M.Sc. Engg. Thesis

The Consensus String Matching Problem and The Diagnosis of Allelic Heterogeneity

by

Fatema Tuz Zohora

Submitted to

Department of Computer Science and Engineering

in partial fulfilment of the requirements for the degree of
Master of Science in Computer Science and Engineering

Department of Computer Science and Engineering

Bangladesh University of Engineering and Technology (BUET)

Dhaka 1000

October 2014

The thesis titled "The Consensus String Matching Problem and the Diagnosis of Allelic Heterogeneity", submitted by Fatema Tuz Zohora, Roll No. **0412052027**, Session April 2012, to the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, has been accepted as satisfactory in partial fulfillment of the requirements for the degree of Master of Science in Computer Science and Engineering and approved as to its style and contents. Examination held on October 14, 2014.

Candidate's Declaration

This is hereby declared that the work titled "The Consensus String Matching Problem and The Diagnosis of Allelic Heterogeneity" is the outcome of research carried out by me under the supervision of Dr. M. Sohel Rahman, in the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka 1000. It is also declared that this thesis or any part of it has not been submitted elsewhere for the award of any degree or diploma.

Fatema Tuz Zohora
Candidate

Acknowledgment

First of all I would like to thank my supervisor, Dr. M. Sohel Rahman, for introducing me to the amazingly interesting and diverse world of the application of consensus string matching and bioinformatics. Without his continuous supervision, guidance and advice it would not have been possible to complete this thesis. I am especially grateful to him for giving us his time whenever we needed, for his encouragement and help at times of disappointment, and always providing continuous support in our effort.

I would like to take this opportunity to thank A. S. M. Shohidul Islam for his kind assistance.

I also want to thank the other members of my thesis committee: Dr. M. Kaykobad, Dr. Mahmuda Naznin and specially the external member Dr. Md. Rezaul Karim for their valuable suggestions.

Last but not the least, I am grateful to my guardians, friends and families for their patience, support and encouragement during this period.

Abstract

The consensus problems in strings is motivated by the requirement of finding commonality of a large number of strings and has a variety of applications in Bioinformatics. This thesis presents important theoretical and algorithmic results determining the complexity class of the consensus string problem and provides a road map for diagnosing unknown genetic diseases that show *Allelic Heterogeneity*, a case where a normal gene mutates in different orders resulting in two different gene sequences causing two different genetic diseases.

In this thesis, we first show the NP-hardness of the consensus string problem under a well known mutation type, namely transposition as the distance metric. Then we propose a polynomial time algorithm for the relaxed version of the problem which determines the existence of a consensus sequence given two input sequences under the inversion and transposition metric. Our algorithm detects the existence of a common ancestor gene sequence given two input DNA sequences with theoretical worst case time complexity of $O(n^4)$ for both the non-overlapping inversion (reversed complement) metric and transposition metric. Here n is the common length of the input sequences. However, for both the inversion and transposition metric, practically the average and worst case time complexity have been found to be $O(n^2)$ and $O(n^3)$ respectively, where the worst case occurs when both input sequences have similarity of around 90%. Similarly, theoretical worst case space complexity is $O(n^3)$ for both the inversion and transposition metric, whereas it is $O(n^2)$ practically for the inversion metric. Finally, we present a pathway of detecting Allelic Heterogeneity, a challenging genetic disease, using our algorithm.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Consensus String problem is one of the fundamental problems in Stringology. In the literature, it is also known as the Center String problem or the Closest String problem. This problem can be defined as follows: given a set of k strings $S = \{s_1, \ldots, s_k\}$ and a constant d, find, if it exists, a string $s^\star$ such that the distance of $s^\star$ from each of the strings in S does not exceed d, for some suitable and meaningful definition of the term ‘distance’. This version of Consensus String problem is NP-complete. However, if we also need to find out the parameter d, then it becomes NP-hard problem. This problem has been widely studied in computational biology and combinatorial pattern matching [4, 39, 42]. This thesis presents theoretical analysis and algorithms that are at the core of several biological problems. The National Center for Biotechnology Information (NCBI 2001) defines bioinformatics as: “Bioinformatics is the field of science in which biology, computer science, and information technology merge into a single discipline”. Main motivation of this thesis comes from the applicability of the Concensus String problem in several problems of computational biology and bioinformatics. The ultimate goal of bioinformatics is to uncover the wealth of biological information hidden in the mass of sequence, structure, literature and other biological data and obtain a clearer insight into the fundamental biology of organisms and to use this information to enhance the standard of life for mankind. The field of bioinformatics is ever changing and rapidly evolving. There are at least four different specialized areas within the field of bioinformatics: acquiring of data (working with machines and equipment, sequencing DNA), storing data (typically working with databases), developing tools to analyze and visualize data (programming) and analyzing data (statistics, analysis). In this thesis, we first prove the NP-hardness of the Consensus String problem for distance metric Transposition. Then we derive an algorithm for a relaxed version of Consensus String problem that can be used in several biological problems. Here, we target a specific problem “Diagnosis of Allelic Heterogeneity, a genetic disorder”. This is a case where different mutations in the same gene results in different phenotypes which may lead to diseases with entirely different clinical features [53]. We have mapped the problem of determining allelic heterogeneity to the well known Consensus String problem by proposing an algorithm that can be used for deciding whether two input DNA sequences x and y are mutated (by non-overlapping inversions or transpositions) from the same sequence p. This is equivalent to determining the *existence* of a Concensus String ($s^\star$), given two strings x and y of length n on an alphabet of size $k = 4$ (DNA bases A, T, C, G) under the distance metric called non overlapping inversion, i.e., reversed complements and transposition. Since the minimum distance d is not present as a parameter, our problem can be thought of as a relaxed version of the original Consensus String problem.

One of the first distance metrics studied in the context of strings is the Hamming distance [36]. Subsequently, the Levenshtein edit distance [43] adds insertions and deletions to the mismatches as error possibilities. Lowrance and Wagner [46, 64] added the swap operation to the set of operations defining the distance metric. However, these distance functions assume that changes between strings occur only locally, i.e., only a small portion of the string is involved in the mutation event. However, in the biological context, evidence shows that large scale changes are also possible. For example, large pieces of DNA can be moved from one location to another (transpositions), or replaced by their reversed complements (inversions). In [17] the authors first solve approximate string matching problem under a string distance whose edit operations are transpositions of equal length adjacent factors and inversions of factors. Further works regarding these metrics were done in [32, 1, 3, 2].

The problem of Consensus String has been intensively addressed in different contexts, namely, in computational geometry [66], in combinatorial pattern matching [38, 14], and in string matching where Hamming distance, Swap Distance, and Reversal Distance have been considered. In this paper, we investigate the Consensus problem under another important metric, namely, the Transposition metric. It is known that the Consensus String problem is NP-complete for Hamming distance, even when the characters in the strings are drawn from the binary alphabet [30, 56]. Amir et al. [7] have proved that Consensus String problem is NP-complete as well for both swap metric and reversal metric. However, there is good number of recent works where several genetic algorithms [49], such as, parallel simulated annealing [45], parallel multi start algorithm [31], ant colony optimization algorithm [28], memetic algorithm [8] etc. are applied for finding the closest string. Besides these, parameterized complexity of the Concensus String problem has been discussed several times in the literature. Gramm et al. [33, 34] have shown that exact solutions for Concensus String and related problems exist for constant distance parameter (d) and constant number of strings (k). Bodlaender et al. [13] have worked on parameterized complexity of sequence alignment and Concensus String. The authors in [7] conjectured that the Consensus String problems for the interchange metric, the transposition metric, and the block interchange metric are also NP-Complete. In this thesis we partially prove the above-mentioned conjecture. In particular, we prove that the Consensus String problems for the transposition metric is NP-complete.

Genome rearrangement problems have been proven so interesting from a combinatorial point of view that the field now belongs as much to computer science as to biology. From one cell to another, from one individual to another, and from one species to another, the content of DNA molecules is often similar. The organization of these molecules, however, differs dramatically, and the mutations that affect this organization are known as genome rearrangements [29]. Multiple genome rearrangement and breakpoint phylogeny has been discussed by Sankoff et al. [54]. Since the genome rearrangement problem for reversal and transposition have been proven to be NP-hard [18, 15], several approximation solutions have been proposed for this problem. A 2-approximation algorithm for genome rearrangements by reversals and transpositions has been proposed by Gu et al. [35]. Later, 1.5 and 1.345 approximation algorithm for sorting by transpositions are proposed by Hartman et al. [37] and Elias et al. [27] respectively. Yancopoulos et al. [65] has proposed efficient sorting of genomic permutations by translocation, inversion and block interchange. Bader et al. [9] has presented a linear-time algorithm for computing only the inversion distance between signed permutations with an experimental study.

Computer Alignment of molecular sequences is widely used for biological sequence comparisons. Amir et al. [6] proposed a new pattern matching paradigm *Pattern Matching with Rearrangements* being motivated by the *Sorting by Reversals* problem [11, 18]. In general, alignment with inversions does not have a known polynomial time algorithm and a simplification to the problem considers only non-overlapping inversions. In the previous works of Schoniger et al. [55] and Vellozo et al. [63], a non-overlapping inversion occurs only in one string and transforms the string to the other string. On the other hand, the more difficult version where non overlapping inversions are allowed in both the strings simultaneously, has been introduced very recently by Cho et al. [21]. The authors in [21] have provided an $O(n^3)$ algorithm using $O(n^2)$ space, where n is the size of the two input strings, though their algorithm fails in returning the correct answers in some cases because of not tracking the prefixes of the common ancestors. In what follows, whenever we refer to the term 'inversion', we mean non-overlapping inversions.

Motivations behind this research work and our main contributions are discussed in the rest of this chapter.

1.1 Motivation Behind the Consensus String Problem

The Consensus problem in strings is motivated by the requirement of finding commonality of a large number of strings.

1. *Computational Geometry:* The problem of Concensus String has been intensively addressed in computational geometry [66] for the minimum enclosing ball problem and others [38].

2. *Stringology:* Consensus string in stringology with Hamming distance [30], Levenshtein edit distance [43], Swap Distance, and Reversal Distance [7], etc. have been considered.

3. *Bioinformatics:* Consensus string problem has a variety of applications in bioinformatics [20]. The closest string was first introduced and studied in the context bioinformatics by Lanctot et al. [41]. It has biological applications concerning finding similar regions in multiple DNA, RNA, or protein sequences. It plays an important role in many application, including universal PCR primer design [26, 41, 47, 60], genetic probe design [41], antisense drug design [41, 23], finding transcription factor binding sites in genomic data [58], determining an unbiased consensus of a protein family [10], and motif-recognition [41, 51, 52]. The closest string problem formalizes these tasks.

4. *Networking:* It also has application in web searching as a clustering aid. For example, clustering methods based on closest string via rank distance [25].

1.2 Motivation Behind the Diagnosis of Allelic Heterogeneity

Inversion and transposition are the two most common mutations that result in several interesting properties in human genome. Genetic disease is caused by gene mutation, which can be inherited through generations and can result in new sequences from a normal gene [67]. It is very interesting to know that different mutations in the same gene results in different phenotypes which may lead to diseases with entirely different clinical features [53]. This scenario is defined as *Allelic Heterogeneity*. For example, mutations in the *RET* gene have been implicated in the etiology of *Hirshprung* disease as well as *Multiple Endocrine Neoplasia (MEN) Type 2*[1].

Allelic Heterogeneity is considered to be the greatest challenge for molecular genetic diagnosis as stated in the book by Meisenberg et al. [50]. It makes the use of usual clinical diagnostic approach like allele-specific oligonucleotide probes impractical and needs different approaches like mismatch scanning, gene sequencing, linkage analysis etc., all of which are highly expensive solutions. Allelic heterogeneity motivates us with its importance in the field of medical science. It also causes autism and rigid-compulsive behaviors [57]. Very recently Castellani et al. [19] presents CFTR2, a novel approach for the clinical diagnosis of genetic disorders emphasizing specially the allelic heterogeneity[2]. But since the clinical diagnosis is extremely expensive it is worth investigating whether a tractable/polynomial time algorithm exists to detect the possibility of allelic heterogeneity.

1.3 Contributions

The main contributions of this thesis can be summarized as follows.

- We have investigated the complexity class of the Concensus String problem under the transposition metric. The Consensus String problem under the Transposition metric is proven to be NP-hard by reduction from the already proven NP-hard problem: Concensus String problem under the Swap Metric.

- We have developed polynomial time algorithms for a relaxed version of the Consensus String problem under the inversion and transposition metric. In this relaxed version we have to output the existence of closest string between two input strings.

  1. For the non overlapping inversion metric, theoretical running time of our algorithm is $O(n^4)$, whereas it is $O(n^3)$ practically, for the worst case scenario. Moreover, for the average case, our algorithm runs in $O(n^2)$ practically. Space complexity of the algorithm is $O(n^3)$.

     Cho et al. [21] have provided an $O(n^3)$ algorithm using $O(n^2)$ space (n is the size of the two input strings) for this same problem we have worked on (non overlapping inversion metric). But we have found through experimentation that

[1]http://www.jpgmonline.com/article.asp?issn=0022-3859;year=2007;volume=53;issue=4;spage=257;epage=261;aulast=Prasun

[2]http://www.irdirc.org/wp-content/uploads/2013/06/Cutting_IRDiRC_2013_Public.pdf

their algorithm fails in returning the correct answers in some cases because of not tracking the prefixes of the common ancestors. In this thesis, our presented algorithm correctly solves this problem with the same time and space complexity.

2. For non overlapping transposition metric, we have analyzed the running time for fixed length transpositions and all length transpositions. For fixed length transpositions, the running time and space complexity are $O(n^3)$ and $O(n^2)$. On the other hand, for all length transpositions, theoretical running time is $O(n^4)$ and space complexity is $O(n^3)$. However, practical running time in worst case and average case are found to be $O(n^3)$ and $O(n^2)$ respectively.

- We have presented a roadmap for a non-clinical efficient scheme to aid in the diagnosis of Allelic Heterogeneity. To this end, this is the first attempt to map the Concensus String problem to the biomedical problem of detecting the allelic heterogeneity. In particular, here we use the term *common ancestor* to indicate the same gene sequence from which different mutation order gives different gene sequences x and y. Our aim is to find the common ancestors given x and y as input, where x is the gene sequence of a known disease caused by mutation of some ancestor gene p, and y is the gene sequence of an unknown disease. If there exist common ancestors between x and y, and we find a match with p, then we diagnose that unknown disease y to be allelic heterogeneous to x. Currently available medical diagnostic techniques, such as, mismatch scanning, linkage analysis, gene sequencing, etc. all are expensive and time consuming operations. Our algorithm is not an alternative option for diagnosis of the allelic heterogeneity. Because, even if our algorithm returns YES, still medical diagnostic techniques may find those diseases as not allelic heterogeneous. But if our algorithm returns NO, then those diseases can never be allelic heterogeneous, and further medical diagnostic approach is unnecessary. So before going through such costly medical diagnostic techniques, it is better to test first if there is even any possibility of allelic heterogeneity between two diseases, using our proposed algorithms.

1.4 Organization of This Thesis

The rest of the chapters are organized as follows. In Chapter 2, we describe the basic concepts on complexity class, Concensus String, distance metrics, and genetic mutations required to understand the problems and algorithms presented in this thesis. Besides that, we present the formal definitions of the problems dealt in this thesis. In Chapter 3, we prove the NP-hardness of the Concensus String problem under the transposition metric. The algorithm developed for the relaxed version under the inversion metric and transposition metric are explained in Chapter 4 and Chapter 5 respectively, along with detailed proofs, lemmas, counter examples, and experimental analysis. Then in Chapter 6, we discuss the road map of applying the algorithm in detecting Allelic Heterogeneity, and some other applications of our algorithm. Finally, in Chapter 7, we conclude our thesis with a brief overview and future research directions.

Chapter 2

Preliminaries with Problem Definitions

This chapter presents the ideas necessary to comprehend the topics covered in this thesis. We also formally present the three problems we mainly covered in this thesis.

2.1 String

In computer programming, a string is traditionally a sequence of characters. Let Σ be a non-empty finite set of symbols (alternatively called characters), called the alphabet. A string (or word) over Σ is any finite sequence of symbols from Σ. For example, if $\Sigma = \{A, T, C, G\}$, then ATCGGAC is a string over Σ.

2.2 Distance Metrics in String Comparison

In mathematics and computer science, a string metric (also known as a string similarity metric or string distance function) is a metric that measures similarity or dissimilarity (distance) between two text strings for approximate string matching or comparison and in fuzzy string searching[1]. String metrics are used heavily in information integration and are currently used in areas including fraud detection, fingerprint analysis, DNA analysis, RNA analysis, image analysis, evidence-based machine learning, database data duplication, data mining, web interfaces, e.g. Ajax-style suggestions as you type, data integration,

[1]http://en.wikipedia.org/wiki/String_metric

and semantic knowledge integration. Some frequently used distance metrics are Hamming distance, Euclidean distance, Levenshtein distance, swap distance, etc.

2.2.1 Hamming Distance

In information theory, the Hamming distance between two strings of equal length is the number of positions at which the corresponding symbols are different [36]. In another way, it measures the minimum number of substitutions required to change one string into the other. For example,

- Hamming distance between ka**rol**in and ka**thr**in is 3.
- Hamming distance between 10**1**1**1**01 and 10**0**1**0**01 is 2

2.2.2 Euclidean distance

In mathematics, the Euclidean distance or Euclidean metric is the 'ordinary' distance between two points that one would measure with a ruler, and is given by the Pythagorean formula[2] [24]. In Cartesian coordinates, if $p = (p_1, p_2, \ldots, p_n)$ and $q = (q_1, q_2, \ldots, q_n)$ are two points in Euclidean n-space, then the distance (d) from p to q, or from q to p is given by, $d(p,q) = d(q,p) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2, \ldots, (q_n - p_n)^2}$.

2.2.3 Levenshtein distance

In information theory and computer science, the Levenshtein distance is a string metric for measuring the difference between two sequences [44]. Informally, the Levenshtein distance between two words is the minimum number of single-character edits (i.e. insertions, deletions or substitutions) required to change one word into the other. For example, the Levenshtein distance between kitten and sitting is 3, since the following three edits change one into the other, and there is no way to do it with fewer than three edits:

- **k**itten to **s**itten (substitution of 's' for 'k')
- sitt**e**n to sitt**i**n (substitution of 'i' for 'e')
- sittin to sittin**g** (insertion of 'g' at the end)

[2]http://www.encyclopediaofmath.org/index.php?title=Pythagoras_theorem&oldid=19490

Biological mutation operation, e.g. inversion, reversal, transposition etc. (defined later in Section 2.5, 2.5.1, 2.5.2) are also considered as distance metric in bioinformatics for sequence analysis.

2.3 Complexity Class

In computational complexity theory, a complexity class is a set of problems that can be solved by an abstract machine M using $O(f(n))$ of resource R, where n is the size of the input [22]. We will discuss four major complexity classes P, NP, NP-complete (NPC), and NP-hard, since these classes are related with the thesis. In this thesis we prove NP-Completeness of the Concensus String problem under the transposition metric in Chapter 3.

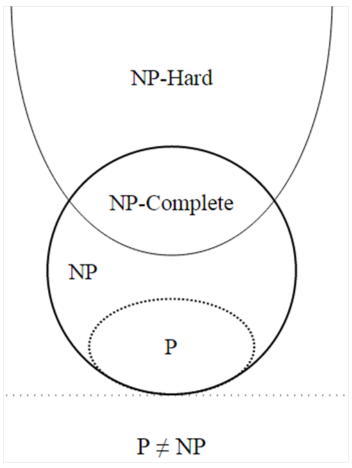


Figure 2.1: Hierarchy of complexity classes
3

- P Class: In computational complexity theory, P, also known as PTIME or DTIME ($O(1)$), is one of the most fundamental complexity classes. It contains all decision problems that can be solved by a deterministic Turing machine using a polynomial amount of computation time, or polynomial time.
- NP Class: NP is the set of decision problems where the 'yes' instances can be accepted in polynomial time by a non-deterministic Turing machine.
- NP-hard: In computational complexity theory, Non-deterministic Polynomial-time hard (NP-hard) is a class of problems that are, at least as hard as the hardest problems in NP. Formally, a problem H is NP-hard when every problem L in NP can be reduced in polynomial time to H [62].

- NPC Class: A decision problem C is NP-complete if,
    1. C is in NP, and
    2. Every problem in NP is reducible to C in polynomial time [61].

  Although any given solution to an NP-complete problem can be verified quickly (in polynomial time), there is no known efficient way to locate a solution.

2.4 Consensus String Problem

Given a set of strings $S = s_1, ..., s_N$ and a constant d, it finds, if exists, a string s^* such that the distance of s^* from each of the strings in S does not exceed d, for some suitable and meaningful definition of the term distance. Also known as Closest String problem in literature. Please refer to the Figure 2.2 for an illustration.

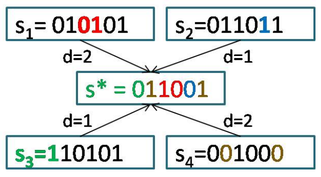


Figure 2.2: $s^\star = 011001$ is the Concensus String of $S = \{s_1, s_2, s_3, s_4\}$, under the Hamming distance, with minimum distance, $d \leq 2$

This version of the Concensus String problem is NP-complete. However, if we also need to find out the parameter d, then it becomes NP-hard problem.

2.5 Mutation

In genetics, a mutation is a change of the nucleotide sequence of the genome of an organism, virus, or extra chromosomal genetic element. Mutations result from errors in the process of replication, from the insertion or deletion of segments of DNA by mobile genetic elements, or from unrepaired damage to DNA or to RNA genomes [12, 5, 16]. Large-scale mutations in chromosomal structure includes deletion, chromosomal translocation, chromosomal inversion etc.

2.5.1 Inversions

Inversion is a chromosomal rearrangement in which a segment of a chromosome is reversed and complemented. For an illustration please refer to the Figure 2.3.

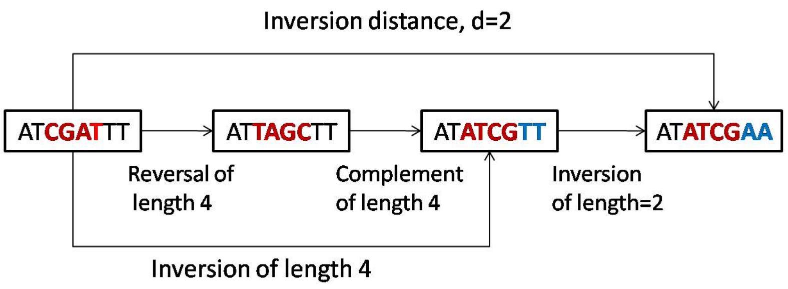
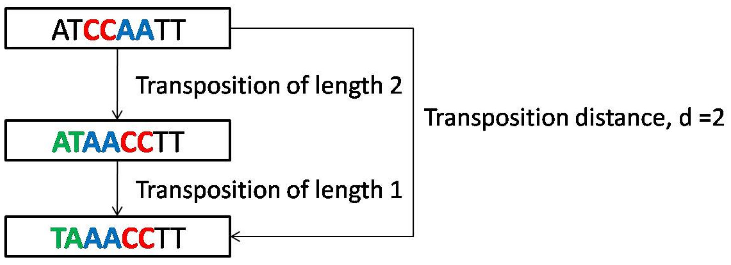


Figure 2.3: Illustration of an inversion operation

In this example, we consider a DNA sequence $x = ATCGATTT$. In a DNA sequence, $A-T$ are complemented base and $C-G$ are complemented base. First inversion of length 4, at index 3 to 6 over x converts it into $x' = AT\underline{TAGC}TT$. Then again, another inversion of length 2, at index 7 of x' converts it into $x'' = ATATCG\underline{AA}$. Therefore, two inversion operations (first one is of length 4 and second one is of length 2) converts x into x''. Thus, the inversion distance between x and x'' is 2.

2.5.2 Transpositions

Transposition is a genetic mutation in which two chromosomal segments of the same size (on the same or different chromosomes) interchange their positions. For an illustration please refer to the Figure 2.4.



ATCCAATT
Transposition of length 2
ATAACCTT
Transposition of length 1
TAAACCTT
Transposition distance, d =2

Figure 2.4: Illustration of a transposition operation

Here. we consider a DNA sequence $x = AT\underline{CCAA}TT$. First transposition of length 2 at index 3 to 6, swaps the position of two blocks CC (length 2) and AA (length 2). It results in $x' = AT\underline{AACC}TT$. Then again, another transposition of length 1, at index 1 to 2, swaps two one sized block A and T, and results in $x'' = \underline{TA}AACCTT$. Thus, two transposition operations (first one is of length 2 and second one is of length 1) converts x into x''. So the transposition distance between x and x'' is 2.

2.5.3 Allelic Heterogeneity

Allelic Heterogeneity is a genetic disease where different mutations in the same gene result in different phenotypes which may lead to diseases with entirely different clinical features [59]. For an illustration please refer to the Figure 2.5. Here, the gene sequence $P = ATTCGCGGTACAG$ is mutated in different order in disease 1 and disease 2. In disease 1, a transposition over x at index 3 to 6 (block $TCGC$) takes place. Also an inversion mutation at index 11 to 13 (block CAG) occurs. These two operations results in gene sequence $X = AT\underline{CGAG}GGTA\underline{CTG}$, which cause disease 1. On the other hand, a transposition operation at index 3 to 6 and an inversion operation at index 7 to 10 over x results in gene sequence $Y = AT\underline{CGAG}\ \underline{TACC}CAG$, causing disease 2. So here disease 1 and disease 2 are called allelic heterogeneous with each other, since both of them are resulted from same parent gene sequence P.

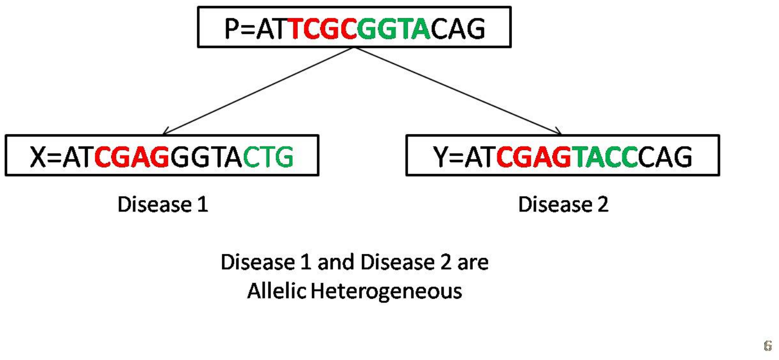


Figure 2.5: Illustration of Allelic Heterogeneity

Some real life examples of allelic heterogeneity are collected from the Journal of Postgraduate Medicine and reported in Table 2.1[4].

We end this section with a formal definition of the problem we handle in this thesis.

2.5.4 Formal Definitions of The Target Problems

Problem 1. *Determining the complexity class of Concensus String under the distance metric or distance function: transposition.*

[4][http://www.jpgmonline.com/viewimage.asp?img=jpgm_2007_53_4_257_33968_1.jpg]

Table 2.1: Examples of Allelic Heterogeneity

Hurler syndrome	IDUA	Scheie syndrome
Charcot-marie-tooth neuropathy	PMP22	Hereditary neuropathy with pressure palsy
Hyperkalemic periodic paralysis	SCN4A	Paramyotonia congenita
Creutzfeldt - Jacob disease	PRNP	Familial fatal insomnia
Pseudohypoparathyroidism IA	GNAS1	Albright hereditary osteodystrophy
Kennedy disease	AR	Androgen insensitivity
Cystic fibrosis	CFTR	Congenital bilateral absense of vas deferens
Duchenne muscular dystrophy	DMD	Becker muscular dystrophy
Hirschprung disease	RET	Multiple endorcrine neoplasia Type 2

Problem 2. *Determining the existence of a Concensus String ($s*$), given two strings x and y of length n on an alphabet of size $k = 4$ under the distance metric called non overlapping inversion (i.e., reversed complements) and transposition. Since the minimum distance d is not present as a parameter, it can be thought of as a relaxed version of the original Concensus String problem.*

Problem 3. *Present a road map for the application of our algorithm in detecting the genetic disease Allelic Heterogeneity. For this purpose, the goal is to find the common ancestors given two DNA sequences x and y as input, where x is the gene sequence of a known disease caused by mutation of some ancestor gene p, and y is the gene sequence of an unknown disease. If there exist common ancestors between x and y, and we find a match with p, then we can diagnose that unknown disease y to be an allelic heterogeneous to x.*

The Problem 2 and Problem 3 are almost identical. This is illustrated in Figure 2.6.

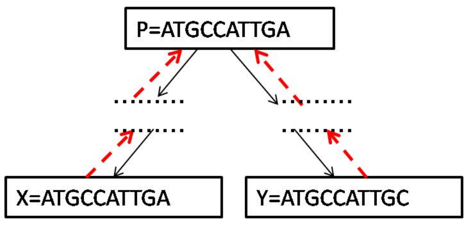


Figure 2.6: Mapping the Concensus String problem to the diagnosis of Allelic Heterogeneity

In allelic heterogeneity, a perfect gene p is mutated in different order giving x and y two defective genes, shown by solid arrows. On the other hand, having x and y, going on reversed direction (shown by dotted arrow), we can get p as a Concensus String of x and y with mutations as distance metric. We denote p as the common ancestor gene of x and y.

- Among the Concensus Strings of x and y, if a match with p is found *the test is positive.* Then we can perform additional clinical diagnostic approaches to validate the positive output.

- If no match is found, *the test is negative.* Then no need of performing expensive clinical diagnostic tests, which saves huge energy and costs.

For Problem 3, we use the algorithm developed in Problem 2, but with some additional steps for detecting the disease.

2.6 Conclusion

In this chapter, we have presented the preliminaries required to understand the following chapters. We also formally have defined the problem we handle in this thesis. In the next chapters, we will formally present our proofs for the complexity class of Concensus String problem under transposition metric and present the algorithms for the relaxed version.

Chapter 3

NP-Hardness of The Consensus String Problem Under the Transposition Metric

In this chapter, we present the proofs of NP-hardness of the Concensus String problem under the transposition metrics. This chapter starts with some primary definitions related with our works, and then we go for the formal proofs.

3.1 Notations and Definitions

Let U be a set, U_ℓ be the Cartesian product of U with itself ℓ times, and $dist : U_\ell \times U_\ell \to \mathcal{R}$ be a distance function. Let $S \subseteq U_\ell$ and let $d \in \mathcal{R}$.

Then $s_d \in U_\ell$ is a distance d consensus of S (alternately a radius r center of S, or a center of the radius d ball enclosing S), if $dist(s, s_d) \leq d \forall s \in S$.

The Consensus String problem has as its input a set $S \subseteq U_\ell$ and as its output the minimum $d \in \mathcal{R}$ for which there exists an $s^* \in U_\ell$ where s^* is a distance d consensus of S. We call s^* a consensus or center of S.

Let $s = s[1] \ldots s[\ell]$ be a string over alphabet Σ. A swap permutation for s is a permutation $\Pi : 1, \ldots, \ell \to 1, \ldots, \ell$ such that

1. if $\Pi(i) = j$ then $\Pi(j) = i$ (characters are swapped);

2. for all i, $\Pi(i) \in i-1, i, i+1$ (only adjacent characters are swapped);

3. if $\Pi(i) \neq i$ then $s[\Pi(i)] \neq s[i]$ (identical characters are not swapped).

For a given string $s = s[1] \dots s[\ell]$ and a swap permutation Π for s we use $\Pi(s) = s[\Pi(1)]s[\Pi(2)]...s[\Pi(\ell)]$. We call $\Pi(s)$ a swapped version of s. The number of swaps in swapped version $\Pi(s)$ of s is the number of pairs $(i, i+1)$ where $\Pi(i) = i+1$ and $\Pi(i+1) = i$. For strings $p = p[1] \dots p[\ell]$ and $t = t[1] \dots t[\ell]$, we say that p swap matches t if t is a swapped version of p. It is not difficult to see that if p swap matches t then there is a unique swap permutation which converts p into t. The number of swaps in that swap permutation is the *swap distance* of p and t. Consensus string problem under the swap distance is NP-hard [7].

Given two strings p and t, the mutation distance $md(p, t)$ is based on the following edit operation:

1. Transposition: A factor of the form ZW is transformed into WZ, provided that $|Z| = |W| > 0$. The transposition size in this case is said to be $|W| = |Z|$. For example, transposition of size 3 at index 3 of the binary string $s =$0 1 $\underline{1\ 1\ 0}$ $\underline{0\ 0\ 0}$ 1 1 is 0 1 $\underline{0\ 0\ 0}$ $\underline{1\ 1\ 0}$ 1 1.

Each of the operations above is assigned unit cost.

There are strings p, t such that p can not be converted into t by any sequence of transpositions, in which case $md(p, t) = \infty$. When $md(p, t) < \infty$, we say that p and t have an md-match. If only transpositions with fixed size ($|Z| = k$) is allowed then p and t is said to have a *fixed-length transposition match* or *flt-match*, for short. In this case, the mutation distance is the *flt-distance* of p and t.

3.2 The Concensus String problem under the transposition metric

In this section, we show that the Consensus String problem under the Fixed-Length Transposition distance (CSFLT) for binary alphabet is NP-hard by reduction from Consensus String problem under the Swap distance (CSS) for binary alphabet.

The CSS problem: *Instance*: A finite alphabet Σ, a finite set $S \subseteq U_\ell$ of strings over Σ with $|S| = K$, and a positive integer swap distance $d \in \mathcal{R}$.

Question: find the string $s^* \in U_\ell$ where s^* is a distance d consensus of S.

The CSS problem is NP-hard even if $\Sigma = \{0, 1\}$ [7]. We assume that $\Sigma = \{0, 1\}$ and

$|S| = K$.

The CSFLT problem: *Instance*: A finite alphabet $\Sigma^{'}$, a finite set $S^{'} \subseteq U_{\ell}^{'}$ of strings over $\Sigma^{'}$ with $|S^{'}| = K'$, a positive integer fixed-length transposition distance $d^{'} \in \mathcal{R}^{'}$, and positive integer fixed-length transposition size k.
Question: find the string $s^{'*} \in U_{\ell}^{'}$ where $s^{'*}$ is a distance d consensus of $S^{'}$.
Now we transform an instance of the CSS problem to an instance of CSFLT problem according to following rules.

1. $\Sigma^{'} \in \{\alpha^k, \beta^k\}$ can be found from Σ such that,
   (a) $\alpha^k \leftarrow 0$, (each 0 is encoded by a block of k consecutive α).
   (b) $\beta^k \leftarrow 1$, (each 1 is encoded by a block of k consecutive β).

2. Each string $s_j^{'} \in U_l^{'}$ can be found from a string $s_j \in U_l$ such that,
   (a) $|s_j^{'}| = k\ell$, where $|s_j| = \ell$.
   (b) Mapping between the symbols of $s_j^{'}$ and s_j is such that, $s_j^{'}[k \times (i-1)+n] = s_j[i]$, where $i = 1, 2, \ldots \ell$ and $1 \leq n \leq k$.
   (c) Starting of each block (of k consecutive α or β) of $s^{'}$ is recorded in an array *strt_indx* of size ℓ such that, the i^{th} block starts at index *strt_indx*$[i] = k \times (i-1) + 1$ in $s^{'}$, where $1 \leq i \leq \ell$. That is, the i^{th} symbol of s_j is replaced by the i^{th} block of $s_j^{'}$ starting at $s_j^{'}[$*strt_indx*$[i]]$.

3. A swap operation over s_j at index i is transformed to a transposition operation over $s_j^{'}$ of size k starting at index *strt_indx*$[i] = k \times (i-1) + 1$.

4. Starting index of any transposition operation over $s_j^{'}$ is to be picked up from *strt_index* array.

Theorem 1. *The CSFLT problem for a binary alphabet is NP-hard.*

Proof. (if part:) Given a binary string s_j, each character 0 is encoded by k consecutive α and each character 1 is encoded by k consecutive β (by transformation rule 1 and 2) in $s_j^{'}$ and corresponding *strt_indx* array is formed in linear time (by transformation rule 1 and 2). Therefore, swap of two characters in s_j, can be effectively transformed to transposition of size k in $s_j^{'}$. That is, swapping of $s_j[i]$ and $s_j[i+1]$ is transformed to a transposition operation in $s_j^{'}$ which interchanges the two k sized blocks $s_j^{'}[strt\_indx[i]]$ and $s_j^{'}[strt\_indx[i+1]]$ (by transformation rule 3). Clearly, a solution for the new input of the fixed-length transposition consensus problem (CSFLT) comes from a solution of the swap consensus problem (CSS) with the original input, since a swap of 01 to 10 or vice versa in the original input is equivalent to a transposition that changes $\alpha_k\beta_k$ (two k sized blocks giving total $2k$ characters) to $\beta_k\alpha_k$ or vice versa, and starting index of these characters in the new input can be found by transformation rule 2. Please refer to Example 1 for an illustration.

(only if part:) We have $s_j^{'} \in U_l^{'}$ such that each $s_j^{'}$ consists of a block of k consecutive α or β, where k is the fixed size of transposition operation. Let z be the number of blocks in $s_j^{'}$. During initializing $s_j^{'}$, starting of each k sized block is stored in $strt\_indx$ array in linear time where $|strt\_indx| = z$. By transformation rule 1, this $s_j^{'} \in U_l^{'}$ can be transformed to a $s_j \in U_l$ by replacing each k sized block of α by a single 0 and each k sized block of β by a single 1. So the i^{th} block of s'_j starting at index $strt\_index[i] = j$, maps to the i^{th} character in s_j (by transformation rule 2). Any arbitrary transposition operation over $s_j^{'}$ should start at some index j picked up from $strt\_indx$ array (by transformation rule 4). For example, let us apply transposition at index $strt\_indx[i]$. Then it actually interchanges the blocks $s_j^{'}[strt\_indx[i]]$ and $s_j^{'}[strt\_indx[i+1]]$. This transposition can be transformed to swapping of $s_j[i]$ and $s_j[i+1]$ (by transformation rule 3). So now we can say that transposition of size k (interchange of two adjacent k sized blocks, that is $2k$ characters) in $s_j^{'}$ is effectively a swap of two consecutive characters in s_j. Clearly, a solution for the original input of the fixed-length transposition consensus (CSFLT) problem can be transformed to a solution of the swap consensus problem (CSS) with the new input, since the transpositions in the original input that change $\alpha_k\beta_k$ (total $2k$ characters) to $\beta_k\alpha_k$ or vice versa, can be reduced to only a possible swap of 01 to 10 or vice versa in the new input such that the character at the i'^{th} index of $s_j^{'}$ is mapped to the character at index $i = \lceil \frac{i'}{k} \rceil$ in s_j (by transformation rule 2). Please refer to Example 2 for an illustration. □

Example 1. In this example we illustrate the *if part* of the reduction explained in Theorem 1. Consider the Consensus problem under the Swap distance with $K = 3$ and $s_1 = 0000$, $s_2 = 0110$ and $s_3 = 1100$. The reduction to a fixed-length transposition instance (with transposition size $k = 2$) is as follows:
$s_1 = 0110 \rightarrow \alpha\alpha\beta\beta\beta\beta\alpha\alpha = s_1^{'}$,
$s_2 = 1001 \rightarrow \beta\beta\alpha\alpha\alpha\alpha\beta\beta = s_2^{'}$,
$s_3 = 1100 \rightarrow \beta\beta\beta\beta\alpha\alpha\alpha\alpha = s_3^{'}$.

The string $s^* = 1010$ is a Concensus String under the Swap distance for $\{s_1, s_2, s_3\}$, since $Swap(1010, 0110)$=$Swap(1010, 1001)$=$Swap(1010, 1100)$=1.

$s^{'*} = \beta\beta\alpha\alpha\beta\beta\alpha\alpha$ is the Concensus String of $\{s_1^{'}, s_2^{'}, s_3^{'}\}$ since transposition at index 1 (interchanging blocks at index 1 and index 3) over $s_1^{'}$ makes it equal to $s^{'*}$, transposition at index 5 (interchanging blocks at index 5 and index 7) over $s_2^{'}$ makes it equal to $s^{'*}$, and transposition at index 3 (interchanging blocks at index 3 and index 5) over $s_3^{'}$ makes it equal to $s^{'*}$.

Example 2. In this example we illustrate the *only if part* of the reduction explained in Theorem 1. Consider the Consensus problem under the Transposition distance with $K = 3$ and $s_1 = 0000$, $s_2 = 0110$ and $s_3 = 1100$ and fixed transposition size $k = 2$. The reduction to a Swap instance is as follows:
$s_1^{'} = \alpha\alpha\beta\beta\beta\beta\alpha\alpha \rightarrow 0110 = s_1$,
$s_2^{'} = \beta\beta\alpha\alpha\alpha\alpha\beta\beta \rightarrow 1001 = s_2$,
$s_3^{'} = \beta\beta\beta\beta\alpha\alpha\alpha\alpha \rightarrow 1100 = s_3$.

Starting positions of the transpositions over $s_j^{'}$ should be picked up from strt_indx. The $s^{'*} = \beta\beta\alpha\alpha\beta\beta\alpha\alpha$ is the Concensus String of $\{s_1^{'}, s_2^{'}, s_3^{'}\}$ since transposition at index $strt\_indx[1] = 1$ (interchanging blocks at index $strt\_indx[1] = 1$ and index strt_indx[2]=3) over $s_1^{'}$ makes it equal to $s^{'*}$, transposition at index $strt\_indx[3] = 5$ (interchanging blocks at index $strt\_indx[3] = 5$ and index strt_indx[4]=7) over $s_2^{'}$ makes it equal to $s^{'*}$, and transposition at index $strt\_indx[2] = 3$ (interchanging blocks at index $strt\_indx[2] = 3$ and index strt_indx[3]=5) over $s_3^{'}$ makes it equal to $s^{'*}$.

The string $s^* = 1010$ is a Concensus String under the Swap distance for $\{s_1, s_2, s_3\}$, since $Swap(1010, 0110)$=$Swap(1010, 1001)$=$Swap(1010, 1100)$=1.

The following theorem readily follows from Theorem 1.

Theorem 2. *The Consensus String problem for the transposition metric is NP-hard.*

Proof. Clearly, the CSFLT problem for a binary alphabet is a restricted version of the Consensus String problem for the transposition metric. The solution space for general transposition problem is definitely larger than the fixed length transposition since in general transposition, same segment can be transposed multiple times. Therefore, solution space for the Concensus String problem for the transposition metric is larger than that of the fixed length transposition. So, by the method of proof by restriction the result follows from Theorem 1. □

3.3 Conclusion

Finding a Concensus String from a given set of strings is a hard and challenging problem. In this chapter we have proved that the Consensus String problem is NP-hard for the transposition metric even for a binary alphabet. Future research endeavor could be directed towards further investigation of other aspects from computational complexity, such as approximation and fixed parameter complexity for the Concensus String problem under transposition and inversion metric.

Chapter 4

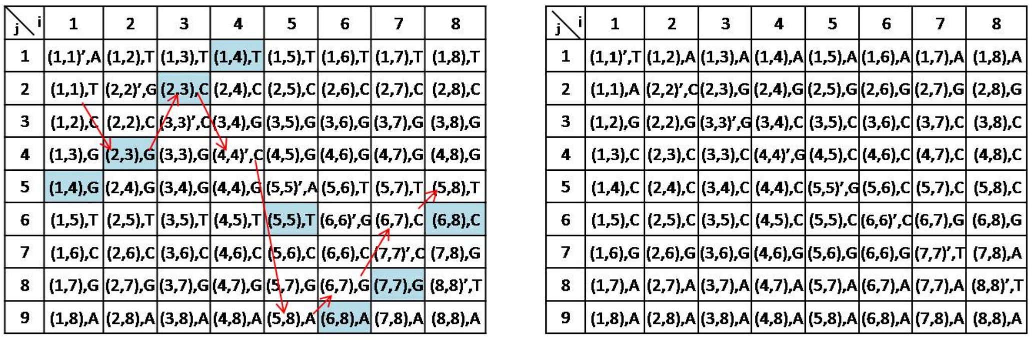
Existence of Consensus String Under The Inversion Metric

In this chapter, we present a polynomial time algorithm for determining the *existence* of a consensus string ($s^\star$), given two strings x and y of length n on an alphabet of size $k = 4$ (DNA bases A, T, C, G) under the distance metric called *non overlapping inversion*, i.e., reversed complements. Since the minimum distance d is not present as a parameter, our problem can be thought of as a relaxed version of the original consensus string problem. In Section 4.1, we provide some definitions and observations necessary for presenting the algorithm. Then in Section 4.3 we discuss the main algorithm. We prove the correctness of our algorithm in Section 4.4. In Section 4.5 and Section 4.6, we discuss the time and space complexity respectively. We show the experimental result in Section 4.7. Finally we conclude in Section 4.8 discussing some future research directions.

4.1 Definitions

We consider the biological operation *Inversion* which is the *reverse* and *complement* of a DNA sequence x. *Inversion Sequence*, θ is defined as a set of the non overlapping inversions. So the *Inversed Sequence*, $\theta(x)$ is the resultant DNA sequence after applying the set of inversions, θ over x. Suppose $x = \mathbf{AG}GC$ is a DNA sequence and $\theta' = \{(1,2),(4,4)\}$, then $\theta'(x) = CTGG$. Again, $\theta'(x)$ upto index 3 is CTG.

Given two sequences x and y of length n, we follow the notation of [21] and use $T_x[n{+}1][n]$ and $T_y[n+1][n]$ to denote the sets of all possible inversions of x and y respectively. In Figure 4.1, each $T_x[j][i]$ is called an *Inversion fragment*; it represents a tuple $\langle (p,q), \alpha \rangle$



j \ i	1	2	3	4	5	6	7	8
1	(1,1)′,A	(1,2),T	(1,3),T	(1,4),T	(1,5),T	(1,6),T	(1,7),T	(1,8),T
2	(1,1),T	(2,2)′,G	(2,3),C	(2,4),C	(2,5),C	(2,6),C	(2,7),C	(2,8),C
3	(1,2),C	(2,2),C	(3,3)′,C	(3,4),G	(3,5),G	(3,6),G	(3,7),G	(3,8),G
4	(1,3),G	(2,3),G	(3,3),G	(4,4)′,C	(4,5),G	(4,6),G	(4,7),G	(4,8),G
5	(1,4),G	(2,4),G	(3,4),G	(4,4),G	(5,5)′,A	(5,6),T	(5,7),T	(5,8),T
6	(1,5),T	(2,5),T	(3,5),T	(4,5),T	(5,5),T	(6,6)′,G	(6,7),C	(6,8),C
7	(1,6),C	(2,6),C	(3,6),C	(4,6),C	(5,6),C	(6,6),C	(7,7)′,C	(7,8),G
8	(1,7),G	(2,7),G	(3,7),G	(4,7),G	(5,7),G	(6,7),G	(7,7),G	(8,8)′,T
9	(1,8),A	(2,8),A	(3,8),A	(4,8),A	(5,8),A	(6,8),A	(7,8),A	(8,8),A

(a)

j \ i	1	2	3	4	5	6	7	8
1	(1,1)′,T	(1,2),A	(1,3),A	(1,4),A	(1,5),A	(1,6),A	(1,7),A	(1,8),A
2	(1,1),A	(2,2)′,C	(2,3),G	(2,4),G	(2,5),G	(2,6),G	(2,7),G	(2,8),G
3	(1,2),G	(2,2),G	(3,3)′,G	(3,4),C	(3,5),C	(3,6),C	(3,7),C	(3,8),C
4	(1,3),C	(2,3),C	(3,3),C	(4,4)′,G	(4,5),C	(4,6),C	(4,7),C	(4,8),C
5	(1,4),C	(2,4),C	(3,4),C	(4,4),C	(5,5)′,G	(5,6),C	(5,7),C	(5,8),C
6	(1,5),C	(2,5),C	(3,5),C	(4,5),C	(5,5),C	(6,6)′,C	(6,7),G	(6,8),G
7	(1,6),G	(2,6),G	(3,6),G	(4,6),G	(5,6),G	(6,6),G	(7,7)′,T	(7,8),A
8	(1,7),A	(2,7),A	(3,7),A	(4,7),A	(5,7),A	(6,7),A	(7,7),A	(8,8)′,T
9	(1,8),A	(2,8),A	(3,8),A	(4,8),A	(5,8),A	(6,8),A	(7,8),A	(8,8),A

(b)

Figure 4.1: (i)$T_x[][]$ for $x = AGCCAGCT$; (ii)$T_y[][]$ for $y = TCGGGCTT$

where α is the base, $\alpha \in \{A, C, T, G\}$ yielded at index i after applying the inversion (p, q) over x according to the following equation:

$$\langle (p,q), \alpha \rangle = \begin{cases} \langle (i,i)', x[i] \rangle & \text{if } j = i \text{ (no inversion at index } i\text{)} \\ \langle (i, j-1), \theta(x[j-1]) \rangle & \text{if } j > i \\ \langle (j,i), \theta(x[j]) \rangle & \text{if } j < i \end{cases} \tag{4.1}$$

The $\theta(x)$ can be constructed by connecting the inversion fragments in a path specified by θ and concatenating their yielded base letters in that order. In Figure 4.1, for a given $\theta' = \{(1,4),(5,5),(6,8)\}$, the $\theta'(x) = GGCTTAGC$ is presented by the path shown by shaded cells in T_x. The $\theta'(x)$ up to index $i = 3$ is GGC. Note that the same inversion fragment can belong to different inversion sequences and thus can present different inversed sequences. For θ', $T_x[4][2]$ presents a fragment that belongs to the inversion $(1,4)$ according to the path: $T_x[5][1] \rightarrow \mathbf{T_x[4][2]} \rightarrow T_x[2][3] \rightarrow T_x[1][4] \equiv \langle (1,4), G \rangle \rightarrow \langle (\mathbf{2},\mathbf{3}), \mathbf{G} \rangle \rightarrow \langle (2,3), C \rangle \rightarrow \langle (1,4), T \rangle$. For $\theta'' = \{(1,1),(\mathbf{2},\mathbf{3}),(4,4)',(5,8)\}$, the same fragment belongs to the inversion $(2,3)$ according to the path: $\mathbf{T_x[4][2]} \rightarrow T_x[2][3] \equiv \langle (\mathbf{2},\mathbf{3}), \mathbf{G} \rangle \rightarrow \langle (2,3), C \rangle$. Clearly, $\theta'(x)$ and $\theta''(x)$ are two different inversed sequences of the same DNA sequence x. Note that, for a fixed θ, there is only one choice as we move from i to $i+1$. For example, only one path, i.e., one inversed sequence can be derived for θ' (shaded cells) and θ'' (arrow). In this way we can generate all possible inversed sequences of x (though not necessary for our problem). In what follows for the sake of notational ease we will drop x or y from $T[][]$ when it is clear from the context.

Two inversion fragments $T[j'][i] = \langle (p_1, p_2), \alpha_1 \rangle$ and $T[j''][i+1] = \langle (q_1, q_2), \alpha_2 \rangle$ are called

Agreed Fragments if one of the following two conditions holds.

Condition 1. $p_1 + p_2 = q_1 + q_2$ *and* $j' > j''$*: As an example, in Figure 4.1, this condition holds when we move from* $T_x[5][1] \rightarrow T_x[4][2] \equiv \langle(1,4), G\rangle \rightarrow \langle(2,3), G\rangle$ *for the inversion* $(1,4)$*.*

Condition 2. $q_1 = p_2 + 1$ *and* $j'' \geq j'$*: As an example, in Figure 4.1, this condition holds when we move from* $T_x[1][4] \rightarrow T_x[6][5] \equiv \langle(1,4), T\rangle \rightarrow \langle(5,5), T\rangle$*, i.e., the inversion* $(1,4)$ *finishes and next inversion* $(5,5)$ *starts.*

Otherwise, we call them disagreed fragments. For example, $T_x[5][1] \equiv \langle(1,4), G\rangle$ and $T_x[5][2] \equiv \langle(2,4), G\rangle$ are disagreed as none of the conditions holds. Again, for two pairs of agreed fragments $(\langle(p_1,p_2),\alpha_1\rangle, \langle(q_1,q_2),\alpha_2\rangle)$ and $(\langle(q_1,q_2),\alpha_2\rangle, \langle(r_1,r_2),\alpha_3\rangle)$, we say these two pairs are connected by $\langle(q_1,q_2),\alpha_2\rangle$ and thus all these three inversion fragments are agreed fragments. The *Agreed Sequence* is formed by taking an *inversion fragment* from each column $i = 1, 2, \ldots, n$, such that, for any two consecutive fragments $T[j'][i] = \langle(p_1,p_2),\alpha_1\rangle$ and $T[j''][i+1] = \langle(q_1,q_2),\alpha_2\rangle$, they are *agreed fragments.* So an *agreed sequence* actually presents an *inversed sequence*, $\theta(x)$. In an *agreed sequence*, we have the following two cases.

Case 1 - Upward movement at index i. It happens in an *agreed sequence* at index i when, $T[j'][i]$ and $T[j''][i+1]$ are agreed fragments based on Condition 1. It implies that an inversion (p,q) is continuing from some index $i' \leq i$. Inversion fragment $T[j'][i]$ involved in such a scenario is called a *continuing_inversion fragment* for the corresponding θ'. In Figure 4.1, for θ', $T_x[4][2]$ is following Case 1 for the inversion $(1,4)$, and thus is a continuing inversion fragment.

Case 2 - Horizontal or Downward movement at index i. This happens when $T[j'][i]$ and $T[j''][i+1]$ constitute an agreed fragments based on Condition 2. It implies that an inversion (p,q) has started at some index $i' \leq i$, ends at index i (having $j' = p = i'$ and $j'' = q = i$), and next inversion starts at index $i+1$. Involved $T[j'][i]$, is called the *ending_inversion fragment* for the corresponding θ. In Figure 4.1, for θ', $T_x[1][4]$ belongs to Case 2 for the inversion $(1,4)$ and thus is the ending inversion fragment.

Observation 1. *In Table* $T[][]$*, an inversion* (p,q) *starts at inversion fragment* $T[q+1][p]$*, continue moving upward up to* $T[p][q]$*, and then it moves horizontal or downward indicating no change (i.e.,* $(p,p)'$*) or start of a new inversion.*

The term $Pair(t, r)$ is defined for any inversion fragment $T[j][i]$, where t is the starting index $i' \leq i$ of the last inversion in the inversion sequence θ it belongs to, and r is the current row j. If the same inversion fragment belongs to multiple inversion sequences then multiple $Pair(t, r)$ exist for it. In such a case the value of t would be different for different inversion sequences. *Pairs* corresponding to the agreed fragments are called *Agreed Pairs*. Similarly, a *Pair* corresponding to *continuing_inversion fragment* is called *continuing_inversion Pair* which has $t \neq -1$ and $t \leq i$. Also the *ending_inversion Pair* is defined for an *ending_inversion fragment* and has $t = -1$; in this case r gives the starting index of the last inversion (by Observation 1). We define another important set, *cont_inv_i'*, containing only the *continuing_inversion Pairs* presenting inversions that started at index i' and still exist as *continuing_inversion Pair*, at index i, $i \geq i'$. Subset of any *cont_inv_i'* is denoted as *cont_inv*, that contains single or multiple *continuing_inversion Pairs* (each having the same value for t) presenting all those inversions which produce the same prefix from index $t \leq i$, up to i.

We define S_x and S_y to be the sets of all possible inversion sets θ over x and y respectively. In general, $\theta_x \in S_x$ and $\theta_y \in S_y$ are used to present the matching phase. Deciding whether any consensus sequence exists between two given DNA sequences x and y having the same length n, involves finding out the existence of common agreed sequences of x and y. For this purpose we track the matched pairs between $T_x[n+1][n]$ and $T_y[n+1][n]$ for each index or column $i = 1, 2, \ldots, n$. For the same index i, if an inversion fragment in T_x mapped by the Pair (t', r') and another in T_y mapped by Pair (t'', r''), yield the same α, and the respective inversed sequences $\theta_x(x)$ and $\theta_y(y)$ up to i is the same, then those two pairs are called *Matched Pairs* and corresponding inversion fragments are called *Matched Fragments*. The matched pairs are denoted as $\langle Xsibling \rangle$ - $\langle Ysibling \rangle$ for the ease of representation. Both of $Xsibling$ and $Ysibling$ may contain one or more Pairs. In the rest of the section, we define some table like data structures that will be used in our algorithm. Each table will record some information of the matched pairs and will be named based on the type of $\theta(x)$ at each column i. Column i of each table presents some alignment of $\theta_x(x)$ and $\theta_y(y)$ up to index i.

ICA_table[i] - Inversions Completed at i. This table holds rows of $\langle Xsibling \rangle$-$\langle Ysibling \rangle \equiv \langle (t', r') \rangle$-$\langle (t'', r'') \rangle \equiv \langle (-1, r') \rangle$-$\langle (-1, r'') \rangle$ presenting an alignment of $\theta_x(x)$ with $\theta_y(y)$ up to i, where the last inversion in θ_x and θ_y are $(p', q') \equiv (r', i)$ and $(p'', q'') \equiv (r'', i)$ respectively by Observation 1. That is, the last inversions in $\theta_x(x)$ and $\theta_y(y)$ were started at index r' and r'' respectively and ends at current index i.

ISA_table_x[ ][i] - Inversions Started At i. This table presents an alignment of $\theta_x(x)$ (upto i), having the last inversion ended at $i-1$, and a new inversion starting from i, with $\theta_y(y)$ (upto i), having the last inversion started before or at i, still continuing or ended at i. It contains the pairs $\langle(t', r_1), \ldots, (t', r_s)\rangle$ of x, presenting the **Inversions Started at** i and ended at i or later. These pairs map to the inversion fragments $T_x[j'][i]$, where $i \leq j' \leq n+1$.

$ISA\_table\_x[][i]$ holds $k = 4$ rows, one for each of the base letters $\alpha \in \{A, T, C, G\}$ such that the row $ISA\_table\_x[\alpha][i]$, holds pairs yielding base letter α at index i. Each row consists of two fields: $Xsibling$ and $Ysibling$.

The $Xsibling$ consists of a x_cont_inv set (having type $cont\_inv$) and a x_end_inv (having type $ending\_inversion$) $Pair$. The x_cont_inv holds the $continuing\_inversion$ $Pairs$ starting from index i, and thus have $t = i$ and $r = j$, $j \geq i+2$. The x_end_inv is the $ending\_inversion$ $Pair$ having $t = -1$ and $r = i$ or $i+1$, representing inversion $(i,i)'$ (no change) or (i,i) (flip) respectively.

Initially $Ysibling$ is empty. In the matching phase, $Ysibling$ maintains a list of pointers to the matched $Pairs$ of $Xsibling$ in T_y, and is categorized into two types, namely, single $ending\_inversion$ $Pair$ named as y_end_inv (Type 1) and set $cont\_inv$ named as y_cont_inv (Type 2) where all $Pairs$ have the same t, $t \leq i$.

Now we explain the intuition behind keeping these records. Both types of pointers ($Ysibling$) mentioned above are considered as matched pairs of x_cont_inv set. But for x_end_inv, only Type 2 pointers are considered as the matched $Pairs$ in this table. For each Type 1 pointer, i.e., y_end_inv in $Ysibling$ list, we keep a separate record $\langle Xsibling\rangle$ - $\langle Ysibling\rangle \equiv \langle x\_end\_inv\rangle$ - $\langle y\_end\_inv\rangle$ in the $ICA\_table[i]$. Though this creates redundancy but this separation makes the data structure conceptually simpler and keeps the final decision checking simple at the end of the algorithm. Please refer to the Figure 4.2 for an illustration.

Observation 2. *At any* i, $\sum_{\alpha\in\{A,T,C,G\}} |Xsibling| = n-i+2$, *where* $Xsibling \in ISA\_table\_x[\alpha][i]$. *Here the total number of* $continuing\_inversion$ *pairs is* $n-i$ *and the number of* $ending\_inversion$ *pairs is 2.*

ISB_table[i] - Inversions Started Before i. It holds rows $\langle Xsibling\rangle$ - $\langle Ysibling\rangle$ just as before presenting alignments of $\theta_x(x)$ yielding α at index i (but having the last inversion started **before** i, and still continuing or ended at i), with $\theta_y(y)$ yielding the same base letter α at index i (having the last inversion started before or at i, and still continuing, or ended at i). Here the x_cont_inv set of $Xsibling$ has $t = i' < i$ and the x_end_inv holds some upper diagonal $ending\_inversion$ pair. Structure of $Ysibling$ and the intuition behind the records are the same as that in $ISA\_table[][i]$ (refer to Figure 4.3).

ISA_table_y[][i]. It contains $\langle Ysibling \rangle \equiv \langle y\_cont\_inv, y\_end\_inv \rangle$ just like the $Xsibling$ in $ISA\_table\_x[][i]$. This $Ysibling$ is actually get pointed by the $Ysibling$ lists of $Xsibling$s, at $ISA\_table\_x[][i]$, $ISB\_table[i]$ and $ICA\_table[i]$.

The row $\langle Xsibling \rangle - \langle Ysibling \rangle$ in a table ($ICA\_table[i]$, $ISA\_table[i]$, or $ISB\_table[i]$) presents an alignment between $\theta_x(x)$ and $\theta_y(y)$ starting from the first index up to index i. If $\|Xsibling\| = N_1$(number of $Pairs$ in $Xsibling$) and $\|Ysibling\| = N_2$, then we call it an $[N_1 : N_2]$ alignment.

4.2 Inaccuracy of the Existing Algorithm by Cho et al. [21]

Cho et al. [21] have provided an $O(n^3)$ algorithm using $O(n^2)$ space (n is the size of the two input strings) for the same problem we have worked on. But we have found through experimentation that their algorithm fails in returning the correct answers in some cases because of not tracking the prefixes of the common ancestors. For example, there can never exists any common ancestors between $x = GTGGC$ and $y = CTGGT$, as the number of complement bases ($A - T$ and $C - G$) is different in x and y. But the algorithm of Cho et al. [21] returns positive for this input and input having the same characteristics. Erroneous output also produced when number of complement bases is the same. In this thesis we present a new algorithm which correctly solves this problem with the same time and space complexity. We further present experimental evidence that our algorithm in practice runs in quadratic time for the average case in contrast to its theoretical cubic time constraint.

4.3 The Algorithm

Common inversed sequences between x and y are computed by tracking the matched pairs between $T_x[][]$ and $T_y[][]$ from column $i = 1$ to n. The following procedures are used in our algorithm.

Procedure 1. ***Next_Calculation((t', r'), i, T_x):*** If the input $pair(t', r')$ is of type $continuing\_inversion$, it returns the pointer to one unique next agreed pair with $t =$

-1 (if the next agreed pair is *ending_inversion*) or $t = t'$ (if the next agreed pair is *continuing_inversion*). Otherwise, if the input pair (t', r') is of type *ending_inversion*, it returns the pointer to the $ISA\_table\_x[][i+1]$ as a new inversion is supposed to start from $i+1$. Similar actions are performed for y if T_y is the input.

Procedure 2. ***PairUp_xColl_yColl(collection_x, collection_y, i):*** This step is called at iteration i, with the matched pairs for index $i+1$ as input. It sets the $\langle collection\_y\rangle \equiv \langle y\_cont\_inv, y\_end\_inv\rangle$ as $Ysibling$ of $\langle collection\_x\rangle \equiv \langle x\_cont\_inv, x\_ending\_inv\rangle$. Thus it lets the alignment (up to i) of $\theta_x(x)$ and $\theta_y(y)$ proceed one step forward, i.e., from i to $i+1$. It executes following steps.

step a: If *x_end_inv* and *y_end_inv* both exist, then pair them up and insert into $ICA\_table[i+1]$.

step b: Insert a pointer to the *y_cont_inv* into the $Ysibling$ list of *collection_x*.

step c: Insert a pointer to the *y_end_inv* into the $Ysibling$ list of *collection_x*.

Procedure 3. ***PairUp_xColl_ySingle(collection_x, single_y, i):*** It works as above but here the *single_y* is a single pair (t, r). If both *collection_x* and *single_y* are nonempty (*Compatibility Check*), it performs the following steps.

step a: If *single_y* is an *ending_inversion* and *collection_x* has *x_end_inv* pair, pair them up and insert into $ICA\_table[i+1]$

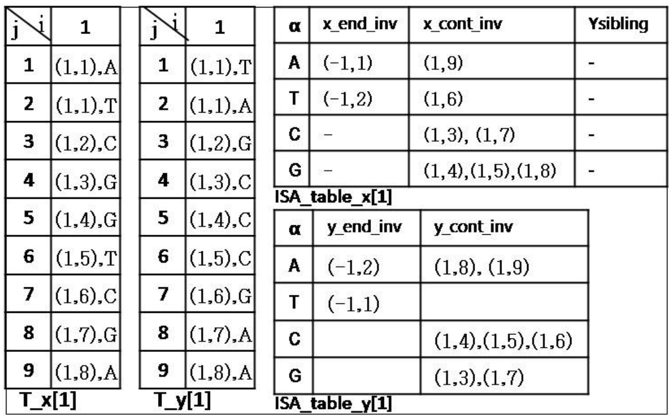
step b: Insert a pointer to *single_y* into the $Ysibling$ list of *collection_x*.

Procedure 4. ***next_calculation_collection(x_cont_inv, x_next_atcg[], i):***
It finds the next agreed pairs of *x_cont_inv* and keep those in a child table $x\_next\_atcg[]$ such that $x\_next\_atcg[\alpha]$ holds the agreed pairs yielding α. For example, suppose, $x\_cont\_inv = \langle (t', r_1), (t', r_2), \ldots, (t', r_p)\rangle$. For each of these pairs we call $next\_calculation((t', r'), i, T\_x)$, $r' = 1, 2, \ldots, p$. Each time as soon as one unique next agreed pair is returned, we add that to $x\_next\_atcg[]$ as follows.

case 1: If the next agreed pair is a *continuing_inversion* pair, yielding α, then insert into *x_cont_inv* of $x\_next\_atcg[\alpha]$.

case 2: If the next agreed pair ends at $i+1$ (has $t = -1$) yielding α, then we assign this *Pair* to *x_end_inv* of $x\_next\_atcg[\alpha]$.

Procedure 5. ***Four_Iteration_Loop(table_x, table_y, i):***
It pairs up the $Xsibling$ in *table_x* with the $Ysibling$ in *table_y*. For each base letters $\alpha \in \{A, T, C, G\}$, if $table\_x[\alpha]$ has non empty $Xsibling$ and $table\_y[\alpha]$ has non empty $Ysibling$ (*Compatibility Check*), then it calls $PairUp\_xColl\_yColl(collection\_x, collection\_y, i)$ with $collection\_x = table\_x[\alpha]$, and $collection\_y = table\_y[\alpha]$.



j \ i	1
1	(1,1),A
2	(1,1),T
3	(1,2),C
4	(1,3),G
5	(1,4),G
6	(1,5),T
7	(1,6),C
8	(1,7),G
9	(1,8),A

T_x[1]

j \ i	1
1	(1,1),T
2	(1,1),A
3	(1,2),G
4	(1,3),C
5	(1,4),C
6	(1,5),C
7	(1,6),G
8	(1,7),A
9	(1,8),A

T_y[1]

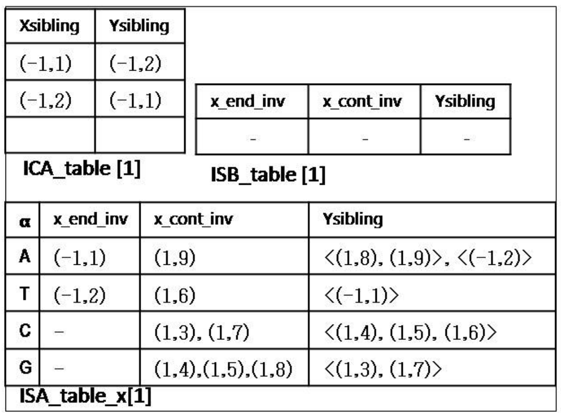
α	x_end_inv	x_cont_inv	Ysibling
A	(−1,1)	(1,9)	-
T	(−1,2)	(1,6)	-
C	–	(1,3), (1,7)	-
G	–	(1,4),(1,5),(1,8)	-

ISA_table_x[1]

α	y_end_inv	y_cont_inv
A	(−1,2)	(1,8), (1,9)
T	(−1,1)	
C		(1,4),(1,5),(1,6)
G		(1,3),(1,7)

ISA_table_y[1]

(a)



Xsibling	Ysibling
(−1,1)	(−1,2)
(−1,2)	(−1,1)

ICA_table [1]

x_end_inv	x_cont_inv	Ysibling
-	-	-

ISB_table [1]

α	x_end_inv	x_cont_inv	Ysibling
A	(−1,1)	(1,9)	⟨(1,8), (1,9)⟩, ⟨(−1,2)⟩
T	(−1,2)	(1,6)	⟨(−1,1)⟩
C	–	(1,3), (1,7)	⟨(1,4), (1,5), (1,6)⟩
G	–	(1,4),(1,5),(1,8)	⟨(1,3), (1,7)⟩

ISA_table_x[1]

(b)

Figure 4.2: (i)Before Initialization; (ii)After Initialization

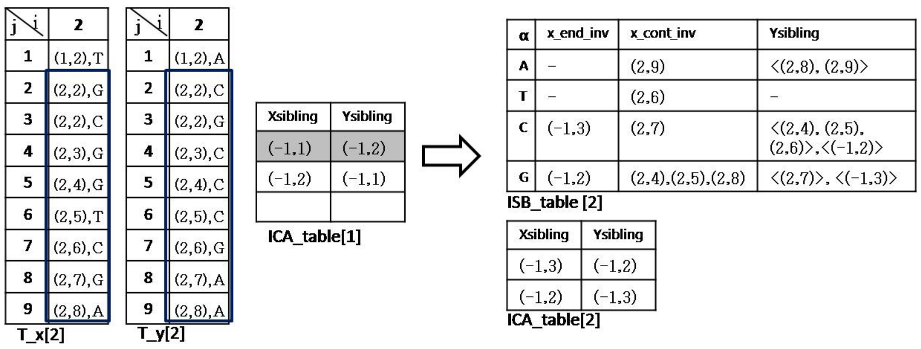
Now we explain the algorithm using the procedures stated above. The main algorithm iterates over $i = 1$ to $n - 1$. The column i of each of the tables described above actually represent the alignment of $\theta_x(x)$ and $\theta_y(y)$ up to index i for some θ_x and θ_y. So at each iteration i, it processes the rows in three tables: $ICA\_table[i]$, $ISA\_table[][i]$, and $ISB\_table[i]$ to calculate the next agreed pairs, pair up the matched pairs and insert those into the column $i + 1$ of the appropriate table. If for any row $\langle Xsibling\rangle$-$\langle Ysibling\rangle$, next agreed pairs of $Xsibling$ does not get matched pair from next agreed pairs of $Ysibling$, then it means no alignment with the inverted sequence of x presented by that $Xsibling$ exists in y. Thus this alignment $\langle Xsibling\rangle$-$\langle Ysibling\rangle$ is not passed forward anymore and is rather dropped here. We will explain the algorithm using an illustrative example. Consider, $x = AGCCAGCT$ and $y = TCGGGCTT$ given in Figure 4.1.

4.3.1 Initialization

$ISA\_table\_x[1]$ and $ISA\_table\_y[1]$ are shown in the Figure 4.2. It executes Procedure 5, i.e., $Four\_Iteration\_Loop$ to start aligning x with y by pairing up these two tables. While calling the procedure, input parameters are set as: $table\_x = ISA\_table\_x[1]$, $table\_y = ISA\_table\_y[1]$, and $i = 1$.

4.3.2 Iteration

For each iteration $i = 1, 2, \ldots, n - 1$, following steps are performed.



j \ i	2
1	(1,2),T
2	(2,2),G
3	(2,2),C
4	(2,3),G
5	(2,4),G
6	(2,5),T
7	(2,6),C
8	(2,7),G
9	(2,8),A

T_x[2]

j \ i	2
1	(1,2),A
2	(2,2),C
3	(2,2),G
4	(2,3),C
5	(2,4),C
6	(2,5),C
7	(2,6),G
8	(2,7),A
9	(2,8),A

T_y[2]

Xsibling	Ysibling
(−1,1)	(−1,2)
(−1,2)	(−1,1)

ICA_table[1]

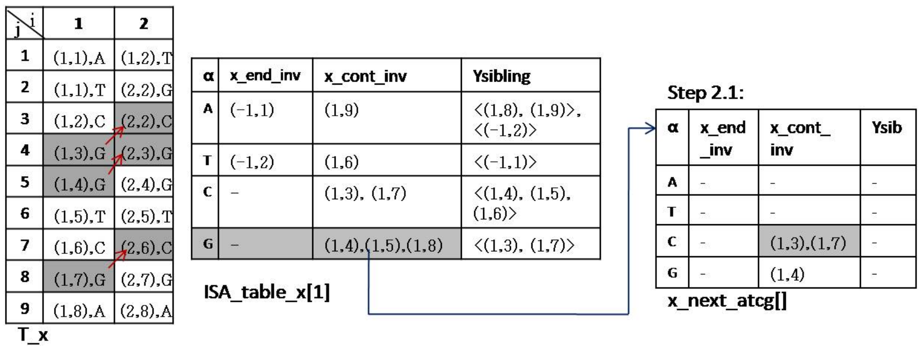
α	x_end_inv	x_cont_inv	Ysibling
A	–	(2,9)	<(2,8), (2,9)>
T	–	(2,6)	–
C	(−1,3)	(2,7)	<(2,4), (2,5), (2,6)>,<(−1,2)>
G	(−1,2)	(2,4),(2,5),(2,8)	<(2,7)>, <(−1,3)>

ISB_table [2]

Xsibling	Ysibling
(−1,3)	(−1,2)
(−1,2)	(−1,3)

ICA_table[2]

Figure 4.3: Demonstration of Step 1 for iteration 1



j \ i	1	2
1	(1,1),A	(1,2),T
2	(1,1),T	(2,2),G
3	(1,2),C	(2,2),C
4	(1,3),G	(2,3),G
5	(1,4),G	(2,4),G
6	(1,5),T	(2,5),T
7	(1,6),C	(2,6),C
8	(1,7),G	(2,7),G
9	(1,8),A	(2,8),A

T_x

α	x_end_inv	x_cont_inv	Ysibling
A	(−1,1)	(1,9)	<(1,8), (1,9)>, <(−1,2)>
T	(−1,2)	(1,6)	<(−1,1)>
C	–	(1,3), (1,7)	<(1,4), (1,5), (1,6)>
G	–	(1,4),(1,5),(1,8)	<(1,3), (1,7)>

ISA_table_x[1]

Step 2.1:

α	x_end _inv	x_cont_ inv	Ysib
A	-	-	-
T	-	-	-
C	-	(1,3),(1,7)	-
G	-	(1,4)	-

x_next_atcg[]

Figure 4.4: Demonstration of Step 2.1 for $\alpha = G$ in iteration 1

Step 1

Process ICA_table[i]: For the first row $\langle Xsibling\rangle$-$\langle Ysibling\rangle$=$\langle(-1,r')\rangle$-$\langle(-1,r'')\rangle$, we call Procedure 1, i.e., $next\_calculation((-1,r'),T\_x,i)$ and $next\_calculation((-1,r''),T\_y,i)$. They return pointers to $ISA\_table\_x[i+1]$ and $ISA\_table\_y[i+1]$ respectively. After that, we call the $Four\_Iteration\_Loop(ISA\_table\_x[i+1],\ ISA\_table\_y[i+1])$. Other rows of $ICA\_table[i]$ are not processed as they involve doing the same assignments (according to the Merging Case 1 explained later in Observation 5). See Figure 4.3 for an illustration.

Step 2

Process ISA_table_x[][i]: For each $\alpha \in \{A,T,C,G\}$ we perform Step 2.1, Step 2.2 and Step 2.3.

Step 2.1

It calls Procedure 4, with x_cont_inv of $ISA\_table\_x[\alpha][i]$, which finds its next agreed pairs and keeps those in a child table $x\_next\_atcg[]$ (see Figure 4.4)

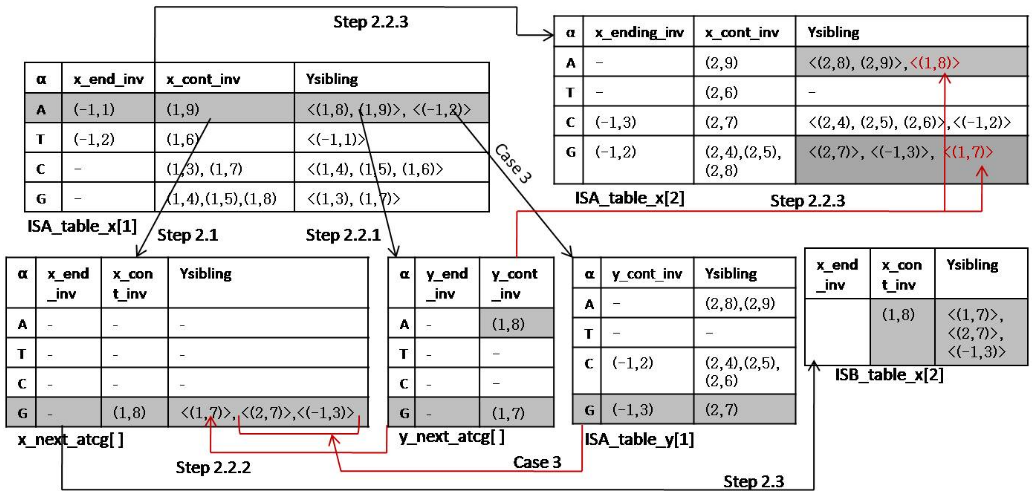

Step 2.2.3
α | x_end_inv | x_cont_inv | Ysibling
A | (-1,1) | (1,9) | <(1,8), (1,9)>, <(-1,2)>
T | (-1,2) | (1,6) | <(-1,1)>
C | – | (1,3), (1,7) | <(1,4), (1,5), (1,6)>
G | – | (1,4),(1,5),(1,8) | <(1,3), (1,7)>
ISA_table_x[1]
Step 2.1
Step 2.2.1
Case 3
α | x_ending_inv | x_cont_inv | Ysibling
A | – | (2,9) | <(2,8), (2,9)>, <(1,8)>
T | – | (2,6) | –
C | (-1,3) | (2,7) | <(2,4), (2,5), (2,6)>, <(-1,2)>
G | (-1,2) | (2,4),(2,5),(2,8) | <(2,7)>, <(-1,3)>, <(1,7)>
ISA_table_x[2]
Step 2.2.3
α | x_end_inv | x_cont_inv | Ysibling
A | - | - | -
T | - | - | -
C | - | - | -
G | - | (1,8) | <(1,7)>, <(2,7)>,<(-1,3)>
x_next_atcg[ ]
α | y_end_inv | y_cont_inv
A | - | (1,8)
T | - | –
C | - | –
G | - | (1,7)
y_next_atcg[ ]
α | y_cont_inv | Ysibling
A | – | (2,8),(2,9)
T | – | –
C | (-1,2) | (2,4),(2,5),(2,6)
G | (-1,3) | (2,7)
ISA_table_y[1]
x_end_inv | x_cont_inv | Ysibling
 | (1,8) | <(1,7)>, <(2,7)>, <(-1,3)>
ISB_table_x[2]
Step 2.2.2
Case 3
Step 2.3

Figure 4.5: Demonstration of Steps 2.2 and 2.3 for $\alpha = A$ in iteration 1

Step 2.2

For each list item $Ysibling[p]$, in this step we find the alignment of the pairs in $x\_next\_actg[]$ (calculated in the previous step) with the next agreed pairs found from $Ysibling[p]$. We need to deal with one of the following cases.

Step 2.2 Case 1. The $Ysibling[p]$ is of type y_cont_inv having $size > 1$ (Step 2.2.1 to Step 2.2.3):

Step 2.2.1: If $y\_next\_atcg[]$ of $Ysibling[p]$ is not calculated yet, then call Procedure 4, i.e., $next\_calculation\_collection(Ysibling[p], y\_next\_actg[], i)$.

Step 2.2.2: Now both the $x\_next\_atcg[]$ and $y\_next\_actg[]$ are ready to be paired up. So we call the *Four_Iteration_Loop*($x\_next\_atcg[]$, $y\_next\_atcg[]$).

Step 2.2.3: If $Xsibling$ has x_end_inv pair, and $y\_next\_actg[]$ has not been paired with $ISA\_table\_x[][i+1]$ yet (*Merging Case 2* explained later in Observation 6), then pair them up by calling *Four_Iteration_Loop*($ISA\_table\_x[][i+1]$, $y\_next\_actg[]$). Please refer to Figure 4.5 for an illustration.

Step 2.2 Case 2. The $Ysibling[p]$ is of type y_cont_inv having $size = 1$ (Step 2.2.4 to Step 2.2.6):

Step 2.2.4: We call $next\_calculation((t', r'), i, T_y)$, where (t', r')=y_cont_inv. Let the returned unique next agreed pair yield α and name it $pair\_y$.

Step 2.2.5: We call $PairUp\_xColl\_ySingle(x\_next\_actg[\alpha], single\_y, i)$.

Step 2.2.6: If $Xsibling$ has x_end_inv pair, and $pair\_y$ has not been paired with

$ISA\_table\_x[\alpha][i{+}1]$ yet (*Merging Case 2*), then we call $PairUp\_xColl\_ySingle(ISA\_table\_x[\alpha][i+1], pair\_y, i)$.

Step 2.2 Case 3. If $Ysibling[p]$ is of type y_end_inv: If $x\_next\_atcg[]$ has not been paired up with $ISA\_table\_y[][i{+}1]$ yet (*Merging Case 3*, explained later in Observation 7), then we call the procedure *Four_Iteration_Loop* with input tables: $x\_next\_atcg[]$ and $ISA\_table\_y[i+1]$. Please refer to Figure 4.5.

Step 2.3

Update the ISB_table[i+1]: For each new $x\_next\_atcg[\alpha]$ created in Step 2.1, if it has non empty $Ysibling$ list, then we insert it into $ISB\_table[i+1]$ as new rows, where $\alpha \in \{A, T, C, G\}$. Please see Figure 4.5 for an illustration.

Step 3

Process ISB_table[i]: For each row p of $ISB\_table[i]$: $\langle Xsibling\rangle$ - $\langle Ysibling\rangle$, we execute the Steps 3.1, 3.2, and 3.3. They are identical to Step 2.1, 2.2, 2.3 except the fact that the row items $ISB\_table[p][i]$ are used instead of $ISA\_table\_x[\alpha][i]$.

4.3.3 Termination

After the iterations complete, if the $ICA\_table[n]$ contains no row, we return NO indicating the absence of any consensus sequence between x and y. Otherwise we return YES, indicating the existence of some consensus sequence between x and y.

Note here that, the algorithm presented by Cho et al. [21] does not return correct results because of not tracking the prefix of common ancestors properly which is kept in our algorithm using the ISA_table, ISB_table and ICA_table.

4.4 Correctness of the Algorithm

Correctness of the algorithm is proven by Lemma 1 and Lemma 2 by showing that no valid alignment is missed and invalid alignments are canceled as soon as detected. Necessity and sufficiency of the termination step of the algorithm is proven in Lemma 3 as well.

We observe that each row at column i of each table actually presents an alignment between $\theta_x(x)$ and $\theta_y(y)$ up to index i. For Lemma 1 we need the following two Observations.

Observation 3. ***Split case 1:*** *One alignment is split into multiple new alignments when ending of the last continuing inversion is reached.* This case is ensured by the step b of Procedure 4, step a of Procedure 2 and 3, and Steps: 2.2.3, 2.2.6, 3.2.3, 3.2.6, case 3 (under Step 2 & 3) in the algorithm. For an illustration see Example 3 below.

Example 3. In this example we explain Observation 3 with the help of Figure 4.6(i). First, we explain initialization. Here, three T's in the first column of T_x and the two T's in the first column of T_y are the matched $Pairs$. So $Xsibling = [ending\_inv, cont\_inv] = [null, \langle T(1,4), T(1,7), T(1,8)\rangle]$ has $Ysibling = [\langle T(1,7), T(1,8)\rangle]$. All the $Pairs$ in $cont\_inv$ have $t = 1$. For simplicity, let us call the three T's in $T_x[1]$, $Xsib_1$ as they are presenting continuing inversions started from index 1. Furthermore let us call the two T's in $T_y[1]$, $Ysib_1$ for the same reason. Now we simply present the alignment by $\langle Xsib_1\rangle$-$\langle Ysib_1\rangle$=$\langle T(1,4), T(1,7), T(1,8)\rangle$-$\langle T(1,7), T(1,8)\rangle$. This row appears in the $ISA\_table\_x[1]$ for index 1.

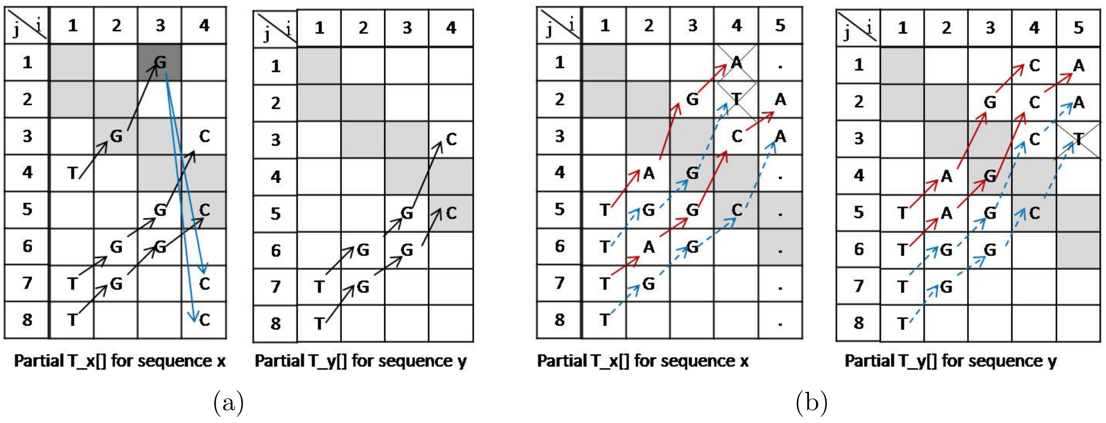

Partial T_x[] for sequence x
Partial T_y[] for sequence y
Partial T_x[] for sequence x
Partial T_y[] for sequence y
(a)
(b)

Figure 4.6: (i) Split case 1 ; (ii) Split case 2

Then in iteration 1, we proceed with this alignment one step forward to index 2 by transferring this $\langle Xsib_1\rangle$-$\langle Ysib_1\rangle$ into the $ISB\_table[2]$ as $\langle Xsib_1\rangle$-$\langle Ysib_1\rangle$=$\langle G(1,3), G(1,6), G(1,7)\rangle$-$\langle G(1,6), G(1,7)\rangle$.

Then we come to iteration 2. Whenever a pair in a set $cont\_inv$ reaches ending, we split them into different paths, even if they continue maintaining the same prefix. In other words, we will break the $cont\_inv$ into two different sets but each having the same $Ysibling$ if none of the pairs in $Ysibling$ reach ending. For example, at iteration 2, when we calculate next agreed pairs of $\langle Xsib_1\rangle$-$\langle Ysib_1\rangle$ = $\langle G(1,3), G(1,6), G(1,7)\rangle$-$\langle G(1,6), G(1,7)\rangle$, we get: $\langle G(-1,1), G(1,5), G(1,6)\rangle$-$\langle G(1,5), G(1,6)\rangle$. Note here, one pair $G(-1,1)$ has reached ending. So for next index, i.e., 3, in $ISB\_table[3]$, we keep it in a separate field named as $ending\_inv = G(-1,1)$. So the new row looks like: $\langle Xsib_1 : [ending\_inv], [cont\_inv]\rangle$-$\langle Ysib_1\rangle = \langle[(-1,1)], [(1,5)(1,6)]\rangle$-$\langle G(1,5), G(1,6)\rangle$.

In iteration 3, after calculating the next agreed pairs, the $Xsib_1 = \langle[(-1,1)], [(1,5),(1,6)]\rangle$ is split into $Xsib_1 = \langle C(1,3), C(1,5)\rangle$ and $Xsib_4 = \langle C(4,7), C(4,8)\rangle$ (resulted by the $ending\_inv = (-1,1)]$ in third index). However, each of those are still pointing at the same $Ysib_1 = \langle C(1,3), C(1,5)\rangle$. One alignment $\langle Xib_1\rangle$-$\langle Ysib_1\rangle$ is kept in $ISB\_table[4]$ and another alignment $\langle Xsib_4\rangle$-$\langle Ysib_1\rangle$ is kept in $ISA\_table\_x[4]$. This is necessary because from now on, $Xsib_1$ and $Xsib_4$ will be following different paths. Just like this, whenever a pair that is an inversion reaches ending, and new inversion starts, new rows are formed to record the new alignment.

Observation 4. ***Split case 2:*** *Alignments can be split before reaching the ending if new prefix appears.* This happens when the next agreed pairs differ by yielding base letter $\alpha \in \{A, C, T, G\}$. This split is ensured by the strategy followed in Procedure 4, 2 and 3. For clarification see the Example 4 below.

Example 4. In this example we explain Observation 4 with the help of Figure 4.6(ii). In initialization step, we have, $\langle Xsib_1\rangle$-$\langle Ysib_1\rangle$=$\langle T(1,5), T(1,6), T(1,7), T(1,8)\rangle$-$\langle T(1,5), T(1,6), T(1,7), T(1,8)\rangle$ stored at $ISA\_table\_x[T][1]$.

In iteration 1, we proceed with this alignment one step forward to index 2 by transferring this $\langle Xsib_1\rangle$-$\langle Ysib_1\rangle$ into $ISB\_table[2]$. But here next agreed pairs are different for different pairs. Two different sets are resulted from $Xsib_1$ as $Xsib'_1 = \langle A(1,4), A(1,6)\rangle$ and $Xsib''_1 = \langle G(1,5), G(1,7)\rangle$. Also $Ysib_1$ gives $Ysib'_1 = \langle A(1,4), A(1,5)\rangle$ and $Ysib''_1 = \langle G(1,6), G(1,7)\rangle$. So the single alignment row presenting a [4:4] alignment is divided into two different rows as $\langle Xsib'_1\rangle$-$\langle Ysib'_1\rangle$ (solid arrow) and $\langle Xsib''_1\rangle$-$\langle Ysib''_1\rangle$ (dotted arrow) each presenting [2:2] alignment and they are placed into the $ISB\_table[2]$.

Then in iteration 2, these two rows proceed to $ISB\_table[3]$ as $\langle Xsib'_1\rangle$-$\langle Ysib'_1\rangle$=$\langle G(1,2), G(1,5)\rangle$-$\langle G(1,2), G(1,4)\rangle$ (solid arrow) and $\langle Xsib''_1\rangle$-$\langle Ysib''_1\rangle$=$\langle G(1,4), G(1,6)\rangle$-$\langle G(1,5), G(1,6)\rangle$ (dotted arrow).

Lemma 1. *No valid alignment is missed*

Proof. Each row at column i of each table presents an alignment between $\theta_x(x)$ and $\theta_y(y)$ up to index i. Based on next agreed pairs, if necessary we split that alignment into multiple new alignments as explained in Observations 3 and 4. Thus no valid alignment is missed. □

Lemma 2. *Invalid alignments that is agreed sequences of x not existing in y are canceled as soon as detected.*

Proof. If in iteration i, for an alignment $\langle Xsibling\rangle$-$\langle Ysibling\rangle$, next agreed pairs of $Xsibling$ get no matched pair from the next agreed pairs of $Ysibling$, then the alignment is not passed forward and rather dropped immediately. This case is ensured by the *Compatibility Check* executed inside the Procedures 3 and 5. □

We illustrate Lemma 2 using Example 5 below.

Example 5. We explain Lemma 2 using the example in Figure 4.6(ii). In iteration 3, let us consider the match showed by a solid arrow first. Next agreed pair again splits the $Xsib'_1$ into two sets as $Xsib'''_1$=$\langle A(-1,1)\rangle$ and $Xsib''''_1$=$\langle C(1,3)\rangle$. The next agreed pairs of $Ysib'_1$ are $\langle C(-1,1), C(1,3)\rangle$. So $Ysib'_1$ is not split by next agreed pairs, since all next agreed pairs yield the same base letter C. Now, $Xsib'''_1$ does not get any match from y, so it is dropped here and only one row, $\langle Xsib''''_1\rangle$-$\langle Ysib_1\rangle$ = $\langle C(1,3)\rangle$-$\langle C(-1,1), C(1,3)\rangle$ is passed to the $ISB\_table[4]$. That is, no inversed sequence having prefix $TAGA$ exists. Or we can say that in x, no inversion sequence will be a consensus sequence if it starts with the inversion $(p,q) = (1,4)$. So if any $Xsibling$ gets split because of different next agreed pairs (yielding different base $\alpha \in \{A,T,C,G\}$) and a set does not get a matched set from the corresponding next agreed pairs produced by $Ysibling$, then that alignment will be dropped immediately.

Lemma 3. *Checking non emptiness of $ICA\_table[n]$ is necessary and sufficient to decide on the existence of consensus sequence.*

Proof. Rows in $ICA\_table[n]$ indicates alignment of $\theta_x(x)$ and $\theta_y(y)$ up to the last index such that, the last inversion ends at i for both of them. Thus it indicates the existence of a consensus sequence. If $ICA\_table[n]$ is empty it means no $\theta_x(x)$ can align with any $\theta_y(y)$ up to the last index, thus indicating the absence of a consensus sequence among x and y. □

4.5 Time Complexity

Before deriving the theoretical time complexity of our algorithm, we first discuss how the polynomiality of the algorithm is ensured. The number of list items $\langle Xsibling\rangle$-$\langle Ysibling\rangle$ (in ISA_table_x & ISB_table) and the size of $Ysibling$ for each such row affect the runtime. Here, some alignment presented by a row $\langle Xsibling\rangle$-$\langle Ysibling\rangle$ having $\|Xsibling\| = N_1$ and $\|Ysibling\| = N_2$ is actually presenting alignment $[N_1 : N_2]$ using a single row. We avoid keeping separate rows for each of them as in that case it needs $N_1 * N_2$ rows each presenting the same prefix. Besides that, our algorithm prevents unpredictable increment of the number of rows in $ISA\_table\_x[]$ and $ISB\_table[]$ by merging overlapping portions

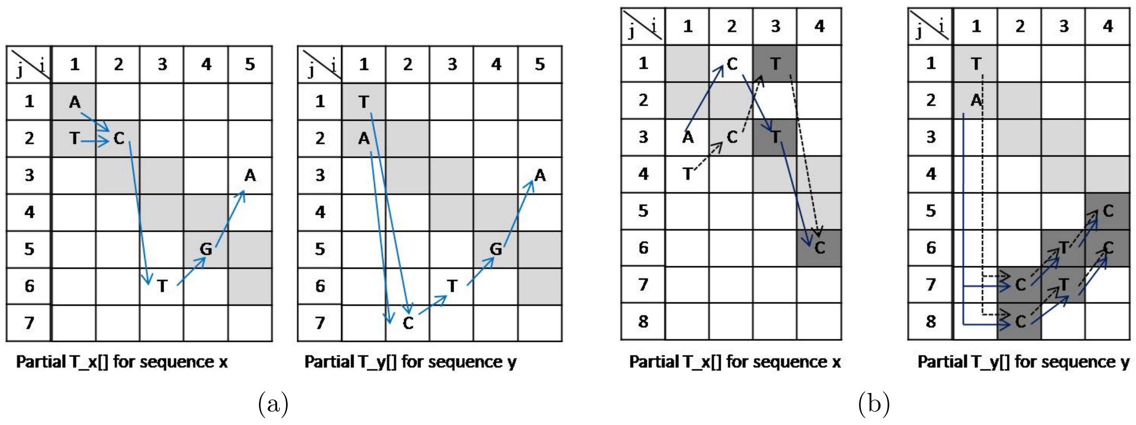

Partial T_x[] for sequence x
Partial T_y[] for sequence y
(a)
Partial T_x[] for sequence x
Partial T_y[] for sequence y
(b)

Figure 4.7: (i) Merging case 1; (ii) Merging case 2 and 3

of the alignments, thus ensuring a polynomial run time as explained below by observations and lemma.

Observation 5. *Merging case 1: Merging in $ICA\_table[i]$.* In each iteration $i = 1$ to $n - 1$ of the algorithm, at Step 1, we pair up $ISA\_table\_x[i + 1]$ and $ISA\_table\_y[i + 1]$ through procedure $Four\_Iteration\_Loop$ once only for the non empty $ICA\_table[i]$. This ensures the merging all the alignments presented by the rows of $ICA\_table[i]$.

Proof. From $i + 1$, destiny of all those alignments in $ICA\_table[i]$ is the same, i.e., sequence of next agreed pairs of all those alignments is the same for following successive iterations, until the next ending is reached. In other words, we can say, if any alignment residing in $ICA\_table[i]$ is dropped at some later index due to some mismatch, then it will happen for all other alignments in $ICA\_table[i]$ as well. So instead of keeping separate rows, we merge the overlapping portion to avoid the redundant calculation. Notably, it may merge multiple alignments having different prefixes as well; but this does not create problem as they have the same destiny from index $i + 1$ up to the next ending. Please refer to the Example 6 for an illustration. □

Example 6. In this example we explain Observation 5 with the help of Figure 4.7(i). Two alignments, one having A as the first letter and the other having T as the first letter are merged into one at index 2 as they are destined to the same result from that point. Here the alignment is shown up to index 5. At iteration 4 if the next agreed pair for $Ysibling$ yields T or some base letter other than A (next agreed pair of $Xsibling$) then this alignment will be canceled and not proceeded further. This will happen for both merged sequences.

Observation 6. ***Merging case 2:*** *Merging alignments in $ISA\_table\_x[i + 1]$ on iteration i is* ensured by Step 2.2.3, 2.2.6, 3.2.3, and 3.2.6 in the algorithm.

Proof. If different *x_end_inv* pairs of x at index i are paired with the same *y_cont_inv* from y, then for each of those *x_end_inv* pairs, $ISA\_table\_x[i+1]$ and the next agreed pairs of that *y_cont_inv* need pairing up. As this same pairing up operation is required for all those matching *x_end_inv* pairs, so this is done once only. So we can say that those alignments presented by the *x_end_inv* pairs are merged into one as from now on, their destiny is the same. For an illustration please refer to Example 7. □

Example 7. In this example we explain Observation 6 with the help of Figure 4.7(ii). In the initialization step, two alignments: $\langle A(1,3)\rangle$-$\langle A(-1,2)\rangle$ and $\langle T(1,4)\rangle$-$\langle T(-1,1)\rangle$ are kept in $ISA\_table\_x[1]$. Then in iteration 1, each of those is passed to $ISB\_table\_x[2]$ as $\langle C(-1,1)\rangle$-$\langle C(2,7), C(2,8)\rangle$ and $\langle C(1,3)\rangle$-$\langle C(2,7), C(2,8)\rangle$. Let us call these $\langle Xsib'_1\rangle$-$\langle Ysib_2\rangle$ and $\langle Xsib''_1\rangle$-$\langle Ysib_2\rangle$. From this point, two different alignment start overlapping. But still we can't merge them into one row until the inverted sequences presented by $Xsib'_1$ and $Xsib''_1$ both reach the ending (reduce to *x_end_inv* pair) later at the same index i. But here only the $\langle Xsib'_1\rangle$ has reached the ending.

In iteration 2, next agreed pairs are calculated for$\langle Xsib'_1\rangle$-$\langle Ysib_2\rangle$ and $\langle Xsib''_1\rangle$-$\langle Ysib_2\rangle$ and passed to $ISB\_table[3]$ as $\langle Xsib_3f'\rangle$-$\langle Ysib_2\rangle$=$\langle T(-1,3)\rangle$-$\langle T(1,6), T(1,7)\rangle$ and $\langle Xsib''_1\rangle$-$\langle Ysib_2\rangle$=$\langle T(-1,1)\rangle$-$\langle T(1,6), T(1,7)\rangle$ respectively.

Finally, in iteration 3, we see that, the same $\langle Ysib_2\rangle$=$\langle T(1,6), T(1,7)\rangle$ is paired up with different *ending_inv* pairs: $\langle Xsib_3f'\rangle$ (solid arrow) and $\langle Xsib''_1\rangle$ (dotted arrow). That is, two inversed sequences of x having the same or different prefix have reached the ending at the same index, i.e., 3 while both have the same $\langle Ysib_2\rangle$. So now they can be merged into one record safely as their destiny is the same now. Only one record $\langle C(4,6)\rangle$-$\langle C(1,5), C(1,6)\rangle$ is kept in $ISA\_table\_x[4]$.

Observation 7. ***Merging case 3:*** *Merging alignments in* $ISB\_table\_x[i+1]$ *on iteration* i *are* ensured by case 3 of Steps 2.2 and 3.2 in the algorithm.

Proof. The scenario explained in previous Lemma also happens for the opposite case. That is, if the same *x_cont_inv* from x matches with different *y_end_inv* pair of y at the same index i, then from $i+1$, they (different alignments presented by those matched *y_end_inv* pairs) are merged into one record in $ISB\_table\_x[i+1]$. Please see the Example 8 for an illustration. □

Example 8. In this example we explain Observation 7 with the help of Figure 4.7(ii). Interchanging the x and y, we get the merged alignment $\langle Xsib_2\rangle$-$\langle Ysib_4\rangle$=$\langle C(1,5), C(1,6)\rangle$-$\langle C(4,6)\rangle$ in $ISB\_table[4]$. Note that, in the previous iteration, the corresponding $\langle Xsib\_2\rangle$ can be found in either $ISA\_table\_x[3]$ or $ISB\_table[3]$.

Lemma 4. *The algorithm merges overlapping portions of the alignments to avoid redundant operations and unnecessary increment of rows in ICA_table, ISA_table, and ISB_table.*

Proof. This Lemma is proven by Observations 5, 6 and 7. □

Now we begin the discussion for deriving theoretical time complexity of our algorithm. Theoretical worst case and average case time complexity of the algorithm are $O(n^4)$ and $O(n^3)$ respectively proven by Lemma 13 and Lemma 15. For deriving the time complexity of the algorithm, we first show it for the worst case. The worst case scenario is defined as when each pair from T_x gets some matched pair from T_y thus no alignment is canceled because of mismatch. We define average case as fifty percent match between the inverted x and inverted y at each index $i = 1, 2, \ldots, n$ thus some alignments are canceled in each iteration as we step forward. Some Lemmas are provided based on the worst case which are later used in defining the average case time complexity.

Observation 8. *For any $i' = 1, \ldots, n-1$, the size of $cont\_inv\_i'$ is $n - i'$ at iteration i' (by Observation 2) and is reduced by one at each iteration $i = i'+1, i'+2, \ldots, n-1$, leaving no continuing_inversion pair (starting at i') at the last index n.*

Proof. We observe that, at iteration i', $cont\_inv\_i'$ are kept in the x_cont_inv sets of $ISA\_table\_x[i']$ and has size $\|cont\_inv\_i'\| = n-i'$ by Observation 2. At iteration i', all of these pairs: $(t,r) = (i', i'+2), (i', i'+3), (i', i'+4), \ldots, (i', n+1)$ map to inversions $(i', i'+2), (i', i'+3), (i', i'+4), \ldots, (i, n)$. For example, for index 1, these pairs $(t,r) = (1,3), (1,4), \ldots, (1, n+1)$ maps inversions $(p,q) = (1,2), (1,3), \ldots, (1,n)$ respectively. Now let us see how its size is reduced. Let us start with iteration i'. If all of them have matched pair in y then at iteration i', we have to perform the *next_calculation* step for each of these pairs, and one of them namely the pair $(t,r) = (i', i'+2)$ (mapping the inversion $(p,q) = (i', i'+1)$) reaches ending in the next index and thus becomes *end_inv* pair$=(-1, i')$ and its eliminated from the set $cont\_inv\_i'$ (see the next calculation steps for clarification). So the size of $cont\_inv\_i'$ is reduced by one at iteration $i = i'+1$. Similarly, in iteration $i = i'+1$ the pair $(i', i'+3)$ (mapping $(p,q) = (i', i'+2)$) reaches ending and gets removed from $cont\_inv\_i'$. Thus at iteration $i = i'+2$, the size of $cont\_inv\_i'$ is again reduced by one. This continues for each of the next iterations and finally at iteration $n-1$, next calculation of $(i', n+1)$ (last pair in $cont\_inv\_i$) gives pair $(-1, i')$ and thus reaches ending at index n. So for index n, no *continuing_inversion* started at i' is left. □

Observation 9. *At any iteration $i \geq i'$, the size of $cont\_inv\_i'$ can be at most $n - i$. Hence, the total number of existing continuing_inversion pairs (for x or y) considering all $cont\_inv\_i'$ equals to $i(n-i)$, where $1 \leq i' \leq i$.*

Proof. At index i', the size of *cont_inv_i'* is $n - i'$. Then in each next iteration its size is reduced by one according to Observation 8. So at index $i = i' + 1$, its size is $n - i' - 1 = n - (i' + 1) = n - i$. At the next index $i = i' + 2$, size is $n - i' - 2 = n - (i' + 2) = n - i$, and so on. This happens for all $1 \leq i' \leq i$. So the total number existing *continuing_inversion* pairs is $i(n - i)$. □

Observation 10. *Total number of end_inv pairs at itearation i is $i + 1$*

Proof. We know by Observation 8 that at each iteration i, one *end_inv* pair is created from *cont_inv_i'*. So at iteration i, we get one *end_inv* pair from each *cont_inv_i'*, $1 \leq i' < i$. Again, two flip pairs (i, i) and $(i, i)'$ are also *end_inv* found at index i. So the total number of *end_inv* pairs at itearation i is $(i - 1) + 2 = i + 1$. □

Lemma 5. *At iteration i, Procedure 1 (finding the next agreed pairs) is called once for each existing continuing_inversion Pair, resulting in $O(2i(n - i))$ calls considering both x and y.*

Proof. By Observation 9, at iteration i, the size of *continuing_inversion Pair*s is $O(i(n-i))$ for x and $O(i(n - i))$ for y. We know *continuing_inversion Pair*s reside in *x_cont_inv* sets by definition. At iteration i, for each distinct *x_cont_inv* set, we calculate next agreed pairs only once by calling Procedure 1. This is true even if the same *x_cont_inv* exists multiple times in $ISA\_table\_x[i]$ or $ISB\_table\_x[i]$, since we use pointer to *x_cont_inv*. This is ensured by the Step 2.2.1 and 3.2.1 in algorithm. This holds true for each *y_cont_inv* as well. So we call Procedure 1 $O(i(n - i))$ times for both x and y, and thus $O(2i(n - i))$ in total. (However, we call the *Four_Iteration_Loop* (Step 2.2.2 and 3.2.2) each time the set *x_cont_inv* is encountered in $ISA\_table\_x[i]$ or $ISB\_table\_x[i]$.) For an illustration please see the Example 9. □

Example 9. We use the same scenario used for split case 1 (Figure 7(i)) for an illustration of Lemma 5. In iteration 3, though $Ysib_1$ is pointed by the *Ysibling* list member of both the $Xsib_1$ and $Xsib_4$, once the *next_calculation* step for $Ysib_1$ is performed (*next_atcg_y*[] is generated) for, say, $Xsib_4$, we do not need to perform the same task again while processing the alignment holding $\langle Xsib_1 \rangle - \langle Ysib_1 \rangle$; Rather, we only need to run the *Four_Iteration_Loop* for pairing up the *next_atcg_x*[α] of $Xsib\_1$ and *next_atcg_y*[α] of $Ysib_1$, $\alpha = \{A, T, C, G\}$.

Total number of calls to the *Four_Iteration_Loop* depends on the size of *Ysibling* list for each $\langle Xsibling \rangle - \langle Ysibling \rangle$ list items in $ISA\_table\_x[i]$ and $ISB\_table[i]$. In order to find the total number of calls to Procedure 5: *Four_Iteration_Loop*, we first present some Observations and Lemmas.

At some index $i > i'$, all the existing *x_cont_inv_i'* of x resides in *ISB_table*$[i]$. Now let us see how many *Ysibling*s they can have in total at index i. For simplicity, let us think of *x_cont_inv_4* only ($i' = 4$). All *continuing_inversion Pair*s in *x_cont_inv_4* represent the inversion starting from index $4 < i$ and at index 4, they reside in *ISA_table_x*$[4]$. These can be divided into $k = 4$ sets each having on average $(n - 4)/k$ pairs, yielding α, $\alpha \in \{A, T, C, G\}$. As they proceeds, they may be divided into several more child sets (introduce new rows) based on the yielding base letter of the next agreed pairs by Observation 4. At index i, the size of *x_cont_inv_4* is $n - i$ by Observation 9. Similar case happens for all *x_cont_inv_i'*, $i' \leq i$. This is also true for all the *y_cont_inv_i'*, $i' \leq i$. First, let us see how many *x_cont_inv* sets: $\{Ci'_1, Ci'_2, \ldots, Ci'_s\} \in$ *x_cont_inv_i'* exists at index i. We have the following Observation.

Observation 11. *At any index $i > i'$, $\{Ci'_1, Ci'_2, \ldots, Ci'_{j'}, \ldots, Ci'_s\} \in$ x_cont_inv_i' are disjoint sets.*

Proof. Same *continuing_inversion* pair does not belong to multiple $Ci'_{j'}$'s. This is so, because each *continuing_inversion* pair follows a unique path from i' to i by definition and each $Ci'_{j'}$ presents all those inversions where the last inversion started from the same index $t = i'$ producing the same inverted sequence, i.e., same prefix from index i' up to index i. Any two $Ci'_{j'}$'s say Ci'_1 and Ci'_2, if got split by Observation 4 somewhere between i' to i, then they can not have any common *continuing_inversion Pair*. □

Lemma 6. *At any index $i > i'$, the number of disjoint x_cont_inv set: $\{Ci'_1, Ci'_2, \ldots, Ci'_{j'}, \ldots, Ci'_s\} \in$ x_cont_inv_i' in the worst case is $(n - i)/k$.*

Proof. We define the worst case such that, the number of *x_cont_inv* sets from *x_cont_inv_i'* is maximized and *Four_Iteration_Loop* is called for each *x_cont_inv* set where pairing up operation is performed in each iteration of the loop. To make this happen, each of the existing $Ci'_{j'}$ must consist of four *continuing_inversion Pair* to produce at least $k = 4$ next agreed pairs. Using this approach, and by Observation 9 and 11 the Lemma is proved. □

So we have $\{C4_1, C4_2, \ldots, C4'_j, \ldots, C4_{(n-i)/k}\} \in$ *x_cont_inv_4* at iteration i. Then, we need to know the total size of *Ysibling* lists considering all the $C4'_j$'s. We have the following Observation, which basically follows readily following the arguments of Observation 11.

Observation 12. *At any index $i > i'$, $\{Si'_1, Si'_2, \ldots, Si'_s\} \in$ y_cont_inv_i' are disjoint sets.*

Observation 13. *The Four_Iteration_Loop is called each time $Ci'_{j'}$ encounters y_cont_inv in its Ysibling list (Steps 2.2.2, 3.2.2 in the algorithm)*

From Observation 13 we can say that, the number of calls to *Four_Iteration_Loop* is maximized when the number of *y_cont_inv* sets is maximized. We also want to ensure that pairing up operation is performed in each iteration of *Four_Iteration_Loop*. Thus we have the following Lemma which is identical to Lemma 6.

Lemma 7. *At any index* $i > i'$*, the number of disjoint y_cont_inv sets:* $\{Si'_1, Si'_2, \ldots, Si'_{j'}, \ldots, Si'_s\} \in$ *y_cont_inv_*i' *in the worst case is* $(n-i)/k$*.*

So we have $\{Si'_1, Si'_2, \ldots, Si'_{(n-i)/k}\} \in$ *y_cont_inv_*i' for each $1 \leq i' \leq i$ to be considered as *Ysibling* of $C4'_j$'s. To make it more simple, let us first consider *only the sets* $C4'_j$*'s yielding* $\alpha = A$. Then based on the arguments provided for $\alpha = A$, we can consider the scenario for all $\alpha \in \{A, T, C, G\}$. Now, the number of collection from *x_cont_inv_*4 each yielding A and having $k = 4$ *continuing_inversion Pair*s is $(n-i)/k^2$ (by similar argument as in Lemma 6). Let us call them $C4\_A_{j''}$ where $j'' = 1, 2, \ldots, (n-i)/k^2$. This is true for all *x_cont_inv_*i'. So we have the following two Observations.

Observation 14. *The number of distinct sets x_cont_inv from x_cont_inv_*i' *yielding A is* $(n-i)/k^2$ *at iteration i in the worst case, where* $1 \leq i' \leq i$ *. Let us call them* $Ci'\_A_{j''}$*, where* $j'' = 1, 2, \ldots, (n-i)/k^2$*.*

Observation 15. *The number of distinct sets y_cont_inv from y_cont_inv_*i' *yielding A is* $(n-i)/k^2$ *at iteration i in the worst case, where* $1 \leq i' \leq i$*. Let us call them* $Si'\_A_{j''}$*, where* $j'' = 1, 2, \ldots, (n-i)/k^2$*.*

At iteration i, a $C4\_A_{j''}$ can have matched pairs from any *y_cont_inv_m* where $1 \leq m \leq i$. But two different cases occurs as follows.

Case 1: *At iteration i, for each y_cont_inv_m, where* $1 \leq m < i' < i$*, the total number of y_cont_inv in Ysibling considering all* $Ci'\_A_{j''}s$ *is* $(n-i)/k^2$*. So considering all y_cont_inv_m, total number of calls to the* Four_Iteration_Loop *for this case is* $\sum_{m=1,\ldots,i'}(n-i)/k^2 = (i'-1)(n-i)/k^2$*.*

For simplicity, we first prove the Case 1 for $i' = 4 < i$, that is for all $C4\_A_{j''} \in$ *x_cont_inv_*4, by Lemma 8 and Lemma 9 below. Then based on the arguments provided for $i' = 4$, we can prove the case for all $i' < i$.

Lemma 8. *For* $1 \leq m < 4$*, each* $C4\_A_{j''}$ *can have multiple y_cont_inv sets in its Ysibling from the same y_cont_inv_m.*

Proof. Let us consider the sets $\{S1\_A_1, \ldots, S1\_A_{j''}, \ldots, S1\_A_{(n-i)/k^2}\} = y\_cont\_inv\_1$. They may be paired with different or the same x_end_inv pairs at index $4-1$. For each of those x_end_inv pairs in x, those matching sets $S1\_A_{j''} \in y\_cont\_inv\_1$ are paired with $ISA\_table\_x[A][4]$ and thus each $ISA\_table\_x[A][k]$ can pair with multiple number of collections from $y\_cont\_inv\_1$. See the Example 8 provided for Observation 7 for an illustration. □

Lemma 9. *For $1 \leq m < 4$, all sets $Sm\_A_{j''} \in y\_cont\_inv\_m$ that exist in the $Ysibling$ list of all these $C4\_A_{j''}$'s are disjoint.*

Proof. The same $Sm\_A_{j''}$ can not exists in the $Ysibling$ list of two different $C4_A_{j''}$s. We prove it by contradiction. Suppose, two different $C4\_A_1$ and $C4\_A_2$ align with the same $S1\_A_1$ at i. Pairing between $C4\_A_{j''}$ and $S1\_A_{j''}$ indicates an alignment of inversed x where the last inversion started from 4, with inversed y having the last inversion continuing from 1. Two $C4\_A_1$ and $C4\_A_2$ are disjoint by Observation 3. It implies that they present two inversed sequences that yield different base letters at some index $4 \leq i'' \leq i$ and that is why they were split into two by the *split case 2* (otherwise they would have belonged to the same set). If they align with the same $S1\_A_1$ at i, we get a contradiction, because $S1\_A_1$ presents all those inversions for which inversed sequences (prefixes) are the same from index 1 up to i, i.e., prefixes do not differ at any index i'', where $4 \leq i'' \leq i$. Thus the Lemma is proved. □

Now from Observations 14, 15, and Lemmas 8, 9, we can say, considering all $C4\_A_{j''}$, that the total number of y_cont_inv sets in $Ysibling$ from $y\_cont\_inv\_m$ is $(n-i)/k^2$ for each m, where $1 \leq m < 4$. So considering all m, in total we get $(4-1)(n-i)/k^2$ sets in the $Ysibling$. For each of these sets the *Four_Iteration_Loop* is called, and this is true for all $i' < i$. Thus case 1 is proved.

Case 2: *Considering all $y\_cont\_inv\_m$, where $4 \leq m < i$, the total number of calls to the* Four_Iteration_Loop *is $(i-i')$ for each $Ci'\_A_{j''}$, and considering all $Ci'\_A_{j''} \in x\_cont\_inv\_i'$, it is $(i-i')(n-i)/k^2$.* Again, for the sake of simplicity, we prove it for $i' = 4$ by Lemma 10. Later, based on the arguments provided for $i' = 4$, we can prove the case for all $i' < i$.

Lemma 10. *Each $C4\_A_{j''}$ can have only one y_cont_inv set from each $y\_cont\_inv\_m$, for $4 \leq m < i$, resulting a total of $i-4$ y_cont_inv sets.*

Proof. For index m after 4, all of $C4_A_{j''}$s will reside in $ISB\_table[m]$. We prove this Lemma by contradiction. Let us assume that at iteration i, $C4\_A_1$ is paired up with two sets say

$S6\_A_1$ and $S6\_A_2$ from $y\_cont\_inv\_6$, $(6 < i)$. By Observation 4 and Observation 6, all $S6_A_{j''}$s are disjoint. Two different $S6\_A_1$ and $S6\_A_2$ means two inversed sequence who get different at somewhere between index $6 \leq i' \leq i$ and thus get split by the Observation 4 (split case 2). But at iteration i, all pairs in $C4\_A_1$ are presenting the same inversed sequence from index 4 to i. $C4\_A_1$ does not change anywhere up to i, if it were changed then it would have been split into two child set say $C4\_A_{1'}$ and $C4\_A_{1''}$ from that index by split case 2. So we reach a contradiction. So $C4\_A_1$ can pair with only one collection say $S6\_A_1$. So the total number of y_cont_inv sets for each $C4\_A''_j$ is $(i-4)$. Thus the Lemma is proved. ☐

Therefore, considering all $C4_A_{j''}$s, the total number of y_cont_inv sets is $(i-4)(n-i)/k^2$ (using Observation 5). For each of these sets the *Four_Iteration_Loop* is called, and this is true for all $i' < i$. So case 2 is proved.

Lemma 11. *Total number of calls to the Procedure 5:* Four_Iteration_Loop *for* $ISB\_table[i]$ *at iteration* i *is* $O(ni^2/k)$

Proof. Continuing from the proof of Lemma 10, considering all $C4\_A_{j''}$'s, the number of y_cont_inv sets is $(4-1)(n-i)/k^2$ (by case 1)+$(i-4)(n-i)/k^2$ (by case 2)=$(i-1)(n-i)/k^2$. Finally, considering all $\alpha \in \{A, T, C, G\}$ the total number of y_cont_inv sets in their $Ysibling$ is $(i-1)(n-i)/k$.

Presence of y_end_inv pairs in $Ysibling$ lists of $C4_{j'}$ cannot dominate the total number of calls to the *Four_Iteration_Loop*, as, for each $C4'_j$, pairing up between its $x\_next\_atcg[]$ and $ISA\_table\_y[][i+1]$ is done once only, by Observation 7.

Therefore, considering all $x\_cont\_inv\_i'$, $i' < i$, the *Four_Iteration_Loop* is called $(i-1)(i-1)(n-i)/k = O(ni^2/k)$ times. So Lemma 11 is proved. ☐

Lemma 12. *Total number of calls to the* Four_Iteration_Loop *for* $ISA\_table[i]$ *at iteration* i *is* $O(i(n-i)/k)$.

Proof. $ISA\_table\_x[\alpha][i]$ gets $Ysibling$ pairs only if in the previous index $i-1$, some x_end_inv have matched pairs from y. Number of x_end_invs at index $i-1$ is i by Observation 10. In the worst case all of those x_end_invs have matched pairs in y. Let us see how many y_cont_inv sets each $ISA\_table\_x[\alpha][i]$ can have in their $Ysibling$s. Let us calculate for A first. At iteration i, we have y_cont_inv sets $Si'\_A_{j''} \in y\_cont\_inv\_i'$, where $1 \leq j'' \leq (n-i)/k^2$ (by Observation 6), for all $i' \leq i$. Let us think all of their parent collection were paired up with the x_end_inv pairs at index $i-1$. No $Si'\_A_{j''}$ will exist multiple times into the $Ysibling$ of

$ISA\_table\_x[A][i]$ by the Observation 6. So for A, $ISA\_table\_x[A][i]$ has a total of $(n-i)/k^2$ y_cont_inv sets from each $y\_cont\_inv\_i'$, where $1 \leq i' \leq i$. Thus in total, we have $i(n-i)/k^2$ y_cont_inv each having size $k = 4$ and yielding A. Considering all $\alpha \in \{A, T, C, G\}$, the total number of y_cont_inv sets from all $y\_cont\_inv\_i'$ is $i(n-i)/k$. So for each of these sets, the *Four_Iteration_Loop* can be called. Again, for each $ISA\_table\_x[\alpha][i]$, pair up step between $x\_next\_atcg[]$ and$ISA_table_y[\alpha][i]$ is executed once only by Observation 7. So the number of y_end_inv pairs in $Ysibling$ does not dominate the total number of calls to the *Four_Iteration_Loop*. Thus Total number of calls to the *Four_Iteration_Loop* for $ISA\_table[i]$ at iteration i is $O(i(n-i)/k)$. □

Observation 16. *Total run time taken by $ICA\_table[i]$ at iteration i is $O(k)$.* □

Lemma 13. *Worst case runtime of the algorithm is $O(n^4)$.*

Proof. Using the Observations and Lemmas provided above we explain the worst case run time for each steps of the algorithm.

Initialization:

It involves filling up the T_x, T_y, and pairing up the $ISA\_table\_x[][1]$ and $ISA\_table\_y[][1]$. So it takes $O(2n^2) + O(k) = O(n^2)$.

Iteration:

At each iteration $i = 1, 2, \ldots, n-1$, the algorithm calls Steps 1, 2, and 3.
Step 1: It needs $O(k)$ at each iteration i, by Observation 16.
Step 2: Processing $ISA\_table\_x[][i]$ depends on two factors:

1. *next_calculation* step: This is done in Steps 2.1 and 2.2.1. Step 2.1 takes $O(n-i)$ by Observation 8 and Lemma 5. Step 2.2.1. takes $O(i(n-i))$ by Observation 9 and Lemma 5. So the total time complexity is $n - i + in - i^2 = O(ni)$.

2. *Four_Iteration_Loop*: This is performed in Step 2.2.2 (always), Step 2.2.3 (conditionally) and once only for case 3 (under Step 2.2). Total number of calls by Step 2.2.2 and 2.2.3 is $O(i(n-i)/k)$ By Lemma 12. Again, pairing up operation is performed in each iteration. So each call to the *Four_Iteration_Loop* performs $k = 4$ pairing up operation. So the total number of pairing up operation is $O(i(n-i))$.

Step 2.3 takes $O(k)$ if all $ISA\_table\_x[\alpha][i]$ for $\alpha = \{A, T, C, G\}$ have non empty $Ysibling$ list. So for Step 2, the total time complexity at iteration i is $O(ni) + O(i(n-i)) = O(ni)$.
Step 3: Processing $ISB\_table[i]$ depends on three factors:

1. *next_calculation* step: This is done in Steps 3.1 and 3.2.1. The pairs in $x\_cont\_inv\_i''$ sets ($1 \leq i'' < i$) are responsible for forming the x_cont_inv sets of $\langle Xsibling : [x\_cont\_inv, x\_end\_inv]\rangle - \langle Ysibling \rangle$ rows of the $ISB\_table[i]$. Again pairs in $y\_cont\_inv\_i'$ are pointed by the $Ysibling$ list of these rows, where $1 \leq i' \leq i$. So by Lemma 5, the number of total steps here is $O(2i(n-i))$.

2. *Four_Iteration_Loop*: This is performed in Step 3.2.2 (always), Step 3.2.3 (conditionally) and once only for case 3. Total number of calls by Step 3.2.2 is $O(ni^2/k)$ By Lemma 11. Again, pairing up operation is performed in each iteration. So total number of pairing up operation is $O(ni^2)$. For each x_end_inv, the loop is called at step 3.2.3. But it is negligible because the total number of x_end_inv pairs considering all $Xsibling$s in $ISB\_table[i]$ is only $i-1$ by Observation 10.

3. Transferring step: This is done in Step 3.3. If all the $x\_next\_atcg[\alpha]$ of the $Xsibling$s created in Step 2.1, has non empty $Ysibling$ list for $\alpha = \{A, T, C, G\}$, then each of them are passed to $ISB\_table[i+1]$. So it takes $O(\frac{k(i-1)(n-i)}{k})$=$O(ni)$.

So for step 3, the total time complexity at iteration i is $O(2i(n-i)) + O(ni^2) + O(ni) = O(2ni(i+1)) = O(ni^2)$.

So Step 1, Step 2, and Step 3 take $O(k) + O(ni) + O(ni^2) = O(ni^2)$ for each iteration, resulting in $O(n^4)$ in total. Here we can see that Step 3 is the dominating step.

Termination:

Decision making takes $O(1)$ that just checks the emptiness of the $ICA\_table[n]$ (n =last index of the table).

So in total, worst case time complexity of the algorithm is $O(n^4)$. Thus Lemma 13 is proved. □

Now we deduce the run time for the average case. Here, we consider fifty percent match between the pairs of $\theta_x(x)$ and $\theta_y(y)$ at each iteration i. So we present the following Lemma 14 to define the maximized size of $cont\_inv\_i'$ considering the average case.

Lemma 14. *For the average case, at some iteration i, the size of any $cont\_inv\_i'$, where $i' < i$, is $(n-i')/2^{i-i'}$.*

Proof. When the $cont\_inv\_i'$ is passed from $ISA\_table\_x[][i']$ to $ISB\_table[i'+1]$ in iteration i', we perform *next_calculation* step $n-i'$ times by Step 2.1. But all of the next agreed pairs do not get matched pairs from y. We assume that fifty percent of the pairs in *cont_inv_i'* gets matched pairs for the next agreed pairs from y. So, in the next iteration, $i = i' + 1$, the size of *cont_inv_i'* becomes $\frac{(n-i')}{2}$ in the average case. Again, in iteration $i = i' + 1$, the new set *cont_inv_*$(i' + 1)$ may get introduced which again takes $n - (i' + 1)$ steps for *next_calculation* by Step 2.1. This pattern continues. Observing the pattern in Table 4.1, at some iteration i, the size of any *cont_inv_i'*, where $i' < i$, is $\frac{(n-i')}{2^{i-i'}}$. □

Iteration, i	1	2	3	4
Size of *x_cont_inv_1*	$(n-1)$	$\frac{(n-1)}{2}$	$\frac{(n-1)}{2^2}$	$\frac{(n-1)}{2^3} = \frac{n-1}{2^{4-1}}$
Size of *x_cont_inv_2*	–	$(n-2)$	$\frac{(n-2)}{2}$	$\frac{(n-2)}{2^2} = \frac{(n-2)}{2^{4-2}}$
Size of *x_cont_inv_3*	–	–	$(n-3)$	$\frac{(n-3)}{2} = \frac{(n-3)}{2^{4-3}}$
Size of *x_cont_inv_4*	–	–	–	$(n-4) = \frac{(n-4)}{2^{4-4}}$

Table 4.1: Size of *cont_inv_i'* set for average case at iteration $i > i'$

Lemma 15. *Average case runtime of the algorithm is $O(n^3)$.*

Proof. From previous Lemmas we see that Step 3 of the algorithm is the dominating step so we show how its run time gets reduced in the average case. The total number of *x_cont_inv* sets forming the *Xsiblings*, and the total number of *y_cont_inv* sets pointed by the *Ysibling* list of *Xsiblings*, are the main factors that determine the runtime of Step 3. All the observations remain the same as before. However, the size of *cont_inv_i'*, changes to $(n-i')/2^{i-i'}$, instead of $(n - i)$. We restate Case 1 and Case 2 below skipping the proofs.

Case 1: For each *y_cont_inv_m*, where $1 \leq m < i'$, number of *y_cont_inv* in *Ysibling* considering all $Ci'_{j'} \in$ *x_cont_inv_i'* is $\frac{(n-m)}{2^{i-m}k}$. So the total number of calls to the *Four_Iteration_Loop* at iteration i, for *x_cont_inv_i'* is $\frac{1}{k}\sum_{m=1,\ldots,i'} \frac{(n-m)}{2^{i-m}}$. Simplifying the term we get $O(n(\frac{2^{i'}}{k2^i}))$.

Case 2: Considering all *y_cont_inv_m*, where $i' \leq m < i$, the total number of calls to the *Four_Iteration_Loop* is $(i - i')$ for each $Ci'\_A_{j''}$. So considering all $Ci'\_A_{j''} \in$ *x_cont_inv_i'* it is $(i-i')\frac{(n-i')}{2^{i-i'}k^2}$. Therefore, considering all $Ci'_{j'} \in$ *x_cont_inv_i'*, the total number of calls to the *Four_Iteration_Loop*, at iteration i is $(i-i')\frac{(n-i')}{2^{i-i'}k}$.

Considering Case 1 and Case 2, at iteration $i > i'$, for each *x_cont_inv_i'*, the total number of *y_cont_inv* sets pointed by their *Ysiblings* is $n\frac{2^{i'}}{k2^i} + (i-i')\frac{(n-i')}{2^{i-i'}k}$. So, for $i' = 1, 2, \ldots, i-1$, we get $\sum_{i'=1,\ldots,(i-1)} n\frac{2^{i'}}{k2^i} + (i-i')\frac{(n-i')}{2^{i-i'}k} = O(\frac{ni}{k})$. So time complexity of Step 3 becomes $O(n^3)$ in the average case. Similarly, the time complexity of Step 2 becomes $O(n^2)$ as well in the average case. So in total the average case time complexity for the algorithm is $O(n^3)$. □

4.6 Space Complexity

The table T_x and T_y takes $O(n^2)$ space. We implement the $ISA\_x\_table[]$ and $ISB\_table[]$ as linked lists. So extra memory is not alocated in advance. Thus space complexity is dominated by the number of list elements $\langle Xsibling \rangle - \langle Ysibling \rangle$ and the size of $Ysibling$ for each such element in $ISA\_table\_x[]$ and $ISB\_table[]$. Processing at iteration i does not need the contents of column $i-1$ of any tables. So, only two columns are needed for those tables which can be used alternatingly. Therefore, as in the worst case, the total size of all the $Ysibling$ lists is $O(ni)$ and $O(ni^2)$ (by Lemmas 12 and 11) for $ISA\_table\_x[i]$ and $ISB\_table[i]$ respectively, so the theoretical worst case space complexity becomes $O(n^3)$. However, as in average case, worst case runtime of the algorithm is $O(n^3)$ (by Lemma 15), thus space complexity becomes $O(n^2)$.

4.7 Experimental Results

Theoretical worst case and average case time complexity of the algorithm are $O(n^4)$ and $O(n^3)$, and theoretical worst case and average case space complexity are $O(n^3)$ and $O(n^2)$, proven in Section 4.5 and Section 4.6 respectively. However, practical runtime of the algorithm in average and the worst case are $O(n^2)$ and $O(n^3)$ respectively. This is apparent from the experimental results reported in Table 4.2.

Table 4.2: Total number of steps taken by the algorithm for $n = 30, 50, 70, 90, 120$

n^3	Similarity / Length, n	Runtime									
		Result: YES									Result: NO
		100%	90%	80%	70%	60%	50%	30%	20%	Average	Average
27000	30	14849.18	8394.05	6419.14	5556.35	4967.8	4811	4239.89	4167.05	4436.30	3126.20
125000	50	70298.30	31995.65	21814.60	18522.4	16859.3	15980.85	15665.8	15344.95	14502.36	7299.40
343000	70	193353.05	72637.53	46755.95	39899.6	35267	34171.90	32686.6	32861.55	26055.48	20505.75
729000	90	412793.40	117034.50	84772.90	67255.35	61626.85	61766.45	56221.2	56567.7	49514.79	23087.36
1728000	120	980591.30	264070.30	160443.85	127785.05	117576.9	112994.70	110491.15	109703.55	94221.95	58573.14

In our experiments, x and y are selected such that both contain the same number of bases from the complement category, $A-T$ and $G-C$. We define the term *performance factor* (equivalent to the runtime), as a counter that keeps track of the total number of statements executed for finding the next agreed *Pair (t, r)* and pairing the matched agreed *Pairs*. So runtime of a test case is calculated by adding the performance factor of the dominating steps, i.e., Step 2 and Step 3 in the algorithm. For each length n (ranging from 30 to 120), we run the experiment under six categories (columns 3 to 8 of Table 4.2) based on the percentage of similarity between the input sequences x and y. Under each category,

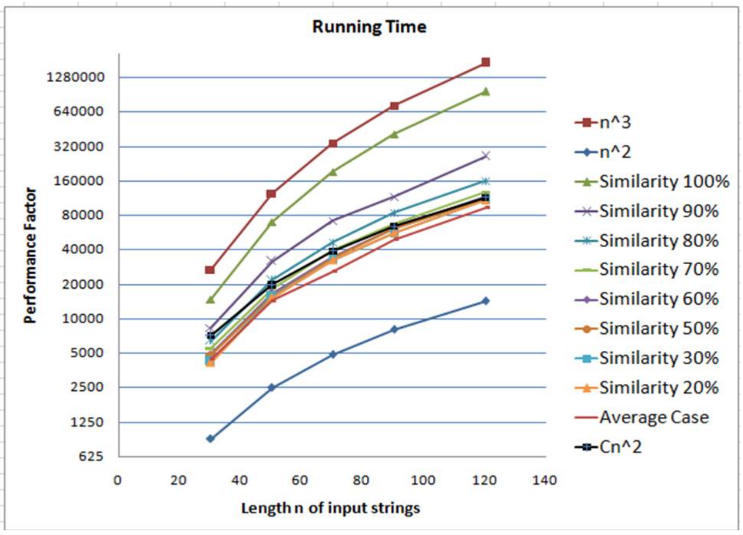


Figure 4.8: Time Complexity of our proposed algorithm

we generate twenty sets of test cases by randomly choosing x and y. Then we calculate the average runtime of the test cases. Worst case occurs when the sequences have a similarity of around 90% or more; otherwise the runtime becomes $O(n^2)$. This is also apparent from the graph in Figure 4.8. With the decrease in the similarity between x and y, the running time drops to $O(n^2) = Cn^2 \approx 7n^2$. Here, no comparison with previous works is provided as there exists no other works on our problem (algorithm by Cho et al. [21] is inaccurate thus not considered).

4.8 Conclusion

In this chapter we have mapped the consensus string problem under the inversion distance metric to the biomedical problem of detecting the allelic heterogeneity. The proposed algorithm finds the common ancestor sequence given two mutated sequences where mutation involves only non overlapping inversions. Future research endeavor could be directed towards other mutation operations such as insertion, deletion, etc. Finding minimum consensus string distance for two input sequences and improving time complexity remain as future works as well.

Chapter 5

Existence of Consensus String Under The Transposition Metric

In this chapter, we present a polynomial time algorithm for determining the *existence* of a Concensus String ($s^\star$), given two strings x and y of length n on an alphabet of size $k = 4$ (DNA bases A, T, C, G) under the distance metric called *non overlapping transposition.* Since the minimum distance d is not present as a parameter, our problem can be thought of as a relaxed version of the original Concensus String problem. In Section 5.1, we provide some definitions and observations necessary for presenting the algorithm. Then in Section 5.2 we discuss the main algorithm. We prove the correctness of our algorithm in Section 5.3. Then in Section 5.4 and Section 5.5, we discuss the running time and space complexity respectively. We show the experimental result in Section 5.6. Finally we conclude in Section 5.7 discussing some future research directions.

5.1 Definitions

We consider the biological operation *Transposition*, a genetic mutation in which two consecutive DNA segments of same size interchange their positions. We denote a transposition operation by (i, j) where the operation takes place between the index i to j of a DNA sequence, and the segment to be interchanged has size $\frac{j-i}{2}$ (transposition size $\frac{j-i}{2}$). For example, a transposition $(3, 8)$ over the DNA sequence $x = AG\ \underline{ACC}\ \underline{CTA}\ GTTCGAA$ results in $AG\ \underline{CTA}\ \underline{ACC}\ GTTCGAA$ (between index 3 to 8). *Transposition Sequence*, θ^T is defined as a set of the non overlapping transpositions. So the *Transposed Sequence*, $\theta^T(x)$ is the resultant DNA sequence after applying the set of transpositions, θ^T over x. Let us consider the transposition sequence $\theta^{T'} = \{(3, 8), (10, 15)\}$, and the same DNA sequence

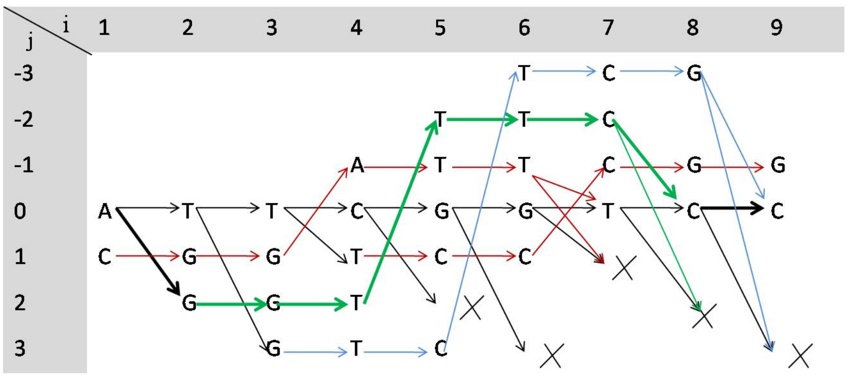


Figure 5.1: $P - graph$ for sequence $x = ATTCGGTCC$ with transposition size 3

$x = AG\mathbf{ACCCTA}G\mathbf{TTCGAA}$. Here, two transposition operations $(3, 8)$ and $(10, 15)$ are applied on x. Then $\theta^{T'}(x) = AG\mathbf{CTAACC}G\mathbf{GAATTC}$. Again, $\theta^{T'}(x)$ upto index 8 is $AGCTAACC$.

In order to find out Concensus Strings based on Transposition metric, we use the $P-graph$ proposed by Pritam et al. [3, 2, 1]. Given a gene sequence $X = X_1, X_2, \ldots, X_n$, and a transposition size k, a $P{-}graph$, denoted by $P^G = (V^P, E^P)$, is a directed graph. The vertex set V^P can be partitioned into three disjoint vertex sets namely V_d^Pown, V_m^Piddle, V_u^Pp such that $k \leq up \leq 1$, $1 \leq down \leq k$, and $middle = 0$. The partition is defined in a $(2k+1) \times n$ matrix M as follows. For the sake of notational symmetry, we use $M[up]$, $M[middle]$ and $M[down]$ to denote respectively the rows $M[k], \ldots, M[1]$, $M[0]$ and $M[1], \ldots, M[k]$ of the matrix M, respectively. Please refer to Figure 5.1 for $k = 3$.

Here the path presented by middle row, that is $M[0]$ presents the original gene sequence. Following the paths in $M[up]$ or $M[down]$ we get the transposed version of the input gene sequence. Given two sequences x and y of length n, we use $M_x[2k+1][n]$ and $M_y[2k+1][n]$ to denote the sets of all possible transpositions of x and y respectively, where k is the transposition size. For example, all possible transposed sequences of x can be found by $M_x[2k+1][n]$. Here the bold path shows the transposed sequence $AGGTTTCCC$ found by $\theta^{T'} = \{(2, 7)\}$.

In practice different sized transposition mutation event might occur simultaneously in a gene sequence. That is, we can have multiple values for the transposition size k, where $1 \leq k \leq k_{max} \leq n/2$. So by superimposing $P-Graph$s for different values of k, we can have the $P-Graph$ showing all possible transpositions of the respective sequence considering all possible transposition size k. For simplicity we show the $P-Graph$ for $k \in \{1, 2, 3\}$ for the same sequence $x = AGACCCTAG$ in Figure 5.2.

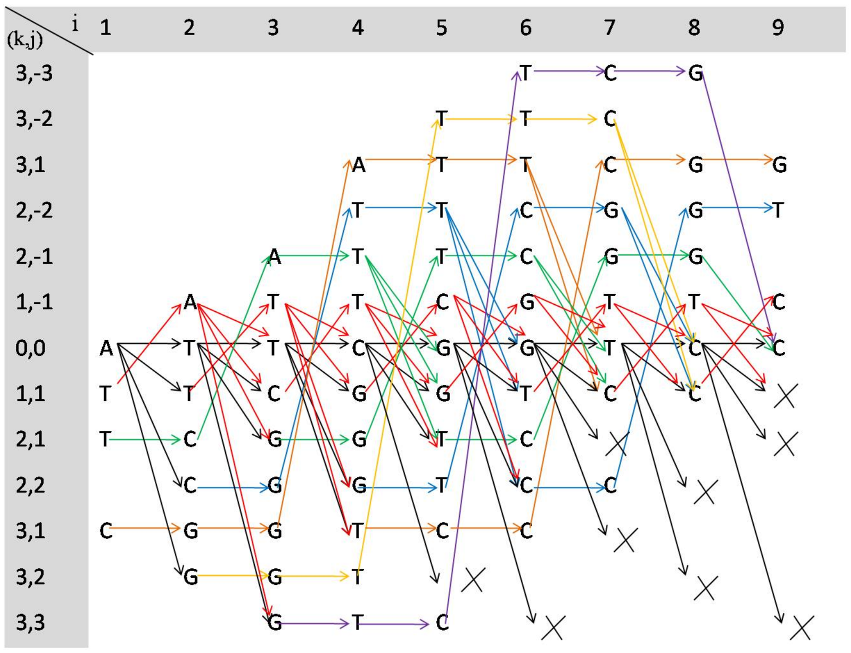


Figure 5.2: $P-graph$ for sequence $x = ATTCGGTCC$ with transposition size $k \in \{1, 2, 3\}$

We define each non-empty $M[k, j][i]$ as a transposition fragment, where k=block size / transposition size, $j \in \{up, mid, down\}$ and i is the index of the sequence. The column $M[i]$ actually presents all possible gene bases at index i considering all possible transpositions or no transposition. Just like the case of inversion fragment presented in previous chapter, from one transposition fragment $M[k, j][i]$, we can find the next agreed transposition fragment according to following rules:

1. **if** $j = mid$,

   (a) if no transposition at index $i + 1$, then $M[0, 0][i].next = M[0, 0][i + 1]$

   (b) if new transposition starts at index i+1, then for each value of $k' \in \{1, 2, 3, \ldots, k\_max\}$, $M[0, 0][i].next = M[k', j'][i + 1]$, where,

$$j' = \begin{cases} k' & \text{if } (i\%k' = 0) \\ i\%k' & \text{Otherwise} \end{cases} \tag{5.1}$$

2. **if** $j = up$,
   if $(i\%k = abs(j) - 1)$, it indicates the end of the ongoing transposition at $M[k, j][i]$. So the next agreed fragment,

(a) if no transposition at index $i+1$, $M[k,j][i].next = M[0,0][i+1]$.

(b) if we consider the start of new transposition at index $i+1$, then for each value of $k' \in \{1,2,3,\dots,n/2\}$,
$M[k,j][i].next = M[k',j'][i+1]$, where,

$$j' = \begin{cases} k' & \text{if } (i\%k' = 0) \\ i\%k' & \text{Otherwise} \end{cases} \tag{5.2}$$

Otherwise, if $(i\%k \neq abs(j) - 1)$, it indicates the ongoing transposition at $M[k,j][i]$. So, the next agreed fragment of $M[k,j][i]$ is $M[k,j][i+1]$

3. **if** $j = down$, for any k,

$$M[k,j][i].next = \begin{cases} M[k,-j][i+1] & \text{if } (i\%k = abs(j)-1)\text{, level change/start of the} \\ & \text{second block (of size } k\text{) in the ongoing transposition} \\ M[k,j][i+1] & \text{otherwise, just ongoing transposition} \end{cases} \tag{5.3}$$

One complete agreed sequence represents a transposed sequence. We need to find out common agreed sequences given M_x and M_y.

$Pair(t,r)$ which was used in inversion (in previous chapter), is actually not necessary for transposition. It was used for the algorithm using inversion because the same fragment could belong to several ongoing inversions starting from several different indexes. But here, for transposition, the $P - graph$ is built in a way such that one transposition fragment belongs to only one ongoing transposition starting from one unique index. Thus transposition fragment itself is enough for keeping track of the alignments between $\theta_x^T(x)$ and $\theta_y^T(y)$. For the same index i, if a transposition fragment in M_x and another in M_y, yield the same $\alpha \in \{A,T,C,G\}$, and the respective transposed sequences $\theta_x^T(x)$ and $\theta_y^T(y)$ up to i is the same, then those two fragments are called *Matched Fragments* and denoted as $\langle Xsibling\rangle - \langle Ysibling\rangle$.

We define, *ending_transposition* fragment at index i such that it indicates the ending or completion of an ongoing transposition at index i (which started at index $i-2k$) for some $\theta_x^T(x)$. We also define $M[mid,mid][i]$ as *no_transposition* fragment at index i. Similarly, we define, *cont_trans_i'*, the set of row indexes of the transposition fragments presenting $\theta_x^T(x)$ where the current ongoing transposition started at index i' and continue through the index i, where $i' \leq i \leq i' + 2k$.

We define S_x and S_y to be the sets of all possible transposition sets θ^T over x and y respectively. In general, $\theta_x^T \in S_x$ and $\theta_y^T \in S_y$ are used to present the matching phase. Deciding whether any consensus sequence exists between two given DNA sequences x and y having the same length n, involves finding out the existence of common agreed sequences of x and y. For this purpose we track the matched fragments between M_x and M_y for each index or column $i = 1, 2, \dots, n$. The set of matched fragments are denoted as $\langle Xsibling \rangle$ - $\langle Ysibling \rangle$ for the ease of representation. Both $Xsibling$ and $Ysibling$ may contain one or more fragments.

In the rest of the section, we define some table like data structures that will be used in our algorithm. Each table will record some information of the matched fragments and will be named based on the type of $\theta^T(x)$ at each column i. Column i of each table presents some alignment of $\theta_x^T(x)$ and $\theta_y^T(y)$ up to index i.

TSA_table_x[ ][i] - Transposition Started At i. This table presents an alignment of $\theta_x^T(x)$ (upto i), having the last transposition ended at $i-1$, and a new transposition starting from i or no transposition at i. with $\theta_y^T(y)$ (upto i), having the last transposition started before or at i, still continuing or ended at i or no transposition at i. $TSA\_table\_x[][i]$ holds $b = 4$ rows, one for each of the base letters $\alpha \in \{A, T, C, G\}$ such that, $TSA\_table\_x[\alpha][i]$ keeps the (k, j)s (the row indexes of $M\_x[k, j][i]$) in $Xsibling$, that yields the base α, and indicates starting of a new transposition at index i ($j \in down$) or no transposition at i ($j = mid$). With such fragments of M_x, it keeps corresponding matching fragments from M_y in $Ysibling$.

The $Xsibling$ in particular consists of two members only. One is the *mid* fragment which actually indicates no transposition at index i and a set of fragments in *down* which indicates start of new transposition operation at index i. So we can call the first one as $x\_no\_trans\_i$ and the second one as $x\_cont\_trans\_i$.

Initially $Ysibling$ is empty. In the matching phase, $Ysibling$ maintains a list of pointers to the matched fragments of $Xsibling$ in T_y, and is categorized into two types, namely, single *ending_transposition* fragment called *y_end_trans* and single *no_transposition* fragment called *y_no_trans* (both belongs to Type 1) and *cont_trans* called *y_cont_trans* (Type 2) where all fragments represent the ongoing transpositions started at $i' \leq i$.

Now we explain the intuition behind keeping these records. Both types of pointers ($Ysibling$) mentioned above are considered as matched fragments of *x_cont_trans* set and kept in TSA_table_x[ ][i]. But for *x_no_trans*, only Type 2 pointers are considered as the matched fragments in this table. For each Type 1 pointer, i.e., *y_end_trans* and *y_no_trans* in $Ysibling$ list, we keep a separate record $\langle Xsibling \rangle$ - $\langle Ysibling \rangle \equiv \langle x\_no\_trans \rangle$ - $\langle y\_end\_trans \rangle$ or $\langle x\_no\_trans \rangle$ - $\langle y\_no\_trans \rangle$ in the $TCA\_table[i]$. Though this creates redundancy but this separation makes the data structure conceptually simpler and keeps the final decision checking simple at the end of the algorithm. Please refer to the Figure 5.5 for an illustration.

TCA_table[i] - Transposition Completed at i. This table holds rows of $\langle Xsibling \rangle$-$\langle Ysibling \rangle \equiv \langle (k', j') \rangle$-$\langle (k'', j'') \rangle$ presenting an alignment of $\theta_x^T(x)$ with $\theta_y^T(y)$ up to i, where the last transposition in θ_x^T and θ_y^T started at index $i - 2k'$ and $i - 2k''$ respectively, and both ends at i, or none has transposition at index i. So j' or j'' should be *up* or *mid*. And both matched fragments from x and y will have $TSA\_x[][i+1]$ and $TSA\_y[][i+1]$ respectively as the next agreed fragments.

TSB_table[i] - Transpositions Started Before i. It holds rows $\langle Xsibling \rangle$ - $\langle Ysibling \rangle$ just as before presenting alignments of $\theta_x^T(x)$ yielding α at index i (but having the last transposition started **before** i, and still ongoing or ended at i), with $\theta_y^T(y)$ yielding the same base letter α at index i (having the last transposition started before or at i, and still ongoing, or ended, or no transposition at i). So $Xsibling$ of $TSB\_table[\alpha][i]$ keeps the (k, j)s for x, where j should be *up* or *down* but not *mid*. And the corresponding $Ysibling$ keeps the (k', j')s for y, where j' can be *up*, *down*, or *mid*. Structure of $Ysibling$ and the intuition behind the records are the same as that in $TSA\_table\_x[][i]$.

TSA_table_y[][i]. It contains $\langle Ysibling \rangle \equiv \langle y\_no\_trans, y\_cont\_trans \rangle$ just like the $Xsibling$ in $ISA\_table\_x[][i]$. This $Ysibling$ is actually get pointed by the $Ysibling$ lists of $Xsibling$s, at $TSA\_table\_x[][i]$, $TSB\_table[i]$ and $TCA\_table[i]$.

5.2 The Algorithm

Common transposed sequences between x and y are computed by tracking the matched pairs between M_x and M_y from column $i = 1$ to n. The following procedures are used in our algorithm.

Procedure 6. ***Next_Calculation(*j*,* i*,* k*,* M_x*):*** If the input fragment $M_x[k, j][i]$ is of type *cont_trans* (continuing transposition), it returns the row index of one unique next agreed fragment. Otherwise, if the input fragment is of type *ending_transposition* or *no_transposition*, it returns the pointer to the $TSA\_table\_x[][i+1]$ as a new transposition is supposed to start from $i + 1$. Similar actions are performed for y if M_y is the input.

Procedure 7. ***next_calculation_collection(x_cont_trans, x_next_atcg[], i):***
It finds the next agreed fragments of *x_cont_trans* and keep those in a child table *x_next_atcg*[] such that *x_next_atcg*[α] holds the agreed fragments yielding α. For example, suppose, *x_cont_trans* $= \langle r_1, r_2, \ldots, r_p \rangle$. For each of these fragments we call *next_calculation*(r', i, k, M_x), $r' = 1, 2, \ldots, p$. Each time as soon as one unique next agreed fragment is returned, we add that to *x_next_atcg*[] as follows.

case 1: If the next agreed fragment is a *cont_trans* fragment, yielding α, then insert into *x_cont_trans* of *x_next_atcg*[α].

case 2: If the next agreed fragment ends at $i + 1$ or is subject to no transposition at index $i + 1$ and yield α, then we assign this fragment to *x_end_trans* or *x_no_trans* of *x_next_atcg*[α].

Procedure 8. ***PairUp_xColl_yColl(collection_x, collection_y, i):*** This step is called at iteration i, with the matched fragments for index i+1 as input. It sets the $\langle$*collection_y*$\rangle \equiv \langle$*y_cont_trans*, *y_end_trans*, *y_no_trans*$\rangle$ as *Ysibling* of $\langle$*collection_x*$\rangle \equiv \langle$*x_cont_trans*, *x_ending_trans*, *x_no_trans*$\rangle$. Thus it lets the alignment (up to i) of $\theta_x^T(x)$ and $\theta_y^T(y)$ proceed one step forward, i.e., from i to $i + 1$. It executes following steps.

step a: Insert the pairs like (*x_end_trans*,*x_no_trans*)$\times$(*y_end_trans*,*y_no_trans*) into *ICA_table*[$i + 1$].

step b: Insert a pointer to the *y_cont_trans* into the *Ysibling* list of *collection_x*.

step c: Insert a pointer to the *y_end_trans* into the *Ysibling* list of *collection_x*.

step d: Insert a pointer to the *y_no_trans* into the *Ysibling* list of *collection_x*.

Procedure 9. ***PairUp_xColl_ySingle(collection_x, single_y, i):*** It works as above but here the *single_y* is a row index (k, r) of single fragment. If both *collection_x* and *single_y* are nonempty (*Compatibility Check*), it performs the following steps.

step a: If *single_y* is an *ending_transposition* or *no_transposition* and *collection_x* also has *x_end_trans* or *x_no_trans*, then pair them up and insert into *ICA_table*[$i + 1$]

step b: Insert a pointer to *single_y* into the *Ysibling* list of *collection_x*.

Procedure 10. ***Four_Iteration_Loop(table_x, table_y, i):***
It pairs up the *Xsibling* in *table_x* with the *Ysibling* in *table_y*. For each base letters $\alpha \in \{A, T, C, G\}$, if *table_x*[$\alpha$] has non empty *Xsibling* and *table_y*[α] has non empty *Ysibling* (*Compatibility Check*), then it calls *PairUp_xColl_yColl*(*collection_x*, *collection_y*, i) with *collection_x*=*table_x*[α], and *collection_y*=*table_y*[α].

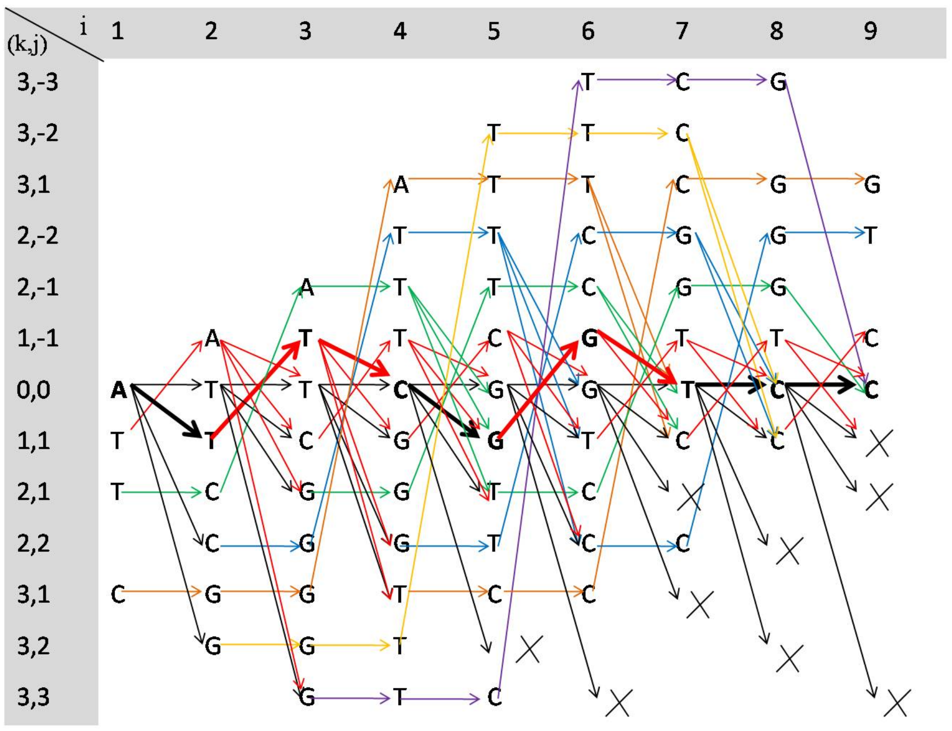


Figure 5.3: $P-graph$ for sequence $x = ATTCGGTCC$ with transposition size $k \in \{1, 2, 3\}$

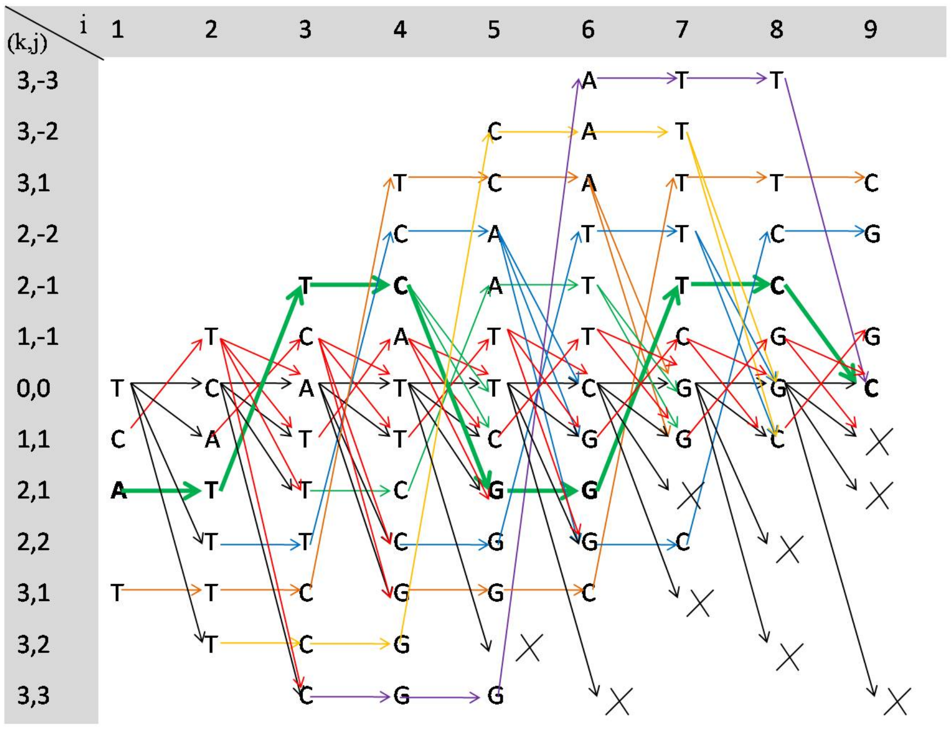


Figure 5.4: $P-graph$ for sequence $y = TCATTCGGC$ with transposition size $k \in \{1, 2, 3\}$

Now we explain the algorithm using the procedures stated above. The main algorithm iterates over $i = 1$ to $n-1$. The column i of each of the tables described above actually represents the alignment of $\theta_x^T(x)$ and $\theta_y^T(y)$ up to index i for some θ_x^T and θ_y^T. So at each iteration i, it processes the rows in three tables: $TCA\_table[i]$, $TSA\_table[][i]$, and $TSB\_table[i]$ to calculate the next agreed fragments, pair up the matched agreed fragments and insert those into the column $i+1$ of the appropriate table. If for any row $\langle Xsibling\rangle$-$\langle Ysibling\rangle$, next agreed fragments of $Xsibling$ does not get matched fragments from next agreed fragments of $Ysibling$, then it means no alignment with the transposed sequence of x presented by that $Xsibling$ exists in y. Thus this alignment $\langle Xsibling\rangle$-$\langle Ysibling\rangle$ is not passed forward anymore and is rather dropped here. We will explain the algorithm using an illustrative example. Consider, $x = ATTCGGTCC$, $y = TCATTCGGC$ and transposition size $k \in \{1, 2, 3\}$ given in Figures 5.3 and 5.4. One of the consensus sequences between those is $ACCTGACAG$.

5.2.1 Initialization

$TSA\_table\_x[1]$ and $TSA\_table\_y[1]$ are shown in the Figure 5.5. It executes Procedure 10, i.e., $Four\_Iteration\_Loop$ to start aligning x with y by pairing up these two tables. While calling the procedure, input parameters are set as: $table\_x = TSA\_table\_x[1]$, $table\_y = TSA\_table\_y[1]$, and $i = 1$.

5.2.2 Iteration

For each iteration $i = 1, 2, \ldots, n-1$, following steps are performed.

Step 1

Process TCA_table[i]: For the first row $\langle Xsibling\rangle$-$\langle Ysibling\rangle$=$\langle(k', r')\rangle$-$\langle(k'', r'')\rangle$, we call Procedure 1, i.e., $next\_calculation(r', i, k', T\_x)$ and $next\_calculation(r'', i, k'', T\_y)$. They return pointers to $TSA\_table\_x[i+1]$ and $TSA\_table\_y[i+1]$ respectively. After that, we call the $Four\_Iteration\_Loop(TSA\_table\_x[i+1], TSA\_table\_y[i+1])$. Other rows of $TCA\_table[i]$ are not processed as they involve doing the same assignments (according to the Merging Case 1 explained later in Observation 19).

α	x_no_trans	x_cont_trans	Ysibling
A	(0,0)		
T		<(1,1), (2,1)>	
C	-	<(3,1)>	-
G	-	-	-

TSA_table_x[1]

α	y_no_trans	y_cont_trans
A		<(2,1)>
T	<(0,0)>	<(3,1)>
C		<(1,1)>
G		

TSA_table_y[1]

(a)

Xsibling	Ysibling
-	-

TCA_table[1]

x_end_trans	x_cont_trans	Ysibling
-	-	-

TSB_table[1]

α	x_no_trans	x_cont_trans	Ysibling
A	(0,0)		<(2,1)>
T		<(1,1), (2,1)>	(0,0), <(3,1)>
C	-	<(3,1)>	<(1,1)>
G	-	-	-

TSA_table_x[1]

(b)

Figure 5.5: (a)The condition of *TSA_tables* Before Initialization; (b)After Initialization the *ysibling* fields of *TSA_table_x*[1] are updated, that is the alignment between x and y is initiated. The *TSB_table* is empty as it should be since no transposition exists that starts before index 1. The *TCA_table* is empty as well, since in this example no alignment exist where a transposition completes at index 1.

α	x_no_trans	x_cont_trans	Ysibling
A	(0,0)		<(2,1)>
T		<(1,1), (2,1)>	(0,0), <(3,1)>
C	-	<(3,1)>	<(1,1)>
G	-	-	-

TSA_table_x[1]

α	x_end_trans	x_cont_trans	Ysibling
A	(1,-1)		
T			
C		(2,1)	
G			

X_next_atcg[]

Figure 5.6: Demonstration of Step 2.1 for $\alpha = T$ in iteration 1

Step 2

Process TSA_table_x[][i]: For each $\alpha \in \{A, T, C, G\}$ we perform Step 2.1, Step 2.2 and Step 2.3.

Step 2.1

It calls Procedure 4, with *x_cont_trans* of *TSA_table_x*[α][i], which finds its next agreed fragments and keeps those in a child table *x_next_atcg*[] (see Figure 5.6)

Step 2.2

For each list item $Ysibling[p]$, in this step we find the alignment of the fragments in $x\_next\_actg[]$ (calculated in the previous step) with the next agreed fragments found from $Ysibling[p]$. We need to deal with one of the following cases.

Step 2.2 Case 1. The $Ysibling[p]$ is of type y_cont_trans having $size > 1$ (Step 2.2.1 to Step 2.2.3):

Step 2.2.1: If $y\_next\_atcg[]$ of $Ysibling[p]$ is not calculated yet, then call Procedure 4, i.e., $next\_calculation\_collection(Ysibling[p], y\_next\_actg[], i)$.

Step 2.2.2: Now both the $x\_next\_atcg[]$ and $y\_next\_actg[]$ are ready to be paired up. So we call the *Four_Iteration_Loop*($x\_next\_atcg[]$, $y\_next\_atcg[]$).

Step 2.2.3: If $Xsibling$ has x_end_trans or x_no_trans fragment, and $y\_next\_actg[]$ has not been paired with $ISA\_table\_x[][i+1]$ yet (*Merging Case 2* explained later in Observation 20), then pair them up by calling *Four_Iteration_Loop(*$ISA\_table\_x[][i+1]$, $y\_next\_actg[]$*)*.

Step 2.2 Case 2. The $Ysibling[p]$ is of type y_cont_trans having $size = 1$ (Step 2.2.4 to Step 2.2.6):

Step 2.2.4: We call $next\_calculation(r', i, M_y)$, where $M_y[r'][i]$ is the y_cont_trans fragment. Let the returned unique next agreed fragment yield α and name it $fragment\_y$.

Step 2.2.5: We call $PairUp\_xColl\_ySingle(x\_next\_actg[\alpha], fragment\_y, i)$.

Step 2.2.6: If $Xsibling$ has x_end_trans or x_no_trans, and $fragment\_y$ has not been paired up with $TSA\_table\_x[\alpha][i+1]$ yet (*Merging Case 2*), then we call $PairUp\_xColl\_ySingle(ISA\_table\_x[\alpha][i+1], fragment\_y, i)$. Please refer to Figure 4.5 for an illustration.

Step 2.2 Case 3. If $Ysibling[p]$ is of type y_end_trans or y_no_trans: If $x\_next\_atcg[]$ has not been paired up with $TSA\_table\_y[][i+1]$ yet (*Merging Case 3*, explained later in Observation 21), then we call the procedure *Four_Iteration_Loop* with input tables: $x\_next\_atcg[]$ and $TSA\_table\_y[i+1]$. Please refer to Figure 4.5.

Step 2.3

Update the TSB_table[i+1]: For each new $x\_next\_atcg[\alpha]$ created in Step 2.1, if it has non empty $Ysibling$ list, then we insert it into $TSB\_table[i+1]$ as new rows, where

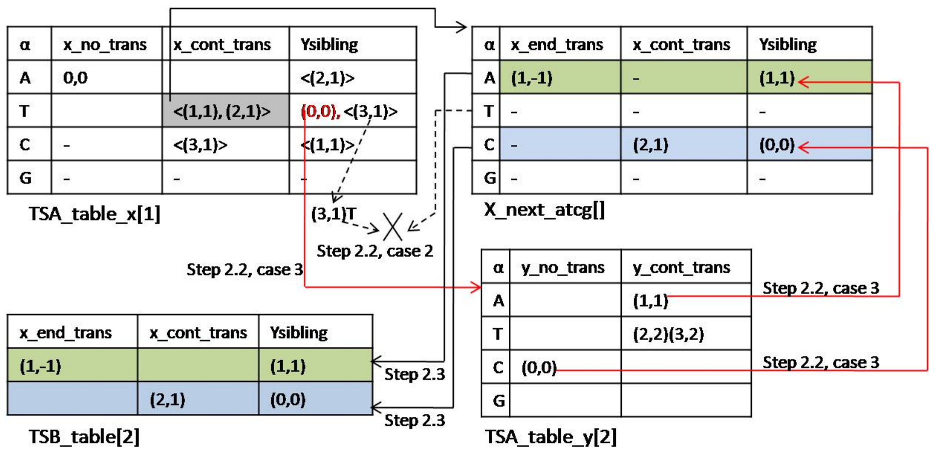


Figure 5.7: Demonstration of Steps 2.2 and 2.3 for $\alpha = A$ in iteration 1

$\alpha \in \{A, T, C, G\}$. Please see Figure 4.5 for an illustration.

Step 3

Process TSB_table[i]: For each row p of $TSB\_table[i]$: $\langle Xsibling \rangle$ - $\langle Ysibling \rangle$, we execute the Steps 3.1, 3.2, and 3.3. They are identical to Step 2.1, 2.2, 2.3 except the fact that the row items $TSB\_table[p][i]$ are used instead of $TSA\_table\_x[\alpha][i]$.

5.2.3 Termination

After the iterations complete, if the $TCA\_table[n]$ contains no row, we return NO indicating the absence of any consensus sequence between x and y. Otherwise we return YES, indicating the existence of some consensus sequence between x and y.

5.3 Correctness of the Algorithm

Correctness of the algorithm is proven by Lemma 16 and Lemma 17 by showing that no valid alignment is missed and invalid alignments are cancelled as soon as detected. Necessity and sufficiency of the termination step of the algorithm is proven in Lemma 18. Since the observations and lemmas are similar to those for inversion (Section 4.4), so detailed explanation with example is not provided.

We observe that each row at column i of each table actually presents an alignment between $\theta^T_x(x)$ and $\theta^T_y(y)$ up to index i. For Lemma 16 we need the following two observations.

Observation 17. ***Split case 1:*** *One alignment is split into multiple new alignments when ending of any ongoing transposition (of that alignment) is reached.* This case is ensured by the step b of Procedure 4, step a of Procedure 2 and 3, and Steps: 2.2.3, 2.2.6, 3.2.3, 3.2.6, case 3 (under Step 2 & 3) in the algorithm. For an illustration see the Example 10 below.

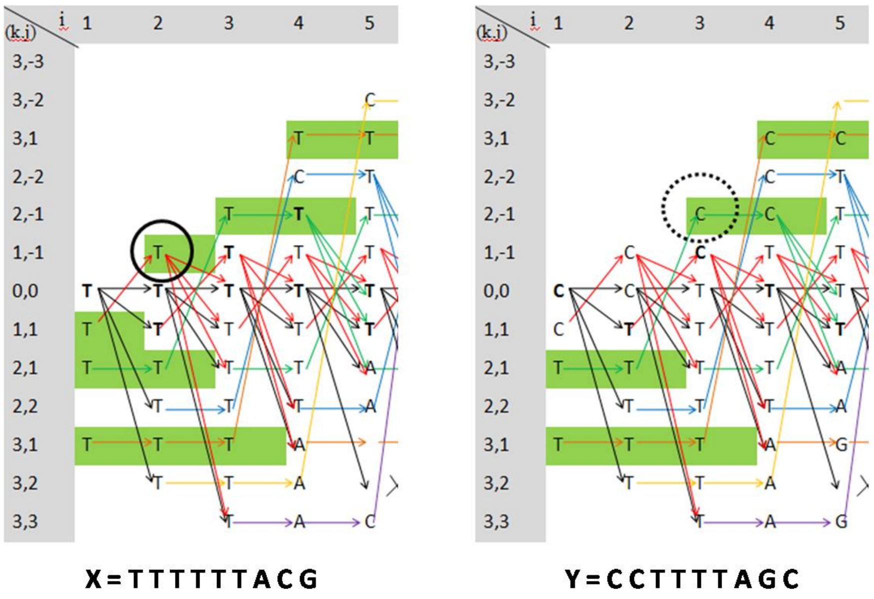


Figure 5.8: Split case 1 (marked by solid circle) and Split case 2 (marked by dotted circle)

Example 10. In this example we explain Observation 17 with the help of Figure 5.8. First, we explain the initialization. Please refer to the shaded alignments only. At the initialization step, the transposition fragment $M_x[1,1][1]$, $M_x[2,1][1]$ and $M_x[3,1][1]$ (presenting the base T) reside in the $Xsibling$ of $TSA\_x[T][1]$ as $x\_cont\_trans\_1$ (we ignore the alignment with $M[0,0][1]$ for this example). Similarly, the matched transposition fragments from y, namely $M_y[2,1][1]$ and $M_y[3,1][1]$ are aligned with $x\_cont\_trans\_1$ thus included in the $Ysibling$ list of $x\_cont\_trans\_1$ as y_cont_trans. Now we for the next iterations, we consider only the alignment between $x\_cont\_trans\_1$ and $y\_cont\_trans\_1$. Let us denote this alignment as A_T.

In iteration 1, these alignment A_T is to be passed one step forward. Here, the next calculation step on $x\_cont\_trans\_1$, returns next agreed fragments $M_x[1,-1][1]$,$M_x[2,1][1]$, $M_x[3,1][1]$. Similarly, the next calculation step on $y\_cont\_trans\_1$, returns next agreed fragments $M_y[2,1][1]$, $M_y[3,1][1]$. one transposition fragment $M_x[1,-1][1]$ of $x\_cont\_trans\_1$ reach ending at index 2. So we exclude it from $x\_cont\_trans\_1$ and add to x_end_trans. Then we pass this alignment one step forward by inserting $\langle Xsibling = [x\_end\_trans], x\_cont\_trans\_1\rangle - \langle Ysibling\rangle \equiv \langle[(1,-1)],(2,1),(3,1)\rangle - \langle(2,1),(3,1)\rangle$ into the $TSB\_table[2]$.

In iteration 2, while processing the alignment A_T residing in $TSB\_table[2]$, we find that the transposition fragment $M_x[1,-1][1]$ reaches ending at index 2. Thus from this point, *we split the alignment into two alignments* (note the solid circle marked around $M_x[1,-1][1]$). We call them A_T_a and A_T_b. Let us denote the next agreed fragments of $y\_cont\_trans\_1$ returning T base as $Ysib_T\_1 \equiv \langle(3,1)\rangle$. The next agreed fragments of $x\_cont\_trans\_1$ (all are yielding base T) who are still ongoing, are denoted as $Xsibling \equiv \langle(2,-1),(3,1)\rangle$. Now, in this iteration, we insert one row $\langle Xsibling\rangle - \langle Ysib_T\_1$ into the $TSB\_table[3]$ representing alignment A_T_a. We also include $Ysib_T\_1$ as $Ysibing$ of the $TSA\_table[T][3]$ which indicates another alignment A_T_b. In this way we split the alignment when ending of any ongoing transposition of that alignment is reached.

Observation 18. ***Split case 2:*** *Alignments can be split before reaching the ending if new prefix appears.* This happens when the next agreed fragments differ by yielding base letter $\alpha \in \{A, C, T, G\}$. This split is ensured by the strategy followed in Procedure 4, 2 and 3. For clarification see the Example 11 below.

Example 11. In this example we explain Observation 18 with the help of Figure 5.8. Please refer to the previous example 10. In iteration 2, while calculating the next agreed fragments of $Ysibling \equiv \langle(2,1),(3,1)\rangle$, we get $M_y[2,-1][3]$ yielding base C and $M_y[3,1][3]$ yielding base T. Here no transposition has reached ending but two transposition are yielding different bases. *That is why we split the alignment into two as* $Ysib_T\_1 = \langle(3,1)\rangle$ *and* $Ysib_C\_1 = \langle(2,-1)\rangle$ (note the dotted circle marked around $M_y[2,-1][3]$). Only $Ysib_T\_1$ has matched fragments from corresponding $Xsibling$. So, $Ysib_T\_1$ is passed forward to $TSB\_table[3]$ and $TSA\_table[T][3]$.

Lemma 16. *No valid alignment is missed.*

Proof. Each row at column i of each table presents an alignment between $\theta^T_x(x)$ and $\theta^T_y(y)$ up to index i. Based on next agreed fragments, if necessary we split that alignment into multiple new alignments as explained in Observations 17 and 18. Thus no valid alignment is missed. □

Lemma 17. *Invalid alignments that is agreed sequences of x not existing in y (or vice versa) are cancelled as soon as detected.*

Proof. If in iteration i, for an alignment $\langle Xsibling\rangle$-$\langle Ysibling\rangle$, next agreed fragments of $Xsibling$ get no matched fragments from the next agreed fragments of $Ysibling$, then the alignment is not passed forward and rather dropped immediately. This case is ensured by the *Compatibility Check* executed inside the Procedures 3 and 5. □

We illustrate Lemma 17 using Example 12 below.

Example 12. We explain Lemma 17 using the example in Figure 5.8. Please refer to the Example 11. Here, in iteration 2, from $Ysibling$, we get $Ysib_T\_1 = \langle(3,1)\rangle$ and $Ysib_C\_1 = \langle(2,-1)\rangle$ presenting two sequences TTC and TTT respectively. Here only the $Ysib_T\_1$ is passed forward by inserting into the $TSB\_table[3]$ and $TSA\_table[T][3]$. But the $Ysib_C\_1$ is dropped here since it has no matched fragments from the next agreed fragments of corresponding $Xsibling$. Thus no alignment with TTC can be found from x. In this way no invalid alignment is passed forward.

Lemma 18. *Checking non emptiness of* $TCA\_table[n]$ *is necessary and sufficient to decide on the existence of consensus sequence.*

Proof. Rows in $TCA\_table[n]$ indicates alignment of $\theta_x^T(x)$ and $\theta_y^T(y)$ up to the last index such that, the last transposition ends at i or no transposition occurs at i for both of them. Thus it indicates the existence of a consensus sequence. If $TCA\_table[n]$ is empty it means no $\theta_x^T(x)$ can align with any $\theta_y^T(y)$ up to the last index, thus indicating the absence of a consensus sequence among x and y. □

5.4 Time Complexity

Before deducing the time complexity of our algorithm, we first discuss how the polynomiality of the algorithm is ensured. The number of list items $\langle Xsibling\rangle$-$\langle Ysibling\rangle$ (in TSA_table_x & TSB_table) and the size of $Ysibling$ for each such row affect the running time. Our algorithm prevents unpredictable increment of the number of rows in $TSA\_table\_x[]$ and $TSB\_table[]$ by merging overlapping portions of the alignments, thus ensuring a polynomial run time as explained below by observations and lemma. Since the observations and lemmas presented below are similar to those for inversion (Section 4.5), detailed explanation with examples is not provided.

Observation 19. *Merging case 1: Merging in* $TCA\_table[i]$. In each iteration $i = 1$ to $n-1$ of the algorithm, at Step 1, we pair up $TSA\_table\_x[i+1]$ and $TSA\_table\_y[i+1]$ through procedure $Four\_Iteration\_Loop$ once only for the non empty $TCA\_table[i]$. This ensures the merging all the alignments presented by the rows of $TCA\_table[i]$.

Proof. From $i+1$, destiny of all those alignments in $TCA\_table[i]$ is the same, i.e., sequence of next agreed fragments of all those alignments is the same for following successive iterations, until the next ending is reached. In other words, we can say, if any alignment residing in

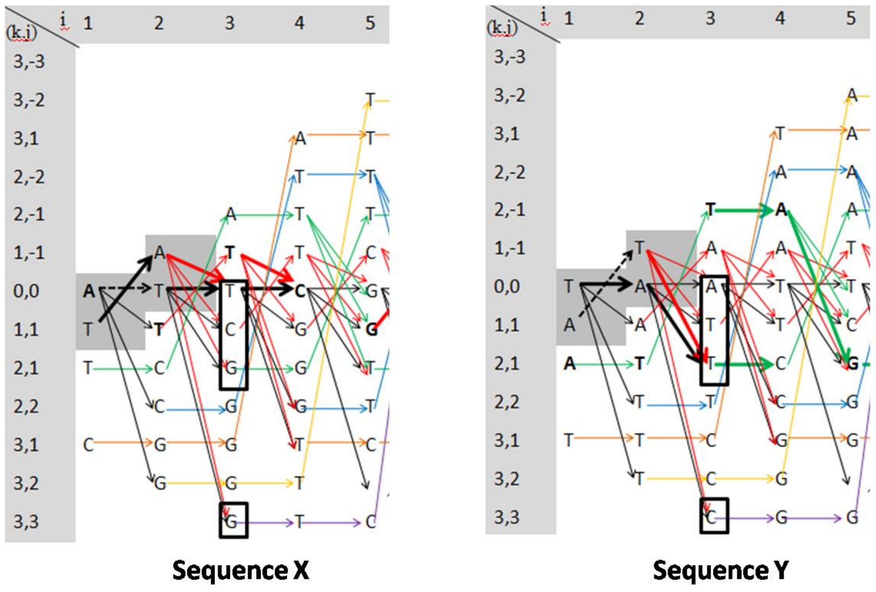


Figure 5.9: Merging case 1

$TCA\_table[i]$ is dropped at some later index due to some mismatch, then it will happen for all other alignments in $TCA\_table[i]$ as well. So instead of keeping separate rows onwards, we merge the overlapping portion to avoid the redundant calculation. Notably, it may merge multiple alignments having different prefixes as well; but this does not create problem as they have the same destiny from index $i + 1$ up to the next ending. Please refer to the Example 13 for an illustration. □

Example 13. In this example we explain Observation 19 with the help of Figure 5.9(i). Please refer to the shaded alignments only. Two alignments, one having A as the first letter (dotted arrow) and the other having T as the first letter (bold solid arrow) are present in the $TCA\_table[2]$. Thus they are merged into one from index 3 as they are destined to the same result from that point. Here the alignment is shown up to index 5. The alignment can go forward up to index 4 (shown by bold arrow), at index 5 both of them are dropped due to mismatch in the next agreed fragments of $Xsibling$ and $Ysibling$.

Observation 20. ***Merging case 2:*** *Merging alignments in* $TSA\_table\_x[i+1]$ *on iteration* i *is* ensured by Step 2.2.3, 2.2.6, 3.2.3, and 3.2.6 in the algorithm.

Proof. If different x_end_trans or x_no_trans fragments of x at index i are paired with the same y_cont_trans from y, then for each of those x_end_inv or x_no_trans fragments, $TSA\_table\_x[i + 1]$ and the next agreed fragments of that y_cont_trans need pairing up. As this same pairing up operation is required for all those matching x_end_trans or x_no_trans

fragments, so this is done once only. So we can say that those alignments presented by the *x_end_trans* or *x_no_trans* fragments are merged into one as from now on, their destiny is the same. For an illustration please refer to Example 14. □

Example 14. In this example we explain Observation 20 with the help of Figure 5.3 and 5.4. Here, in iteration 3, the $TSA\_table[T][3]$ contains $\langle Xsibling \rangle - \langle Ysibling \rangle \equiv \langle (0,0) \rangle - \langle (2,-1) \rangle$. Also the $TSB\_table[3]$ contains $\langle Xsibling' \rangle - \langle Ysibling \rangle \equiv \langle (1,-1) \rangle - \langle (2,-1) \rangle$. We can see, the transposition fragment $M_y[2,-1][3]$ is aligned with two different transposition fragments from x, namely, the *x_no_trans* fragment $M[0,0][3]$ and the *x_end_trans* $M[1,-1][3]$. So for both of these fragments from x, the next calculation step returns $TSA\_table[][4]$ as next agreed fragments. Thus for both of the alignments, the next agreed fragment of $Ysibling$ is paired up with $TSA\_table[][4]$ once only since from now on, their destiny is the same.

Observation 21. ***Merging case 3:*** *Merging alignments in* $TSB\_table\_x[i+1]$ *on iteration* i *are* ensured by case 3 of Steps 2.2 and 3.2 in the algorithm.

Proof. The scenario explained in previous lemma also happens for the opposite case. That is, if the same *x_cont_trans* from x matches with different *y_end_trans* or or *y_no_trans* fragments of y at the same index i, then from $i+1$, they (different alignments presented by those matched *y_end_trans* or *x_no_trans* fragments) are merged into one record in $TSB\_table[i+1]$. Please see the Example 15 for an illustration. □

Example 15. In this example we explain Observation 21 with the help of Figure 5.3 and 5.4. In iteration 5, the $Xsibling = (2,1)$ is paired up with different transposition fragment from y, namely, the $y\_end\_trans = M_y[1,-1][5]$ and $y\_no\_trans = M_y[0,0][5]$. Since both of the transposition fragments from y returns the $TSA\_table\_y[][6]$ as next agreed fragments, thus only one row (one alignment) for the next agreed fragment of $Xsibling = (2,1)$ is inserted into the $TSB\_table[6]$. The inserted row is $\langle Xsibling \rangle - \langle Ysibling \rangle \equiv \langle (2,1) \rangle - \langle (0,0) \rangle$. This ensures the merging case 3.

Lemma 19. *The algorithm merges overlapping portions of the alignments to avoid redundant operations and unnecessary increment of rows in TCA_table, TSA_table, and TSB_table.*

Proof. This lemma is proven by Observations 19, 20 and 21. □

5.4.1 Running Time Complexity for Fixed Length Transposition

Now we first deduce the time complexity for *Fixed Length Transposition* where the transposition size k is fixed. We will prove shortly that the running time complexity of finding the existence of consensus sequence given two input sequences considering fixed length transpositions is $O(nk^2)$, where n is the length of input sequences and k is the fixed transposition size. For an illustration, we will use an example with transposition size $k = 3$, $x = AGACCCTAG$, and $y = ACACACTGG$. $P - graph$ for x and y are given in Figure 5.10 and 5.11. One consensus sequence between those is $ACCTGACAG$.

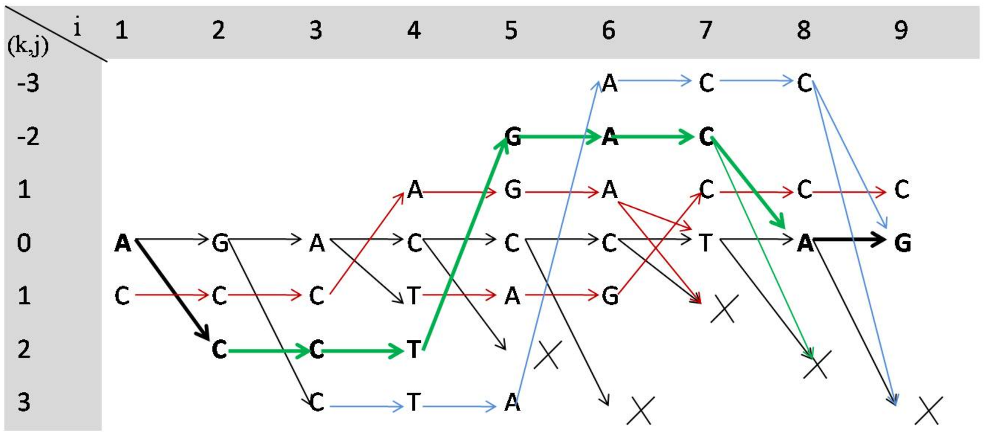


Figure 5.10: $P - graph$ for sequence $x = AGACCCTAG$ with transposition size $k = 3$

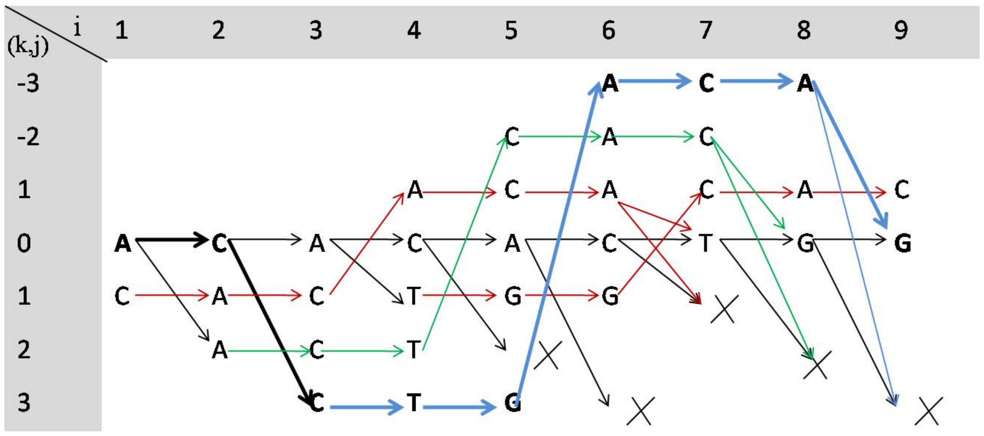


Figure 5.11: $P - graph$ for sequence $y = ACACACTGG$ with transposition size $k = 3$

Now, from the figures, we can state following observations and lemmas.

Observation 22. *At each index, i.e., each column i, at most two transposition fragments $M[j][i]$ can give two next agreed fragments at a time (as an indication of ending of a trans-*

position and beginning of new transposition or no transposition), all other gives one unique next agreed fragment (as an indication of ongoing transposition).

Observation 23. *At each index, i.e., each column i, the middle transposition fragment M[mid][i] can be reached from at most two transposition fragments:* $M[mid][i-1]$ *(previous index was not subject to any transposition) or* $M[Up][i-1]$ *(one transposition ended at previous index).*

cont_trans_i' is actually a singleton set which reduces to an *ending_transposition* fragment $M[j][i]$ at index $i = i' + 2k$ and $j = up$. Then for an *ending_transposition* fragment there are two available next agreed fragments, either $M[-j][i+1]$ (starting of new transposition) or $M[0][i+1]$ (no transposition). Note here, $M[-r][i+1]$ and $M[0][i+1]$ both are mapped by $TSA_table[][i+1]$. That is, next agreed fragments of *ending_transposition* fragment maps to the $TSA\_table[][i+1]$.

Observation 24. *Size of cont_trans_i' at* i' *is one. It reaches ending at* $i'' = i' + 2k$.

Observation 25. *At any* i, $\sum_{\alpha \in \{A,T,C,G\}} |Xsibling| = 2$, *where* $Xsibling \in ISA\_table\_x[\alpha][i]$. *Here the number of members in set cont_trans fragments is* 1 *and no_transposition fragment is 1.*

Observation 26. *At index* $i > i'$, *maximum size of cont_trans_i' is one throughout the index* i, *where* $i' \leq i \leq i' + 2k$. *Otherwise the set is empty.*

Observation 27. *At index* $i > i'$, *total number of cont_trans_i', considering all* $1 \leq i' \leq i$ *is*
=0 for $1 \leq i' \leq i - 2k$ *or for* $i' > i$
$=\sum_{i-2k \leq i' \leq i} 1$
$=2k - 1$

Observation 28. *at index i Total number of ending_transposition fragments is one and no_transposition fragments is one.*

Observation 29. *A fragment in* M_x *can have at most* $2k-1+2 = 2k+1$ *matched fragments from* M_y.

Proof. In M_y, the number of *cont_trans* fragments, *ending_trans* fragment, and *no_transposition* fragment at any iteration is in total $(2k-1)+2 = 2k+1$, each containing single fragment since transposition size k is fixed. Similar is true for M_x also. So if each of them has match with all the fragments in M_y, then total number of match is $(2k+1)\times(2k+1) = O(k^2)$. Lets consider the worst case where all of the matched fragments belong to separate set of matched fragments, e.g., $\langle Xsibling_1\rangle - \langle Ysibling_1\rangle$, $\langle Xsibling_1\rangle - \langle Ysibling_2\rangle$, $\ldots$. In such case each of those will take constant time for finding next agreed fragments and pairing up the next agreed fragments according to the algorithm. Also no same matched fragment $\langle Xsibling_1\rangle - \langle Ysibling_1\rangle$ will occur multiple times in TSA, TSB or TCA, as ensured by the merging cases. So in each iteration, running time complexity is $O(k^2)$. This holds true for each iteration $1 \le i \le n$. So the next lemma is directly followed by the observation. □

Lemma 1. *The running time complexity of the algorithm is $O(nk^2)$.* □

5.4.2 Running time Complexity for All Length Transpositions

Observation 30. *At each index, i.e., each column i, at most k_{max} transposition fragments $M[j][i]$ can give k_{max} next agreed fragments at a time (as an indication of ending of a transposition and beginning of new transposition or no transposition), all other gives one unique next agreed fragment (as an indication of ongoing transposition).*

Observation 31. *At each index, i.e., each column i, the middle transposition fragment M[mid][i] can be reached from at most one $M[mid][i-1]$ (previous index was not subject to any transposition) or at most k_{max} transposition fragments $M[Up][i-1]$ (one transposition ended at previous index).*

Observation 32. *Maximum size of cont_trans_i' at i' is k_{max}. It reaches ending at $i'' = i' + 2k$. For example, in figure 5.3, the size of cont_trans_1 at index 1 is $k_{max} = 3$, since it contains the transposition fragments $M[1,1][1] = T$, $M[2,1][1] = T$, and $M[3,1][1] = C$. The cont_trans_1 gets empty at iteration $i'' = 1 + 2\times 6 - 1 = 6$, since the last cont_transposition fragment $M[3,1][5]$ from cont_trans_1 turns into ending_transposition fragments at iteration 6.*

Observation 33. *At any i, $\sum_{\alpha\in\{A,T,C,G\}} |Xsibling| = k_{max}+1$, where $Xsibling \in ISA\_table\_x[\alpha][i]$. Here the number of members in set cont_trans fragments is k_{max} and no_transposition fragment is 1.*

Now we begin the discussion for deriving the theoretical time complexity of our algorithm. Theoretical worst case time complexity of the algorithm is $O(n^4)$ proven by lemma 28. For deriving the time complexity of the algorithm, we first show it for the worst case. The worst case scenario arises as when each pair from M_x gets some matched pair from M_y thus no alignment is canceled because of mismatch.

Observation 34. *For any $i' = 1, \ldots, n-1$, the size of cont_trans_i' is k_{max} at iteration i' (by Observation 25) and is reduced by one after each 2 iterations $i = i'+1, i'+3, i'+5, \ldots, n-1$, leaving no continuing_transposition pair (starting at i') at the index $i' + 2k_{max} - 1$.*

Proof. We observe that, at iteration i', *cont_trans_i'* are kept in the *x_cont_trans* sets of $ISA\_table\_x[i']$ and has size $\|cont\_trans\_i'\| = k_{max}$ by Observation 25. Now let us see how its size is reduced. Let us start with iteration i'. If all of them have matched fragments in y then at iteration i', we have to perform the *next_calculation* step for each of these pairs, and one of them namely the fragment indicating the transposition of size one reaches ending in the next index $i = i' + 1$ at $M[1,1][i]$, and thus becomes *end_trans* fragment and it is eliminated from the set *cont_trans_i'* (see the next calculation steps for clarification). So the size of *cont_trans_i'* is reduced by one at iteration $i = i' + 1$. Similarly, in iteration $i = i' + 3$ the fragment M[2,j][i] indicates ending of a transposition of size 2, and gets removed from *cont_trans_i'*. Thus at iteration $i = i' + 3$, the size of *cont_trans_i'* is again reduced by one. Then, in iteration $i = i' + 5$ the fragment M[3,j][i] reaches ending and gets removed from *cont_trans_i'*. This continues for each of the next iterations and finally at iteration $i = i' + 2k_{max} - 1$, *cont_trans_i'* becomes empty since $M[k_{max}, j][i]$ becomes ending fragment indicating the completion of transposition of size k_{max} that started at i'. So by index $i' + 2k_{max} - 1$, all the transpositions started at i' are completed. □

Observation 35. *At any iteration $i \geq i'$, the size of cont_trans_i' can be at most $k_{max} - \lceil (i - i')/2 \rceil$. Hence, the total number of existing continuing_transposition pairs (for x or y) considering all cont_inv_i' equals to $O(k_{max}^2)$, where $i - 2k_{max} + 2 \leq i' \leq i$.*

Proof. At index i', the maximum size of *cont_trans_i'* is k_{max}. Then after each two next iteration its size is reduced by one according to Observation 34. Let us consider the iteration $i = 7$. At this iteration, the transpositions started at index 3 to 7 exists as *cont_transposition* fragments. Transpositions started at index 1 and index 2 already complete by index 3. For example, the *cont_trans_3* contains the fragment $M[3,-3][7]$; *cont_trans_4* contains fragment $M[3,1][7]$; *cont_trans_5* contains $M[2,-1][7]$ and $M[3,2][7]$; *cont_trans_6* contains $M[2,2][7]$ and $M[3,3][7]$; finally, *cont_trans_7* cantains $M[1,1][7]$, $M[2,2][7]$, $M[3,3][7]$. So if we observe the patern, we can say that at index i, the number of fragments existing in *cont_trans_i'* is $k_{max} - \lceil (i - i')/2 \rceil$. □

Observation 36. *Maximum number of end_trans fragments at iteration i is k_{max}.*

Lemma 20. *At iteration i, Procedure 1 (finding the next agreed pairs) is called once for each existing continuing_transposition fragments, resulting in $O(2k_{max}^2)$ calls considering both x and y.*

Proof. From Observation 35, we can say that, at index i, the oldest existing $cont\_trans$ set is $cont\_trans\_i'$ where, $i' = i - 2k_{max} + 2$. So we have to take the sum of existing $cont\_transposition$ fragments from index i' to i. Let us consider $k_{max} = 4$ and iteration $i = 8$. So $i' = 2$. Then we can formulate the result in according to the following table. This patern is preserved for any value of k_{max} and iteration i. So we can say, the number of $cont\_transposition$ fragments: $(1 + 2 + 3 + \cdots + k_{max} - 1) + (1 + 2 + 3 + \cdots + k_{max} - 1) + k_{max} = O(k_{max}^2)$. This is true for both x and y. So we can say total number of $continuing\_transposition$ fragments existing at index i is $O(2k_{max}^2)$.

Now, according to the Algorithm, at iteration i, for each distinct $cont\_trans$ set of M_x, we calculate next agreed pairs only once by calling Procedure 1. This is true even if the same $cont\_trans$ exists multiple times in $TSA\_table\_x[i]$ or $TSB\_table\_x[i]$, since we apply pointer calculation on $cont\_trans$. This is ensured by the Step 2.2.1 and 3.2.1 in algorithm. This holds true for M_y as well. So we call Procedure 1 $O(k_{max}^2)$ times for both x and y, and thus $O(2k_{max}^2)$ in total. (However, we call the *Four_Iteration_Loop* (Step 2.2.2 and 3.2.2) each time the set x_cont_inv is encountered in $TSA\_table\_x[i]$ or $TSB\_table\_x[i]$.) □

Total number of calls to the *Four_Iteration_Loop* depends on the size of $Ysibling$ list for each $\langle Xsibling\rangle - \langle Ysibling\rangle$ list items in $TSA\_table\_x[i]$ and $TSB\_table[i]$. In order to find the total number of calls to Procedure 5: *Four_Iteration_Loop*, we first present some observations and lemmas.

At some index $i > i'$, all the existing $cont\_trans\_i'$ of x resides in $TSB\_table[i]$. Now let us see how many $Ysibling$s they can have in total at iteration i. For simplicity, let us think of $x\_cont\_trans\_4$ only ($i' = 4$). All $continuing\_transposition$ $Pair$s in $x\_cont\_trans\_4$ represent the transposition starting from index $4 < i$ and at index 4, they reside in $TSA\_table\_x[4]$. These can be divided into $b = 4$ sets each having on average k_{max}/b fragments, yielding α, $\alpha \in \{A, T, C, G\}$. As they proceeds, each set may be divided into several more child sets based on the yielding base letter of the next agreed fragments by Observation 4, thus increase the number of $\langle Xsibling\rangle - \langle Ysibling\rangle$ rows in $TSB\_table[i]$. At index i, the size of $x\_cont\_trans\_4$ is $k_{max} - \lceil (i-4)/2 \rceil$ by Observation 35 assuming that $4 \geq i - 2k_{max} + 2$. Similar case happens for all those $x\_cont\_trans\_i'$, where $i - 2k_{max} + 2 \leq i' \leq i$. That is, total number of existing $x\_cont\_trans\_i'$ set is $(2k_{max} - 1)$. This is also true for all such $y\_cont\_trans\_i'$. First, let us see how many sets of $cont\_transposition$ fragments: $\{Ci'_1, Ci'_2, \ldots, Ci'_s\} \in x\_cont\_trans\_i'$ exists at index i. We have the following observation.

Observation 37. *At any index $i > i'$, $\{Ci'_1, Ci'_2, \ldots, Ci'_{j'}, \ldots, Ci'_s\} \in x\_cont\_trans\_i'$ are disjoint sets.*

Proof. Same *continuing_transposition* pair does not belong to multiple $Ci'_{j'}$'s. This is so, because each *continuing_transposition* fragment follows a unique path from i' to i by definition and each $Ci'_{j'}$ presents all those transpositions where the last transposition started from the same index $t = i'$ producing the same transposed sequence, i.e., same prefix from index i' up to index i. Any two $Ci'_{j'}$'s say Ci'_1 and Ci'_2, if got split by Observation 4 somewhere between i' to i, then they can not have any common *continuing_transposition Pair*. □

Lemma 21. *At any index $i > i' \geq i - 2k_{max} + 2$, the maximum number (the worst case) of disjoint x_cont_trans sets: $\{Ci'_1, Ci'_2, \ldots, Ci'_{j'}, \ldots, Ci'_s\} \in$ x_cont_trans_i′ is $\frac{k_{max} - \lceil (i-i')/2 \rceil}{b}$.*

Proof. We define the worst case such that, the number of *x_cont_trans* sets from *x_cont_trans_i′* is maximized and *Four_Iteration_Loop* is called for each *x_cont_trans* set where pairing up operation is performed in each iteration of the loop. To make this happen, each of the existing $Ci'_{j'}$ must consist of four *continuing_transposition* fragments to produce at least $b = 4$ next agreed fragments. Using this approach, and by Observations 35 and 37 the lemma is proved. □

So we have $\{C4_1, C4_2, \ldots, C4'_j, \ldots, C4_{k_{max} - \lceil (i-4)/2 \rceil / b}\} \in$ *x_cont_trans_4* at iteration i. Then, we need to know the total size of *Ysibling* lists considering all the $C4'_j$'s. We have the following observation, which basically follows readily following the arguments of Observation 37.

Observation 38. *At any index $i > i'$, $\{Si'_1, Si'_2, \ldots, Si'_s\} \in$ y_cont_trans_i′ are disjoint sets.* □

Observation 39. *The Four_Iteration_Loop is called each time $Ci'_{j'}$ encounters y_cont_trans in its Ysibling list (Steps 2.2.2, 3.2.2 in the algorithm)*

□

From Observation 39, we can say that, the number of calls to *Four_Iteration_Loop* is maximized when the number of *y_cont_trans* sets is maximized. We also want to ensure that pairing up operation is performed in each iteration of *Four_Iteration_Loop*. Thus we have the following lemma which is similar to Lemma 21.

Lemma 22. *At any index $i > i'$, the maximum number of (worst case) disjoint y_cont_trans sets: $\{Si'_1, Si'_2, \ldots, Si'_{j'}, \ldots, Si'_s\} \in$ y_cont_trans_i′ is $\frac{k_{max} - \lceil (i-i')/2 \rceil}{b}$.*

So we have $\{Si'_1, Si'_2, \ldots, Si'_{\frac{k_{max}-\lceil (i-i')/2\rceil}{b}}\} \in y\_cont\_inv\_i'$ for each $i-2k_{max}+2 \leq i' \leq i$ to be considered as $Ysibling$ of $C4'_j$'s. To make it more simple, let us first consider *only the sets* $C4'_j$*'s yielding* $\alpha = A$. Then based on the arguments provided for $\alpha = A$, we can consider the scenario for all $\alpha \in \{A, T, C, G\}$. Now, the number of collection from $x\_cont\_trans\_4$ each yielding A and having $k = 4$ *continuing_transposition* fragments is $\frac{k_{max}-\lceil (i-i')/2\rceil}{b^2}$ (by similar argument as in Lemma 21). Let us call them $C4\_A_{j''}$ where $j'' = 1, 2, \ldots, \frac{k_{max}-\lceil (i-i')/2\rceil}{b^2}$. This is true for all $x\_cont\_trans\_i'$. So we have the following two observations.

Observation 40. *The number of distinct sets of cont_transposition fragments from* $x\_cont\_trans\_i'$ *yielding* A *is* $\frac{k_{max}-\lceil (i-i')/2\rceil}{b^2}$ *at iteration* i *in the worst case, where* $i - 2k_{max} + 2 \leq i' \leq i$ *. Let us call them* $Ci'\_A_{j''}$*, where* $j'' = 1, 2, \ldots, \frac{k_{max}-\lceil (i-i')/2\rceil}{b^2}$.

Observation 41. *The number of distinct sets* y_cont_trans *from* $y\_cont\_trans\_i'$ *yielding* A *is* $\frac{k_{max}-\lceil (i-i')/2\rceil}{b^2}$ *at iteration* i *in the worst case, where* $i - 2k_{max} + 2 \leq i' \leq i$*. Let us call them* $Si'\_A_{j''}$*, where* $j'' = 1, 2, \ldots, \frac{k_{max}-\lceil (i-i')/2\rceil}{b^2}$.

At iteration i, a $Ci'\_A_{j''}$ can have matched pairs from any $y\_cont\_trans\_m$ where $i - 2k_{max} + 2 \leq m \leq i$. But two different cases occur as follows.

Case 1: *At iteration* i*, for each* $y\_cont\_inv\_m$*, where* $i - 2k_{max} + 2 \leq m < i' < i$*, the total number of* y_cont_trans *in* $Ysibling$ *considering all* $Ci'\_A_{j''}s$ *is* $\frac{k_{max}-\lceil (i-m)/2\rceil}{b^2}$*. So considering all such* $y\_cont\_trans\_m$*, total number of calls to the* Four_Iteration_Loop *for this case is* $\sum_{m=i-2k_{max}+2,\ldots,i'} \frac{k_{max}-\lceil (i-m)/2\rceil}{b^2} \leq \sum_{m=i-2k_{max}+2,\ldots,i'} \frac{k_{max}}{b^2} = (i'-i+2k_{max}-2+1)(k_{max}/b^2) = (i' + 2k_{max})(k_{max}/b^2)$.

For simplicity, we first prove the Case 1 for $i' = 4 < i$, that is for all $C4\_A_{j''} \in x\_cont\_trans\_4$, by Lemma 23 and Lemma 24 below. Then based on the arguments provided for $i' = 4$, we can prove the case for all $i' < i$.

Lemma 23. *For* $1 \leq m < 4$*, each* $C4\_A_{j''}$ *can have multiple* y_cont_trans *sets in its* $Ysibling$ *from the same* $y\_cont\_trans\_m$.

Proof. Let us consider the sets $\{S1\_A_1, \ldots, S1\_A_{j''}, \ldots, S1\_A_{\frac{k_{max}-\lceil (i-i')/2\rceil}{b^2}}\} = y\_cont\_trans\_1$. They may be paired with different or the same x_end_trans pairs at index $4 - 1$. For each of those x_end_trans pairs in x, those matching sets $S1\_A_{j''} \in y\_cont\_trans\_1$ are paired with $TSA\_table\_x[A][4]$ and thus each $TSA\_table\_x[A][k]$ can pair with multiple number of distinct collections from $y\_cont\_trans\_1$. See the Example 15 provided for Observation 21 for an illustration. But remember, same $S1\_A_{j''}$ is not paired multiple times with $TSA\_table\_x[A][4]$ ensured by the *merging case 2* at $TSA\_table\_x[A][4]$. □

Lemma 24. *For $1 \leq m < 4$, all sets $Sm\_A_{j''} \in y\_cont\_trans\_m$ that exist in the $Ysibling$ list of all these $C4\_A_{j''}$'s are disjoint.*

Proof. The same $Sm\_A_{j''}$ can not exists in the $Ysibling$ list of two different $C4_A_{j''}$s. We prove it by contradiction. Suppose, two different $C4\_A_1$ and $C4\_A_2$ align with the same $S1\_A_1$ at i. Pairing between $C4\_A_{j''}$ and $S1\_A_{j''}$ indicates an alignment of inversed x where the last transposition started from 4, with transposed y having the last transposition continuing from 1. Two $C4\_A_1$ and $C4\_A_2$ are disjoint by Observation 3. It implies that they present two transposed sequences that yield different base letters at some index $4 \leq i'' \leq i$ and that is why they were split into two by the *split case 2* (otherwise they would have belonged to the same set). If they align with the same $S1\_A_1$ at i, we get a contradiction, because $S1\_A_1$ presents all those inversions for which inversed sequences (prefixes) are the same from index 1 up to i, i.e., prefixes do not differ at any index i'', where $4 \leq i'' \leq i$. Hence the result follows. □

Now from Observations 40, 41, and Lemmas 23, 24, we can say, considering all $C4\_A_{j''}$, that the total number of y_cont_trans sets in $Ysibling$ from $y\_cont\_trans\_m$ is $\frac{k_{max} - \lceil (i-m)/2 \rceil}{b^2}$ for each m, where $i - 2k_{max} + 2 \leq m < 4$. So considering all m, in total we get $(4 - i + 2k_{max} - 2) \times \frac{k_{max} - \lceil (i-m)/2 \rceil}{b^2} = (4 + 2k_{max})(k_{max}/b^2)$ sets in the $Ysibling$. For each of these sets the *Four_Iteration_Loop* is called. We have shown the Case 1 for $i' = 4$ and this is true for any $i' < i$. Thus Case 1 is proved.

Case 2: For each $Ci'\_A_{j''}$, *considering all $y\_cont\_trans\_m$, where $i' \leq m < i$, the total number of calls to the* Four_Iteration_Loop *is $(i - i')$, and considering all $Ci'\_A_{j''} \in x\_cont\_trans\_i'$, it is $O((i - i')\frac{k_{max}}{b^2})$.* Again, for the sake of simplicity, we prove it for $i' = 4$ by Lemma 25. Later, based on the arguments provided for $i' = 4$, we can prove the case for all $i' < i$.

Lemma 25. *Each $C4\_A_{j''}$ can have only one y_cont_trans set from each $y\_cont\_trans\_m$, for $4 \leq m < i$, resulting a total of $i - 4$ y_cont_trans sets.*

Proof. We prove this lemma by contradiction. For index m after 4, all of $C4_A_{j''}$s will reside in $TSB\_table[m]$. Let us assume that at iteration i, $C4\_A_1$ is paired up with two sets say $S6\_A_1$ and $S6\_A_2$ from $y\_cont\_trans\_6$, $(6 < i)$. By Observation 4 and Observation 6, all $S6_A_{j''}$s are disjoint. Two different $S6\_A_1$ and $S6\_A_2$ means two transposed sequence who get different at somewhere between index $6 \leq i' \leq i$ and thus get split by the Observation 4 (split case 2). But at iteration i, all pairs in $C4\_A_1$ are presenting the same inversed sequence from index 4 to i. $C4\_A_1$ does not change anywhere up to i, if it were changed then it would have been split into two child set say $C4\_A_{1'}$ and $C4\_A_{1''}$ from that index by split case 2. So we reach a contradiction. So $C4\_A_1$ can pair with only one collection say $S6\_A_1$. So the total number of *y_cont_trans* sets for each $C4\_A''_j$ is $(i-4)$. Hence the result follows. □

Therefore, considering all $C4_A_{j''}$s, the total number of *y_cont_trans* sets is $\sum_{i' \leq m \geq i}(k_{max} - \lceil (i-m)/2 \rceil)/b^2 = O((i-i')k_{max}/b^2)$ (using Observation 5). For each of these sets the *Four_Iteration_Loop* is called, and this is true for all $i' < i$. So Case 2 is proved.

Lemma 26. *Total number of calls to the Procedure 5:* Four_Iteration_Loop *for* $ISB\_table[i]$ *at iteration* i *is* $O((2k_{max}^2(i + k_{max}))/b)$.

Proof. Continuing from the proof of Lemma 25, considering all $C4\_A_{j''}$'s, the number of *y_cont_trans* sets is $(i' + 2k_{max})(k_{max}/b^2)(bycase1) + (i-i')\frac{k_{max}}{b^2})(bycase2) = i'\frac{k_{max}}{b^2} + 2\frac{k_{max}^2}{b^2} + i\frac{k_{max}}{b^2} - i'\frac{k_{max}}{b^2} = i\frac{k_{max}}{b^2} + 2\frac{k_{max}^2}{b^2}$. Finally, considering all $\alpha \in \{A, T, C, G\}$ the total number of *y_cont_trans* sets in their $Ysibling$ is $i\frac{k_{max}}{b} + 2\frac{k_{max}^2}{b}$.

Presence of *y_end_trans* pairs in $Ysibling$ lists of $C4_{j'}$ cannot dominate the total number of calls to the *Four_Iteration_Loop*, as, for each $C4'_j$, pairing up between its $x\_next\_atcg[]$ and $TSA\_table\_y[][i+1]$ is done once only, by Observation 21.

Therefore, at iteration i, considering all $x\_cont\_trans\_i'$, $i - 2k_{max} + 2 \leq i' < i$, the *Four_Iteration_Loop* is called $(i - i + 2k_{max} - 2)(i\frac{k_{max}}{b} + 2\frac{k_{max}^2}{b}) = 2i\frac{k_{max}^2}{b} + 4\frac{k_{max}^3}{b} = O(\frac{k_{max}^2(i+k_{max})}{b})$ times. So Lemma 26 is proved. □

Lemma 27. *Total number of calls to the* Four_Iteration_Loop *for* $TSA\_table[i]$ *at iteration* i *is* $O(\frac{k_{max}^2}{b})$.

Proof. $TSA\_table\_x[\alpha][i]$ gets $Ysibling$ pairs only if in the previous index $i-1$, some *x_end_trans* have matched pairs from y. Number of *x_end_trans*s at index $i-1$ is k_{max} by Observation 10. In the worst case all of those *x_end_trans*s have matched pairs in y. Let us see how many *y_cont_trans* sets each $TSA\_table\_x[\alpha][i]$ can have in their $Ysibling$s. Let us calculate for A first. At iteration i, we have *y_cont_trans* sets $Si'\_A_{j''} \in y\_cont\_trans\_i'$, where $1 \leq j'' \leq \frac{(k_{max} - \lceil (i-i')/2 \rceil)}{b^2}$ (by Observation 6), for all $i - 2k_{max} + 2 \leq i' \leq i$. Let us think all of their parent collection were paired up with the *x_end_trans* pairs at index $i-1$. No $Si'\_A_{j''}$ will exist multiple times into the $Ysibling$ of $TSA\_table\_x[A][i]$ by the Observation 20. So for A, $TSA\_table\_x[A][i]$ has a total of $\frac{(k_{max} - \lceil (i-i')/2 \rceil)}{b^2}$ *y_cont_trans* sets from each $y\_cont\_trans\_i'$, where $i - 2k_{max} + 2 \leq i' \leq i$. Thus in total, we have $\sum_{i - 2k_{max} + 2 \leq i' \leq i}(\frac{(k_{max} - \lceil (i-i')/2 \rceil)}{b^2} = O(\frac{k_{max}^2}{b^2})$. *y_cont_trans* each having size $k = 4$ and yielding A. Considering all $\alpha \in \{A, T, C, G\}$, the total number of *y_cont_trans* sets from all *y_cont_trans_i'* is $O(\frac{k^2_{max}}{b})$. So for each of these sets, the *Four_Iteration_Loop* can be called. Again, for each $TSA\_table\_x[\alpha][i]$, pair up step between $x\_next\_atcg[]$ and $TSA\_table\_y[\alpha][i]$ is executed once only by Observation 21. So the number of *y_end_trans* pairs in $Ysibling$ does not dominate the total number of calls to the *Four_Iteration_Loop*. Thus Total number of calls to the *Four_Iteration_Loop* for $TSA\_table[i]$ at iteration i is $O(\frac{k^2_{max}}{b})$. □

Observation 42. *Total running time for processing the* $TCA\_table[i]$ *at iteration* i *is* $O(b)$. □

If $TCA\_table[i]$ is non empty then the *Four_Iteration_Loop* runs performing $O(b)$ assignments. See the step 1 and merging case 1 for clarification.

Lemma 28. *Worst case running time of the algorithm is* $O(n^4)$.

Proof. Using the observations and lemmas provided above it is easy to deduce the worst case running time for each step of the algorithm, as follows.

Initialization:

It involves filling up the M_x, M_y, and pairing up the $TSA\_table\_x[][1]$ and $TSA\_table\_y[][1]$. So it takes $O(nk^2_{max}) + O(k) = O(nk^2_{max})$.

Iteration:

At each iteration $i = 1, 2, \ldots, n-1$, the algorithm calls Steps 1, 2, and 3.
Step 1: It needs $O(k)$ at each iteration i, by Observation 42.
Step 2: Processing $TSA\_table\_x[][i]$ depends on two factors:

1. *next_calculation* step: This is done in Steps 2.1 and 2.2.1. Step 2.1 takes $O(k_{max})$ by Observation 25 and 34. Step 2.2.1. takes $O(k^2_{max})$ by Observation 35 and Lemma 20. So the total time complexity is $O(k_{max}) + O(k^2_{max}) = O(k^2_{max})$.

2. *Four_Iteration_Loop*: This is performed in Step 2.2.2 (always), Step 2.2.3 (conditionally) and once only for case 3 (under Step 2.2). Total number of calls by Step 2.2.2 and 2.2.3 is $O(k^2_{max}/b)$ By Lemma 27. Again, pairing up operation is performed in each iteration. So each call to the *Four_Iteration_Loop* performs $b = 4$ pairing up operation. So the total number of pairing up operation is $O(k^2_{max})$.

Step 2.3 takes $O(b)$ if all $TSA\_table\_x[\alpha][i]$ for $\alpha = \{A, T, C, G\}$ have non empty $Ysibling$ list. So for Step 2, the total time complexity at iteration i is $O(k^2_{max}) + O(\frac{k^2_{max}}{b}) = O(k^2_{max})$.
Step 3: Processing $TSB\_table[i]$ depends on three factors:

1. *next_calculation* step: This is done in Steps 3.1 and 3.2.1. The pairs in $x\_cont\_trans\_i''$ sets ($i - 2k_{max} + 2 \leq i'' < i$) are responsible for forming the x_cont_trans sets of $\langle Xsibling : [x\_cont\_trans, x\_end\_trans]\rangle - \langle Ysibling\rangle$ rows of the $TSB\_table[i]$. Again pairs in $y\_cont\_trans\_i'$ are pointed by the $Ysibling$ list of these rows, where $i-2k_{max}+2 \leq i' \leq i$. So by Lemma 20, the number of total steps here is $O(2k^2_{max})$.

2. *Four_Iteration_Loop*: This is performed in Step 3.2.2 (always), Step 3.2.3 (conditionally) and once only for case 3. Total number of calls by Step 3.2.2 is $O((2k^2_{max}(i + k_{max}))/b)$ By Lemma 26. Again, pairing up operation is performed in each iteration. So total number of pairing up operation is $O(2k^2_{max}(i+k_{max}))$. For each x_end_trans, the loop is called at step 3.2.3. But it is negligible because the total number of x_end_trans pairs considering all $Xsibling$s in $TSB\_table[i]$ is only k_max by Observation 10.

3. Transferring step: This is done in Step 3.3. If all the $x\_next\_atcg[\alpha]$ of the $Xsibling$s created in Step 2.1, has non empty $Ysibling$ list for $\alpha = \{A, T, C, G\}$, then each of them are passed to $ISB\_table[i+1]$. So it takes $O(b \times (k_{max}\frac{k_{max}-\lceil (i-i')/2\rceil}{b})) = O(k^2_{max})$. Tcsize

So for Step 3, the total time complexity at iteration i is $O(2k^2_{max}) + O(2k^2_{max}(i + k_{max})) + O(k^2_{max}) = O(3k^2_{max} + 2ik^2_{max} + 2k^3_{max}) = O(2k^2_{max}(i + k_{max}))$.

So Step 1, Step 2, and Step 3 take $O(b) + O(k^2_{max}) + O(2k^2_{max}(i + k_{max})) = O(2k^2_{max}(i + k_{max}))$ for each iteration i, resulting in $O(k^2_{max}(n^2 + nk_{max})) = O(n^4)$ (if we consider $k_{max} = n/2$) in total. Here we can see that Step 3 is the dominating step.

Termination:

Decision making takes $O(1)$ that just checks the emptiness of the $TCA\_table[n]$ (n =last index of the table).

So in total, worst case time complexity of the algorithm is $O(n^4)$. Thus Lemma 28 is proved. □

5.5 Space Complexity

The space needed for storing the $P - graph$ of x and y is $O(nk^2_{max})$. For storing the agreed fragments, $TSA\_table[i]$ allocates $O(k\_max^2)$ space (by Lemma 27), and $TSB\_table[i]$ allocates $O((2k^2_{max}(i + k_{max})))$ space (by Lemma 26), for each iteration $i = 1, 2, \ldots, (n-1)$. While processing the i^{th} column of TSA_table or TSB_table, values of previous columns are not required, thus can be freed. So storing the agreed fragments needs $O(n^3)$ space in total. Thus, the space complexity of the algorithm is $O(n^3)$.

5.6 Experimental Results for All Length Transposition

Theoretical worst case time complexity of the algorithm is $O(n^4)$ proven in Lemma 28. However, practical running time of the algorithm in the worst case and average case are $O(n^3)$ and $O(n^2)$ respectively. This is apparent from the experimental results reported in Table 5.1.

Table 5.1: Total number of steps taken by the algorithm for $n = 10, 20, 30, 40, 50, 60, 70, 90, 120$

n^2	n^3	Length, n	Running Time		
			Result: YES		Result: NO
			Worst	Average	
100	1000	10	501.4	212.6	87.25
400	8000	20	3483.4	1034.6	201.75
900	27000	30	11623.2	2047.6	486.5
1600	64000	40	26745.4	3722.2	1000.5
2500	125000	50	48951.6	4629	1382.75
4900	343000	70	133693.8	12616.2	2046.25
8100	729000	90	290609.4	27454.4	3001
14400	1728000	120	687939.4	84960.8	5052.5
Comments			$O(n^3)$	$O(n^2)$	$O(n^2)$

In our experiments, x and y are selected such that y is a permutation of x (otherwise there can never be any Concensus String among them since in transposition only the blocks in a string are transposed or swapped but no new base is generated, thus our algorithm returns NO as well). We define the term *performance factor* (equivalent to the running time), as a counter that keeps track of the total number of statements executed for finding the next agreed fragments $M[j,k][i]$ and pairing the matched agreed fragments. So running time of a test case is calculated by adding the performance factor of the dominating steps, i.e., Step 2 and Step 3 in the algorithm. For each length n (ranging from 10 to 120), we run the experiment under three categories (columns 4 to 6 of Table 5.1). Under each category, we generate ten sets of test cases by randomly choosing x. Then we calculate the average running time of the test cases. The column 4 shows the worst case running time (when the y is equal to x). Then the column 5, presents the running time in average case. The average

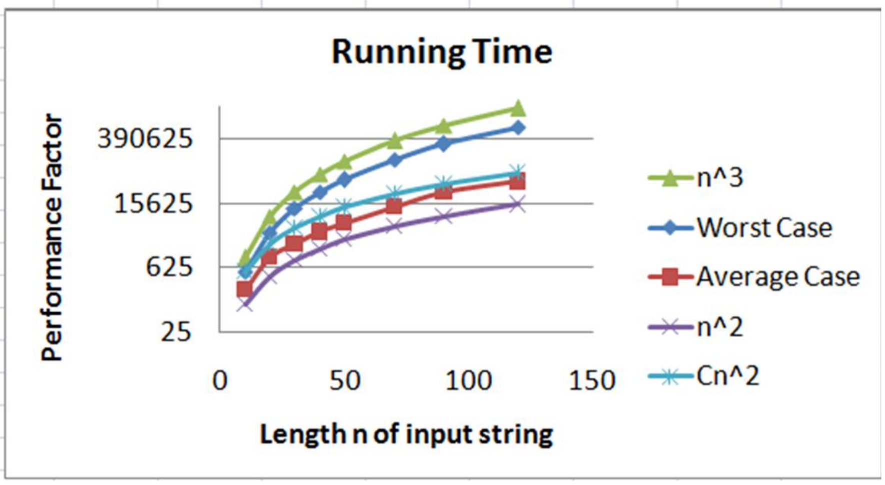


Figure 5.12: Time Complexity of our proposed algorithm

case is generated by selecting y as an arbitrary permutation of x. Finally column 5, (where the y is selected as an arbitrary permutation of x) shows the running time of the algorithm for returning NO when there exist no Concensus String between x and y.

Worst case running time is $O(n^3)$ as apparent from the experiments. We can see from the Table 5.1, the average case and false case running time are $O(n^2)$. This is also apparent from the graph in Figure 5.12. With the decrease in the similarity between x and y, the running time drops to $O(n^2) = Cn^2 \approx 5n^2$.

Here, no comparison with previous works is provided since there exists no other works on our problem.

5.7 Conclusion

In this chapter we have mapped the Concensus String problem under the transposition metric to the biomedical problem of detecting the allelic heterogeneity. Our proposed algorithm finds the common ancestor sequence given two mutated sequences where mutation involves only non overlapping transposition. Future research endeavor could be directed towards other mutation operations as distance metric and simultaneous application of inversion and transposition mutations.

Chapter 6

Diagnosis of Allelic Heterogeneity

In this chapter, we first discuss the traditional clinical approaches for diagnosing allelic heterogeneity in Section 6.1. Then we see how our algorithms can help in detecting alellic heterogeneity in Section 6.2. We also present some other utilities of our algorithm in Section 6.3.

6.1 Clinical Approach for Diagnosing Allelic Heterogeneity

Clinically several approaches are available for the detection of allelic heterogeneity. The allele-specific oligonucleotide probe is used in some cases for detection of allelic heterogeneity[1]. But the same approach is not applicable for all types of allelic heterogeneity. In the X-linked clotting disorder hemophilia B, for example, more than 2000 different mutations in the gene for clotting factor IX have been observed in different patients. This degree of allelic heterogeneity makes the use of allele-specific oligonucleotide probes impractical. In such cases, following clinical techniques are followed to obviate this problem:

1. Mismatch Scanning: This is done by amplifying the exons of the gene and hybridizing the PCR products from the patient with the corresponding products from the normal gene. The mismatch can be detected either by chemical reagents that cleave selectively at the site of the mismatch or by electrophoresis under partially denaturing conditions.

[1]https://www.inkling.com/read/principles-of-medical-biochemistry-meisenberg-simmons-3rd/chapter-11/allelic-heterogeneity-is-the

2. Gene Sequencing: Sequencing all exons of the gene is used in genetic disorder according to the report of National Center for Biotechnology Information[2] and [40].

3. Linkage Analysis: In this case, no attempt is made to identify the mutation. Instead, a known genetic marker that is located next to the mutated gene is analyzed. The mutation is inherited along with the marker simply because they are close together on the same DNA molecule, and meiotic recombination between the gene and the marker is very rare.

However, all of these are expensive and time consuming operations. Our algorithms are never the alternative of all these medical diagnostic approaches. Because, even if our algorithm returns YES, still medical diagnostic techniques may find those diseases as not allelic heterogeneous. But if our algorithm returns NO, then those diseases can never be allelic heterogeneous, and further medical diagnostic approach is unnecessary. So before going through such costly techniques, it is better to test first if there is even any possibility of allelic heterogeneity between two diseases, using our proposed algorithms.

Detection of an unknown disease as allelic heterogeneous with a known genetic disease helps in medication and treatment. For this purpose whole genome sequencing is required which takes around 12 to 13 weeks (data collected from `https://www.genetests.org`). Besides, diagnosis of such disease needs approaches like mismatch scanning, gene sequencing, linkage analysis etc., all of which are highly expensive solutions as apparent from the cost estimates provided in Table 6.1. For example, diagnosis of Hurler syndrome or Scheie syndrome I takes three to four weeks with gene sequencing approach and costs around $2,050.

Table 6.1: Gene name with corresponding allelic heterogeneous diseases and diagnosis details (data is collected from: https://www.genetests.org/tests; http://www.ggc.org/)

Gene	Allelic Heterogeneous Disease		Diagnosis Details		
	Disease 1	Disease 2	Diagnostic Method	Cost	Time
IDUA	Hurler syndrome	Scheie syndrome I	Sequencing	$2,050	3 weeks
CFTR	Cystic Fibrosis	Congenital Absence of the Vas Deferens	Sequencing	$1,310.00	3 - 4 weeks
DMD	Duchenne Muscular Dystrophy	Becker Muscular Dystrophy	MLPA	$500	2 weeks
RET	Hirschsprung Disease	Multiple endocrine neoplasia Type 2	Sequencing	$1,160.00	3 - 4 weeks

[2]http://www.ncbi.nlm.nih.gov/pubmed/24066368

6.2 Steps for Detecting Allelic Heterogeneity Using Our Algorithms

In this section we explain the process for both the inversion and transposition operations. For detecting allelic heterogeneity, generating all the consensus sequences is not mandatory. We just need to check if some specific ancestor gene sequence p exists as a consensus sequence of input sequences x and y. For instance, suppose we have an unknown disease χ and we want to see if it is allelic heterogeneous with the disease Cystic Fibrosis, i.e., if both of these are mutated from the same gene CFTR according to the Table 2.1 in Section 2.5.3. For this purpose we input gene sequence of χ and Cystic Fibrosis as x and y respectively. We denote the ancestor CFTR gene sequence as p. Let the length of the sequences be n. After T_x (or M_x) and T_y (or M_y) are initialized, we do a small trick for detecting allelic heterogeneity. For each index i, we keep $T_x[j][i] = \langle(p,q),\alpha\rangle$ if the base α matches with the base at $p[i]$, where $1 \leq i \leq n$ and $1 \leq j \leq n+1$. Otherwise we set null to $T_x[j][i]$ (similar is done on M_x in case of transposition). Similar approach is followed for $T_y[j][i]$ (or M_y) as well. Reinitializing T_x and T_y in this approach needs $O(n^2)$ time (in case of all length transposition, reinitializing M_x and M_y needs $O(n^3)$ time). Then we run our main algorithm and ignore the nullified cells in T_x and T_y (or M_x and M_y). If the algorithm terminates returning YES, it indicates existence of the ancestor gene p as a consensus sequence. That means there is a possibility of allelic heterogeneity among the two diseases. So we can perform additional clinical diagnostic approaches to validate the output. On the other hand, if the algorithm returns NO, it indicates nonexistence of the common gene sequence p from which both χ and Cystic Fibrosis could be derived. So they are definitely not allelic heterogeneous. Therefore there is no need of performing expensive clinical diagnostic tests, which saves huge energy and costs. For an illustration please refer to the flowchart shown in Figure 6.1

6.3 Other Applications

Though detecting allelic heterogeneity does not demand generating all the common ancestor sequences, but if we maintain predecessor links among the agreed pairs (in case of inversion) or agreed fragments (in case of transposition) for either x (links in T_x for inversion and links in M_x for transposition) or y (links in T_y for inversion and links in M_y for transposition), then it keeps track of all those agreed sequences starting at index 1 and ending at index n. After the algorithm terminates, these connections resemblance a tree type structure (from right to left). So applying DFS on this structure gives us all possible common ancestor

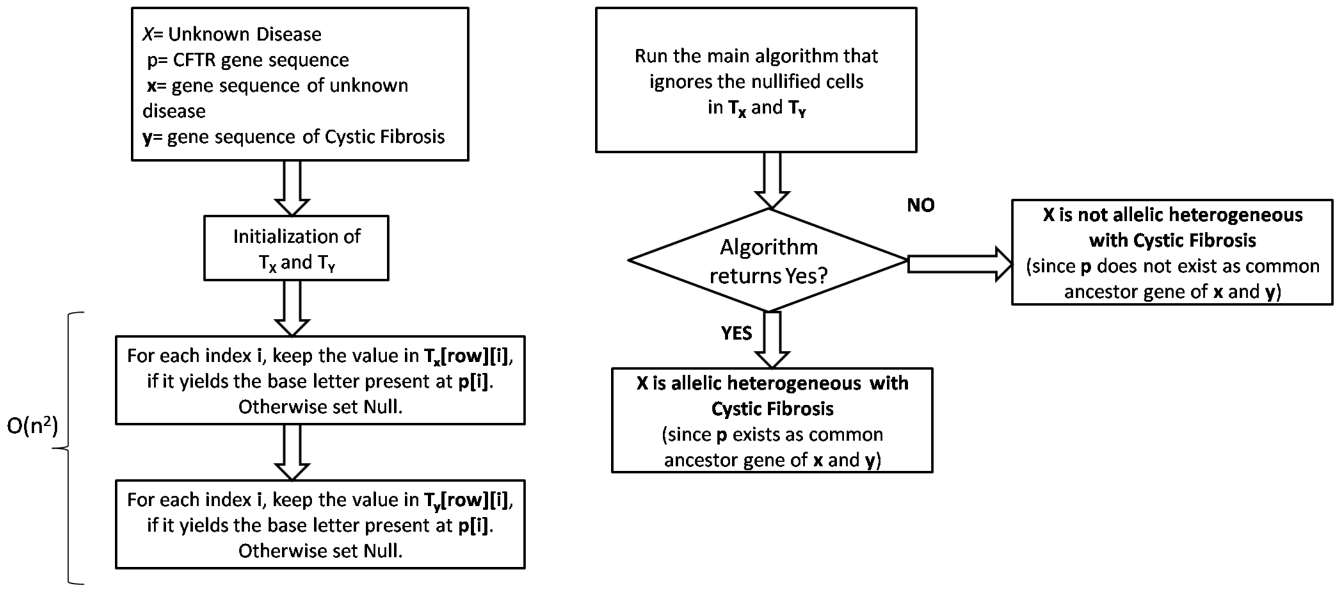

X= Unknown Disease
p= CFTR gene sequence
x= gene sequence of unknown disease
y= gene sequence of Cystic Fibrosis
Initialization of T_X and T_Y
For each index i, keep the value in T_x[row][i], if it yields the base letter present at p[i]. Otherwise set Null.
For each index i, keep the value in T_y[row][i], if it yields the base letter present at p[i]. Otherwise set Null.
O(n²)
Run the main algorithm that ignores the nullified cells in T_X and T_Y
Algorithm returns Yes?
NO
X is not allelic heterogeneous with Cystic Fibrosis (since p does not exist as common ancestor gene of x and y)
YES
X is allelic heterogeneous with Cystic Fibrosis (since p exists as common ancestor gene of x and y)

Figure 6.1: Steps for diagnosing allelic heterogeneity that involves only inversion mutation. For the case of transposition, similar steps are followed (just use M_x and M_y instead of T_x and T_y).

sequences. For example, applying this approach for the transposition mutation on the sequences $x = ATTCGGTCC$ and $y = TCATTCGGC$ gives following common ancestor gene sequences:

1. ATTCGGTCC
2. TACTGGTCC
3. TCATGGTCC
4. ATTCTCGGC
5. TACTTCGGC
6. TCATTCGGC

This actually extends the utility of our algorithm since it can meet other scenarios in computational biology where retrieving all possible common ancestor gene sequences is necessary. For example, when studying breed-related hereditary conditions, a common practice for breeders and medical experts alike is to compare pedigrees of affected or carrier dogs. In doing so, there is a tendency to trace back to common ancestors and blame these individuals as carriers or progenitors of a defective gene. For this purpose the *closest common ancestor analysis* is performed to determine the minimum age of a defective gene in the population - and therefore its possible genetic spread[3,4]. This allows breeders to determine the minimum breadth of the gene pool that is liable for carrying the defective gene, and that requires genetic counseling. This is usually done by analyzing the family tree which is time consuming and expensive in term of genetic tests. However, it is possible that no common ancestor is found after traversal in the family tree. So before doing this complex task of identifying the common ancestor career gene, its helpful if we can run a simple algorithm that returns existence of common ancestors. If it returns true/yes, only then biologists can attempt for analyzing the family tree. This is what our algorithms does in $O(n^3)$ running time (practical running time).

[3]http://pawpeds.com/pawacademy/genetics/commonancestor/

[4]http://www.pcagenetics.com/ARTICLES/095-Epidemiological-Studies.html

Chapter 7

Conclusion

Algorithms for sequence analysis are of central importance in computational molecular biology and coding theory. One of the widely known problem in this field is the Closest String Problem (CSP), or Consensus String Problem. In this thesis we add a new problem to the NP-hard family: Consensus String problem with transposition metric. Then we provide algorithm for a relaxed version of the Consensus String problem under inversion and transposition metric. The algorithm can be applied in several biological problems, including diagnosis of allelic heterogeneity, a challenging problem in molecular genetic diagnosis.

In this chapter, we draw conclusion by highlighting the major contributions made in this thesis. We have also provided some directions for future research.

7.1 Major Contribution

The contributions that have been made in this thesis are enumerated as follows.

- We have investigated the complexity class of the Consensus String problem under the inversion and transposition metrics. The Consensus String problem under the transposition metric has been proven to be NP-hard by reduction from the already proven NP-hard problem: Consensus String problem under the Swap Metric.

- We have develop polynomial time algorithms for a relaxed version of the Consensus String problem under the inversion and transposition metric. In this relaxed version we have to output the existence of Closest String between two input strings.

    1. For the non overlapping inversion metric, theoretical run time of our algorithm is $O(n^4)$, whereas it is $O(n^3)$ practically, for the worst case scenario. Moreover, for

the average case, our algorithm runs in $O(n^2)$. Space complexity of the algorithm is $O(n^3)$.

Cho et al. [21] have provided an $O(n^3)$ algorithm using $O(n^2)$ space (n is the size of the two input strings) for this same problem we have worked on (non overlapping inversion metric). But we have found through experimentation that their algorithm fails in returning the correct answers in some cases because of not tracking the prefixes of the common ancestors. In this thesis, our presented algorithm correctly solves this problem with the same time and space complexity.

2. For non overlapping transposition metric, we have analyzed the running time for fixed length transpositions and all length transpositions. For fixed length transpositions, the running time and space complexity are $O(n^3)$ and $O(n^2)$. On the other hand, for all length transpositions, theoretical running time is $O(n^4)$ and space complexity is $O(n^3)$. However, practical running time in worst case and average case are found to be $O(n^3)$ and $O(n^2)$ respectively for the all length transpositions.

- We have presented a roadmap for a non-clinical efficient scheme to aid in the diagnosis of allelic heterogeneity. In particular, here we use the term *common ancestor* to indicate the same gene sequence from which different mutation order gives different gene sequence x and y. Our aim is to find the common ancestors given x and y as input, where x is the gene sequence of a known disease caused by mutation of some ancestor gene p, and y is the gene sequence of an unknown disease. If there exist common ancestors between x and y, and we find a match with p, then we diagnose that unknown disease y to be allelic heterogeneous to x. Currently available medical diagnostic techniques, such as, mismatch scanning, linkage analysis, gene sequencing, etc. all are expensive and time consuming operations. Our algorithm is not an alternative option for diagnosis of the allelic heterogeneity. Because, even if our algorithm returns YES, still medical diagnostic techniques may find those diseases as not allelic heterogeneous. But if our algorithm returns NO, then those diseases can never be allelic heterogeneous, and further medical diagnostic approach is unnecessary. So before going through such costly medical diagnostic techniques, it is better to test first if there is even any possibility of allelic heterogeneity between two diseases, using our proposed algorithms.

7.2 Future Plan

A number of future research directions arise out of our work as discussed below.

1. The issue of the parameterized complexity of Consensus String has been raised several times in the literature [13, 33, 34, 48]. The approximation and fixed parameter complexity for the Consensus String problem under transposition and inversion metrics are still unknown and a good topic to work on in future.

2. Our algorithm mainly detects the existence of Closest Strings, and keep track of all possible common ancestor strings given two input strings x and y. Finding the Closest String or the Consensus String under the transposition metrics is NP-hard, already proven in Chapter 3. However, there is good number of recent works where several genetic algorithms [49], such as, parallel simulated annealing [45], parallel multi start algorithm [31], ant colony optimization algorithm [28], memetic algorithm [8] etc. are applied for finding the Closest String. An interesting future work would be the analysis of such genetic algorithms for the inversion metric and transposition metric.

3. Future research endeavor could be directed towards developing algorithms considering other mutation operations such as insertion, deletion, etc [44] (levenshtein distance) since in many allelic heterogeneity such mutations occur frequently.

4. Developing algorithm for finding **minimum** Consensus String distance for two input sequences (fixed parameter version considering only two input strings) under the transposition and inversion metrics remains as future work as well.

5. Another research direction could be to improve the time complexity of the current algorithms.

6. Another interesting direction could be to devise algorithms that can handle simultaneous application of inversion and transposition. It will increase the utility of our algorithm since in many practical cases these two mutations happen side by side.

7. Finally, testing with real dataset to prove the validity of our pathway of detecting allelic heterogeneity. For this purpose we need gene sequence of some genetic diseases involving allelic heterogeneity, where only inversions or only transpositions cause the disorder.

Bibliography

[1] Pritom Ahmed, A. S. M. Sohidull Islam, and M. Sohel Rahman. A graph theoretic model to solve the approximate string matching problem allowing for translocations. In *IWOCA*, pages 169–181, 2012.

[2] Pritom Ahmed, A. S. M. Sohidull Islam, and M. Sohel Rahman. A graph theoretic model to solve the approximate string matching problem allowing for translocations. In *Combinatorial Algorithms*, pages 169–181. Springer, 2012.

[3] Pritom Ahmed, A. S. M. Sohidull Islam, and M. Sohel Rahman. A graph-theoretic model to solve the approximate string matching problem allowing for translocations. *Journal of Discrete Algorithms*, 23:143–156, 2013.

[4] Stephen F Altschul and David J Lipman. Trees, stars, and multiple biological sequence alignment. *SIAM Journal on Applied Mathematics*, 49(1):197–209, 1989.

[5] Yael T Aminetzach, J Michael Macpherson, and Dmitri A Petrov. Pesticide resistance via transposition-mediated adaptive gene truncation in drosophila. *Science*, 309(5735):764–767, 2005.

[6] 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.

[7] Amihood Amir, Haim Paryenty, and Liam Roditty. On the hardness of the consensus string problem. *Inf. Process. Lett.*, 113(10-11):371–374, 2013.

[8] Maryam Babaie and Seyed Rasoul Mousavi. A memetic algorithm for closest string problem and farthest string problem. In *Electrical Engineering (ICEE), 2010 18th Iranian Conference on*, pages 570–575. IEEE, 2010.

[9] David A Bader, Bernard ME Moret, and Mi Yan. A linear-time algorithm for computing inversion distance between signed permutations with an experimental study. *Journal of Computational Biology*, 8(5):483–491, 2001.

[10] Amir Ben-Dor, Giuseppe Lancia, R Ravi, and Jennifer Perone. Banishing bias from consensus sequences. In *Combinatorial Pattern Matching*, pages 247–261. Springer, 1997.

[11] Piotr Berman and Sridhar Hannenhalli. Fast sorting by reversal. In *Combinatorial Pattern Matching*, pages 168–185. Springer, 1996.

[12] John S Bertram. The molecular biology of cancer. *Molecular aspects of medicine*, 21(6):167–223, 2000.

[13] Hans L Bodlaender, Rodney G Downey, Michael R Fellows, and Harold T Wareham. The parameterized complexity of sequence alignment and consensus. *Theoretical Computer Science*, 147(1):31–54, 1995.

[14] Christina Boucher and Bin Ma. Closest string with outliers. *BMC bioinformatics*, 12(Suppl 1):S55, 2011.

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

[16] Vincent Burrus and Matthew K Waldor. Shaping bacterial genomes with integrative and conjugative elements. *Research in microbiology*, 155(5):376–386, 2004.

[17] Domenico Cantone, Simone Faro, and Emanuele Giaquinta. Approximate string matching allowing for inversions and translocations. In *Stringology*, pages 37–51, 2010.

[18] Alberto Caprara. Sorting by reversals is difficult. In *Proceedings of the first annual international conference on Computational molecular biology*, pages 75–83. ACM, 1997.

[19] Carlo Castellani. Cftr2: How will it help care? *Paediatric respiratory reviews*, 14:2–5, 2013.

[20] Zhi-Zhong Chen, Bin Ma, and Lusheng Wang. A three-string approach to the closest string problem. *Journal of Computer and System Sciences*, 78(1):164–178, 2012.

[21] Da-Jung Cho, Yo-Sub Han, and Hwee Kim. Alignment with non-overlapping inversions on two strings. In *Algorithms and Computation*, pages 261–272. Springer, 2014.

[22] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein, et al. *Introduction to algorithms*, volume 2. MIT press Cambridge, 2001.

[23] Xiaotie Deng, Guojun Li, Zimao Li, Bin Ma, and Lusheng Wang. Genetic design of drugs without side-effects. *SIAM Journal on Computing*, 32(4):1073–1090, 2003.

[24] Michel Marie Deza and Elena Deza. *Encyclopedia of distances*. Springer, 2009.

[25] Liviu P Dinu and R-T Ionescu. Clustering methods based on closest string via rank distance. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2012 14th International Symposium on*, pages 207–213. IEEE, 2012.

[26] Joaquin Dopazo, A Rodríguez, JC Sáiz, and F Sobrino. Design of primers for pcr ampiification of highly variable genomes. *Computer applications in the biosciences: CABIOS*, 9(2):123–125, 1993.

[27] Isaac Elias and Tzvika Hartman. A 1.375-approximation algorithm for sorting by transpositions. *Computational Biology and Bioinformatics, IEEE/ACM Transactions on*, 3(4):369–379, 2006.

[28] Simone Faro and Elisa Pappalardo. Ant-csp: An ant colony optimization algorithm for the closest string problem. In *SOFSEM 2010: Theory and Practice of Computer Science*, pages 370–381. Springer, 2010.

[29] Guillaume Fertin. *Combinatorics of genome rearrangements*. MIT press, 2009.

[30] Moti Frances and Ami Litman. On covering problems of codes. *Theory Comput. Syst.*, 30(2):113–119, 1997.

[31] Fernando C Gomes, Cláudio N Meneses, Panos M Pardalos, and Gerardo Valdisio R Viana. A parallel multistart algorithm for the closest string problem. *Computers & Operations Research*, 35(11):3636–3643, 2008.

[32] Szymon Grabowski, Simone Faro, and Emanuele Giaquinta. String matching with inversions and translocations in linear average time (most of the time). *Inf. Process. Lett.*, 111(11):516–520, 2011.

[33] Jens Gramm, Rolf Niedermeier, and Peter Rossmanith. Exact solutions for closest string and related problems. In *Algorithms and Computation*, pages 441–453. Springer, 2001.

[34] Jens Gramm, Rolf Niedermeier, Peter Rossmanith, et al. Fixed-parameter algorithms for closest string and related problems. *Algorithmica*, 37(1):25–42, 2003.

[35] Qian-Ping Gu, Shietung Peng, and Hal Sudborough. A 2-approximation algorithm for genome rearrangements by reversals and transpositions. *Theoretical Computer Science*, 210(2):327–339, 1999.

[36] Richard W Hamming. Error detecting and error correcting codes. *Bell System technical journal*, 29(2):147–160, 1950.

[37] Tzvika Hartman and Roded Sharan. A 1.5-approximation algorithm for sorting by transpositions and transreversals. *Journal of Computer and System Sciences*, 70(3):300–320, 2005.

[38] Yishan Jiao, Jingyi Xu, and Ming Li. On the k-closest substring and k-consensus pattern problems. In *Combinatorial Pattern Matching*, pages 130–144. Springer, 2004.

[39] Richard M Karp. Mapping the genome: some combinatorial problems arising in molecular biology. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 278–285. ACM, 1993.

[40] Chee-Seng Ku, David N Cooper, Constantin Polychronakos, Nasheen Naidoo, Mengchu Wu, and Richie Soong. Exome sequencing: dual role as a discovery and diagnostic tool. *Annals of neurology*, 71(1):5–14, 2012.

[41] J Kevin Lanctot, Ming Li, Bin Ma, Shaojiu Wang, and Louxin Zhang. Distinguishing string selection problems. In *Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, pages 633–642. Society for Industrial and Applied Mathematics, 1999.

[42] Eric S Lander, Robert Langridge, and Damian M Saccocio. Mapping and interpreting biological information. *Communications of the ACM*, 34(11):32–39, 1991.

[43] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10:707–710, 1966.

[44] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. In *Soviet physics doklady*, volume 10, page 707, 1966.

[45] Xuan Liu, Hongmei He, and Ondrej Sỳkora. Parallel genetic algorithm and parallel simulated annealing algorithm for the closest string problem. In *Advanced Data Mining and Applications*, pages 591–597. Springer, 2005.

[46] Roy Lowrance and Robert A. Wagner. An extension of the string-to-string correction problem. *J. ACM*, 22(2):177–183, 1975.

[47] K Lucas, M Busch, S Mössinger, and JA Thompson. An improved microcomputer program for finding gene-or gene family-specific oligonucleotides suitable as primers for polymerase chain reactions or as probes. *Computer applications in the biosciences: CABIOS*, 7(4):525–529, 1991.

[48] Dániel Marx. Closest substring problems with small distances. *SIAM Journal on Computing*, 38(4):1382–1410, 2008.

[49] Holger Mauch, Michael J Melzer, and John S Hu. Genetic algorithm approach for the closest string problem. In *Bioinformatics Conference, 2003. CSB 2003. Proceedings of the 2003 IEEE*, pages 560–561. IEEE, 2003.

[50] Gerhard Meisenberg and William H Simmons. *Allelic heterogeneity is the greatest challenge for molecular genetic diagnosis.* Elsevier Health Sciences, 2011.

[51] Giulio Pavesi, Giancarlo Mauri, and Graziano Pesole. An algorithm for finding signals of unknown length in dna sequences. *Bioinformatics*, 17(suppl 1):S207–S214, 2001.

[52] Pavel A Pevzner, Sing-Hoi Sze, et al. Combinatorial approaches to finding subtle signals in dna sequences. In *ISMB*, volume 8, pages 269–278, 2000.

[53] P Prasun, M Pradhan, and S Agarwal. One gene, many phenotypes. *Journal of Postgraduate Medicine*, 53(4):257–261, 2007.

[54] David Sankoff and Mathieu Blanchette. Multiple genome rearrangement and breakpoint phylogeny. *Journal of Computational Biology*, 5(3):555–570, 1998.

[55] Michael Schöniger and Michael S Waterman. A local algorithm for dna sequence alignment with inversions. *Bulletin of Mathematical Biology*, 54(4):521–536, 1992.

[56] Jeong Seop Sim and Kunsoo Park. The consensus string problem for a metric is np-complete. *Journal of Discrete Algorithms*, 1(1):111–117, 2003.

[57] James S Sutcliffe, Ryan J Delahanty, Harish C Prasad, Jacob L McCauley, Qiao Han, Lan Jiang, Chun Li, Susan E Folstein, and Randy D Blakely. Allelic heterogeneity at the serotonin transporter locus (slc6a4) confers susceptibility to autism and rigid-compulsive behaviors. *The American Journal of Human Genetics*, 77(2):265–279, 2005.

[58] Martin Tompa, Nan Li, Timothy L Bailey, George M Church, Bart De Moor, Eleazar Eskin, Alexander V Favorov, Martin C Frith, Yutao Fu, W James Kent, et al. Assessing computational tools for the discovery of transcription factor binding sites. *Nature biotechnology*, 23(1):137–144, 2005.

[59] Peter D Turnpenny and Sian Ellard. *Emery's elements of medical genetics.* Elsevier Health Sciences, 2011.

[60] Proutski V and Holme E. Primer master: A new program for the design and analysis of pcr primers. *Computer Applications in the Biosciences*, 12:253–255, 1996.

[61] Jan Van Leeuwen and Jan Leeuwen. *Handbook of theoretical computer science: Algorithms and complexity*, volume 1. Elsevier, 1990.

[62] Jan Van Leeuwen and Jan Leeuwen. *Handbook of theoretical computer science: Algorithms and complexity*, volume 1. Elsevier, 1998.

[63] Augusto F Vellozo, Carlos ER Alves, and Alair Pereira do Lago. Alignment with non-overlapping inversions in o (n^3)-time. In *Algorithms in Bioinformatics*, pages 186–196. Springer, 2006.

[64] Robert A. Wagner. On the complexity of the extended string-to-string correction problem. In *STOC*, pages 218–223, 1975.

[65] Sophia Yancopoulos, Oliver Attie, and Richard Friedberg. Efficient sorting of genomic permutations by translocation, inversion and block interchange. *Bioinformatics*, 21(16):3340–3346, 2005.

[66] E. Alper Yildirim. Two algorithms for the minimum enclosing ball problem. *SIAM Journal on Optimization*, 19(3):1368–1391, 2008.

[67] Karl-Heinz Zimmermann, Israel Martínez-Pérez, and Zoya Ignatova. Dna computing models. *DNA Computing Models:, ISBN 978-0-387-73637-2. Springer-Verlag US, 2008*, 1, 2008.