, Binhai Zhu, and Peng Zou. Genomic problems involving copy number profiles: Complexity and algorithms. In *31st Annual Symposium on Combinatorial Pattern Matching*, volume 161, pages 22:1–22:15, 2020.

[61] Manuel Lafond, Binhai Zhu, and Peng Zou. The tandem duplication distance is NP-hard. In *37th International Symposium on Theoretical Aspects of Computer Science*, volume 154, pages 15:1–15:15, 2020.

[62] Manuel Lafond, Binhai Zhu, and Peng Zou. Computing the tandem duplication distance is NP-hard. *SIAM Journal on Discrete Mathematics*, 36(1):64–91, 2022.

[63] Wenfeng Lai, Adiesha Liyanage, Binhai Zhu, and Peng Zou. Beyond the longest letter-duplicated subsequence problem. In *33rd Annual Symposium on Combinatorial Pattern Matching*, 2022.

[64] Gad M. Landau, Jeanette P. Schmidt, and Dina Sokol. An algorithm for approximate tandem repeats. *Journal of Computational Biology*, 8(1):1–18, 2001.

[65] Gad M. Landau and Uzi Vishkin. Efficient string matching in the presence of errors. In *26th Annual Symposium on Foundations of Computer Science*, pages 126–136. IEEE, 1985.

[66] John Leech. A problem on strings of beads. *The Mathematical Gazette*, 41(338):277–278, 1957.

[67] Ivica Letunic, Richard R. Copley, and Peer Bork. Common exon duplication in animals and its role in alternative splicing. *Human Molecular Genetics*, 11(13):1561–1567, 2002.

[68] Peter Leupold, Carlos Martín-Vide, and Victor Mitrana. Uniformly bounded duplication languages. *Discrete Applied Mathematics*, 146(3):301–310, 2005.

[69] Peter Leupold, Victor Mitrana, and José M. Sempere. Formal languages arising from gene repeated duplication. In *Aspects of Molecular Computing*, pages 297–308. Springer, 2003.

[70] Brook G. Milligan, Janet N. Hampton, and Jefrey D. Palmer. Dispersed repeats and structural reorganization in subclover chloroplast DNA. *Molecular Biology and Evolution*, 6(4):355–368, 1989.

[71] Joseph H. Nadeau, Muriel T. Davisson, Donald P. Doolittle, Patricia Grant, Alan L. Hillyard, Michael R. Kosowsky, and Thomas H. Roderick. Comparative map for mice and humans. *Mammalian Genome*, 3(9):480–536, 1992.

[72] Joseph H. Nadeau and Benjamin A. Taylor. Lengths of chromosomal segments conserved since divergence of man and mouse. *Proceedings of the National Academy of Sciences*, 81(3):814–818, 1984.

[73] Cancer Genome Atlas Research Network. Integrated genomic analyses of ovarian carcinoma. *Nature*, 474(7353):609, 2011.

[74] Michael Conlon O'Donovan. A novel gene containing a trinucleotide repeat that is expanded and unstable on Huntington's disease chromosomes. *Cell*, 72(6):971–983, 1993.

[75] Layla Oesper, Anna Ritz, Sarah J. Aerni, Ryan Drebin, and Benjamin J. Raphael. Reconstructing cancer genomes from paired-end sequencing data. In *BMC Bioinformatics*, volume 13, pages 1–13. BioMed Central, 2012.

[76] Jeffrey D. Palmer and Laura A. Herbo. Unicircular structure of the brassica hirta mitochondrial genome. *Current Genetics*, 11(6):565–570, 1987.

[77] Jeffrey D. Palmer and Laura A. Herbon. Plant mitochondrial DNA evolved rapidly in structure, but slowly in sequence. *Journal of Molecular Evolution*, 28(1):87–97, 1988.

[78] Benny Porat and Ely Porat. Exact and approximate pattern matching in the streaming model. In *50th Annual IEEE Symposium on Foundations of Computer Science*, pages 315–323. IEEE, 2009.

[79] Letu Qingge, Xiaozhou He, Zhihui Liu, and Binhai Zhu. On the minimum copy number generation problem in cancer genomics. In *Proceedings of the 2018 ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics*, pages 260–269, 2018.

[80] David Sankoff. Gene and genome duplication. *Current Opinion in Genetics & Development*, 11(6):681–684, 2001.

[81] David Sankoff, Guillame Leduc, Natalie Antoine, Bruno Paquin, B. Franz Lang, and Robert Cedergren. Gene order comparisons for phylogenetic inference: evolution of the mitochondrial genome. *Proceedings of the National Academy of Sciences*, 89(14):6575–6579, 1992.

[82] Gryte Satas, Simone Zaccaria, Geoffrey Mon, and Benjamin J. Raphael. Scarlet: Single-cell tumor phylogeny inference with copy-number constrained mutation losses. *Cell Systems*, 10(4):323–332, 2020.

[83] Olivier Tremblay Savard, Denis Bertrand, and Nadia El-Mabrouk. Evolution of orthologous tandemly arrayed gene clusters. In *BMC Bioinformatics*, volume 12, pages 1–12. BioMed Central, 2011.

[84] Thomas J. Schaefer. The complexity of satisfiability problems. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, pages 216–226, 1978.

[85] Sven Schrinner, Manish Goel, Michael Wulfert, Philipp Spohr, Korbinian Schneeberger, and Gunnar W. Klau. The longest run subsequence problem. In *20th International Workshop on Algorithms in Bioinformatics*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[86] Roland F. Schwarz, Anne Trinh, Botond Sipos, James D. Brenton, Nick Goldman, and Florian Markowetz. Phylogenetic quantification of intra-tumour heterogeneity. *PLoS Computational Biology*, 10(4):e1003535, 2014.

[87] Ron Shamir, Meirav Zehavi, and Ron Zeira. A linear-time algorithm for the copy number transformation problem. In *27th Annual Symposium on Combinatorial Pattern Matching*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[88] Andrew J. Sharp et al. Segmental duplications and copy-number variation in the human genome. *The American Journal of Human Genetics*, 77(1):78–88, 2005.

[89] Jack W. Szostak and Ray Wu. Unequal crossing over in the ribosomal dna of saccharomyces cerevisiae. *Nature*, 284(5755):426–430, 1980.

[90] Craig A. Tovey. A simplified NP-complete satisfiability problem. *Discrete Applied Mathematics*, 8(1):85–89, 1984.

[91] Ming-wei Wang. On the irregularity of the duplication closure. *Bull. EATCS*, 70:162–163, 2000.

[92] Geoffrey A. Watterson, Warren J. Ewens, Thomas Eric Hall, and Alexander Morgan. The chromosome inversion problem. *Journal of Theoretical Biology*, 99(1):1–7, 1982.

[93] Ruofan Xia, Yu Lin, Jun Zhou, Tieming Geng, Bing Feng, and Jijun Tang. Phylogenetic reconstruction for copy-number evolution problems. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 16(2):694–699, 2018.

[94] Simone Zaccaria, Mohammed El-Kebir, Gunnar W. Klau, and Benjamin J. Raphael. Phylogenetic copy-number factorization of multiple tumor samples. *Journal of Computational Biology*, 25(7):689–708, 2018.

[95] I.A. Zakharov, V.S. Nikiforov, and E.V. Stepaniuk. Homology and evolution of gene orders: combinatorial measure of synteny group similarity and simulation of the evolution process. *Genetika*, 28(7):77–81, 1992.