Two Problems in Computational Genomics

Nahla A. Belal

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

Lenwood S. Heath, Chair
Ruth Grene
Ayman Abdel-Hamid
T. M. Murali
João Carlos Setubal

February 14, 2011
Blacksburg, Virginia

Keywords: DNA sequences, Nucleotides, Alignment, Horizontal gene transfer, Algorithms
Copyright 2011, Nahla A. Belal

Two Problems in Computational Genomics

Nahla A. Belal

(ABSTRACT)

This work addresses two novel problems in the field of computational genomics. The first is whole genome alignment and the second is inferring horizontal gene transfer using posets. We define these two problems and present algorithmic approaches for solving them. For the whole genome alignment, we define alignment graphs for representing different evolutionary events, and define a scoring function for those graphs. The problem defined is proven to be NP-complete. Two heuristics are presented to solve the problem, one is a dynamic programming approach that is optimal for a class of sequences that we define in this work as breakable arrangements. And, the other is a greedy approach that is not necessarily optimal, however, unlike the dynamic programming approach, it allows for reversals. For inferring horizontal gene transfer, we define partial order sets among species, with respect to different genes, and infer genes involved in horizontal gene transfer by comparing posets for different genes. The posets are used to construct a tree for each gene. Those trees are then compared and tested for contradiction, where contradictory trees correspond to genes that are candidates of horizontal gene transfer.

GRANT INFORMATION

This work was supported by NSF-ITR Grant-0428344 for the Computational Models for Gene Silencing project.

Acknowledgments

First of all, no words are enough to thank my advisor Dr. Lenwood Heath, Lenny you have been a great support and you really taught me so many things, I could not have done this thesis without you. I am really grateful to how you helped me get better every day, and how you handled me during my down times, and how you were so understanding of my family commitments, baby Hussein is also very grateful.

I thank my committee members, Dr. Ayman Abdel-Hamid, Dr. Ruth Grene, Dr. Murali, and Dr. Joao Setubal for their support and comments which helped me improve my work all along the way. I also thank Dr. Zanoni Dias for his comments on Chapter 2.

Special thanks to Dr. Yasser Hanafy for just letting me into the VT-MENA program, Dr. Yasser, this was the best thing you let me do, and I know you have been through a lot of difficulties to let us get this degree, I am really grateful. And I also thank Dr. Sedki Riad for making this program available for us in Egypt, I could not have done it otherwise.

And certainly, thanks to Dr. Karen DePauw for allowing me to have a shorter residency period, this was very understanding of you.

I also thank Dr. Dennis Kafura, our former computer science department head, and Dr. Barbara Ryder, our current department head, for making everything as organized and as helpful as possible for foreign students.

I also thank Dr. Naren Ramakrishnan for his endless support and for the help he gave me all along the way, he really never made me feel that I am away from campus at any point in time. He always made the effort to keep me and I am sure all other students on the right track. And, also thanks to Hall Smith and Julie King, thanks for helping me with my plan of study and with my transfers, and my course registration, and just thanks for all the help you provided.

I would like to thank my course professors for the knowledge and experience they have given me in my courses, and for the effort made to make these courses available for us here in Egypt. I would also like to

thank Dr. Bevan for allowing me to take the Biochemistry course from Egypt.

I also thank all my professors that gave me tremendous help and support throughout my undergraduate studies and my masters. And thanks for my colleagues and friends for their endless love and support.

Last but not least, I thank my family, they have given me the strength to get this thesis done, they have given me love, and they have helped me find the time to work on my Ph.D. I thank my Mom for coming with me to Blacksburg for my residency period, thank you Mommy, you have left your work and everything here to come with me, but it was fun though! And also, many thanks for the babysitting you did for me. I also thank my husband for letting me go, he had to stay here all alone and I know it was not so easy on him. I also thank my husband for always helping me with my presentation rehearsals, and my studying schedules. I thank my Dad for his help and all the advice he gave me along the way, it really meant a lot to me. I thank my family in law for always encouraging me and lifting me up when I felt so down. And also thanks to my brother and his family for all the love and support they gave me. And, I also give a million thanks to everyone who have helped me with my son Hussein to help me accomplish this, it would have never ever worked without this help. Now, to my little baby Hussein, I love you so much, you are the greatest gift from God, but you could have let me SLEEP some more to have more concentration!

Finally, I thank God for making everything possible and getting me through this.

TABLE OF CONTENTS

1 Introduction **1**

1.1 Organization 2

2 Whole Genome Alignment **3**

2.1 Introduction 3

2.2 Definitions 4

2.3 Literature Review 6

2.4 Advances in Presented Model 10

2.5 Representation of Different Phenomena 10

2.6 Problem Definition 13

2.7 Scoring Mechanism 13

2.8 Breakable Arrangements 15

2.8.1 Identifying Breakable Arrangements 19

2.8.2 Generating Breakable Arrangements 23

2.8.3 Counting Breakable Arrangements 24

2.9 NP-Completeness Proof 27

2.10 Dynamic Programming for Whole Genome Alignment 28

2.11 Arrangements and Other Alignments 34

2.12 A Greedy Algorithm for Whole Genome Alignment 36

3 Inferring HGT from Posets 45

3.1 Introduction 45

3.2 Definitions 47

3.3 Literature Review 50

3.4 Posets and Trees 57

3.4.1 Constructing an S-tree From a Set of Posets $\mathcal{P}$ 57

3.4.2 Generating a Set of Posets $\mathcal{P}$ From a Given S-tree 70

3.4.3 Relating Posets to Trees 72

3.5 Refinement of Trees 75

3.6 Inferring HGT from Posets 83

3.7 Comparing the Presented Technique to Others 87

4 Conclusions 92

References 94

LIST OF FIGURES

2.1 An example alignment graph for two identical sequences. 5

2.2 An example alignment graph for two non-identical sequences with black and red edges. 5

2.3 Aligning identical sequences. 11

2.4 The effect of inserting AG into the second sequence in Figure 2.3. 11

2.5 The effect of deleting CT from the second sequence in Figure 2.3. 11

2.6 The effect of duplication in the first sequence in Figure 2.3. 12

2.7 The effect of mutation on Figure 2.3. 12

2.8 The effect of reversal on Figure 2.3. 12

2.9 The effect of transposition on Figure 2.3. 13

2.10 An alignment graph that starts by a duplication. 14

2.11 Scoring example 1. 15

2.12 Scoring example 2. 15

2.13 Breakable Arrangement. 16

2.14 Non-breakable Arrangement. 16

2.15 A Break Tree for $A = 2, 1, 7, 5, 6, 8, 9, 10, 4, 3$. 18

2.16 Possible breaks for $i + 2, i, i + 3, i + 1$. 18

2.17 Possible breaks for $i + 1, i + 3, i, i + 2$. 18

2.18 Break Tree for the arrangement $A = 7, 6, 5, 9, 8, 4, 3, 2, 1$. 20

2.19 Identifying Breakable Arrangements. 21

2.20 Subroutine to check if a subarrangement is consecutive. 21

2.21 Generating Breakable Arrangements. 24

2.22 Counting Breakable Arrangements. 26

2.23 Algorithm for RegAlign. 31

2.24 Algorithm for precomputing alignments of subsequences without breaks. 31

2.25 Algorithm for the Whole Genome Alignment. 32

2.26 Alignment graph for AC and CA 37

2.27 The greedy algorithm GreedyAlign. 37

2.28 The Preprocessing Step. 37

2.29 The preprocessing step with specified subsequence size. 38

2.30 The alignment step. 38

2.31 The scoring step. 39

2.32 Alignment graph for S_1 and S_2 with two breaks and two mutations. 41

2.33 Alignment graph for S_1 and S_2 with five breaks and one mutation. 41

2.34 Alignment graph for S_1 and S_2 with five breaks and one mutation. 41

3.1 A trivial S-tree with a minimum number of nodes. 47

3.2 An evolutionary S-tree with 5 taxa. 47

3.3 Refinement of T_1 to T_2. 48

3.4 An example of a Hasse diagram. 49

3.5 An example of a tree to test compatibility with posets. 50

3.6 An example of a poset compatible with the tree in Figure 3.5. 50

3.7 An example of a poset incompatible with the tree in Figure 3.5. 51

3.8 An undirected graph with cliques representing siblings. 60

3.9 Diagram for posets. 62

3.10 An undirected graph corresponding to the matrix shown in Table 3.2. 64

3.11 Tree corresponding to the posets in Figure 3.9. 64

3.12 Diagrams for posets. 65

3.13 Tree corresponding to the posets in Figure 3.12. 65

3.14 Algorithm to construct an S-tree from a set of posets $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$. 66

3.15 Algorithm to validate an $n \times n$ matrix A. 66

3.16 Algorithm to find the maximum of a matrix A. 67

3.17 Algorithm to find a clique with edge labels equal max. 67

3.18 Algorithm to add elements of a clique as siblings in a tree T. 67

3.19 Set of posets $\mathcal{P}$ for the set of species $S = \{s_1, s_2, s_3, s_4, s_5\}$. 69

3.20 The tree corresponding to the set of posets $\mathcal{P}$ in Figure 3.19. 70

3.21 An example of how the poset corresponding to s_1 is generated. 71

3.22 Algorithm to generate a set of posets $\mathcal{P}$ from an S-tree T. 72

3.23 An example to illustrate Theorem 23. 74

3.24 Refinement of T_1 and T_2 into T_3. 76

3.25 Two contradicting trees. 77

3.26 T_3 is the minimum common refinement of T_1 and T_2. 79

3.27 An example to illustrate the algorithm MinCommonRefine. 81

3.28 An example to illustrate the algorithm ConstructTree2Partitions. 83

3.29 Algorithm to find the minimum common refinement of two trees. 83

3.30 Algorithm to find the 2-partitions set of a given tree. 84

3.31 Algorithm to construct a tree from its 2-partitions set. 85

3.32 Algorithm to detect contradiction between two trees. 86

3.33 Algorithm to infer HGT. 87

3.34 An example to illustrate the algorithm InferHGT. 88

3.35 The possible HGT events for the example in Figure 3.34. 89

3.36 Flowchart for Detecting HGT 90

LIST OF TABLES

2.1 Arrangements generated by Trotter's algorithm. 23

2.2 Scoring function used by RegAlign 30

2.3 Dynamic Programming: Example 1. 33

2.4 Output of the preprocessing step for AC and $S_2 = CA$. 43

2.5 Greedy algorithm: Example 1. 44

3.1 Matrix A for the set of posets $\mathcal{P}$. 60

3.2 Matrix A for posets in Figure 3.9. 61

3.3 Matrix A for posets in Figure 3.9 after reducing s_1 and s_2. 61

3.4 Updated matrix A for posets in Figure 3.9 after reducing s_4 and s_5. 61

3.5 Updated matrix A for posets in Figure 3.9. 63

3.6 Matrix A for posets in Figure 3.12. 63

3.7 Updated matrix A for posets in Figure 3.12. 63

3.8 Matrix A for the posets in Figure 3.19. 68

3.9 Updated matrix A for the posets in Figure 3.19. 69

Chapter 1

Introduction

In this work, we introduce two theoretical problems in computational genomics, namely whole genome alignment and detecting horizontal gene transfer (HGT) using partial order sets (posets). The two problems are presented from a theoretical point of view, where some parts have been implemented, and other parts still need to be taken to the practical side.

The first problem discussed in Chapter 2 introduces a more detailed level of genome analysis, where the analysis is performed at the detailed level of a nucleotide. An earlier version of our results is in [8]. The input to the problem is two genomic sequences, that are decomposed and compared to find optimal alignments between subsequences. The problem of optimally aligning two sequences is proven to be NP-complete. Two algorithms are presented to approximately solve the problem using the alignment graphs defined in Chapter 2. The first algorithm is a dynamic programming algorithm that is proven optimal for a class of sequences that is defined in Chapter 2 as breakable arrangements. Whereas, the second algorithm is a greedy algorithm that is not necessarily optimal but it allows for reversals, unlike the dynamic programming approach.

The second problem that is developed in Chapter 3 uses partial order sets (posets) to infer horizontal gene transfers (HGT). An earlier version of our results is in [7]. This is done by using a poset for each species per gene, where those posets are used to construct trees that are then compared and tested for contradiction. Trees that are found to be contradictory correspond to genes that are involved in HGT.

1.1 Organization

This dissertation is organized as follows. The whole genome alignment problem is developed in Chapter 2. Section 2.1 introduces the problem, Section 2.2 gives the necessary notations and definitions, and the related literature is given in Section 2.3. The remaining sections in Chapter 2 present the details of the problem, where a problem definition is given, along with the developed algorithms and theorems. Chapter 3 covers the problem of inferring horizontal gene transfers from partial order sets. The chapter starts by an introduction in Section 3.1, then Section 3.2 gives the definitions and notations, and the literature review is given in Section 3.3. The rest of the chapter covers the problems defined, algorithms, and theorems developed. We summarize and conclude in Chapter 4.

Chapter 2

Whole Genome Alignment

This chapter starts by an introduction to the problem of whole genome alignment, then we give the defintions and notations needed in the rest of the chapter. Section 2.3 summarizes the related literature. The rest of the chapter is dedicated to problem defintions, algorithms, and theorems, where two algorithms are described; one is a dynamic programming algorithm and the other is a greedy approach.

2.1 Introduction

Synteny is a slippery concept. In some contexts, it is used to denote the collocation of several genetic loci on one chromosome. In other contexts, it is used to describe the preserved order of genes of related species on chromosomes. Analysis of this preserved order has many applications in studying evolution and phylogenetic relationships. Most research, the overwhelming majority, simplifies analysis of a genome to the level of "gene" or "synteny block". This simplification has a number of problems. It is difficult to know whether a sequence is actually a gene and, if it is a gene, where the start codon, exons, introns, and stop codon are. In addition, a gene may have multiple transcripts, a sequence may be a pseudogene, and the sequences of two genes may overlap. Especially ignored is the absolute location of genes on a chromosome, as opposed to the less informative relative location.

A new synteny-like term could be defined to compare genomes at the detailed level of nucleotides, taking into account the length of genes and their absolute positions. One could think of genomes as being composed of parts (subsequences) and compare genomes by comparing the parts that constitute each genome. One challenge with that approach is that the parts are not defined in advance. Therefore, an algorithm has to

define the parts so as to optimize a similarity measure. Given two genomes with parts that are very similar and parts that are not, an algorithm may seek to infer how similar the two genomes are. This takes us to our concept, of whole genome alignment.

Let G_1 and G_2 be two genomes. We wish to define a distance between G_1 and G_2 based on an optimal "alignment". Here, "alignment" is something more than the traditional pairwise or multiple sequence alignment. Suppose G_1 and G_2 are identical, except for a substring in the middle of G_2 that is the Watson-Crick complement of the corresponding substring in G_1. In genome rearrangement terms, this is a reversal. Now, real genomes do not evolve so neatly. There will not be crisp boundaries for the reversal. And, there will be point mutations that occur, as well as sequence insertions and deletions. However, ultimately, these two genomes have a common ancestor genome and the parts of the current genome could, in principle, be "mapped" or "aligned" to parts of the ancestor and hence the two genomes could be piecewise aligned to each other.

Because of genome rearrangements and the sizes of genomes, an optimal alignment of the suggested sort will be expensive to compute. However, impractical exact algorithms can be developed, as well as practical approximations. A whole genome alignment looks for a decomposition of G_1 and a decomposition of G_2 into substrings such that paired substrings are optimally aligned. It is possible that, in alignment, some strings will undergo Watson-Crick complementation at the nucleotide level. Two scenarios can occur. One scenario is that there may be gene duplications that separate the two genomes. In this case, multiple sequences from one genome will align to a single sequence in the other genome. The other scenario is actual movement within the genome. A gene sequence or other genomic sequence may move from one part of a genome to another. The sequence may also be reversed (on the opposite strand) from where it started. In these cases, there are "breaks" forced in the alignment.

2.2 Definitions

Let $S_1 = a_1 a_2 \cdots a_m$ and $S_2 = b_1 b_2 \cdots b_n$ be two genomic sequences. We define a class of edge-colored, mixed directed and undirected graphs representing alignments of S_1 and S_2. There are two disjoint sets of nodes, $U = \{u_1, u_2, \ldots, u_m\}$ and $V = \{v_1, v_2, \ldots, v_n\}$. Node u_i is labeled a_i, and node v_i is labeled b_i. The directed edges are colored blue and constitute this set of edges:

$$E_{\text{blue}} \quad = \quad \{(u_i, u_{i+1}) \mid 1 \leq i < m\} \cup \{(v_i, v_{i+1}) \mid 1 \leq i < n\}.$$

There are two sets of undirected edges. The set E_{black} consists of edges colored black, each of which connects a node in U to a node in V. The set E_{red} consists of edges colored red, each of which connects a node in U to a

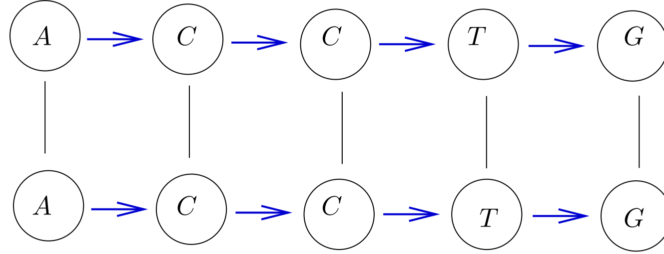


Figure 2.1: An example alignment graph for two identical sequences.

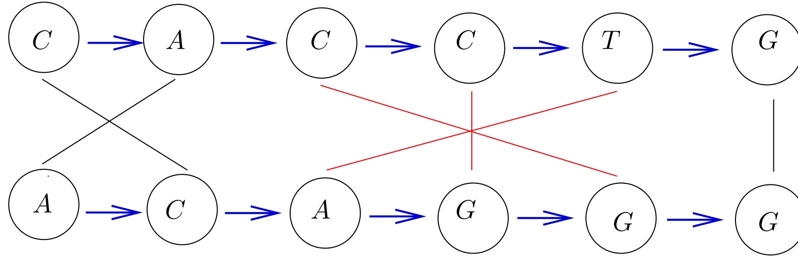


Figure 2.2: An example alignment graph for two non-identical sequences with black and red edges.

node in V. E_{black} and E_{red} are constrained such that any cycle of black and red edges contains an even number of red edges. For each choice of E_{black} and E_{red}, the resulting mixed graph $G = (U, V; E_{\text{blue}}, E_{\text{black}}, E_{\text{red}})$ is an *alignment graph* for S_1 and S_2. The blue edges form two paths, one for each genomic sequence. Intuitively, a black edge connects two nucleotides that are on the same strand, while a red edge connects two nucleotides that are on opposite strands.

Figure 2.1 shows an example of an alignment graph for two identical sequences. And, Figure 2.2 shows an example of an alignment graph for two non-identical sequences with black and red edges.

Let $G = (U, V; E_{\text{blue}}, E_{\text{black}}, E_{\text{red}})$ be an alignment graph for S_1 and S_2. A *free node* is one that is unaligned, that is, having neither black nor red incident edges. Two edges e_{ij} and e_{kl} in G are *adjacent in* S_1 when nodes i and k in S_1 are separated by zero or more free nodes, namely $u_{i+1}, u_{i+2}, \ldots, u_{k-1}$, and nodes j and l in S_2 are also separated by zero or more free nodes, namely $v_{j+1}, v_{j+2}, ..., v_{l-1}$. Similarly, two edges e_{ij} and e_{kl} in G are *adjacent in* S_2 when nodes j and l in S_2 are separated by zero or more free nodes, namely $v_{j+1}, v_{j+2}, \ldots, v_{l-1}$, and nodes i and k in S_1 are also separated by zero or more free nodes, namely $u_{i+1}, u_{i+2}, ..., u_{k-1}$. In case of red edges, free nodes in S_1 are $u_{i-1}, u_{i-2}, ..., u_{k+1}$ and in S_2 are $v_{j-1}, v_{j-2}, ..., v_{l+1}$. A *break* in the alignment graph G occurs when there exists two adjacent edges in S_1 or in S_2, where one edge $\in E_{\text{black}}$ and the other $\in E_{\text{red}}$ or vice versa. A *break* also occurs when two edges are

adjacent in S_1 but not in S_2 or vice versa.

A *mutation* is a change in the value of a single nucleotide. On an alignment graph, a mutation is either a black edge connecting a node from U and a node from V with different labels or a red edge connecting a node from U and a node from V with labels that are not complementary.

A *free node* is either a node in U or a node in V that is unaligned, that is, having neither black nor red incident edges.

A *duplication* is a node in the graph with more than one incident edge, either red or black.

2.3 Literature Review

Sequence alignment is a way of arranging genomic sequences to identify similarities between sub-regions that point to some functional, structural, or evolutionary relationship. If two sequences in an alignment share a common ancestor, then differences between the two sequences could be interpreted as point mutations, insertions, deletions, or other evolutionary events that help us infer the evolutionary distance between the two sequences. Pairwise alignment techniques are those that involve two input sequences to be aligned, whereas multiple sequence alignment, MSA, techniques are those that involve three or more sequences.

According to Blanchette [10], two strategies are known to solve alignment problems, namely, global alignment and local alignment. In global alignment, the sequences are considered as a whole, and the entire sequences are given as input to the alignment program. This imposes the constraint on the alignment that orthologous genes must be colinear, which prevents the detection of rearrangements and duplications. A general global alignment technique is the Needleman-Wunsch algorithm [54], which is based on dynamic programming. On the other hand, local alignment considers fragments of the input sequences, and aligns fragments rather than whole sequences, which overcomes the deficiency of global alignment that co-linearity must be maintained, at the expense of efficiency, where a higher probability of false alignments is expected. The Smith-Waterman [76] algorithm is a general local alignment algorithm, also based on dynamic programming. Some techniques combine both global and local alignment strategies, such as the hybrid technique named glocal [20]. Due to the large amounts of data in each sequence, exact methods are impractical, and require huge amounts of memory and very long running times. Many heuristics have been developed to overcome the impracticality of exact algorithms.

For pairwise alignment, the major alignment methods are highlighted. Dot matrix methods construct plots with dots representing matching characters, where one sequence is placed at the topmost row, and the other

sequence is placed on the leftmost column. This method is time consuming but it is simple and easy to visualize. Some tools were developed using this technique, among which are the DNADot[1] web-based tool [15] and the DOTLET[2] Java based tool [59]. These are both global alignment techniques. Another technique is dynamic programming, which can be used for both global and local alignments, as previously mentioned. Dynamic programming techniques use scoring functions to find optimal solutions, and once a scoring function is defined, and here is where the challenge lies, a dynamic programming algorithm is guaranteed to find the optimal answer, if the scoring function defined is summed column-wise.

Several techniques are also available for performing multiple sequence alignment. These techniques can also be used to perform pairwise alignment. Some of the techniques are sequence-based, like CLUSTAL W [82]. Others use secondary structure like MUMMALS [62] or 3D structures as in M-Coffee [87]. However, since these methods depend on structural information, they can only be used for sequences whose corresponding structures are known through X-ray crystallography or NMR spectroscopy. Other aligners are genome aligners, for example MUMmer [23]. There are several techniques used in multiple sequence alignment. There are programs that use seeded pairwise alignment, these programs use heuristics for aligning large sequences, where a seed is defined as a short highly conserved match, and a local alignment is considered only if it contains this seed. Nearly all seeded pairwise alignment programs are based on BLAST algorithms [3]. Examples of programs that employ this technique are BLASTZ [73], LAGAN [20], CHAOS [19], AVID [16], and MUMmer [23]. Other programs that perform multiple sequence alignment either use multiple pairwise alignments or perform true multiple alignment. MLAGAN [20], CLUSTAL W [82], and MAVID [17] use progressive multiple sequence alignment, a technique based on progressive alignment [26], where a phylogenetic tree is first inferred by performing pairwise alignments. Another non-phylogenetic alignment technique is consistency-based multiple sequence alignment [52, 66, 80, 92]. Also, the MAUVE tool presented in [22] allows alignments with rearrangements. However, the results obtained by MAUVE were shown to be best on closely-related organisms. It also does not support the alignment of large regions shared by subsets of the genomes, nor the rearranged regions shared by subsets of the genomes. Moreover, it does not perform well when there are many duplicated segments. Another drawback of the MAUVE algorithm is that it requires manual data entry for some parameters. The progressive version of the MAUVE algorithm allows for alignment of more divergent genomes and it reduces the manual adjustment of the alignment scoring parameters. It also aligns regions conserved among subsets of the input genomes. However, it still has the limitation of being substantially slow, consumes more memory than the original MAUVE algorithm, and there still remains to be manual adjustment. The algorithm used by the Mauve tool performs a number of steps. First, it performs local alignment to find multiple maximal unique matches (multi-MUMs). Then,

[1] `http://www.vivo.colostate.edu/molkit/dnadot/index.html`
[2] `http://www.isrec.isb-sib.ch/java/dotlet/Dotlet.html`

those multi-MUMs are used to calculate a phylogenetic guide tree. A subset of the multi-MUMs are selected and used as anchors which are partitioned into locally collinear groups (LCBs). Recursive anchoring is then performed within each LCB, and also outside the LCBs to identify additional alignment anchors. The guide tree calculated is then used to perform progressive alignment of each LCB.

Other research directions define the alignment problem in terms of a graph problem. The local multiple sequence alignment problem could be viewed as finding Eulerian paths in a graph [93]. Raphael et al. [68] use de Bruijn graphs to perform multiple sequence alignments. They present a technique, A-Bruijn Alignment, which represents an alignment as a directed graph, and they present methods to detect cycles and reversals. This method consists of two tasks, the first task is to find a graph that represents the domain structure and the second is to find a mapping of each sequence to this graph. The graph is constructed from a set of pairwise local alignments. The graph representation presented in [68] is used in the alignment of protein sequences with shuffled or repeated domain structure, and also in the alignment of proteins containing domains that are not present in all proteins, domains that are present in different orders in different proteins, and domains that are present in multiple copies in some proteins. Moreover, the technique they present detects duplications and inversions. In [86], an algorithm for alignment with non-overlapping inversions is presented, this algorithm uses edit graphs, and those graphs allow for three edit operations, insertions, deletions, and substitutions. Their algorithm considers only non-overlapping inversions. An edit graph consists of one set of vertices, and three sets of edges, edges horizontal to a vertex, edges diagonal to a vertex, and edges vertical to a vertex. A path on this graph corresponds to an alignment of two sequences.

Phuong at al. [64] present an algorithm, ProDA, for aligning protein sequences with repeated and shuffled domains. The algorithm they present computes local alignments for every pair of sequences, then clusters those alignments into blocks of globally alignable subsequences to determine block boundaries and resolve inconsistencies between pairwise alignments to be able to find the multiple alignment between blocks.

Paten et al. [61] present two programs, namely Enredo and Pecan, for multiple genome alignment. They divide the problem of multiple genome alignment into two stages. The first stage partitions the input genomes into a set of colinear segments; and this is carried out by the program Enredo, which handles rearrangements and duplications. The second stage generates a base pair level alignment map for each colinear segment; this is carried out by the program Pecan, which makes the alignment problem practical on a large scale.

Ma et al. [47] present a polynomial-time algorithm to find the most parsimonious evolutionary history of any set of related genomes. They start by a single genome, called the root genome, taken from a species called the original species. Evolution of the root genome takes place through the evolutionary events, namely, loss and gain of chromosomes, duplication, and rearrangement. When a speciation event occurs, an identical

copy of the genome is made, and then copies evolve independently.

Ergun et al. [27] present a linear time greedy algorithm that computes sequence similarity with rearrangements. They define two edit operations, character edits, which allows insertions, deletions and replacements, and segment edits, which allows substring relocations, deletions, and duplications. The distance between two sequences is the minimum number of edit operations needed to transform one string into the other.

An algorithm for multiple genome alignment without a reference genome is implemented as part of the VISTA genome pipeline [24]. The algorithm is based on progressive alignment. After aligning two genomes, the algorithm builds synteny blocks based on the outgroups, where outgroups are the genomes that are not yet aligned. This helps the algorithm then align more distant genomes.

In [91], a method for finding genomic distances is presented. The method is based on a comparison graph generated for two genomes, and the distance is calculated from breakpoints and cycles in the graph. In their method, they also define a double-cut-and-join operation that accounts for the events of inversion, translocation, fission, and fusion.

Otu and Sayood [58] propose a new sequence distance measure. Their method uses Lempel-Ziv complexity for finding the relative distances between sequences, and they use the distance matrix obtained to construct phylogenetic trees.

Varre et al. [85] present another family of genome distances, namely transformation distances. Transformation distances are calculated in terms of segment-based events, like insertion and deletion of sequence segments. The algorithm presented computes the exact distance between two input sequences, without taking the order of residues into account, and hence, the algorithm is able to account for duplications and translocations.

A recent genome alignment tool, Mugsy, was presented by Angiuoli and Salzberg [4]. Mugsy is a fast alignment tool that does not require a reference sequence. It identifies a number of events including duplications, rearrangements, gain and loss. Mugsy identifies homologous regions including rearrangements and duplication using Nucmer, and it also uses the segment-based multiple alignment method provided by the SeqAn C+ library. Locally colinear blocks are identified by Mugsy from an alignment graph.

Comparing our technique with other available techniques, the algorithm is categorized as a heuristic genome alignment technique that uses extended dynamic programming. The algorithm works on DNA sequences, on the level of a nucleotide, where colinearity is not necessarily kept. To allow for more events, we define a break to detect possible events that involve relocation of nucleotide subsequences. This will be further expanded upon in the coming sections.

2.4 Advances in Presented Model

The model presented in this research adds the ability to align at the nucleotide level. This makes it possible to align two sequences without the need to identify a gene sequence, start/end codons, exons, introns, and pseudogenes. The model presented is a graph-based model that uses three kinds of edges, blue, black, and red edges, represented by E_{blue}, E_{black}, and E_{red}, respectively. Blue edges are used to identify the location of nucleotides. Black and red edges make it possible to detect evolutionary events other than insertions, deletions, and mutations identified by ordinary alignment techniques. For example, the model presented allows the detection of rearrangements, reversals, and duplications. In other words, these edges allow alignment at the level of nucleotides.

2.5 Representation of Different Phenomena

Start with two DNA sequences $S_1 = a_1a_2 \cdots a_m$ and $S_2 = b_1b_2 \cdots b_n$. A mixed graph (both directed and undirected edges) can be constructed to represent an alignment of those two sequences. The graph will be $G = (U, V; E_{\text{blue}}, E_{\text{black}}, E_{\text{red}})$, where $U = \{u_1, u_2, \ldots, u_m\}$ is a set of m vertices having u_i labeled with a_i and $V = \{v_1, v_2, \ldots, v_n\}$ is a set of n vertices having v_i labeled with b_i; the vertices represent nucleotide positions in S_1 and S_2. The set E_{blue} of blue edges consists of directed edges linking the U-vertices into a path and the V-vertices into a path. The set E_{black} of black edges consists of undirected edges linking a u_i and a v_j that are descended from a common ancestor and are uncomplemented. The set E_{red} of red edges consists of undirected edges linking a u_i and a v_j that are descended from a common ancestor and are complemented.

For example, if the two sequences are identical, we get the graph shown in Figure 2.3; such a graph is called a *perfect alignment graph.* The nodes constituting the top row represent the first input sequence, while the nodes in the bottom row represent the second input sequence. The blue edges between nodes of the same sequence indicate the order of nodes in the sequence. A black edge connecting a node from the top row to a node from the bottom row represents a match. Red edges connect complemented nodes. To consider different phenomena, we modify the second sequence, the one in the bottom row, and formalize the effect of each evolutionary event with respect to how the edges vary. First, consider the event of insertion. Insertion results in repositioning of the nucleotides and having new nucleotides in the second sequence that are unaligned. Therefore, with insertion, we add the nucleotides without undirected edges. However, blue edges are added to connect the new inserted subsequence. This is illustrated in Figure 2.4, where AG is inserted in the second sequence. The second event to consider is deletion. This is a trivial case, and the effect

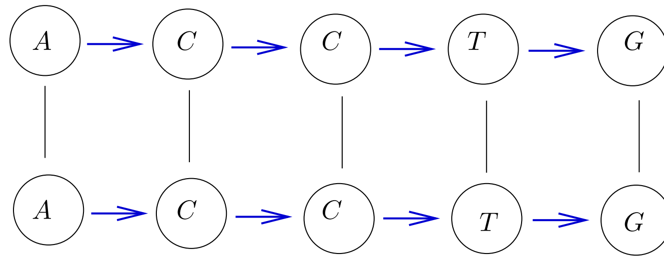


Figure 2.3: Aligning identical sequences.

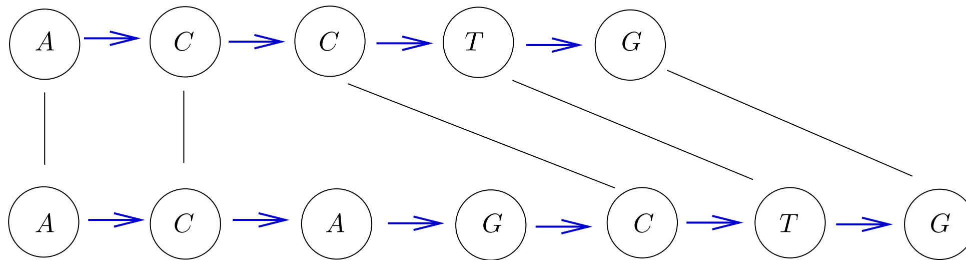


Figure 2.4: The effect of inserting AG into the second sequence in Figure 2.3.

is straightforward, deleting a node results in deleting all its incident edges. The deletion event is illustrated in Figure 2.5, where CT is deleted from the second sequence. The CT in the first sequence is therefore unaligned.

In case of subsequences being duplicated, the duplicate nodes are aligned to the same nodes in the other sequence, this gives a one-to-many relation, as shown in Figure 2.6. New blue edges are added, as well as black edges to align the duplicate nodes.

Point mutations do not have an effect on the edges of the graph. Since edges indicate common ancestry, mutation does not remove this common ancestor and does not establish a non-existent common ancestor.

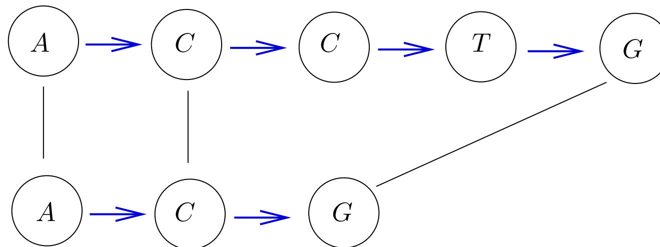


Figure 2.5: The effect of deleting CT from the second sequence in Figure 2.3.

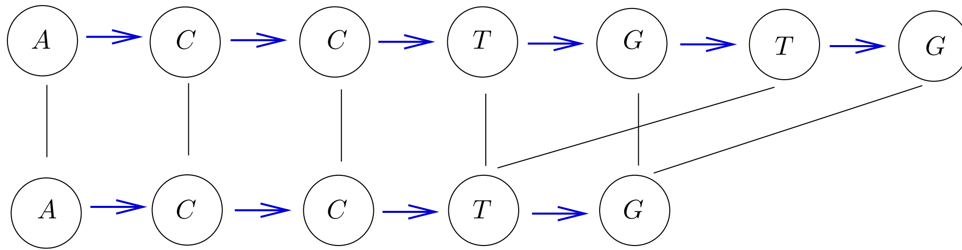


Figure 2.6: The effect of duplication in the first sequence in Figure 2.3.

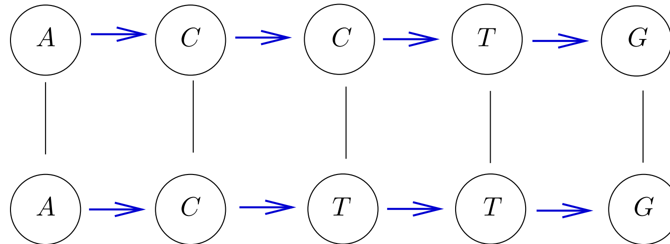


Figure 2.7: The effect of mutation on Figure 2.3.

This is illustrated in Figure 2.7, where a node of value C is changed into T.

In reversals, we have to represent complementation as well. We have a new order for the nucleotides, along with complementation of some nucleotides. The effect on the graph is only in the position of edges, but they still connect the same nucleotides they connected before the reversal. This is shown in Figure 2.8, the subsequence CCT is reversed into AGG, where red edges represent edges with complementation, and black edges represent the regular alignment.

Transposition results in repositioning a subsequence, and hence, the edges are repositioned accordingly, as shown in Figure 2.9. The blue edges indicate the change in location, and hence the black edges are also

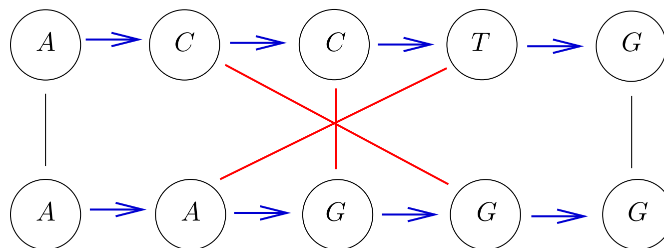


Figure 2.8: The effect of reversal on Figure 2.3.

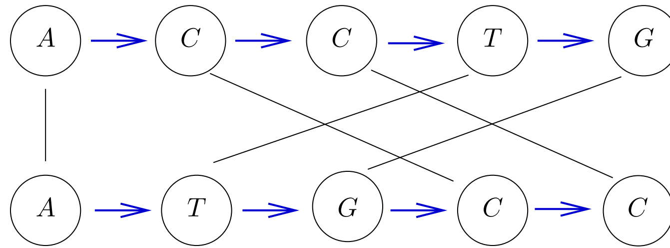


Figure 2.9: The effect of transposition on Figure 2.3.

repositioned.

2.6 Problem Definition

Suppose we have a scoring function s for alignment graphs. (We discuss s in Section 2.7.) Then, we have the following computational problem.

Optimal Whole Genome Alignment Graph

INSTANCE: Two DNA sequences S_1 and S_2 and weights w_b, w_m, w_f, and w_d.

SOLUTION: An alignment graph G for S_1 and S_2 that minimizes the alignment score, $s(G)$.

This is the computational problem that we address.

2.7 Scoring Mechanism

Given an alignment graph $G = (U, V; E_{\text{blue}}, E_{\text{black}}, E_{\text{red}})$, we must calculate a score s that represents the level of alignment intrinsic in G. The score $s(G)$ can be a function of the following components of G:

- The presence, absence, and color of edges.
- The indices of the nodes.
- The labels of the connected nodes (u_i and v_j).
- The number of edges incident to each node in sets U and V.

Special cases can arise at the ends of a sequence, for example, if a sequence starts by a duplication. Therefore, for the purposes of scoring, two additional nodes are added to each sequence, a node at the beginning and

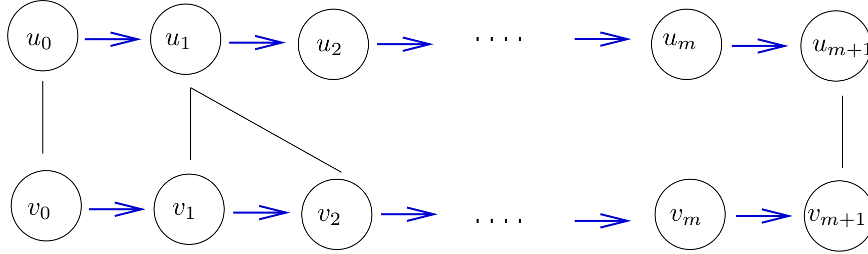


Figure 2.10: An alignment graph that starts by a duplication.

another one at the end. Hence, given two sequences $U = u_1, ..., u_m$ and $V = v_1, ..., v_n$, nodes u_0 and u_{m+1} are added to U, and nodes v_0 and v_{n+1} are added to V, where an alignment graph connects node u_0 to v_0 and node u_{m+1} to v_{n+1} using black edges. This is shown in Figure 2.10.

Those graph characteristics guide us to calculate an alignment score that accounts for different evolutionary events. Those events can be categorized into the following categories:

- Breaks
- Mutations
- Free nodes
- Duplications

Each of the items above are defined in Section 2.2.

Given two identical sequences, an optimal alignment is a perfect alignment, which has a score $s = 0$. Each break, mutation, free node, and duplication adds a penalty to $s(G)$. For each break, we add a penalty w_b to the score, for mutations, we add a penalty w_m, for free (unaligned) node, we add a penalty w_f, and for duplications, we add a penalty w_d. This yields the following formula for the score:

$$s(G) \quad = \quad bw_b + mw_m + fw_f + dw_d,$$

where b is the number of breaks, m is the number of mutations, f is the number of free nodes, and d is the number of duplications.

In the following examples, we use the following values: $w_b = 1$, $w_m = 4$, $w_f = 4$, and $w_d = 4$. Figure 2.11 is an example to illustrate the described scoring mechanism. In Figure 2.11, $U = \{u_1, u_2, u_3, u_4, u_5, u_6, u_7\}$ and $V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8\}$. Two breaks can be detected, at (u_2, v_4) and (u_5, v_5). Also, a mutation is seen between nodes u_6 and v_7. Finally, there is one free node, v_6. This results in $s(G) = 2w_b + w_m + w_f = 10$.

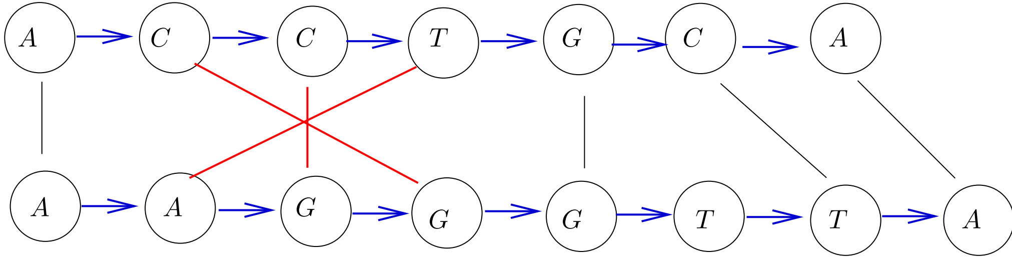


Figure 2.11: Scoring example 1.

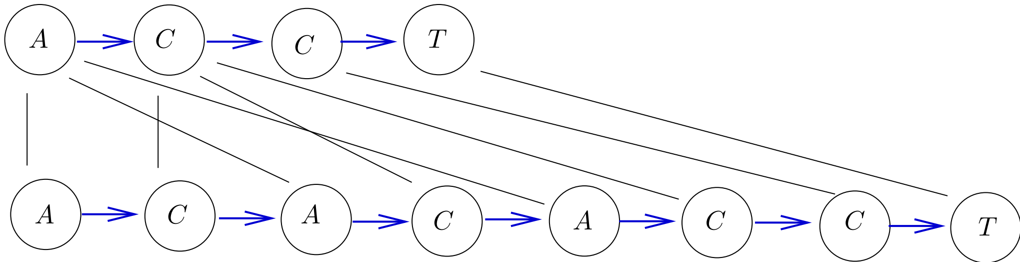


Figure 2.12: Scoring example 2.

Figure 2.12 is another example to illustrate our scoring mechanism. In Figure 2.12, $U = \{u_1, u_2, u_3, u_4\}$ and $V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8\}$. The nodes in U are assigned the following labels, respectively, A, C, C, T, and the nodes in V are assigned the labels A, C, A, C, A, C, C, T, respectively. Here, we see three edges incident to nodes u_1 and u_2, this is translated into four duplications, two per node. There is one break at (u_3, v_7). This gives a score $s(G) = 4w_d + w_b = 17$.

2.8 Breakable Arrangements

The algorithm presented in Section 2.10 is proven optimal for a special category of sequences, which brings us to the term *arrangement*, which is defined later in this section. Before *arrangements* are presented, let's clarify how arrangements relate to nucleotide sequences. Given two aligned nucleotide sequences S_1 and S_2, the two sequences are broken into blocks. The blocks are directly related to the breaks in the alignment, where a break is as defined in Section 2.2. Therefore, the blocks are separated by breaks. Some arrangements, namely breakable arrangements, are defined here and proven to be optimally aligned using the dynamic programming algorithm presented in Section 2.10. As an example, take $S_1 = ACCCGT$ and $S_2 = CACTGC$. An alignment graph for S_1 and S_2 is shown in Figure 2.13.

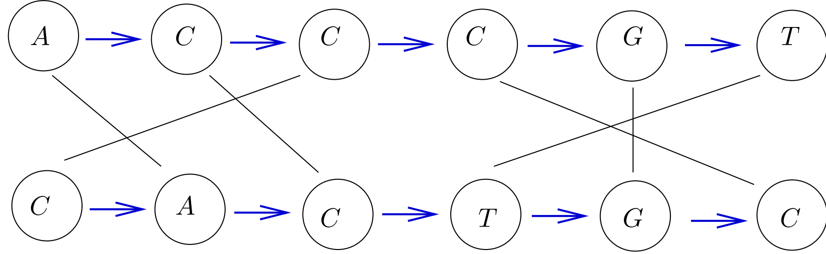


Figure 2.13: Breakable Arrangement.

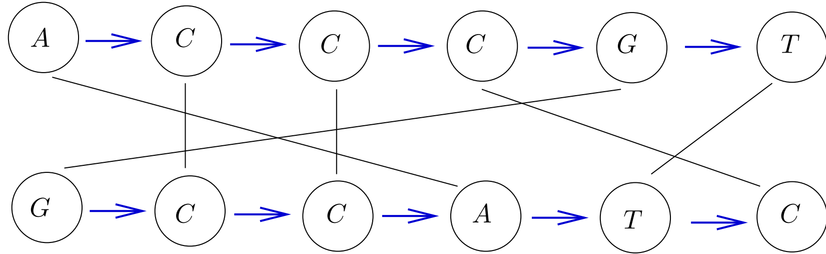


Figure 2.14: Non-breakable Arrangement.

Therefore, the arrangements corresponding to this example are $A_1 = 1, 2, 3, 4, 5$ and $A_2 = 2, 1, 5, 4, 3$. If A_1 is the reference arrangement, then A_2 can be broken into a prefix and a suffix of A_1, therefore, A_2 is breakable.

Another example is when $S_1 = ACCCGT$ and $S_2 = GCCATC$. An alignment graph for S_1 and S_2 is shown in Figure 2.14.

Therefore, the arrangements corresponding to this example are $A_1 = 1, 2, 3, 4, 5$ and $A_2 = 4, 2, 1, 5, 3$. Taking A_1 as a reference, A_2 can not be broken into a prefix and a suffix of A_1, and hence it is not breakable.

Let S be a set of n integers. An *arrangement* A of S is a sequence of integers of length n in which every element of S occurs exactly once. For example, if $S = \{1, 4, 5, 7, 12\}$, then $5, 12, 1, 7, 4$ is an arrangement of S but neither $4, 1, 3, 7, 5$ nor $12, 7, 4, 5, 4$ is. The *reverse* of A is denoted by A^R. If A is an arrangement, then $A(i)$ is the element in position i in the sequence. For example, if $A = 5, 12, 1, 7, 4$, then $A(4) = 7$. If $n \geq 1$ is an integer, then the *identity arrangement* for n is the sequence $I_n = 1, 2, \ldots n$. Let S be any set of n integers. Then sort(S), the *sorted arrangement* of S, is the unique arrangement of S in which the elements appear in increasing order. For example, sort$(\{4, 1, 5, 7, 12\}) = 1, 4, 5, 7, 12$ and sort$(\{1, 2, 3, 4, 5\}) = I_5$. If A is an arrangement of n elements, then the *size* of A is size$(A) = n$.

If $A = a_1, a_2, \ldots, a_n$ is an arrangement and $1 \leq i \leq j \leq n$, then $A[i, j] = a_i, a_{i+1}, \ldots, a_j$ is a *subarrangement*

of A; of course, $A[i, j]$ is an arrangement of the set $\{a_i, a_{i+1}, \ldots, a_j\}$. For example, given an arrangement $A = 5, 7, 6, 4, 9, 8$, then the following is a subarrangement: $A[3, 5] = 6, 4, 9$. An arrangement A is *consecutive* if it is a subarrangement of I_n, for some n. A set S is *consecutive* if sort(S) is consecutive. For example, the arrangement $3, 4, 5, 6, 7, 8$ is consecutive, while neither $4, 3, 5, 6$ nor $3, 4, 7, 8$ is. The set $\{2, 4, 3, 6, 5\}$ is consecutive, but the set $\{2, 4, 6, 5\}$ is not.

The class of *breakable arrangements* is defined recursively, as follows. The base cases are subarrangements of I_n for some n, which are all breakable arrangements. Now, assume that A_1 is a breakable arrangement of a consecutive set S_1 and that A_2 is a breakable arrangement of a consecutive set S_2 such that $S_1 \cap S_2 = \emptyset$ and $S_1 \cup S_2$ is consecutive. Then, both A_1A_2 and A_2A_1 are breakable arrangements. For example, if $A_1 = 6, 4, 5$ and $A_2 = 2, 3, 1$, then it is easy to show that A_1 and A_2 are breakable. Moreover, $\{6, 4, 5\}$, $\{2, 3, 1\}$, and $\{6, 4, 5, 2, 3, 1\}$ are all consecutive sets. Hence, $6, 4, 5, 2, 3, 1$ and $2, 3, 1, 6, 4, 5$ are breakable arrangements. However, $6, 4, 5$ cannot be combined with $2, 1$ to form a breakable arrangement, since $\{6, 4, 5, 2, 1\}$ is not a consecutive set.

The definition of a breakable arrangement implies that, for every breakable arrangement A, there exists a binary tree T with A at the root, breakable subarrangements of A at every node, and subarrangements of I_n at the leaves. Such a tree is called a *break tree* for the arrangement. For example, consider $A = 2, 1, 7, 5, 6, 8, 9, 10, 4, 3$. Starting from the left of A, the first place that we can break yields subarrangements $2, 1$ and $7, 5, 6, 8, 9, 10, 4, 3$. The subarrangement $2, 1$ breaks further into the two subarrangements 2 and 1 of an identity arrangement. The subarrangement $7, 5, 6, 8, 9, 10, 4, 3$ breaks into $7, 5, 6, 8, 9, 10$ and $4, 3$. Continuing recursively, we get the break tree in Figure 2.15.

An example of a non-breakable arrangement is $A = 2, 1, 7, 5, 3, 8, 10, 9, 4, 6$. The first division into consecutive subarrangements is $2, 1$ and $7, 5, 3, 8, 10, 9, 4, 6$. However, it is not possible to further break $7, 5, 3, 8, 10, 9, 4, 6$ into two consecutive subarrangements.

Theorem 1. *The two shortest non-breakable arrangements are* $i + 2, i, i + 3, i + 1$ *and* $i + 1, i + 3, i, i + 2$.

Proof. From the definition of breakable arrangements, we know that a breakable arrangement must be able to break into a prefix and a suffix of the identity arrangement I_n. All size four arrangements are breakable except for $i + 2, i, i + 3, i + 1$ and $i + 1, i + 3, i, i + 2$. For the two arrangements $i + 2, i, i + 3, i + 1$ and $i + 1, i + 3, i, i + 2$, all possible breaks are shown in Figures 2.16 and 2.17, respectively. From the figures, it is seen that none of the possible breaks result in blocks of I_n. Therefore, the two arrangements $i+2, i, i+3, i+1$ and $i + 1, i + 3, i, i + 2$ are non-breakable. We can easily check that any arrangements of sizes one, two, or three are breakable. Therefore, the two arrangements $i+2, i, i+3, i+1$ and $i+1, i+3, i, i+2$ are the shortest non-breakable arrangements. □

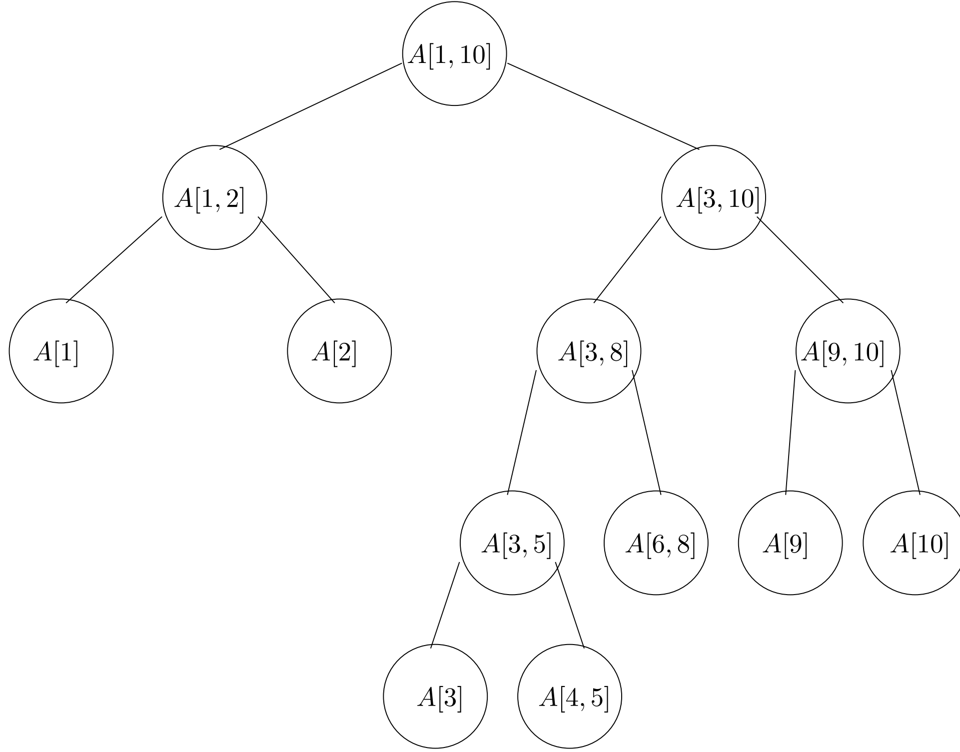


Figure 2.15: A Break Tree for $A = 2, 1, 7, 5, 6, 8, 9, 10, 4, 3$.

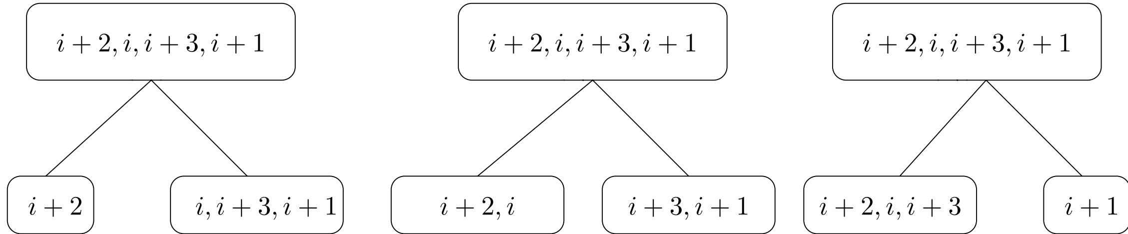


Figure 2.16: Possible breaks for $i + 2, i, i + 3, i + 1$.

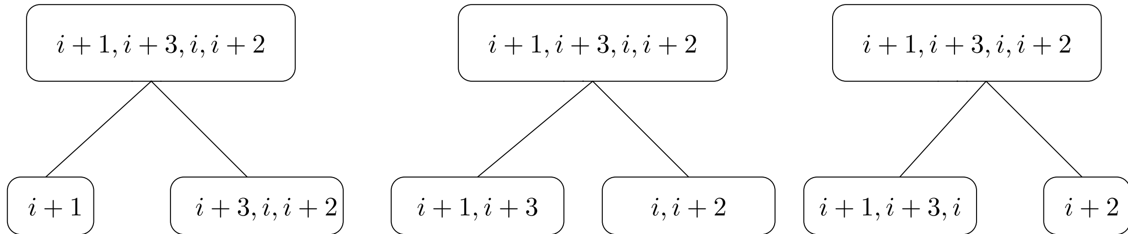


Figure 2.17: Possible breaks for $i + 1, i + 3, i, i + 2$.

The same result in theorem 1 is proven in [14] by Bose et. al.

Theorem 2. *If A is a breakable arrangement, then its reverse A^R is also breakable.*

Proof. Given a breakable arrangement A, we know that A can be recursively broken into a prefix and a suffix of the identity arrangement I corresponding to A. Therefore, it is clear that A^R can similarly be broken in the same way, however, the break will result into two parts, where the first part is a suffix of I and the second part is a prefix of I. Since the definition of a breakable arrangements includes both cases of having either a prefix first or a suffix, therefore, if A is breakable then A^R is also breakable by simply recursively reversing the break tree obtained for A. □

2.8.1 Identifying Breakable Arrangements

In this section, we address the identification problem for breakable arrangements:

IDENTIFY BREAKABLE ARRANGEMENT

INSTANCE: Arrangement A.

QUESTION: Is A breakable?

If A is breakable, we would like the algorithm to construct a break tree for A. An algorithm to solve the Identify Breakable Arrangement problem could build a break tree bottom-up or top-down. We choose to proceed top-down.

Using the recursive definition of a breakable arrangement, at any stage we have a consecutive arrangement A that we need to represent as A_1A_2, where A_1 and A_2 are also consecutive arrangements. For example, given the consecutive arrangement $A = 7, 3, 9, 5, 8, 4, 2, 1, 6$, the algorithm identifies two consecutive subarrangements $A[1, i]$ and $A[i + 1, n]$, where $n = \text{size}(A) = 9$ and i is as small as possible. If we consider $i = 1$, we have $A[1, 1] = 7$ and $A[2, 9] = 3, 9, 5, 8, 4, 2, 1, 6$; we see that $A[2, 9]$ is not a consecutive arrangement. If we consider $i = 2$, we have $A[1, 2] = 7, 3$ and $A[3, 9] = 9, 5, 8, 4, 2, 1, 6$; in this case, neither subarrangement is consecutive. As we consider all other values of i, we see that at least one of the corresponding two subarrangements is not consecutive. We conclude that A is not breakable.

Now consider an example of a breakable arrangement. Let $A = 7, 6, 5, 9, 8, 4, 3, 2, 1$. A can be divided into $A[1, 5]$ and $A[6, 9]$, which are consecutive subarrangements of A. Hence, a top-down algorithm can address each of $A[1, 5]$ and $A[6, 9]$ recursively. Continuing recursively, we ultimately end up with the break tree in Figure 2.18.

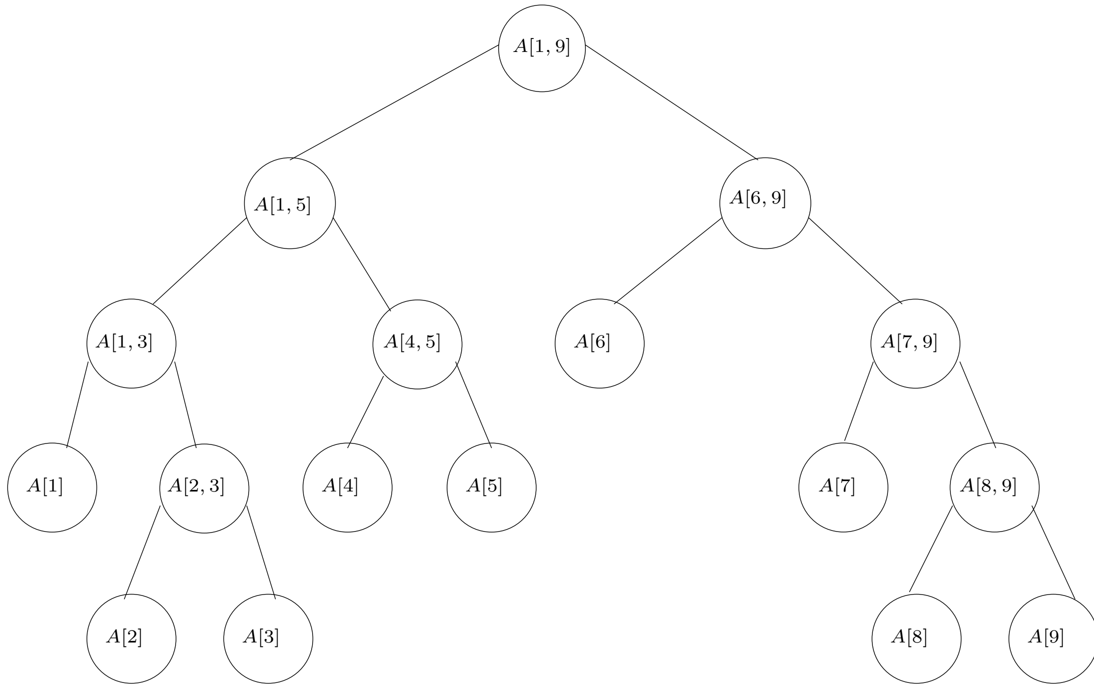


Figure 2.18: Break Tree for the arrangement $A = 7, 6, 5, 9, 8, 4, 3, 2, 1$.

Figure 2.19 shows the algorithm for solving Identify Breakable Arrangement. The algorithm uses the subroutine shown in Figure 2.20 to check if a subarrangement is consecutive.

Theorem 3. *The algorithm Breakable has time complexity $O(n^3)$.*

Proof. Each call to the recursive algorithm Breakable is performed in time Cn^2, for some constant $C > 0$. The loop on line 8 repeats n times, and in each iteration the subroutine Consecutive is called and is $O(n)$. Therefore, the time needed for the loop is Cn^2. The function is recursively called for subarrangements of the input A. The worst case for the sizes of the subarrangements $A[1, i]$ and $A[i+1, n]$ is when $A[1, i]$ has a size of 1. Therefore, the time complexity of the function can be expressed using the following recurrence:

$$
\begin{aligned}
T(1) &= 1 \\
T(2) &= 2 \\
T(n) &= T(1) + T(n-1) + Cn^2.
\end{aligned}
$$

From the above recurrence, it is seen that, in the worst case, the function is called n times. Hence, the recursive algorithm Breakable is of time complexity $O(n^3)$. □

```
1  Breakable(A, N)
2  if A is a subarrangement of an identity arrangement
3    then return TRUE
4  n ← size(A)
5  if Consecutive(A) = FALSE
6    then return FALSE
7  for i ← 1 to n − 1
8      do A_1 ← A[1, i]
9         A_2 ← A[i + 1, n]
10        if Consecutive(A_1) = TRUE and Consecutive(A_2) = TRUE
11          then Add child N_1 labeled A_1 with parent N
12               Add child N_2 labeled A_2 with parent N
13               return Breakable(A_1, N_1) and Breakable(A_2, N_2)
14 return FALSE
```

Figure 2.19: Identifying Breakable Arrangements.

```
1  Consecutive(A)
2  n ← size(()A)
3  min ← MIN(A)
4  max ← MAX(A)
5  if max-min+1 = n
6  THEN RETURN TRUE
7  ELSE RETURN FALSE
```

Figure 2.20: Subroutine to check if a subarrangement is consecutive.

Theorem 4. *The algorithm Breakable identifies all breakable arrangements and only breakable arrangements.*

Proof. If Breakable(A, r) returns TRUE, then it has built a break tree with root r that proves that A is breakable. Hence, Breakable(A, r) returns TRUE only for breakable arrangements.

To prove that algorithm Breakable identifies all breakable arrangements, we proceed by induction on the size of the input arrangement. The base case occurs for arrangements of size 1. Such arrangements are necessarily subarrangements of an identity arrangement and hence are breakable. In this case, Algorithm Breakable returns TRUE, as required.

The inductive hypothesis is that, for $n \geq 1$, all breakable arrangements of size $\leq n$ are identified correctly by algorithm Breakable.

Now, assume that A is a breakable arrangement of size $n+1$. Without loss of generality, assume that A is an arrangement of the set $1, 2, \ldots, n+1$. Let T be a break tree for A, and let $A[1, j]$ and $A[j+1, n+1]$ be the labels of the children of the root of T. Furthermore, let i be the value chosen by the algorithm Breakable so that $A[1, i]$ and $A[i+1, n+1]$ are selected as the labels of N_1 and N_2 in the first recursive call of Breakable. Clearly, $i \leq j$. If $i = j$, then $A[1, i]$ and $A[i+1, n+1]$ are both breakable and the inductive hypothesis implies that Breakable will correctly identify them and hence A is breakable. If $i < j$, then we proceed as follows.

Without loss of generality, assume that every element of $A[1, i]$ is less than every element of $A[i+1, n+1]$ (the remaining case that every element of $A[1, i]$ is greater than every element of $A[i+1, n+1]$ is symmetric). Then 1 is an entry of $A[1, i]$, and $n+1$ is an entry of $A[i+1, n+1]$. We claim that $A[1, i]$ is the label of some node in T. To see this, consider the node labels of the nodes on the leftmost path of T. Without loss of generality, we can write the labels in order as $A[1, s_1], A[1, s_2], \ldots, A[1, s_t]$, where $s_1 = n+1$, $s_2 = j$, and $A[1, s_t]$ is a subarrangement of I_{n+1}. Of necessity, $s_1 > s_2 > \cdots > s_t$. Hence, we can select a unique k such that $s_k > i$ and $s_{k+1} \leq i$. If $s_{k+1} = i$, then the claim that $A[1, i]$ is the label of some node in T is true. Otherwise, to obtain a contradiction, assume that $s_{k+1} < i$. We have that $A[1, s_{k+1}]$ and $A[s_{k+1}+1, s_k]$ are both breakable, since they are labels in T. Since every element of $A[s_{k+1}+1, i]$ is less than every element of $A[i+1, s_k]$, both $A[s_{k+1}+1, i]$ and $A[i+1, s_k]$ must be consecutive. $A[1, s_{k+1}]$ cannot contain the element 1, since then algorithm Breakable would have selected s_{k+1} instead of i. So $A[s_{k+1}, i]$ must contain the element 1. Since it is breakable, $A[s_{k+1}, i]$ must in fact contain the elements $1, 2, \ldots, s_k - s_{k+1}$. However, every element of $A[1, s_{k+1}]$ is smaller than every element of $S[i+1, s_k]$, which gives us the desired contradiction. We conclude that $A[1, i]$ is the label of some node in T and hence breakable. Moreover, since $A[1, i]$ is on the leftmost path of T, we have that $A[i+1, n+1]$ is breakable as well. By the inductive hypothesis, the recursive calls by Breakable correctly identify $A[1, i]$ and $A[i+1, n+1]$ as breakable. Hence, Breakable identifies A as

breakable. By induction, we conclude that algorithm Breakable identifies all breakable arrangements. □

2.8.2 Generating Breakable Arrangements

Another problem of interest is generating all breakable arrangements. This problem takes the identity arrangement as input to generate a list of breakable arrangements, and it is defined as follows.

Generate Breakable Arrangements

INSTANCE: Identity arrangement, I_n.

SOLUTION: A list L of all breakable arrangements of I_n.

A trivial way of generating all breakable arrangements is to generate all possible arrangements and use the algorithm shown in Figure 2.19 to identify which arrangements are breakable. The algorithm for generating all breakable arrangements uses Trotter's algorithm [83] to generate all possible arrangements. Trotter's algorithm generates arrangements by transposing, and it depends on the idea that only one swap between one pair of neighbors is needed to generate the next arrangement. This is done by indicating the direction of the swap of each integer, and an integer that is to be swapped is said to be mobile. A mobile integer is one with its direction of mobility pointing towards a lower value neighbor. The direction of mobility, or of the swap, could be either left or right, but the default direction is left. For example, take $n = 3$, the arrangements will be generated in the order shown in Table 2.1.

1	2	3
1	3	2
3	1	2
3	2	1
2	3	1
2	1	3

Table 2.1: Arrangements generated by Trotter's algorithm.

Each generated arrangement is then passed to the identification algorithm to be tested for breakability. Figure 2.21 shows the generation algorithm.

Theorem 5. *The algorithm GenerateBreakArrange has time complexity* $O((n!)n^3)$.

```
1  GenerateBreakArrange(I)
2  n ← size(I)
3  c ← 0
4  initialize list L to store breakable arrangements
5  for i ← 1 to n!
6      do I' ← Generate the next arrangement using Trotter's algorithm
7          breakable ← Breakable(I')
8          if breakable = 1
9             then L[c] ← I'
10            c ← c + 1
11 return L
```

Figure 2.21: Generating Breakable Arrangements.

Proof. The loop in the algorithm repeats $n!$ times, and each call to Trotter's algorithm is linear in n. The Breakable algorithm has a complexity of $O(n^3)$. Therefore, the GenerateBreakArrange algorithm has a total complexity of $O((n!)n^3)$. □

Theorem 6. *The algorithm GenerateBreakArrange generates all breakable arrangements and only breakable arrangements.*

Proof. The algorithm GenerateBreakArrange tests the breakability of each and every possible arrangement, using the identification algorithm. Therefore, it is guaranteed that all possible breakable arrangements are detected by Theorem 4. □

2.8.3 Counting Breakable Arrangements

For an integer $n \geq 1$, let $C(n)$ be the number of breakable arrangements on the set $\{1, 2, \ldots, n\}$. The counting problem is then the following:

COUNT BREAKABLE ARRANGEMENTS

INSTANCE: A positive integer n.

SOLUTION: $C(n)$.

To compute $C(n)$, we proceed as follows. The base cases are $C(1) = 1$ and $C(2) = 2$. Let $n \geq 3$. Let A be a breakable arrangement of $\{1, 2, \ldots, n\}$. By the definition of breakable arrangements, there exists an i such that $1 \leq i \leq n-1$ and such that there is a breakable arrangement A_1 of $\{1, 2, \ldots, i\}$ and a breakable arrangement A_2 of $\{i+1, i+2, \ldots, n\}$ satisfying $A = A_1A_2$ or $A = A_2A_1$. For a fixed i, there are $2C(i)C(n-i)$ ways to create a breakable arrangement of $\{1, 2, \ldots, n\}$. Hence,

$$C(n) \quad \leq \quad \sum_{i=1}^{n-1} 2C(i)C(n-i).$$

It is not equal because, for $i \geq 2$, some of the $2C(i)C(n-i)$ arrangements for i may be counted in arrangements for some i', where $i' < i$. For example, the breakable arrangement $2, 1, 3, 4$ will be counted both for $i = 2$ and for $i = 3$.

We claim that exactly half of the $2C(i)C(n-i)$ arrangements for i, where $i \geq 2$, are counted earlier. Fix $i \geq 2$. Let A_1 be a breakable arrangement of $\{1, 2, \ldots, i\}$, and let A_2 be a breakable arrangement of $\{i+1, i+2, \ldots, n\}$. Let A_1^R be the reverse of A_1; by Theorem 2, A_1^R is also breakable. We claim that precisely one of A_1A_2 and $A_1^RA_2$ is counted by an i' with $i' < i$. (The argument for A_2A_1 and $A_2A_1^R$ is similar.) Without loss of generality, we may assume that i appears before 1 in A_1. Let $A_1 = a_1, a_2, \ldots, a_i$, let $a_j = i$, and let $a_k = 1$. Then $1 \leq j < k \leq i$. Since A_1 is breakable, there exists m, where $j \leq m < k$, such that $A_1[m+1, i]$ is a breakable arrangement of $\{1, 2, \ldots, i-m\}$, and $A_1[1, m]$ is a breakable arrangement of $\{i-m+1, i-m+2, \ldots, i\}$. Then, $A_1^R[1, i-m]$ is a breakable arrangement of $\{1, 2, \ldots, i-m\}$, and $A_1^R[i-m+1, i]$ is a breakable arrangement of $\{i-m+1, i-m+2, \ldots, i\}$. Because i appears before 1 in A_1, A_1A_2 was not counted earlier. Because $A_1^R[1, i-m]$, $A_1^R[i-m+1, i]$, and A_2 are breakable, we have that $A_1^R[i-m+1, i]A_2$ is breakable and that $A_1^RA_2$ is counted earlier. Hence, exactly half of the $2C(i)C(n-i)$ arrangements for i, where $i \geq 2$, are counted earlier.

We arrive at the following recurrence:

$$\begin{aligned} C(1) &= 1 \\ C(2) &= 2 \\ C(n) &= 2C(1)C(n-1) + \sum_{i=2}^{n-1} C(i)C(n-i), \end{aligned}$$

where $n \geq 3$.

Figure 2.22 shows the algorithm for counting breakable arrangements.

Theorem 7. *The algorithm CountBreakArrange has time complexity $O(n^2)$.*

Proof. There are two nested loops on lines 5 through 7, the loop on line 5 repeats n times, and the loop on line 7 repeats n times, in the worst case. The statement inside the nested loops, on line 8, is of constant

```
1  CountBreakArrange(n)
2  Count[1] ← 1
3  Count[2] ← 2
4  for i ← 3 to n
5      do Count[i] ← 2Count[1]Count[i − 1]
6          for j ← 2 to i − 1
7              do Count[i] ← Count[i] + Count[j]Count[i − j]
```

Figure 2.22: Counting Breakable Arrangements.

time, as it performs access on calculated values. Therefore, the overall complexity of the algorithm is $O(n^2)$. □

The sequence of counts turns out to be the same sequence as the Schröder numbers [74], which can be expressed by the following recurrence [89]:

$$\begin{aligned} S_0 &= 1 \\ S_1 &= 2 \\ S_n &= S_{n-1} + \sum_{i=0}^{n-1} S_i S_{n-1-i}, \end{aligned}$$

where $n \geq 2$. The equivalence of the sequences is embodied in the following theorem.

Theorem 8. *For $n \geq 0$, $C(n+1) = S_n$.*

Proof. The theorem is immediately true for $n = 0$ and $n = 1$. We proceed by induction on n, $n \geq 2$, assuming that the theorem is true for numbers smaller than n and showing that $C(n+1) = S_n$. Fix $n \geq 2$.

Using the inductive hypothesis, we have

$$\begin{aligned}
C(n+1) &= 2C(1)C(n) + \sum_{i=2}^{n} C(i)C(n+1-i) \\
&= 2S_0S_{n-1} + \sum_{i=2}^{n} S_{i-1}S_{n-i} \\
&= 2S_0S_{n-1} + \sum_{i=1}^{n-1} S_iS_{n-1-i} \\
&= S_0S_{n-1} + \sum_{i=0}^{n-1} S_iS_{n-1-i} \\
&= S_{n-1} + \sum_{i=0}^{n-1} S_iS_{n-1-i} \\
&= S_n.
\end{aligned}$$

By induction, the theorem holds for all n. □

2.9 NP-Completeness Proof

The following 3-Partition problem is used in the proof that Optimal Whole Genome Alignment is NP-complete [29].

3-Partition

INSTANCE: Set $\mathbb{A}$ of $3m$ elements, a bound $B > 0$, and a positive size $s(a)$ for each $a \in \mathbb{A}$ such that $B/4 < s(a) < B/2$ and such that $\sum_{a \in \mathbb{A}} s(a) = mB$.

QUESTION: Can $\mathbb{A}$ be partitioned into m disjoint sets $\mathbb{A}_1, \mathbb{A}_2, \ldots, \mathbb{A}_m$, each containing 3 elements, such that, for $1 \leq i \leq m$, we have $\sum_{a \in \mathbb{A}_i} = B$?

The 3-Partition problem is NP-complete in the strong sense, which means that it remains NP-complete if we express the numbers in unary.

Theorem 9. *Optimal Whole Genome Alignment is NP-complete.*

Proof. We first need to show that the Optimal Whole Genome Alignment problem is in NP. Given a whole genome alignment, it is straightforward to compute, in polynomial time, the score of the alignment and check whether that score is less than the given bound. Hence, Optimal Whole Genome Alignment is in NP.

To complete the proof, we reduce 3-Partition to Optimal Whole Genome Alignment in polynomial time. Let $\mathbb{A} = \{a_1, a_2, \ldots, a_{3m}\}$, a set of $3m$ elements, sizes $s(a_i)$ for each $a_i \in \mathbb{A}$, and bound B constitute an instance

of 3-Partition. We assume that all the numbers are represented in unary. The corresponding instance of Optimal Whole Genome Alignment consists of two sequences over the DNA alphabet:

$$U = A^{s(a_1)}TA^{s(a_2)}T \cdots A^{s(a_{3m})}G^{m-1}$$
$$V = (A^B G)^{m-1}A^B T^{3m-1}.$$

Both strings have length $mB + 4m - 2$ and the same counts of A's, G's, and T's. Set the weights for scoring an alignment to $w_b = 1$ and $w_m = w_f = w_d = 4$. The bound for the score is $S = 7m - 3$. It is clear that U, V, and S can be constructed from the instance of 3-Partition in polynomial time, since the sizes are represented in unary.

If only black edges are allowed, then there are $7m - 3$ 'natural' breaks in U, consisting of one before and after each of $3m - 1$ T's and one before each of $m - 1$ G's. If there is a 3-partition of $\mathbb{A}$, then it is straightforward to give a global alignment of U and V of score S: use the partition of $\mathbb{A}$ to match three blocks of A's in U to each A^B block in V, then match each G in U to an arbitrary G in V and each T in U to an arbitrary T in V.

Now assume that there is a global alignment of U and V with score $\leq S$. It is clear that, unless there are red edges, then the global alignment has at least S breaks (all of the natural ones in U). Since we assume that there are no red edges, there can be no mutations, free nodes, or duplicate nodes. Hence, each block of A^B in V specifies three elements of $\mathbb{A}$, and we get the desired 3-partition of $\mathbb{A}$.

If only red edges are allowed, we have the same number of natural breaks, using the same construction above, since there is one break before and after each delimiter, either a T or a G, then the score will still be $7m - 3$. When both black and red edges are allowed, then some of the T's will connect to A's using red edges. Therefore, there will be the same two breaks at each delimiter T connected to an A. In addition to that, there will be a break at each of the A's that will be connected to one of the T's at the end of the second sequence. Therefore, if the number of T's connected to A's by red edges is k, then the number of breaks will be $7m - 3 + 2k$. This shows that allowing red edges will never result in a score less than $7m - 3$.

We have demonstrated that 3-Partition reduces to Optimal Whole Genome Alignment in polynomial time. We conclude that Optimal Whole Genome Alignment is NP-complete. □

2.10 Dynamic Programming for Whole Genome Alignment

A dynamic programming heuristic algorithm can be used to approximately solve whole genome alignment. The algorithm presented optimally solves the following problem:

Whole Genome Alignment for Breakable Arrangements

INSTANCE: Two breakable DNA sequences S_1 and S_2 and weights w_b, w_m, w_f, and w_d.

SOLUTION: An alignment graph G for S_1 and S_2 that minimizes the alignment score, $s(G)$.

The algorithm takes the two sequences S_1 and S_2 as input, of lengths m and n, respectively. The algorithm starts processing all the pairs of subsequences, starting from pairs of length 0 and increasing the lengths until m and n. Finding the alignment score between pairs of lengths 0 or 1 is trivial, and the score can be calculated using one of the following options:

- $S_1[0,0]$ and $S_2[0,0]$ are identical blank subsequences, and their score is 0.
- $S_1[0,0]$ and $S_2[j,j]$ give a free node with a score of w_f.
- $S_1[i,i]$ and $S_2[0,0]$ give a free node with a score of w_f.
- $S_1[i,i]$ and $S_2[j,j]$ are either identical with a score of 0, or different and considered a mutation with a score of w_m.

The alignment scores of the pairs of substrings are stored in an array, AlignScore, where each cell in the array is indexed by four indices, i, k, j, and l, where i indicates the S_1 subsequence starting position, k indicates the S_1 subsequence ending position, j indicates the S_2 subsequence starting position, and l indicates the S_2 subsequence ending position. Processing pairs of subsequences continues to longer subsequences, where the score is calculated using the previously calculated scores of shorter subsequences. This is done by trying each and every possible break, and aligning all possible parts together, to be able to find the minimum score. For example, given $S_1 = AC$ and $S_2 = CA$, a possible break is at $(1,1)$, this gives four parts of the sequences, namely, $S_1[1,1] = A$, $S_1[2,2] = C$, $S_2[1,1] = C$, and $S_2[2,2] = A$. Any of the parts can be aligned to one another, and since the scores of aligning those parts are already calculated, since the parts are all of shorter lengths, then it is easy to compare the scores resulting from different combinations. The optimum score is obtained by aligning $S_1[1,1]$ to $S_2[2,2]$, and aligning $S_1[2,2]$ to $S_2[1,1]$, this gives a score of 0, since all values are identical, plus a break penalty w_b. For longer subsequences, more breaks are considered, and all are processed to find the minimum score. This minimum score is only recorded if it is less than aligning the two subsequences without any breaks, otherwise, a score without breaks is recorded in the AlignScore array.

Therefore the base cases are represented as follows:

$$
\begin{aligned}
\text{AlignScore}[0,0,0,0] &= 0 \\
\text{AlignScore}[0,0,j,j] &= w_f \\
\text{AlignScore}[i,i,0,0] &= w_f \\
\text{AlignScore}[i,i,j,j] &= \begin{cases} 0 & \text{if } S_1[i,i] = S_2[j,j], \\ w_m & \text{otherwise.} \end{cases}
\end{aligned}
$$

The function RegAlign used in the following recursive relation is the classic global alignment algorithm for the case when there are only mutations, insertions, and deletions. The algorithm for RegAlign is illustrated in Figure 2.23, and the scoring function used by RegAlign is shown in Table 2.2. The algorithm for whole genome alignment using a dynamic programming approach is illustrated in Figure 2.25. The algorithm uses a preprocessing step, shown in Figure 2.24, to compute alignments without breaks between all pairs of subsequences. The scores for those alignments are stored in the matrix PrecompRegAlign.

And, the general case:

$$
\text{AlignScore}[i,k,j,l] = \min_{i \le x \le k, j \le y \le l} \begin{cases} \text{RegAlign}(S_1[i,k], S_2[j,l]), \\ \text{AlignScore}[i,x,j,y] + \text{AlignScore}[x+1,k,y+1,l] + w_b, \\ \text{AlignScore}[x+1,k,j,y] + \text{AlignScore}[i,x,y+1,l] + w_b \end{cases}
$$

AlignScore$[i,k,j,l]$ fetches a value from the AlignScore array.

$s(x,y)$	A	C	G	T	-
A	0	4	4	4	4
C	4	0	4	4	4
G	4	4	0	4	4
T	4	4	4	0	4
-	4	4	4	4	0

Table 2.2: Scoring function used by RegAlign

Example 1. This example shows the steps of the dynamic programming algorithm, Table 2.3 shows the alignment scores of pairs of subsequences. Let $w_b = 1$, hence $w_m = w_f = w_d = 4 \times w_b = 4$, and let the empty sequence be λ.

```
1  RegAlign(X[1,m], Y[1,n])
2  for i ← 0 to m
3       do OPT(i,0) ← ∑_{k=1}^{i} s(x_k, −)
4  for j ← 0 to n
5       do OPT(0,j) ← ∑_{k=1}^{j} s(−, y_k)
6  for i ← 0 to m
7       do for j ← 0 to n
8              do OPT[i,j] ← min{ s(x_i, −) + OPT[i−1, j],
9                                 s(−, y_j) + OPT[i, j−1],
10                                s(x_i, y_j) + OPT[i−1, j−1]}
11 return OPT(m,n)
```

Figure 2.23: Algorithm for RegAlign.

```
1  PreProcess(S_1[1,m], S_2[1,n])
2  for i ← 1 to m
3       do for k ← i to m
4           do for j ← 1 to n
5               do for l ← j to n
6                   for PrecompRegAlign[i,k,j,l] ← RegAlign(S_1[i,k], S_2[j,l])
```

Figure 2.24: Algorithm for precomputing alignments of subsequences without breaks.

```
1  GenomeAlign(S_1[1,m], S_2[1,n])
2  for i ← 0 to m
3      do for j ← 0 to n
4          do if S_1[i,i] = S_2[j,j]
5              then  AlignScore[i,i,j,j] ← 0
6              else if i = 0 or j = 0,  and i ≠ j
7                      then AlignScore[i,i,j,j] ← w_f
8                      else if S_1[i,i] ≠ S_2[j,j],  and i ≠ 0 and j ≠ 0
9                              then AlignScore[i,i,j,j] ← w_m
10 PreProcess(S_1, S_2)
11 for h ← 1 to m
12     do for v ← 1 to n
13         do for i ← 1 to m − h + 1
14             do for j ← 1 to n − v + 1
15                 do k ← i + h − 1
16                    l ← j + v − 1
17                    for x ← i to k
18                    do for y ← j to l
19                        do AlignScore[i,k,j,l] ← min{PrecompRegAlign[i,k,j,l],
20                            AlignScore[i,x,j,y] + AlignScore[x+1,k,y+1,l] + wb,
21                            AlignScore[x+1,k,j,y] + AlignScore[i,x,y+1,l] + wb}
22 return AlignScore[1,m,1,n]
```

Figure 2.25: Algorithm for the Whole Genome Alignment.

S_1	S_2	Score
A	λ	4
A	C	4
A	A	0
A	CA	4
C	λ	4
C	C	0
C	A	4
C	CA	4
AC	λ	8
AC	C	4
AC	A	4
AC	CA	1
λ	C	4
λ	A	4
λ	CA	8

Table 2.3: Dynamic Programming: Example 1.

$S_1 = AC$

$S_2 = CA$

Theorem 10. *Algorithm GenomeAlign has time complexity $O(m^3n^3)$.*

Proof. By analyzing the pseudocode in Figure 2.25, the call for the PreProcess subroutine in $O(m^3n^3)$. There are six nested loops, three of which repeat m times, and the other three repeat n times. The statements inside the loops are all $O(1)$, as they just access arrays of precomputed values. Therefore, the overall complexity of GenomeAlign is $O(m^3n^3)$. □

Theorem 11. *The space complexity of the algorithm GenomeAlign is $O(m^2n^2)$.*

Proof. By analyzing the algorithm in Figure 2.25, the algorithm needs to store the alignment scores of all pairs of subsequences, therefore, the space complexity of GenomeAlign is $O(m^2n^2)$, where m and n are the lengths of the two input sequences. □

Theorem 12. *The algorithm GenomeAlign yields optimal results for breakable arrangements.*

Proof. (by contradiction)

Suppose there are two sequences S_1 and S_2. S_1 can be broken into blocks such that each block is numbered to obtain the identity arrangement $R_1 = 123...n$. Suppose S_2 is breakable, that is, S_2 can be broken into blocks that can be numbered then mapped to the blocks of S_1. The blocks of S_2 are represented by the breakable arrangement R_2. Assume S_1 and S_2 can not be optimally aligned by GenomeAlign. From the definition of breakable arrangements, it is known that R_2 can be divided into subsets that are either of size one, or are blocks of the identity arrangement. If GenomeAlign can not optimally align S_1 and S_2, then GenomeAlign can not optimally align sequences of length one, or identical sequences (blocks of the identity arrangement), however, those are trivial cases solved by GenomeAlign, and hence this gives a contradiction. Therefore, the theorem holds. □

2.11 Arrangements and Other Alignments

In this section we study some alignment examples and see how they map to the arrangements we define. For each alignment, the blocks are identified and numbered to obtain arrangements. One of the arrangements is taken as a reference, and the breakability of the other arrangement is tested accordingly.

The first alignment we consider is shown in Figure 6 in [22]. This alignment is obtained for nine genomes listed in Table 1 in [22], and those are *E. coli K12 MG1655*, *E. coli O157:H7 EDL933*, *E. coli O157:H7 VT-2 Sakai*, *E. coli CFT073*, *S. flexneri 2A 2457T*, *S. flexneri 2A*, *S. enterica Typhimurium LT2*, *S. enterica Typhi CT18*, and *S. enterica Typhi Ty2*. The alignment was done using the MAUVE alignment tool [22]. The alignment shown in Figure 6 [22] is color coded, where the corresponding blocks in different genomes have the same color, and the numbers for the blocks were obtained accordingly. The topmost row in the figure is taken as a reference, where 25 blocks were identified, giving the reference identity arrangement $I = 1, 2, 3, \ldots, 25$. The other eight arrangements were obtained in a similar manner, by mapping the color-codes to the first genome. And, all arrangements were found to be breakable. The arrangements obtained are as follows:

$$
\begin{aligned}
A_1 &= 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17, \\
&\quad 18,19,20,21,22,23,24,25 \\
A_2 &= 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17, \\
&\quad 18,19,20,21,22,23,24,25 \\
A_3 &= 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17, \\
&\quad 18,19,20,21,22,23,24,25 \\
A_4 &= 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17, \\
&\quad 18,19,20,21,22,23,24,25 \\
A_5 &= 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,21,22, \\
&\quad 20,19,18,17,16,24,23,25 \\
A_6 &= 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17, \\
&\quad 18,19,20,22,21,24,23,25 \\
A_7 &= 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17, \\
&\quad 18,19,20,21,22,23,24,25 \\
A_8 &= 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,19, \\
&\quad 20,18,17,21,22,23,24,25 \\
A_9 &= 1,15,14,13,12,11,10,9,8,7,6,5,4,3,2,16,19, \\
&\quad 20,18,17,21,22,23,24,25
\end{aligned}
$$

All these arrangements are breakable.

MUMmer [23] is used to align *M. genitalium* and *M. pneumoniae*. Figure 7(bottom) [23] gives the identity arrangement $I_7 = 1,2,3,4,5,6,7$ for the sequence represented on the x-axis, and $A = 7,2,1,3,4,5,6$ for the sequence represented on the y-axis. A is a breakable arrangement.

In [35], the chaos game representation (CGR) of sequences is used to obtain regions of similarity between two input sequences. The CGR representation of a sequence is a dot-matrix plot, where the points on the plot are mathematically calculated. Figure 3 [35] yields two identical arrangements with duplication for the genomes HIV type 1 and CIV, so that is clearly breakable. Also, for Figure 4 [35], we obtain the identity arrangement $I_8 = 1,2,3,4,5,6,7,8$ for the sequence on the x-axis, *Pyrococcus abyssi GE5*, and for that on the y-axis, *Pyrococcus horikoshii*, we get the arrangement $A = 1,7,5,6,4,3,2,8$, which is breakable.

In [40], MUMmer is used to align *A. fumigatus* and *A. nidulans*. The arrangements $I_{10} = 1,2,3,4,5,6,7,8,9,10$ and $A = 10,5,6,7,8,9,4,3,2,1$ are obtained for the x-axis and y-axis, respectively, in Figure 1 [40]. A is clearly breakable.

Other alignments clearly yield breakable arrangements. For example Figure 2(A) [11] shows an alignment between the chloroplast genomes of *Arabidopsis thaliana* and *Oenothera elata* using threaded blockset alignment (TBA). This alignment yields two identical arrangements, with duplication. Also, Figure 5 [46] is an alignment obtained for EBV and EHV2 using GenomeBlast [46], and it is clearly breakable.

The alignment tool CoCoNUT [2] is used to align orthologs of human chromosome X to mouse chromosome X. In Figure 5 [2], the blocks on the solid lines on the x-axis (human) are mapped to the identity arrangement $I_5 = 1,2,3,4,5$. However, the corresponding blocks on the y-axis (mouse) yield the arrangement $A = 3,5,2,4,1$, which is not breakable.

Therefore, it is seen that all but one of the real world examples analyzed are breakable arrangements that can be optimally solved by GenomeAlign.

2.12 A Greedy Algorithm for Whole Genome Alignment

A greedy approach can be used to obtain a heuristic of the whole alignment problem. This heuristic allows for reversals. This greedy approach takes as input two sequences. Let the two sequences be denoted by S_1 and S_2, and let the complement of S_2 be ${S_2}^c$. The algorithm in Figure 2.27 consists of three steps, the first is the preprocessing step, the second is the alignment step, and the third step is the scoring step.

The preprocessing step in Figure 2.28 finds the alignment score between each pair of subsequences of S_1 and each of S_2 and ${S_2}^c$, and the values are kept in a Preprocesstable, where there is an attribute in the table to indicate whether the S_2 is complemented or not. The alignment step in Figure 2.30 starts processing the two subsequences, this is done by greedily aligning pairs of subsequences according to their alignment score, computed in the preprocessing step. The pairs are processed in ascending order of their score. This step continues until the two sequences are finished. If the pair being processed already contains a subsequence that has been aligned in a previous step, then this indicates a duplication. The scoring step in Figure 2.31 then follows to compute the score of the alignment done in step two.

For example, let $S_1 = AC$, $S_2 = CA$, and ${S_2}^c = GT$. The preprocessing will produce Table 2.4.

The alignment step then sorts the scores and results in the alignment shown in Figure 2.26.

This alignment is then scored, in the final step.

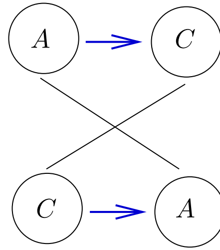


Figure 2.26: Alignment graph for AC and CA

1 GreedyAlign(S_1, S_2)

2 PreprocessPairs(S_1, S_2) ▷ Or alternatively, ReducedPreprocessPairs (S_1, S_2, k)

3 Align(S_1, S_2)

4 Score(G)

Figure 2.27: The greedy algorithm GreedyAlign.

1 PreprocessPairs(S_1, S_2)

2 **for** $i \leftarrow 1$ **to** m

3 **do for** $k \leftarrow i$ **to** m

4 **do for** $j \leftarrow 1$ **to** n

5 **do for** $l \leftarrow j$ **to** n

6 **do** PreprocessTable$[i, k, j, l] \leftarrow$ RegAlign$(S_1[i, k], S_2[j, l])$

PreprocessTable$[i, k, j, l].complement \leftarrow 0$

7 PreprocessTable$[i', k', j', l'] \leftarrow$ RegAlign$(S_1[i, k], S_2^c[j, l])$

8 PreprocessTable$[i', k', j', l'].complement \leftarrow 1$

Figure 2.28: The Preprocessing Step.

```
1   ReducedPreprocessPairs(S_1, S_2, k)
2   for i ← 1 to (m/k) + 1
3   do for j ← 1 to (n/k) + 1
4       do PreprocessTable[i, i + k, j, j + k] ← RegAlign(S_1[i, i + k], S_2[j, j + k])
5          PreprocessTable[i, i + k, j, j + k].cmpl ← 0
6          PreprocessTable[i', i' + k, j', j' + k] ← RegAlign(S_1[i, i + k], S_2^c[j, j + k])
7          PreprocessTable[i', i' + k, j', j' + k].cmpl ← 1
```

Figure 2.29: The preprocessing step with specified subsequence size.

```
1   Align(S_1, S_2)
2   Sort(PreprocessTable)
3   count ← 0
4   while (S_1 ≠ λ or S_2 ≠ λ or S_2^c ≠ λ)
5         do ▷ Preprocess contains the indices of the subsequences to be aligned after the sorting
6            (i, k, j, l) ← Preprocess[count]
7            if PreprocessTable[i, j, k, l].complement = 0
                then if S_1[i, k] and S_2[j, l] are not aligned
8                       then S_1 ← S_1 − S_1[i, k]
9                            S_2 ← S_2 − S_2[j, l]
10                           Align S_1[i, k] and S_2[j, l] according to PreprocessTable[i, j, k, l]
11                           count ← count + 1
12                    else if S_1[i, k] and S_2[j, l] are not aligned
13                            then S_1 ← S_1 − S_1[i, k]
14                            S_2^c ← S_2^c-S_2^c[j, l]
15                            Align S_1[i, k] and S_2^c[j, l] according to Preprocess[i, j, k, l]
16                            count ← count + 1
```

Figure 2.30: The alignment step.

1 Score($G(V, U, E_{\text{blue}}, E_{\text{black}}, E_{\text{red}})$)
2 $m \leftarrow 0$
3 $b \leftarrow 0$
4 $f \leftarrow 0$
5 currentV $\leftarrow 0$
6 currentU $\leftarrow 0$
7 **for** $i = 1$ to $|V|$
8   **do if** v_i is not connected to currentU $+ 1$
9   **then** $b \leftarrow b + 1$
10   **if** v_i OR u_{currentU} are not aligned
11   **then** $f \leftarrow f + 1$
12   **if** $e_{v_i u_{\text{currentU}}} \in E_{\text{red}}$ **and**
            $\text{Label}(v_i) \neq \text{Label}^c(u_{\text{currentU}})$
            or $e_{v_i u_{\text{currentU}}} \in E_{\text{black}}$ **and**
            $\text{Label}(v_i) \neq \text{Label}(u_{\text{currentU}})$
13   **then** $m \leftarrow m + 1$
14   currentU $\leftarrow$ currentU $+ 1$
15   currentV $\leftarrow$ currentV $+ 1$
16 score $\leftarrow mw_m + fw_f + bw_b$
17 **return** score

Figure 2.31: The scoring step.

Theorem 13. *Let S_1 be a sequence of length m, and let S_2 be a sequence of length n. Then GreedyAlign produces a global alignment of S_1 and S_2 in $O(m^3n^3)$ time.*

Proof. The preprocessing step contains four nested loops, two of which repeat m times, and the other two repeat n times, this gives a complexity of $O(m^2n^2)$. The step done inside the inner loop, which is the RegAlign has time complexity $O(mn)$. Therefore, the preprocessing step has time complexity $O(m^3n^3)$. The alignment step firsts sorts the Preprocesstable, this takes $O(m^2n^2 \log m^2n^2)$. Then the alignment step iterates on S_1 and S_2. The number of iterations is either m or n. Therefore, this loop is $O(m)$, if $m > n$, or $O(n)$, otherwise. Therefore, the complexity of the alignment step is $O(m^2n^2 \log m^2n^2)$. The final scoring step loops on the nodes of the alignment graph and scores it. This is done in $O(mn)$. Therefore, the complexity of this greedy approach is $O(m^3n^3)$. □

Theorem 14. *Algorithm GreedyAlign has space complexity $O(m^2n^2)$.*

Proof. By analyzing the algorithm in Figure 2.27, the preprocessing step needs to store the scores for aligning all pairs of subsequences, therefore, the space complexity of GenomeAlign is $O(m^2n^2)$, where m and n are the lengths of the two input sequences.

□

Theorem 15. *Let S_1 be a sequence of length m, let S_2 be a sequence of length n, and let k be the length of subsequences. Then GreedyAlign using ReducedPreprocessPairs produces a whole genome alignment of S_1 and S_2 in $\max\{O(m^2n^2/k^2), O((m/k)^2(n/k)^2 \log(m/k)^2(n/k)^2)\}$ time.*

Proof. Without loss of generality, assume that $m \geq n$. The preprocessing step in Figure 2.29 is $O(m^2n^2/k^2)$. The alignment step first sorts the Preprocesstable, this takes $O((m/k)^2(n/k)^2 \log(m/k)^2(n/k)^2)$. Then the alignment step iterates on S_1 and S_2. The number of iterations is either m or n. Therefore, this loop is $O(m)$. Therefore, the complexity of the alignment step is $O((m/k)^2(n/k)^2 \log(m/k)^2(n/k)^2)$. The final scoring step loops on the nodes of the alignment graph and scores it. This is done in $O(mn)$. Therefore, the complexity of this greedy approach is $\max\{O(m^2n^2/k^2), O((m/k)^2(n/k)^2 \log(m/k)^2(n/k)^2)\}$. □

Example 1. This example shows the steps of the greedy algorithm in detecting reversals. Table 2.5 shows the alignment scores of pairs of subsequences. Let $w_b = 1$, hence $w_m = w_f = w_d = 4 \times w_b = 4$. Starting by the lowest scores, of value 0, it is seen that the lower scores come from aligning S_1 with ${S_2}^c$. $S_1[1,1]$ aligns with ${S_2}^c[2,2]$ using a red edge. Similarly, $S_1[2,2]$ aligns with ${S_2}^c[1,1]$ using a red edge. This gives a score of $w_b = 1$.

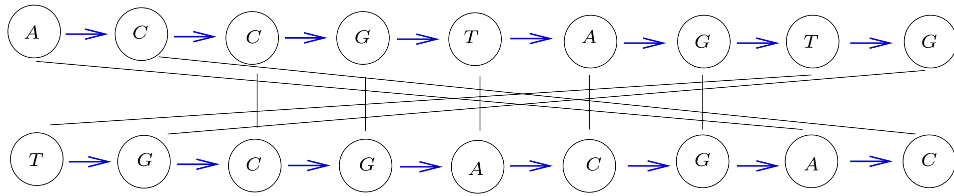


Figure 2.32: Alignment graph for S_1 and S_2 with two breaks and two mutations.

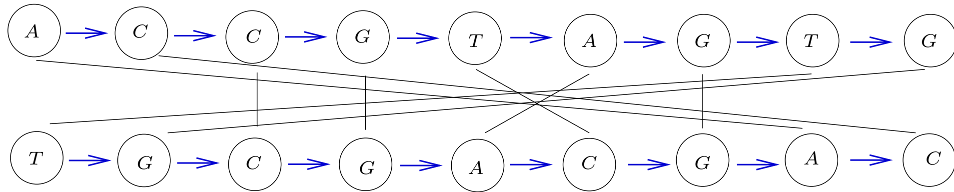


Figure 2.33: Alignment graph for S_1 and S_2 with five breaks and one mutation.

$S_1 = AC$

$S_2 = GT$

${S_2}^c = CA$

The presented greedy algorithm fails to find the optimal alignment for sequences that have long common subsequences that contain a few mutations. The greedy algorithm will choose to align the mutated nucleotides with other matching nucleotides, thus, creating more breaks rather than creating mutations. For example, let $S_1 = ACCGTAGTG$ and $S_2 = TGCGACGAC$, three possible alignments are shown in Figures 2.32, 2.33, and 2.34.

The greedy algorithm will fail to obtain the alignment in Figure 2.32, which has the minimum number of breaks with one extra mutation.

A number of modifications could be made to the greedy algorithm to obtain better results. However, these

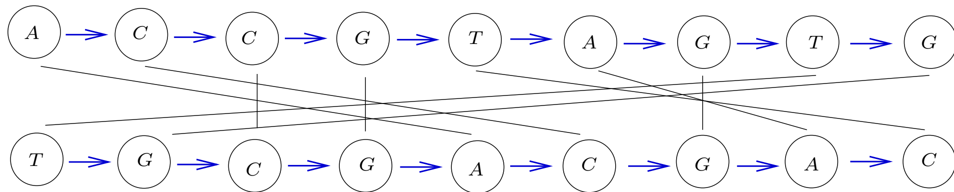


Figure 2.34: Alignment graph for S_1 and S_2 with five breaks and one mutation.

modifications will yield a more complicated algorithm. First, for pairs of sequences with the same score calculated in the preprocessing step, longer sequences should be considered first. And, the other point is that the relative location of aligned pairs of subsequences should be taken into consideration. In other words, if a subsequence is aligned to multiple other subsequences with the same score, then the subsequence that will give better overall alignment should be chosen. For example, if one of the subsequences in S_1 is present in the same corresponding position in S_2, then this pair is better aligned than others.

S_1	S_2or$S_2{}^c$	Complement(1/0)	Score
A	λ	0	4
A	C	0	4
A	A	0	0
A	CA	0	4
C	λ	0	4
C	C	0	0
C	A	0	4
C	CA	0	4
AC	λ	0	8
AC	C	0	4
AC	A	0	4
AC	CA	0	8
λ	C	0	4
λ	A	0	4
λ	CA	0	8
A	G	1	4
A	T	1	4
A	GT	1	8
C	G	1	4
C	T	1	4
C	GT	1	8
AC	G	1	8
AC	T	1	8
AC	GT	1	8
λ	G	1	4
λ	T	1	4
λ	GT	1	8

Table 2.4: Output of the preprocessing step for AC and $S_2 = CA$.

S_1	S_2	Score	${S_2}^c$	Score
A	λ	4		
A	G	4	C	4
A	T	4	A	0
A	GT	8	CA	4
C	λ	4		
C	G	4	C	0
C	T	4	A	4
C	GT	8	CA	4
AC	λ	8		
AC	G	8	C	4
AC	T	8	A	4
AC	GT	8	CA	8
λ	G	4		
λ	T	4		
λ	GT	8		

Table 2.5: Greedy algorithm: Example 1.

Chapter 3

Inferring HGT from Posets

This chapter starts by an introduction to the problem of whole genome alignment, then we give the definitions and notations needed in the rest of the chapter. Section 3.3 summarizes the related literature. The rest of the chapter is dedicated to problem definitions, algorithms, and theorems, where we define a computational problem for inferring horizontal gene transfer (HGT) from partial order sets (posets). The posets do not depend on absolute bit scores but rather on their relative values. In what follows, we show how posets can be obtained from distance matrices, however, it is not necessary to have absolute values either, if the relative distances are the only information we have, posets can still be used to infer horizontal gene transfers. and following that we present an algorithm to solve this problem.

3.1 Introduction

Most work in evolutionary genomics has focused on vertical gene transfer from one species to a lineal descendant. Much recent work has been directed towards the phenomenon of horizontal gene transfer (HGT). Because of the impact of horizontal gene transfers on the ecological and pathogenic character of genomes, algorithms are sought that can computationally determine which genes of a given genome are products of horizontal gene transfer events. Numerous strategies have employed nucleotide composition of coding sequences to predict horizontal gene transfer (see Section 3.3 for references). Early methods flagged genes with atypical G + C content. Later methods evaluated codon usage patterns as predictors of horizontal transfer. A variety of genomic signature models have been proposed, using nucleotide patterns of varying lengths and codon position. These models have been analyzed both individually and in various combinations, using sliding windows, Bayesian classifiers, Markov models, and support vector machines.

Suppose that we have complete, annotated genomes for species $s_1, s_2, \ldots, s_m$. Further, suppose that we have selected a set of genes, from some reference genome or otherwise, for analysis. Let those genes be $g_1, g_2, \ldots, g_n$. Standard methods for obtaining the set of genes, such as the one in Lake and Rivera [41], can be followed. BLASTing gene g_k in species s_i against a database of genes from all m species, we obtain a bit score $B(g_k; s_i, s_j)$ of a best alignment of that gene against the same gene in species s_j. If g_k is not found in s_j, then set $B(g_k; s_i, s_j) = 0$. In general, the higher $B(g_k; s_i, s_j)$ is, the better the match between gene g_k in species s_i and gene g_k in species s_j. There is no need to take special notice of an absent gene, since $B(g_k; s_i, s_j) = 0$ is a meaningful substitute for a Boolean value representing presence or absence of a gene.

There is another quantity associated with the (g_k, s_i, s_j) triple. Define $T(g_k; s_i, s_j)$ to be the true evolutionary distance in time between the g_k gene of s_i and the g_k gene of s_j. For example, if the most recent common ancestor of the two genes existed 20 million years ago, then $T(g_k; s_i, s_j)$ is 40 million years. While these $T(g_k; s_i, s_j)$ values cannot be measured directly, either absolute or relative values for times can be estimated using probabilistic models.

The $B(g_k; s_i, s_j)$ values are not random. In fact, a ranking of the $B(g_k; s_i, s_j)$ values for $1 \leq j \leq m$ should roughly match a ranking of the $T(g_k; s_i, s_j)$ values from the s_i gene g_k to all the other g_k's. In the absence of HGT or other horizontal evolutionary events, we must have $T(g_k; s_i, s_j) = T(g_\ell; s_i, s_j)$ for every pair of genes g_k and g_ℓ. Therefore, we expect that the rankings of the $B(g_k; s_i, s_j)$ and $B(g_\ell; s_i, s_j)$ values will be similar in ways we want to explore. And, under reasonable assumptions, the distribution of relative distances should be consistent with predictions of coalescent theory. In particular, as evolutionary distances increase, there will typically be multiple genes that have the same T value from the g_k gene in species s_i. Moreover, the probability that two evolutionary events occur at the same instance in time is 0.

In the presence of horizontal evolutionary events, the patterns of rankings of the B and T values will be different for different genes, depending on which horizontal events each gene is involved in. Two genes that are involved in exactly the same horizontal events will have identical patterns in their T values and similar patterns in their B values.

If we use the rankings of the B values as an approximate substitute for the rankings of the unknown T values, then the rankings can be compared and clustered to identify groups of genes that participated in the same horizontal events. Fix a gene g_k. Then there is a gene g_k tree that represents the true evolutionary history of the g_k's in all the species. It is rooted at the most recent common ancestor of the m species. Our first goal is to define a computational problem to achieve this clustering and to design an efficient algorithm to solve the problem.

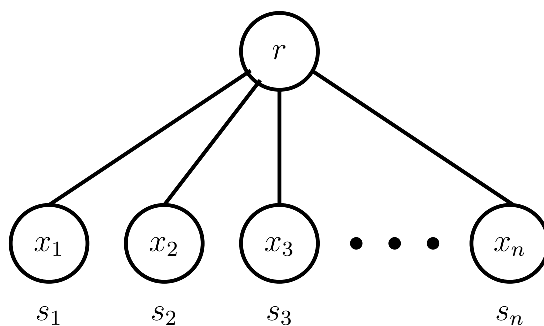


Figure 3.1: A trivial S-tree with a minimum number of nodes.

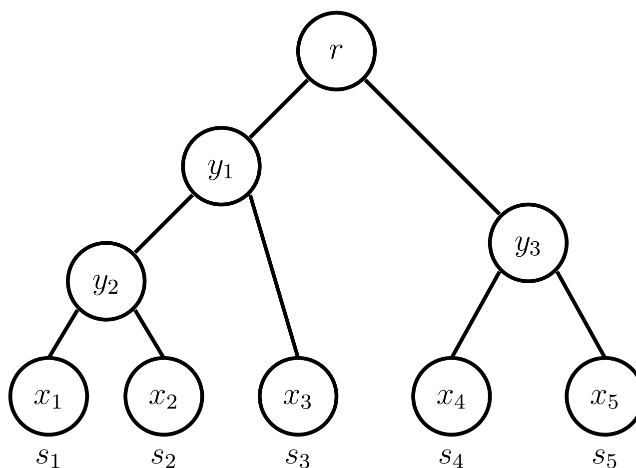


Figure 3.2: An evolutionary S-tree with 5 taxa.

3.2 Definitions

For a rooted (directed) tree T, let $R(T)$ be the root of T, let $I(T)$ be the set of internal nodes of T, and let $L(T)$ be the set of leaves of T.

Let S be a finite set of species. An *S-tree* $T = (V, E)$ is a rooted tree such that every internal node has outdegree at least two and a bijective *labeling function* $\lambda : L(T) \to S$. In particular, every S-tree has precisely $|S|$ leaves. Figure 3.1 illustrates an S-tree for the case $n \geq 2$, where there is only one internal node, the root $r = R(T)$. There are n leaves $x_1, x_2, \ldots, x_n$ and $\lambda(x_i) = s_i$. If every internal node of T has outdegree exactly two, then T is an *evolutionary tree.* Figure 3.2 illustrates an evolutionary tree on five species.

Let $T = (V, E)$ be an S-tree. Let $u \in V$. The subtree rooted at u is $T(u)$. The *species set* $S(u)$ for u is the set of leaf labels in $T(u)$.

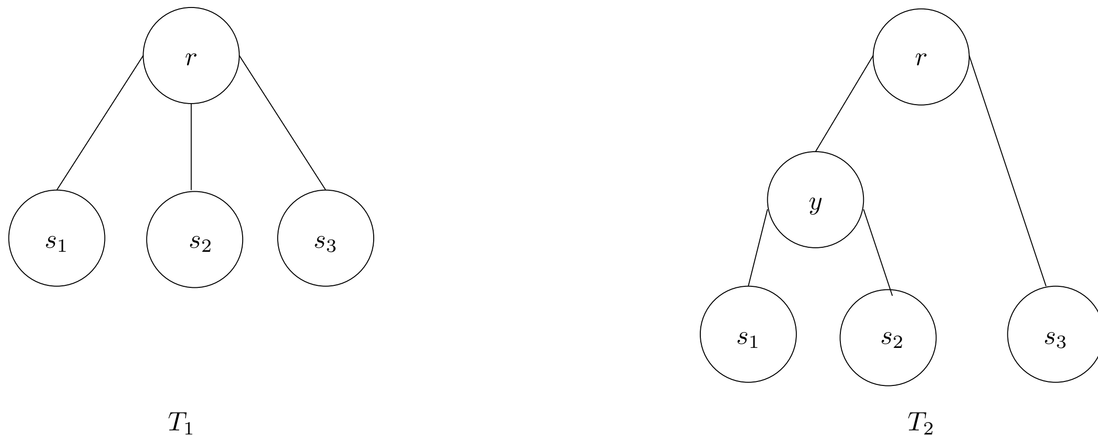


Figure 3.3: Refinement of T_1 to T_2.

Let T be an S-tree with an internal node x that has three or more children. A *refinement step* (on T at x) adds an internal node y to the tree T, where y is the parent of a proper subset of the children of x and y is a new child of x. An S-tree T' is a *refinement* of T if T' can be obtained by performing zero or more refinement steps on T. For example, in Figure 3.3, T_2 is a refinement of T_1 by a refinement step on T_1 at r. The refinement step applied adds one internal node y, which is the parent of s_1 and s_2 in T_2; y and s_3 are the children of r in T_2.

Let $X = \{X_1, X_2\}$ and $Y = \{Y_1, Y_2\}$ be two partitions of S. Call such partitions with two elements each 2-*partitions*. Note that the deletion of an edge from an S-tree induces two connected subtrees and, hence, a 2-partition of S. X and Y are *contradicting partitions* if there exist four species s_1, s_2, s_3, s_4 such that $s_1, s_2 \in X_1$, $s_3, s_4 \in X_2$, $s_1, s_3 \in Y_1$, and $s_2, s_4 \in Y_2$. Two S-trees T_1 and T_2 are *contradictory* if their exists an edge in T_1 and an edge in T_2 such that their induced 2-partitions are contradicting.

Let $u, v \in L(T)$, for some S-tree T. The *most recent common ancestor* MRCA(u, v) of u and v is the node w that is a common ancestor of u and v such that $T(w)$ is the smallest rooted subtree in T containing both u and v.

A *partial order* is a binary relation $\leq$ over a set S that is reflexive, antisymmetric, and transitive, i.e., for all $a, b, c \in S$, we have that

- $a \leq a$ (reflexivity);
- if $a \leq b$ and $b \leq a$ then $a = b$ (antisymmetry); and
- if $a \leq b$ and $b \leq c$ then $a \leq c$ (transitivity).

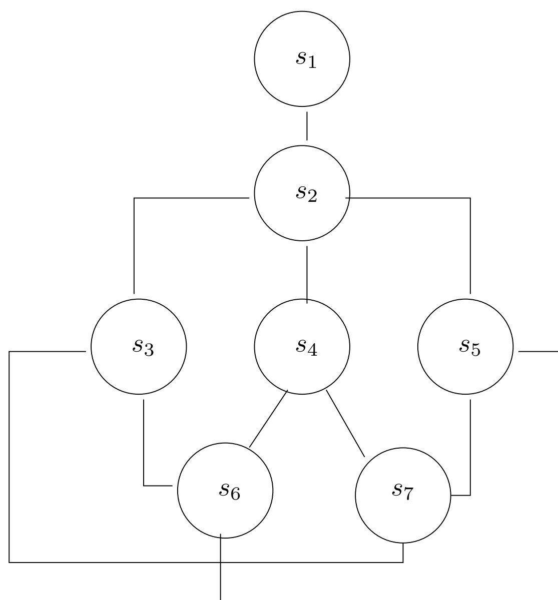


Figure 3.4: An example of a Hasse diagram.

A set with a partial order is a *partially ordered set* or a *poset*. If $(S, \leq)$ is a poset and $a, b \in S$, then $a < b$ if and only if $a \leq b$ and $a \neq b$. Note that $a < b$ is transitive. The directed graph $G = (S, <)$ is clearly a directed acyclic graph (DAG). The *transitive reduction* of G is the DAG on node set S that contains those edges (a, b) such that there is no $c \in S$ satisfying $a < c < b$. A *Hasse diagram* of $<$ (which is also a Hasse diagram of $\leq$) is a drawing of the transitive reduction of $(S, <)$ such that no arrows are included. An example of a Hasse diagram is shown in Figure 3.4. The diagram shown corresponds to the following poset:

$$\begin{aligned} P \quad = \quad & \{(s_1, s_2), (s_1, s_3), (s_1, s_4), (s_1, s_5), (s_1, s_6), (s_1, s_7), \\ & (s_2, s_3), (s_2, s_4), (s_2, s_5), (s_2, s_6), (s_2, s_7), \\ & (s_3, s_6), (s_3, s_7), (s_4, s_6), (s_4, s_7), (s_5, s_6), (s_5, s_7)\}. \end{aligned}$$

Let $s_i \in S$ be a species. An *s_i-poset* $P = (S, \leq_i)$ is a poset with the property that, for every $s_j \in S$, we have $s_i \leq_i s_j$. In other words, s_i is the unique minimum element of P.

The s_i-poset $P_i = (S, \leq_i)$ is *compatible* with S-tree T if, for all distinct triples $x, y, z \in L(T)$ such that $\lambda(x) = s_i$, $\lambda(y) = s_j$, and $\lambda(z) = s_k$ and such that $s_j \leq_i s_k$, then we have the shortest path from either of x or y to z passes through $\text{MRCA}(x, y)$. Given the tree shown in Figure 3.5, Figure 3.6 shows an example of a poset that is compatible with the given tree, while Figure 3.7 shows an incompatible poset, where the poset indicates that s_3 is the closest species to s_1, while, in the tree, the closest species to s_1 is s_2.

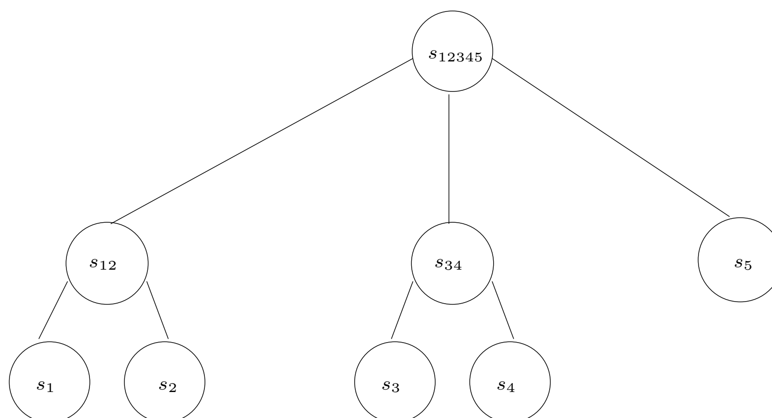


Figure 3.5: An example of a tree to test compatibility with posets.

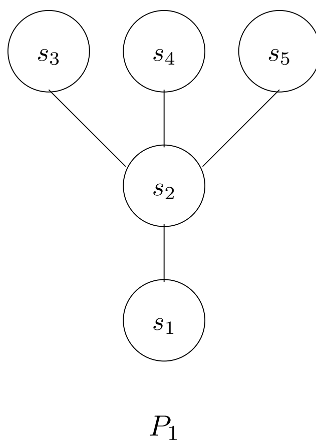


Figure 3.6: An example of a poset compatible with the tree in Figure 3.5.

Let $\mathcal{P} = \{P_1, P_2, \ldots, P_n \mid P_i \text{ is an } s_i\text{−poset}\}$ be a set of posets. $\mathcal{P}$ is *consistent* if, for all posets $P_i, P_j \in P$, whenever $s_j \leq_i s_k$, then $s_i \leq_j s_k$. For example, let $P_1 = \{(s_1, s_2), (s_1, s_3), (s_2, s_3)\}$, $P_2 = \{(s_2, s_1), (s_2, s_3), (s_1, s_3)\}$, and $P_3 = \{(s_3, s_1), (s_3, s_2)\}$. Then, $\{P_1, P_2, P_3\}$ is consistent. However, if $P_4 = \{(s_3, s_1), (s_3, s_2), (s_1, s_2)\}$, then $\{P_1, P_2, P_4\}$ is inconsistent, since P_1 and P_2 indicate that s_1 and s_2 are closer to each other than to s_3, while P_4 indicates that s_1 is closer to s_3 than to s_2.

3.3 Literature Review

Much research is directed towards evolutionary event detection and phylogenetic reconstruction. Here, we concentrate on the detection of horizontal (lateral) gene transfer.

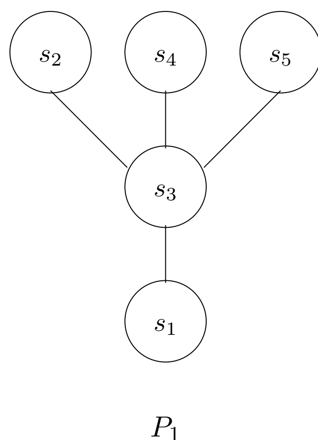


Figure 3.7: An example of a poset incompatible with the tree in Figure 3.5.

Horizontal gene transfers occur when a species transfers genes to other species that are not their direct offspring. In contrast to vertical gene transfer, where genes are transferred from parent to child through sexual or asexual reproduction, HGT allows sharing of genetic material without the relatedness of the species involved. Microorganisms are the most affected by HGT, however, there is evidence that only 2% [30] of the core genes in microorganisms are horizontally transferred. And, the most extreme cases are those that occur between different sub-kingdoms, an example is bacterial genes from *Agrobacterium* species that have been found in tobacco plants, *Nicotiniana* [21]. Perna et al. [63] present analyses to advance the understanding of the evolution of *E. coli*, by comparing it with the genome of the non-pathogenic laboratory strain *E. coli K-12*. *E. coli O157:H7* was discovered as a human disease in 1982 from a contaminated hamburger. The strain *EDL933* sequenced was isolated from Michigan ground beef, that is the reference strain *O157:H7*. *EDL933* was compared to the *E. coli K-12* laboratory strain *MG1655*, where it was found that their most recent common ancestor was 4.5 million years ago. The two strains share a common backbone sequence that is colinear except for one 422 kb inversion. Moreover, there are hundreds of DNA segments that were identified as K-islands and O-islands. Where K-islands are DNA segments found in *MG1655* but not in *EDL933*, and O-islands are those that are uniquely found in *EDL933*. 26% of *EDL933* genes were found to lie within the O-islands. The 26% correspond to 1,387 new genes that they believe are horizontally transferred. In [55], it was found that the genome sequence of *Thermotoga maritima* has 24% of its open reading frames (ORFs) with highest similarity scores to archaeal genes in BLAST analyses. Nesbo et al. [55] screened 16 strains from the *genus Thermotoga* and other related *Thermotogales* to find the origin of the two genes, the gene encoding the large subunit of glutamate synthase (gltB) and the myo-inositol 1P synthase gene (ino1), and they found that the two genes were acquired from Archaea during the divergence of the *Thermotogales*. Similar database searches revealed other horizontal transfers between bacteria and archaea,

where three other bacteria, namely, *Dehalococcoides ethenogenes*, *Sinorhizobium meliloti*, and *Clostridium difficile*, possess archaeal genes.

There are several ways for HGT to occur among organisms. In prokaryotes, there are three ways for HGT to occur:

- Transformation, where the recipient cell takes up from its environment a fragment of DNA, rarely longer than 50 kb, released from a donor cell [18].
- Transduction involves transfer of a small segment of DNA from donor to recipient via a bacteriophage [18].
- Conjugation, where two bacteria come into physical contact and one bacterium (the donor) transfers DNA to the second bacterium (the recipient). The transferred DNA can be a copy of some or possibly all of the donor cell's chromosome, or it could be a segment of chromosomal DNA, up to 1 Mb in length, integrated in a plasmid [18].

Eukaryotes can be classified into different classes, among which are:

- Protists: Protists historically belonged to the kingdom Protista. This kingdom includes mostly unicellular organisms that do not fit into the other kingdoms [75]. However, in modern taxonomy protists are considered a grouping of 30 or 40 phyla with diverse combinations of trophic modes, mechanisms of motility, cell coverings and life cycles [32]. In other words, protists are either unicellular organisms, or multicellular organisms without specialized tissues. An example of HGT is in the protist *Entamoeba histolytica*, where there are 96 cases of HGT from prokaryotes [44].
- Fungi: In [25], it is shown that *Saccharomyces cerevisiae* has received two genes from bacteria by HGT, The analysis also suggests that eight genes from *Yarrowia lipolytica*, five from *Kluyveromyces lactis*, and one from *Debaryomyces hansenii* are horizontally transferred.
- Other eukaryotes: An example of HGT within multicellular eukaryotes, is the transfer of genes from chloroplast and mitochondrial genomes to the nuclear genomes [60].
- Plants: Natural movement of genes between different plant species and from other kingdoms into plants occurs naturally, and the transfer is acquired by means of direct cell to cell transfer, or through microorganisms, parasites, epiphytes, or viruses [69, 79]. Another example of horizontal gene transfer in plants is through mobile genetic elements in plants, such as transposons. These transposons move to new locations in a genome, and can also move to new species. An example of such transfers is found in food crops such as rice [28].

- Animals: An example of HGT in animals is found in the adzuki bean beetle *Callosobruchus chinensis*, it was found to be infected with several strains of bacterial *Wolbachia endosymbionts*, where a genome fragment of one of these endosymbionts is transferred to the X chromosome of the host insect [38].

According to [13], HGT detection methods can be classified into:

- Compositional methods: these methods depend on the fact that horizontally transferred genes come from different background, and hence their nucleotide sequences contain information about their previous location. Some methods use nucleotide or oligonucleotide composition [42, 37], other methods use codon usage patterns [43], or alternatively a combination of both [84]. However, these methods are applicable to recent transfers, because the older the transfer is, the more the gene adapts to the new genome.

- Use of phyletic patterns: these methods find the homologs of a gene by performing similarity searches or clustering. The phyletic pattern of a gene is used to detect HGT events. Some of these methods use a top-scoring approach [65], where the most similar sequence is used to detect HGT. However, the top-scoring BLAST is not always the nearest neighbor [39]. This is overcome by taking into account the taxonomic information to re-rank the BLAST hits and exclude the closely related organisms.

- Use of substitution rates: these methods compare gene substitution rates. Novichkov et al. [56] compare pairwise distances within a gene family to pairwise genomic distances. If the gene distances is the same as genomic distances, this suggests that the gene was vertically transferred. However, horizontally transferred genes would have equal gene distances regardless of the genomic distances. And, gene distances can differ from genome differences for reasons other than HGT. Moreover, these methods only consider genes that have orthologs in other genomes. Another method proposed in [31] uses changes in the nucleotide substitution matrix to detect HGT. The method uses the assumption that horizontally transferred genes should have substitution matrices that differ from the genome as a whole. Drawbacks of this method is that due to sampling errors, it might not be feasible to accurately estimate the substitution matrix for a single gene.

- Phylogenetic incongruence: these methods reconstruct phylogenetic trees for individual gene families, assuming that trees with unexpected histories are results of HGT. These methods rely on a reference phylogenetic tree, and hence, one drawback for these methods is that horizontal transfers between taxa in the reference tree are not detected. Also, the results obtained are dependent on the choice of the reference tree. Among phylogenetic incongruence methods are the ones presented in [48] and [49].

- Scanning for polymorphic sites: these methods are used to detect transferred genes in closely related

organisms. Mau et al. [51] describe a method to identify the segments of genomes resulting from orthologous replacement of homologous regions from closely related genomes.

In what follows, some of the methods developed for HGT detection are presented in more detail. Among the methods addressed by many researchers is conditioned reconstruction. Conditioned reconstruction (CR) is a phylogenetic technique that utilizes gene absence/presence data to reconstruct phylogenetic relationships [5]. CR [41], compares a genomic sequence to another and according to whether a gene ortholog is present or absent supplies a P or A character state. The probability of a state transition is analyzed using Markov models. Given two genes, X and Y, four patterns are possible, PP, PA, AP, and AA. Many questions were raised on how to count the pattern AA. How can one identify genes that are missing from both genomes X and Y. To solve this problem, CR uses a conditioning genome, as a reference to which genes to be considered. A gene has to be present in both the conditioning genome and the genome being coded, in order to be considered present. An absent gene is present in the conditioning genome and absent from the genome under study. The conditioning genome has a big effect on the results obtained, as it represents the full set of orthologous genes coded during matrix development. In our approach, we avoid building our results on a conditioning genome, or any other input that would bias our results. However, the approach we present is similar to CR in the problem addressed and the use of information about all genes in the genomes. CR defines fibers as representatives to gene orthologs, fibers representing present genes are colored gray, and white if they represent absent genes. Those fibers are contained into bands, representing the genomes. The conditioning genome has all fibers colored gray. For the genomes being analyzed, gene duplication and deletion does not affect the band size, as it already contains a fiber for each gene. Therefore, duplication will have no effect, whereas deletion will only change the color of the fiber from gray to white. Similarly, insertion changes the color of the fiber from white to gray. If a gene is horizontally transferred, then the fibers will not be effected if the transferred gene is not orthologous to a gene in the conditioning genome. However, if an orthologous gene to the horizontally transferred one is present in the conditioning genome, then the color of the corresponding fiber in the genome being analyzed will be modified accordingly.

Bailey et al. [5] argue that CR cannot be used to distinguish between HGT and genome fusion. They suggest some refinements that make CR perform better. Bapteste and Walsh [6] question the ring of life hypothesis of Lake and Rivera [41]. They claim that it is not possible to reconstruct the ring of life in the presence of HGT. Bapteste and Walsh [6] see that the conditioning genome (CG) is more a tool than a biological concept, this genome can exist anywhere in the tree of life, and can not be used in evolutionary reconstruction. Using a CG may come in the way of identifying HGT. For example, if the CG has many genes that are acquired by HGT from a genome other than those being compared, then many of the genes in the genomes under study will be considered absent. Moreover, if the CG has many fewer HGTs than the genomes under study,

then many HGTs will not be identified. This is quoted from Lake and Rivera's paper [41]: "Note that at the location of the conditioning genome C, all the fibers are gray since each one represents a gene present in the conditioning genome. Also note that gene deletions and gene duplications do not affect the width of the bands since the set of genes, and hence the number of fibers, is solely determined by the conditioning genome. If a gene that was present in the conditioning genome is duplicated in genome X, this does not affect the number of fibers in the band since only one copy of an ortholog contributes to genome X. If a gene is deleted, then the character state of that fiber goes from P (gray) to A (white). If a new gene is horizontally transferred into a genome and corresponds to a gene that is not present in the conditioning genome, then it will not affect the coloring of the band since there will be no fiber corresponding to this gene. If the new gene is orthologous to a gene present in the conditioning genome, and that gene is lacking in the current organism, then the coloring of the orthologous gene will change from white to gray, A to P. And if the new gene corresponds to an orthologous gene that is already present in the current genome, then the coloring of the fiber will not change. Gene duplications do not change the character state of the fiber. This is because gene duplications must start from an existing gene, implying that its initial character state must have already been P, so that the second gene does not alter its state. If a second copy of a gene is acquired by HGT that, likewise, will not affect the coloring. Finally, as previously discussed, the source of an acquired gene is not coded and hence is irrelevant." This quote shows that detecting HGT is not really possible by their method, however, they claim it can be used to detect fusion. However, their method can mistake a genome fusion with multiple HGT events. Bapteste and Walsh claim that multiple HGT events give the same presence/absence patterns as a single fusion event, and hence, when obtaining multiple incongruent trees, with different bootstrap scores, fusion is one possible interpretation of that. However, Lake and Rivera ignored other possibilities that could lead to other conclusions, and kept their choices between one well-supported tree and a ring, in the case of multiple best trees. However, the bootstraps of multiple trees have high discrepancy, and hence, some conflicts in the data are expected. Spencer et al. [78] investigate the CR method and discuss the effect of the conditioning genome, and how the CR method can avoid choosing the conditioning genome. They suggest combining information from all choices of conditioning genomes.

A distance based method for detecting horizontal gene transfer is presented by Wei at al. [88]. In this method, BLAST is used to identify gene orthologs in multiple species, then multiple alignment is obtained using ClustalW. From the multiple alignment, Phylip is used to calculate distances. Distance vectors are obtained for each pair of species, and the length of the vector is the number of genes considered. The distance vector is then used to identify genes that are outliers; this is done by comparing the distance corresponding to each gene to the mean and standard deviation of the vector. The outliers are candidates to be involved in HGT.

Other methods for detecting horizontal gene transfer are proposed by multiple researchers. Podell and Gaasterland [65] present the DarkHorse method for detecting HGT. They defined the LPI, lineage probability index, to measure HGT and species closeness. This measure relies on lineage key terms. The higher the LPI score for an organism, the closer it is to the query (reference) genome. Groups of closely related organisms, have similar LPI scores. Tsirigos and Rigoutsos [84] also presented a computational method for detecting HGT. The method relies on the nucleotide composition of genes, without the need for knowledge of the codon boundaries. Novozhilov et al. [57] describe a stochastic birth and death model of evolution of horizontally transferred genes in microbial populations.

Moreover, phylogenetic reconstruction research contributed in solving many evolutionary problems. Huson and Bryant [33] outline a statistical framework for applying split networks method. They also present the SplitsTree4 program, a tool for inferring different types of phylogenetic networks from sequences, distances, and trees. Nakhleh et al. [53] present a method for reconstructing phylogenetic networks using maximum parsimony. Their method is then studied and applied in [34]. Willson [90] use TOM-networks to reconstruct tree-like phylogenetic networks using distances between leaves. Marjoram and Tavare [50] highlight some computational models for analyzing molecular genetic variation data and show how coalescence is used for such analysis. Rosenberg and Nordborg [70] develop a statistical framework using coalescent theory to analyze genetic polymorphism. Long et al. [45] list the sources of new genes and highlight the effect and importance of each mechanism. Blomme et al. [12] study the gain and loss of genes during 600 million years of vertebrate evolution. Roth et al. [71] study the evolution of genes after duplication, and they present models and mechanisms to study the process of gene duplication.

Kannan and Warnow [36] study partial order sets and phylogeny, and they define networks that could be reconstructed given the partial order sets among the species. In the problem they define, they are given an n-by-n distance matrix M and they find a tree T with leaves $1, 2, ..., n$ such that the path distance $d + T_{ij}$ in the tree closely approximates the matrix M. Kannan and Warnow [36] use two types of experiments, a total order model (TOM), which determines the total order of the three pairwise distances $d(i, j), d(j, k), d(i, k)$, with equality or strict inequality indicated and a partial order model (POM), which determines the minimum elements of $d(i, j), d(j, k), d(i, k)$. They studied two problems under the two models they defined. The first is the consistency problem, a decision problem stated as follows: Given a partial order on a set of distances, does there exist an edge-weighted tree which realizes this partial order? The second problem is the construction problem, which given the ability to perform an experiment, determines how quickly can we construct an edge-weighted tree realizing the experiments. In [36] they show that the consistency problem is NP-complete. And they also present techniques to construct unweighted binary trees in $O(n^3)$ time from TOM experiments and in $O(n^4)$ time from POM experiments.

Snir and Trifonov [77] present a method for detecting HGT. Their algorithm takes two genomes with their lengths and calculates the expectancy of each identical region's length to obtain a measure of confidence as to exceptional similarity. The algorithm then identifies matching seeds for putative horizontal transfer such that their size is chosen to have very low probability of having occurred by chance. This is done based on the evolutionary distance between the two genomes to filter out random similarities. Using the statistical and biological property that a horizontal transfer is normally not inserted exactly in its homologous counterpart location, it is expected that the flanking regions of a horizontal transfer are non-homologous. The authors use a sliding window algorithm to detect horizontal transfer borders, searching for sharp borders. Duplicate occurrences of possible horizontal transfers are discarded, as these may be a result of some other event.

Abby et al. [1] present a program called Prunier for the detection of HGT. The program searches for a maximum statistical agreement forest between a gene tree and a reference tree.

3.4 Posets and Trees

In this section, we show how, given a set of posets $\mathcal{P}$, an S-tree is constructed. And, we also show how to obtain the posets from a given tree.

3.4.1 Constructing an S-tree From a Set of Posets $\mathcal{P}$

Recall the definition of compatible from Section 3.2: The s_i-poset $P_i = (S, \leq_i)$ is *compatible* with S-tree T if, for all distinct triples $x, y, z \in L(T)$ such that $\lambda(x) = s_i$, $\lambda(y) = s_j$, and $\lambda(z) = s_k$ and such that $s_j \leq_i s_k$, then we have the shortest path from either of x or y to z passes through $\text{MRCA}(x, y)$.

The problem of constructing a tree is defined as follows:

Compatible Tree Construction

INSTANCE: Set $S = \{s_1, s_2, \ldots, s_n\}$ of n taxa; for $1 \leq i \leq n$, an s_i-poset $P_i = (S, \leq_i)$.

SOLUTION: An S-tree T compatible with $P_1, P_2, \ldots, P_n$, if one exists.

Theorem 16. *Let $\mathcal{P}$ be a set of posets that is compatible with an S-tree T. Let T' be a refinement of T. Then $\mathcal{P}$ is compatible with T'.*

Proof. The proof is by induction on the number of refinement steps, k, to obtain T' from T. For the base case of the induction, assume that $k = 0$. Then $T' = T$, and, therefore, $\mathcal{P}$ is clearly compatible with T'. Now assume that $k \geq 1$ and that the result holds for $k - 1$ refinement steps. Then there exists an S-tree T''

such that T'' is obtained by $k-1$ refinement steps from T and T' is obtained from T'' in one refinement step. Let u in T'' have children $v_1, v_2, \ldots, v_p$ such that in T' there is a new node w that is a child of u with children $v_1, v_2, \ldots, v_q$, where u retains children $v_{q+1}, \ldots, v_p$ in T'. Note that $q \geq 2$ and $p-q \geq 1$. Therefore, for $\mathcal{P}$ to be compatible with T', the compatibility condition must hold, and that is:

- For all distinct triples $x, y, z \in L(T)$ such that $\lambda(x) = s_i$, $\lambda(y) = s_j$, and $\lambda(z) = s_k$ and such that $s_j \leq_i s_k$, then there is a shortest path from either of x or y to z passing through MRCA (x, y).

By applying the compatibility condition to T'', the cases for x, y, and z are as follows:

- $x \in v_1, v_2, \ldots, v_p$ or $y \in v_1, v_2, \ldots, v_p$. Since $s_j \leq_i s_k$, therefore, there exists an MRCA for x and y. Let MRCA (x, y) be q. Therefore, the shortest path from either of x or y to z passes through q.

- $x, y \in v_1, v_2, \ldots, v_p$. Therefore, MRCA (x, y) is u, and the shortest path from either of x or y to z passes through u.

- $x, y \notin v_1, v_2, \ldots, v_p$. Since, $s_j \leq_i s_k$, therefore, there exists an MRCA for x and y such that the shortest path from either of x or y to z passes through the MRCA (x, y).

Similarly, by applying the compatibility condition to T', the cases for x, y, and z are as follows:

- $x, y \in v_1, v_2, \ldots, v_q$. Therefore, MRCA (x, y) is w, and the shortest path from either of x or y to z passes through w.

- $x, y \in v_{q+1}, \ldots, v_p$. Therefore, MRCA (x, y) is u, and the shortest path from either of x or y to z passes through u.

- $x \in v_1, v_2, \ldots, v_q$ and $y \in v_{q+1}, \ldots, v_p$. Therefore, MRCA (x, y) is u and the shortest path from either of x or y to z passes through u.

- $y \in v_1, v_2, \ldots, v_q$ and $x \in v_{q+1}, \ldots, v_p$. Therefore, MRCA (x, y) is u and the shortest path from either of x or y to z passes through u.

- $x \in v_1, v_2, \ldots, v_p$ or $y \in v_1, v_2, \ldots, v_p$. Since $s_j \leq_i s_k$, therefore, there exists an MRCA for x and y. Let MRCA (x, y) be q. Therefore, the shortest path from either of x or y to z passes through q.

- $x, y \notin v_1, v_2, \ldots, v_p$. Since, $s_j \leq_i s_k$, therefore, there exists and MRCA for x and y such that the shortest path from either of x or y to z passes through the MRCA (x, y).

Therefore, if the compatibility condition holds for T'', and T' is obtained using one refinement step from T'', then the compatibility condition also holds for T'.

By induction, $\mathcal{P}$ is compatible with T', as required. □

Now we present a data structure that the algorithm uses to identify siblings. For the set of posets, $\mathcal{P}$, a matrix A of size $n \times n$ is defined. We define

$$A(i,j) = \begin{cases} |\{s_x \mid s_j <_i s_x\}| & \text{if } i \neq j; \\ -1 & \text{if } i = j. \end{cases}$$

In other words, for $i \neq j$, $A(i,j)$ is the number of species s_x such that s_j is strictly less than s_x in the poset $(S, \leq_i)$.

Theorem 17. *Let $\mathcal{P}$ be a set of posets, and let A be the matrix representing $\mathcal{P}$. If $\mathcal{P}$ is consistent, then A is symmetric.*

Proof. Let $\mathcal{P} = \{P_1, P_2, \ldots, P_n \mid P_i \text{ is an } s_i\text{−poset}\}$ be a set of posets. $\mathcal{P}$ is *consistent* if, for all posets $P_i, P_j \in P$, whenever $s_j \leq_i s_k$, then $s_i \leq_j s_k$. Let $1 \leq i < j \leq n$. By the consistency condition, $\{s_x \mid s_j <_i s_x\} = \{s_x \mid s_i <_j s_x\}$. Therefore, $A(i,j) = A(j,i)$, and A is symmetric. □

This A matrix represents an undirected graph, where siblings are indicated by cliques in the graph, that is, for a species s_i, all other species connected to s_i with edges having equal labels, then they are siblings. Higher values indicate siblings at lower levels in the tree, in other words, the maximum value indicates leaf siblings. Note that if there is missing data or incorrect data in the posets, there will be a problem in constructing the tree, for example, if the posets have missing information or incorrect information then the algorithm will not be able to construct a tree for that specific gene corresponding to that posets set. To follow is an example to illustrate the defined data structures. Consider the set of posets $\mathcal{P}$, where $\mathcal{P}$ is given as follows:

$$\begin{aligned} P_1 &= (s_1, s_2), (s_1, s_3), (s_1, s_4), (s_2, s_4), (s_3, s_4) \\ P_2 &= (s_2, s_1), (s_2, s_3), (s_2, s_4), (s_1, s_4), (s_3, s_4) \\ P_3 &= (s_3, s_1), (s_3, s_2), (s_3, s_4), (s_1, s_4), (s_2, s_4) \\ P_4 &= (s_4, s_1), (s_4, s_2), (s_4, s_3) \end{aligned}$$

The matrix A corresponding to $\mathcal{P}$ is shown in Table 3.1

And the graph G that is represented by the matrix A given in Table 3.1 is shown in Figure 3.8, where s_1, s_2, and s_3 are siblings, and their parent and s_4 are both children of the root.

For example, the matrix in Table 3.2 is constructed for the posets in Figure 3.9.

	s_1	s_2	s_3	s_4
s_1	-1	1	1	0
s_2	1	-1	1	0
s_3	1	1	-1	0
s_4	0	0	0	-1

Table 3.1: Matrix A for the set of posets $\mathcal{P}$.

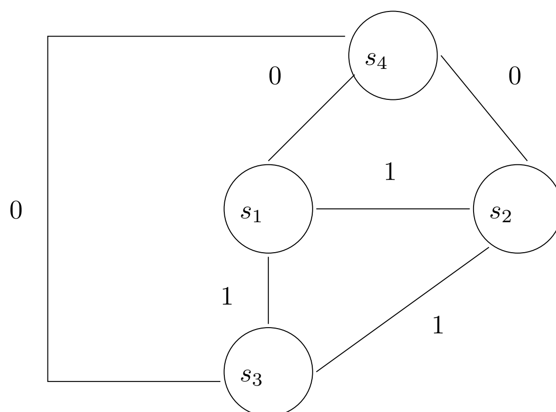


Figure 3.8: An undirected graph with cliques representing siblings.

The graph in Figure 3.10 shows the cliques that represent siblings indicated by matrix A in Table 3.2.

The first row of matrix A indicates that s_2 is a sibling of s_1. The maximum value in the s_1 row is 3, which is in the s_2 column, and it is the only column with this value. This is also clear in the graph shown in Figure 3.10. Since the maximum value found in the s_1 row is 3, and it is only under the s_2 column, therefore, s_2 is the only sibling of s_1. Similarly, s_4 and s_5 are also siblings.

The algorithm starts by the procedure of inferring siblings by detecting cliques in the graph. For each species, the algorithm scans the row corresponding to that species, and detects which species are connected using edges with equal labels. The detected species are all siblings. After detecting each set of siblings comes the updating step. In this step, the rows and columns of the siblings are merged. This procedure is repeated until only one species is remaining, which is the root.

After scanning the s_1 row, the matrix A is reduced as shown in Table 3.3.

Similarly, the matrix A is reduced after detecting the siblings s_4 and s_5, as shown in Table 3.4.

This procedure is repeated, but this time the highest integer is 2, therefore, s_3 is a sibling of s_{12}, the parent

	s_1	s_2	s_3	s_4	s_5
s_1	-1	3	2	0	0
s_2	3	-1	2	0	0
s_3	2	2	-1	0	0
s_4	0	0	0	-1	3
s_5	0	0	0	3	-1

Table 3.2: Matrix A for posets in Figure 3.9.

	x	s_3	s_4	s_5
x	-1	2	0	0
s_3	2	-1	0	0
s_4	0	0	-1	3
s_5	0	0	3	-1

Table 3.3: Matrix A for posets in Figure 3.9 after reducing s_1 and s_2.

	x	s_3	y
x	-1	2	0
s_3	2	-1	0
y	0	0	-1

Table 3.4: Updated matrix A for posets in Figure 3.9 after reducing s_4 and s_5.

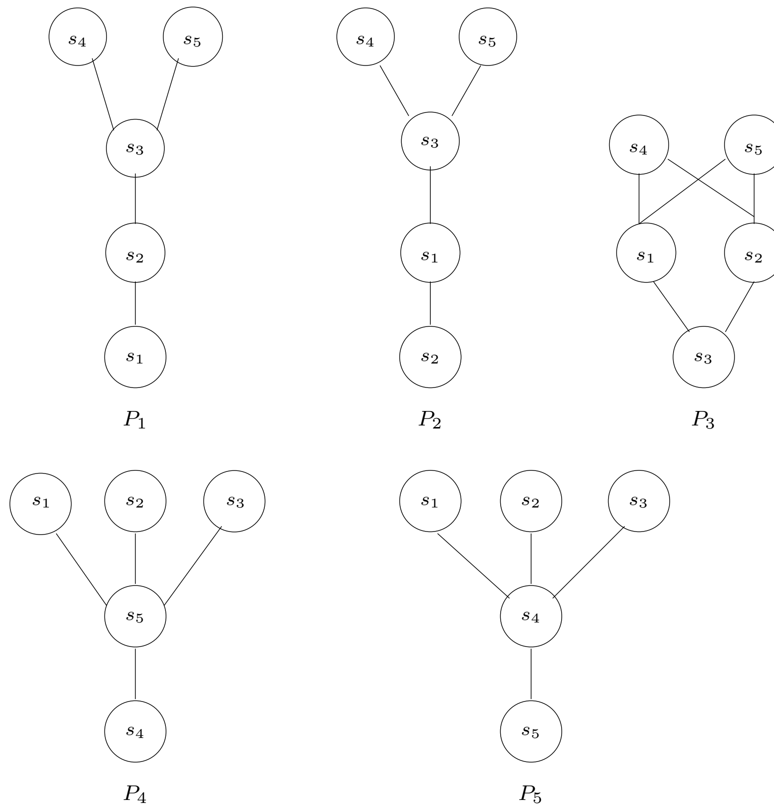


Figure 3.9: Diagram for posets.

of s_1 and s_2. And, the new matrix is shown in Table 3.5.

The final step creates one root for the remaining species because all the values are 0, hence, all the remaining species are at the same level. The tree reconstructed from the posets in Figure 3.9 is shown in Figure 3.11.

Another example to further illustrate the algorithm uses the set of posets $\mathcal{P}$ in Figure 3.12.

The matrix in Table 3.6 is constructed for the set of posets $\mathcal{P}$ in Figure 3.12.

The largest integer is 3, and it indicates that s_1, s_2, and s_3 are siblings, as well as s_4, s_5, and s_6.

The matrix then becomes as shown in Table 3.7.

Therefore, one root is created for the remaining two nodes to construct the tree in Figure 3.13.

Figure 3.14 shows the algorithm for reconstructing a tree from a set of posets $\mathcal{P}$. The algorithm validates the matrix A by testing that $A[i, j] = A[j, i]$, for all i and j, where $1 \leq i \leq n$ and $1 \leq j \leq n$. The algorithm for validation is shown in Figure 3.15. The algorithm also uses a subroutine to find cliques with equal edge

	z	y
z	-1	0
y	0	-1

Table 3.5: Updated matrix A for posets in Figure 3.9.

	s_1	s_2	s_3	s_4	s_5	s_6
s_1	-1	3	3	0	0	0
s_2	3	-1	3	0	0	0
s_3	3	3	-1	0	0	0
s_4	0	0	0	-1	3	3
s_5	0	0	0	3	-1	3
s_6	0	0	0	3	3	-1

Table 3.6: Matrix A for posets in Figure 3.12.

	x	y
x	-1	0
y	0	-1

Table 3.7: Updated matrix A for posets in Figure 3.12.

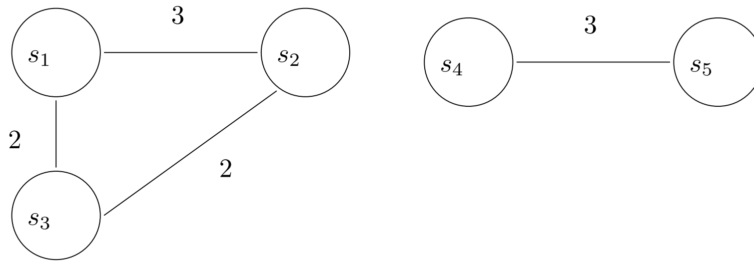


Figure 3.10: An undirected graph corresponding to the matrix shown in Table 3.2.

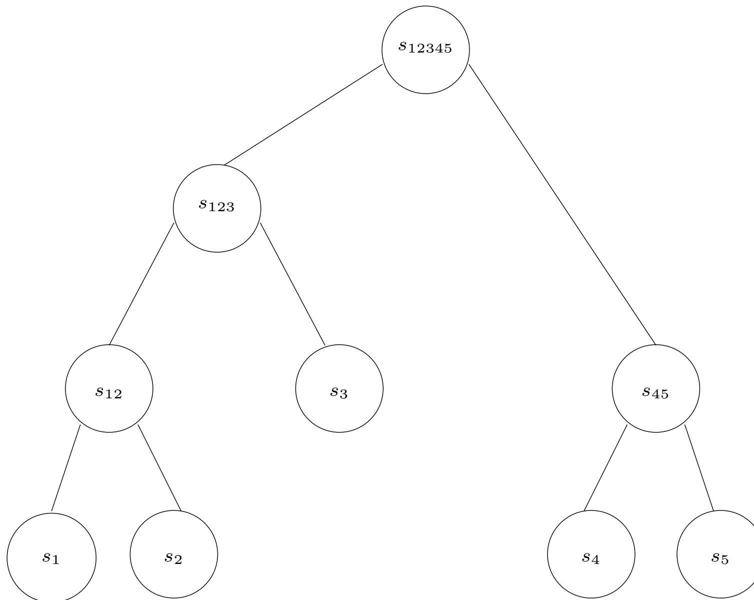


Figure 3.11: Tree corresponding to the posets in Figure 3.9.

labels. The subroutine scans the matrix A to find a clique with maximum edge labels. The subroutine for finding the maximum edge label is shown in Figure 3.16, and the algorithm for finding the cliques is shown in Figure 3.17. The subroutine AddSiblings shown in Figure 3.18 shows the steps for adding the vertices that belong to a certain clique as siblings in the tree T. The subroutine also reduces the graph by merging the rows and columns in the matrix A.

Theorem 18. *The algorithm ConstructTree has $O(n^3)$ time complexity.*

Proof. Lines 2–6 in the algorithm ConstructTree contain two nested loops, each of which repeats n times. The statement in line 6, which is repeated in the nested loops, takes $O(n)$ time, that is because the poset P_i contains, at most, n ordered pairs with $x = s_j$. Therefore, the total amount for these three nested loops

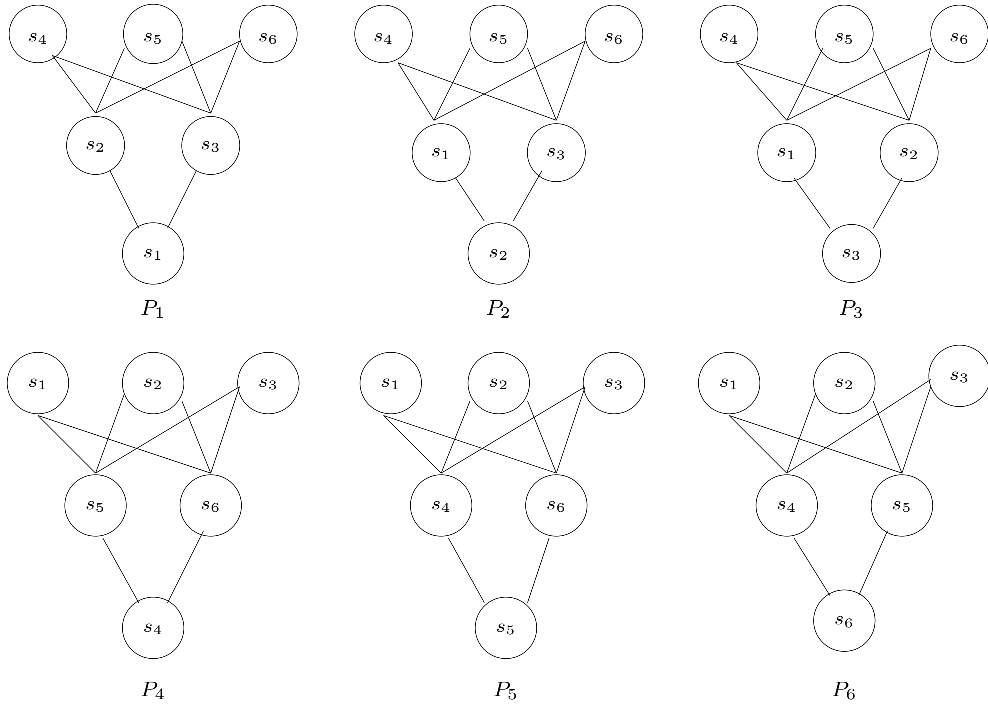


Figure 3.12: Diagrams for posets.

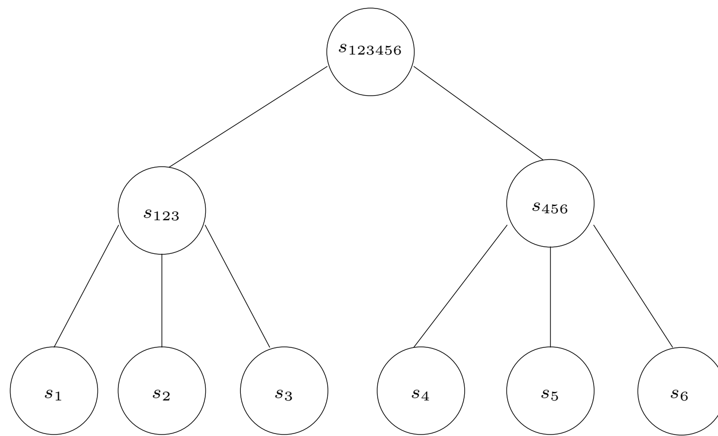


Figure 3.13: Tree corresponding to the posets in Figure 3.12.

```
1  begin ConstructTree(𝒫);
2  ▷ 𝒫 = {P_1, P_2, ..., P_n}
3  for i ← 1 to n
4      do for j ← 1 to n
5          do if i = j
6              then A[i, j] ← −1
7              else A[i, j] ← |{s_x | s_j <_i s_x}|
8  if not Validate(A) then return "No compatible S-tree"
9  T ← ({s_1, s_2, ..., s_n}, ∅)        ▷ Initially, a graph with no edges
10 while n ≥ 1
11         do (max, row, col) ← FindMax(A)
12             clique ← FindClique(A, max, row)
13             (T, A) ← AddSiblings(T, clique)
14             n ← n − (|clique| − 1)
15 return T
```

Figure 3.14: Algorithm to construct an S-tree from a set of posets $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$.

```
1  begin Validate(A)
2  for i ← 1 to n
3      do for j ← 1 to n
4          do if A[i, j] ≠ A[j, i]
5              then return False
6  return True
```

Figure 3.15: Algorithm to validate an $n \times n$ matrix A.

1 **begin** FindMax(A, row, col)
2 max $\leftarrow -1$
3 **for** $i \leftarrow 1$ **to** n
4   **do for** $j \leftarrow 1$ **to** n
5   **do if** $A(i, j) > \max$
6   **then** max $\leftarrow A(i, j)$
7             $(\text{row}, \text{col}) \leftarrow (i, j)$
8 **return** $(\max, \text{row}, \text{col})$

Figure 3.16: Algorithm to find the maximum of a matrix A.

1 **begin** FindClique(A, max, row)
2 clique $\leftarrow$ node corresponding to row
3 **for** $j \leftarrow 1$ **to** n
4   **do if** $A(\text{row}, j) = \max$
5   **then** clique = clique $\cup$ {node corresponding to j}
6 **for** all nodes $i \in$ clique
7   **do for** all nodes $j \in$ clique such that $i \neq j$
8   **do if** $A(i, j) \neq \max$ **then return** "Invalid clique"
9 **return** (clique)

Figure 3.17: Algorithm to find a clique with edge labels equal *max*.

1 **begin** AddSiblings(T, clique)
2 Let clique $= \{s_{i_1}, s_{i_2}, \ldots, s_{i_k}\}$
3 $x \leftarrow s_{i_1, i_2, \ldots, i_k}$ $\triangleright$ New node
4 $V(T) \leftarrow V(T) \cup \{x\}$ $\triangleright$ $V(T)$ is the set of vertices of the tree T
5 $E(T) \leftarrow E(T) \cup \{(x, s_{i_p}) \mid i_p \in \{i_1, i_2, \ldots, i_k\}\}$ $\triangleright$ $E(T)$ is the set of edges of the tree T
6 clique = clique $- s_j$ where s_j is the first element in the set clique
7 **for all** nodes $i \in$ clique
8   remove row i and column i from A
9 **return** (T, A)

Figure 3.18: Algorithm to add elements of a clique as siblings in a tree T.

will be $O(n^3)$. Lines 8 scans the matrix A in $O(n^2)$. The while loop on line 10 repeats at most n times, on line 11, FindMax is $O(n^2)$, on line 12, FindClique is $O(n^2)$, AddSiblings on line 13 is $O(n)$, Therefore, the while loop takes $O(n^3)$. Therefore, the complexity of the algorithm is $O(n^3)$. □

Theorem 19. *The algorithm ConstructTree solves the* Compatible Tree Construction *problem.*

Proof. To prove the theorem, we use induction on the number of species. Let the number of species be n. For $n = 1$ and $n = 2$, there is no maximum value in the matrix A, hence, the tree is trivial. For $n = 3$, there are three possibilities for the third species s_3. Either s_3 is a sibling of s_1 and s_2, a sibling of their parent, or a sibling of either one of them. The algorithm checks the values in the A matrix, if $A(1,3) = A(2,3) = A(1,2)$, then s_3 is a sibling of s_1 and s_2, otherwise, s_3 is a sibling of their parent. In case of s_1 and s_2 not being siblings, then the values in the A matrix will detect s_3 as a sibling of either one of them, that is the third possibility. After detecting siblings, the matrix A is reduced by eliminating the siblings and replacing them by their parent. Therefore, for n species, the algorithm scans the matrix A, and at each step, the siblings are eliminated and replaced by their parent, this reduces the matrix A, until only one species is remaining, which is the root. □

To follow is an example to illustrate how the algorithm works.

Given a set of species, $S = \{s_1, s_2, s_3, s_4, s_5\}$, with the set of posets $\mathcal{P}$ in Figure 3.19.

The corresponding A matrix is shown in Table 3.8.

	s_1	s_2	s_3	s_4	s_5
s_1	-1	3	0	0	2
s_2	3	-1	0	0	2
s_3	0	0	-1	3	2
s_4	0	0	3	-1	2
s_5	2	2	2	2	-1

Table 3.8: Matrix A for the posets in Figure 3.19.

Therefore, the maximum is 3, with the siblings s_1 and s_2, as well as s_3 and s_4.

And, the matrix A becomes as shown in Table 3.9.

Now, s_5 is a sibling of both s_{12} and s_{34}, giving one root for the three nodes. The constructed tree is shown in Figure 3.20.

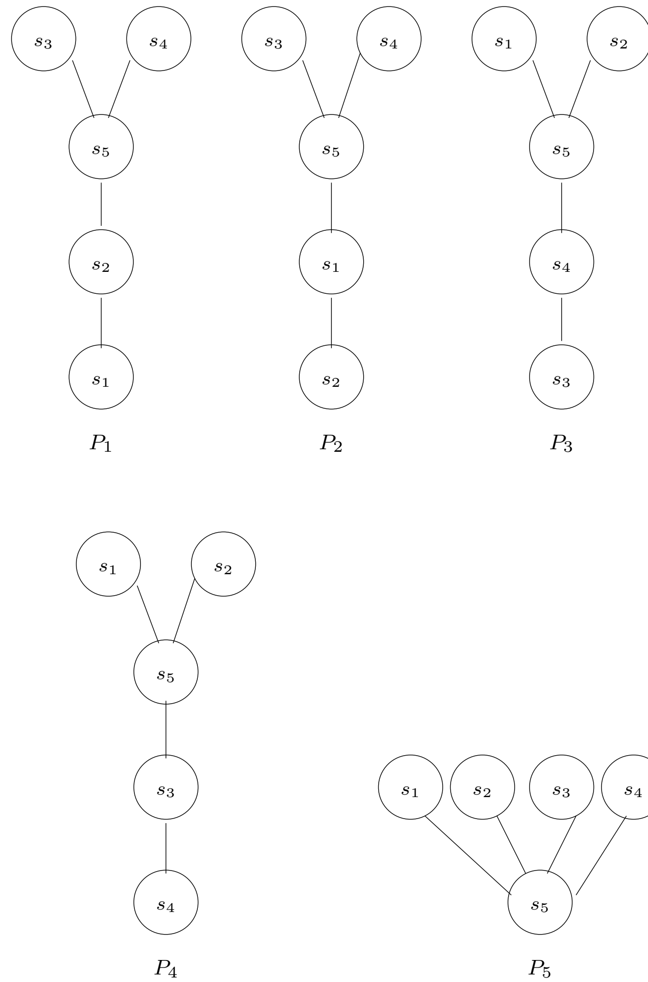


Figure 3.19: Set of posets $\mathcal{P}$ for the set of species $S = \{s_1, s_2, s_3, s_4, s_5\}$.

	s_{12}	s_{34}	s_5
s_{12}	-1	0	2
s_{34}	0	-1	2
s_5	2	2	-1

Table 3.9: Updated matrix A for the posets in Figure 3.19.

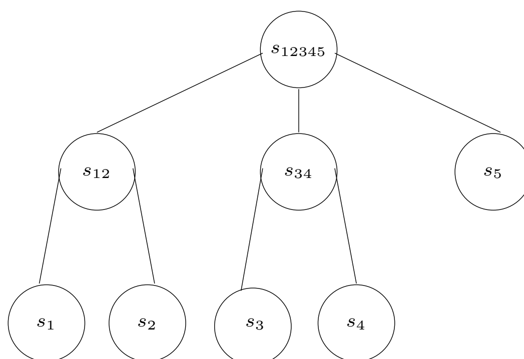


Figure 3.20: The tree corresponding to the set of posets $\mathcal{P}$ in Figure 3.19.

3.4.2 Generating a Set of Posets $\mathcal{P}$ From a Given S-tree

For each tree T, there exists a set of posets $\mathcal{P}$ compatible with T. In this section, we show how given a tree T, the set of compatible posets can be generated.

A set of posets $\mathcal{P}$ is compatible with an S-tree T if, for all distinct triples $x, y, z \in L(T)$ such that $\lambda(x) = s_i$, $\lambda(y) = s_j$, and $\lambda(z) = s_k$ and such that $s_j \leq_i s_k$, then we have the shortest path from either of x or y to z passes through MRCA(x, y). Therefore, the procedure of obtaining posets from a tree is straightforward. Given a tree T, it is clear which species are closer to each other than others, and hence, posets can be generated. By obtaining the path from each species (leaf node) to the root of the tree, and laying this path horizontally, we get the nodes sorted in order of closeness to this specific leaf node. Each node on the path represents a subtree, of which the leaves belonging to the species set represent one level of the poset. For example, given the tree shown in Figure 3.11, we look at each species to generate the corresponding poset. Starting with s_1, the poset P_1 automatically contains the ordered pairs (s_1, s_2), (s_1, s_3), (s_1, s_4), and (s_1, s_5). It is clear from the tree that s_2 is the closest sibling to s_1, this adds the ordered pairs (s_2, s_3), (s_2, s_4), and (s_2, s_5) to the poset P_1. Also, the ordered pairs (s_3, s_4) and (s_3, s_5) are added. In a similar manner the posets P_2, P_3, P_4, and P_5 are generated as shown in Figure 3.9.

An example to illustrate how posets are generated from a tree is shown in Figure 3.21. The tree on the right shows the path from s_1 to the root, where each node on the path is a root to a subtree, and the leaves belonging to each subtree represent a level of the poset P_1. The subtree with the root s_1 has only one leaf and that is s_1. The second level of the poset contains the leaves in the subtree with the root x, and that is only s_2, then comes the last level, in the subtree with the root r, and this subtree contains the leaves s_3 and s_4. Therefore, the poset P_1 is generated as follows. $P_1 = \{(s_1, s_2), (s_1, s_3), (s_1, s_4), (s_2, s_3), (s_2, s_4)\}$.

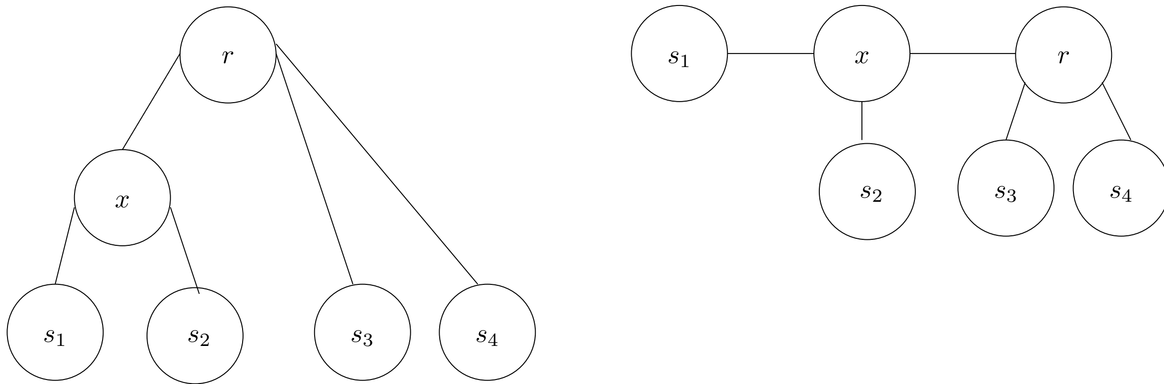


Figure 3.21: An example of how the poset corresponding to s_1 is generated.

The algorithm for generating the posets corresponding to a given tree is shown in Figure 3.22.

Theorem 20. *The algorithm GeneratePosets shown in Figure 3.22 generates the set of posets $\mathcal{P}$ that is compatible with a given tree T.*

Proof. Using a proof by construction, we show that the algorithm GeneratePosets generates the set of posets $\mathcal{P}$ compatible with a given tree T. From the definition of compatible in Section 3.2, we know that an s_i-poset $P_i = (S, \leq_i)$ is compatible with S-tree T if, for all distinct triples $x, y, z \in L(T)$ such that $\lambda(x) = s_i$, $\lambda(y) = s_j$, and $\lambda(z) = s_k$ and such that $s_j \leq_i s_k$, then we have the shortest path from either of x or y to z passes through MRCA (x, y). The algorithm GeneratePosets finds, for a species s_i, the path p from s_i to the root r, on that path, the nodes that come first on the path p are definitely closer to s_i and, hence, come at a lower level in the poset. That follows from the definition of compatible, which indicates that if $s_j \leq_i s_k$, then the shortest path from either of x or y to z passes through MRCA (x, y). Therefore, by scanning the path p, the set of posets $\mathcal{P}$ can be constructed. □

Theorem 21. *The algorithm GeneratePosets has a time complexity of $O(n^3)$.*

Proof. Let the number of species be n. The loop on line 2 iterates n times, and on line 3, finding the path from a certain species to the root is also linear in the number of species, this gives a complexity $O(n^2)$. Then on line 7, the while loop is also linear in n, and on line 9, finding all leaves in a subtree is linear as well. This gives a total complexity of $O(n^3)$. □

```
1  begin GeneratePosets(T, S)
2  for i ← 1 to n
3        do path ← path from s_i to root r
4            U ← S \ {s_i}
5            P_i ← {s_i} × U
6            q ← pointer to s_i
7            while q ≠ r
8                  do q ← next node on path
9                      L ← all leaves in the subtree rooted at q
10                     U ← U \ L
11                     P_i ← P_i ∪ (L × U)
12 return P_i
```

Figure 3.22: Algorithm to generate a set of posets $\mathcal{P}$ from an S-tree T.

3.4.3 Relating Posets to Trees

The following theorems relate posets and trees to one another.

Theorem 22. *Given a set of posets $\mathcal{P}$, if there exists an S-tree T that $\mathcal{P}$ is compatible with, then T can be used to generate the same set of posets $\mathcal{P}$.*

Proof. Given a set of posets $\mathcal{P}$, assume that $\mathcal{P}$ is compatible with a tree T. Assume that T, in turn, generates a different set of posets $\mathcal{P}'$. $\mathcal{P}'$ can now be used to construct a tree T' that is compatible with $\mathcal{P}'$, T' is expected to be equivalent to T. However, since $\mathcal{P}'$ and $\mathcal{P}$ are not equal, then the two trees constructed are also not the same. Since, T and T' are different, therefore, T and T' can yield contradictory 2-partitions, this means that that T and T' may be contradictory trees, and hence, one of them can not be used to give the same set of posets. Hence, there is a contradiction, and T can not be used to generate a set of posets other than $\mathcal{P}$. □

Theorem 23. *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two sets of posets that are compatible with the two S-trees, T_1 and T_2. Then T_1 and T_2 are contradictory if and only if there exists a poset $P_i \in \mathcal{P}_1$ and $P_j \in \mathcal{P}_2$, such that $P_i \in \mathcal{P}_1$ is inconsistent with $P_j \in \mathcal{P}_2$.*

Proof. First, we prove that if T_1 and T_2 are contradictory then there exists a poset $P_i \in \mathcal{P}_1$ and a poset $P_j \in \mathcal{P}_2$, such that $P_i \in \mathcal{P}_1$ is inconsistent with $P_j \in \mathcal{P}_2$. Using a proof by contradiction, assume that T_1

and T_2 are contradictory and there is no poset $P_i \in \mathcal{P}_1$ and $P_j \in \mathcal{P}_2$, such that $P_i \in \mathcal{P}_1$ is inconsistent with $P_j \in \mathcal{P}_2$. Since, T_1 and T_2 are contradictory, therefore, there exists an edge in T_1 and an edge in T_2, that when cut induces contradictory 2-partitions. This means that there exists four species s_1, s_2, s_3, and s_4, such that s_1 and s_2 belong to the same partition in one tree but not in the other. Similarly, s_3 and s_4 belong to the same partition in one tree but not in the other. Since, the set of posets $\mathcal{P}_1$ is compatible with T_1 and the set of posets $\mathcal{P}_2$ is compatible with T_2, and since T_1 and T_2 are contradictory, therefore, there exists a poset $P_i \in \mathcal{P}_1$ and a poset $P_j \in \mathcal{P}_2$ such that $P_i \in \mathcal{P}_1$ is inconsistent with $P_j \in \mathcal{P}_2$. This leads to a contradiction with the assumption.

The second part of the proof proves that if there exists a poset $P_i \in \mathcal{P}_1$ and a poset $P_j \in \mathcal{P}_2$, such that $P_i \in \mathcal{P}_1$ is inconsistent with $P_j \in \mathcal{P}_2$, then T_1 and T_2 are contradictory. Using a proof by contradiction, assume that there exists a poset $P_i \in \mathcal{P}_1$ and a poset $P_j \in \mathcal{P}_2$, such that $P_i \in \mathcal{P}_1$ is inconsistent with $P_j \in \mathcal{P}_2$ while T_1 and T_2 are non-contradictory. If $P_i \in \mathcal{P}_1$ is inconsistent with $P_j \in \mathcal{P}_2$, therefore, $\mathcal{P}_1$ is inconsistent with the set of posets $\mathcal{P}_2$, hence, the two sets of posets can create contradictory 2-partitions in their corresponding trees, and therefore, the trees that are compatible with both sets of posets can not be non-contradictory, and this leads to a contradiction with the assumption. Therefore, the theorem follows. $\square$

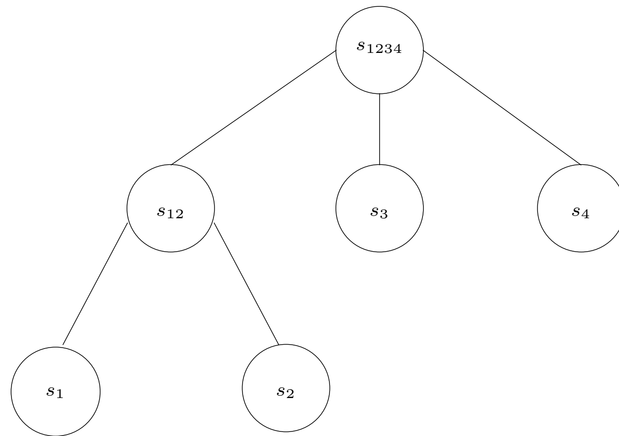
Figure 3.23 shows an example to illustrate Theorem 23. The set of posets $\mathcal{P}_1$ corresponding to the tree at the top consists of the following posets:

$$
\begin{aligned}
P_1 &= (s_1, s_2), (s_1, s_3), (s_1, s_4), (s_2, s_3), (s_2, s_4) \\
P_2 &= (s_2, s_1), (s_2, s_3), (s_2, s_4), (s_1, s_3), (s_1, s_4) \\
P_3 &= (s_3, s_1), (s_3, s_2), (s_3, s_4) \\
P_4 &= (s_4, s_1), (s_4, s_2), (s_4, s_3)
\end{aligned}
$$

And, the set of posets $\mathcal{P}_2$ corresponding to the tree at the bottom consists of the following posets:

$$
\begin{aligned}
P_1 &= (s_1, s_2), (s_1, s_3), (s_1, s_4), (s_3, s_2), (s_3, s_4) \\
P_2 &= (s_2, s_1), (s_2, s_3), (s_2, s_4) \\
P_3 &= (s_3, s_1), (s_3, s_2), (s_3, s_4), (s_1, s_3), (s_1, s_4) \\
P_4 &= (s_4, s_1), (s_4, s_2), (s_4, s_3)
\end{aligned}
$$

The poset $P_1 \in \mathcal{P}_1$ indicates that s_2 is a sibling of s_1, while the poset $P_1 \in \mathcal{P}_2$ indicates that s_3 is a sibling of s_1. Therefore, the two posets are inconsistent.



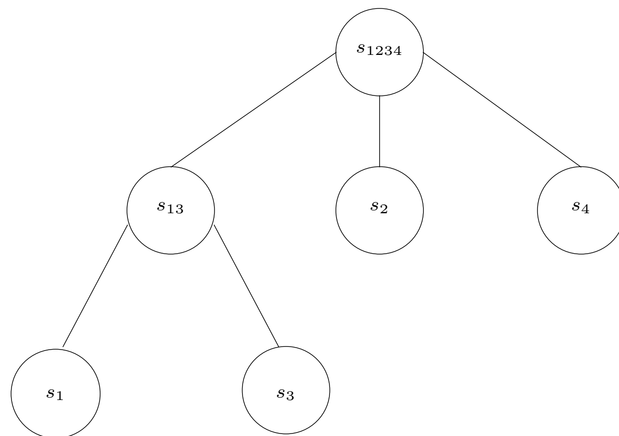


Figure 3.23: An example to illustrate Theorem 23.

3.5 Refinement of Trees

This section is about refinement and contradictory trees. We start with a basic result about refinement (Theorem 26).

Lemma 24. *Let T be an S-tree. Let Q be the 2-partition set of T. Then Q is not contradictory with itself.*

Proof. We show that every pair of 2-partitions in Q is non-contradictory. Consider an arbitrary pair of distinct edges of T. This pair of edges are the ends of a unique path in T. Let $u_0, u_1, \ldots, u_{k-1}, u_k$ be that path. Then the edges are (u_0, u_1) and (u_{k-1}, u_k). These edges partition S into three sets: X, the set of species reachable from u_0 without using (u_0, u_1); Y, the set of species reachable from u_k without using (u_{k-1}, u_k); and Z, the set of species reachable from $u_1, u_2, \ldots, u_{k-1}$ without using (u_0, u_1) or (u_{k-1}, u_k). The 2-partition corresponding to (u_0, u_1) is $(X, Y \cup Z)$, and the 2-partition corresponding to (u_{k-1}, u_k) is $(X \cup Z, Y)$. Recall the definition of contradictory 2-partitions: Two 2-partitions $X = (X_1, X_2)$ and $Y = (Y_1, Y_2)$ are contradictory partitions if there exist four species s_1, s_2, s_3, s_4 such that $s_1, s_2 \in X_1$, $s_3, s_4 \in X_2$, $s_1, s_3 \in Y_1$, and $s_2, s_4 \in Y_2$. Let $s_1, s_2, s_3, s_4 \in S$. If $s_1, s_2 \in X$ and $s_3, s_4 \in Y \cup Z$, then $s_1, s_2 \in X \cup Z$, so the definition definitely does not apply to the 2-partitions corresponding to (u_0, u_1) and (u_{k-1}, u_k). Since the two edges were arbitrary, we conclude that Q is not contradictory with itself. □

Lemma 25. *Let T_1 be an S-tree, and let T_2 be a refinement of T_1. Let Q_1 be the 2-partition set of T_1, and let Q_2 be the 2-partition set of T_2. Then $Q_1 \subseteq Q_2$.*

Proof. Since a refinement step adds one edge to the tree, it adds one 2-partition to the set of 2-partitions. By induction on the number of refinement steps going from T_1 to T_2, we get that $Q_1 \subseteq Q_2$. □

Theorem 26. *If S-tree T_2 can be obtained from S-tree T_1 using a number of refinement steps, then T_1 and T_2 are non-contradictory.*

Proof. Let T_1 be an S-tree, and let T_2 be a refinement of T_1. Let Q_1 be the set of 2-partitions of T_1, and let Q_2 be the set of 2-partitions of T_2. By Lemma 25, $Q_1 \subseteq Q_2$. By Lemma 24, Q_2 is not contradictory with itself. Then Q_1 and Q_2 are non-contradictory, since otherwise Q_2 would be contradictory with itself. By definition, T_1 and T_2 are non-contradictory. □

Given two S-trees, T_1 and T_2, if using a number of refinement steps both trees can be refined into a third S-tree T_3, then it is guaranteed that both trees carry non-contradictory information. For example, the two S-trees, T_1 and T_2 shown in Figure 3.24 are non-contradictory and they are both refined into T_3. In this

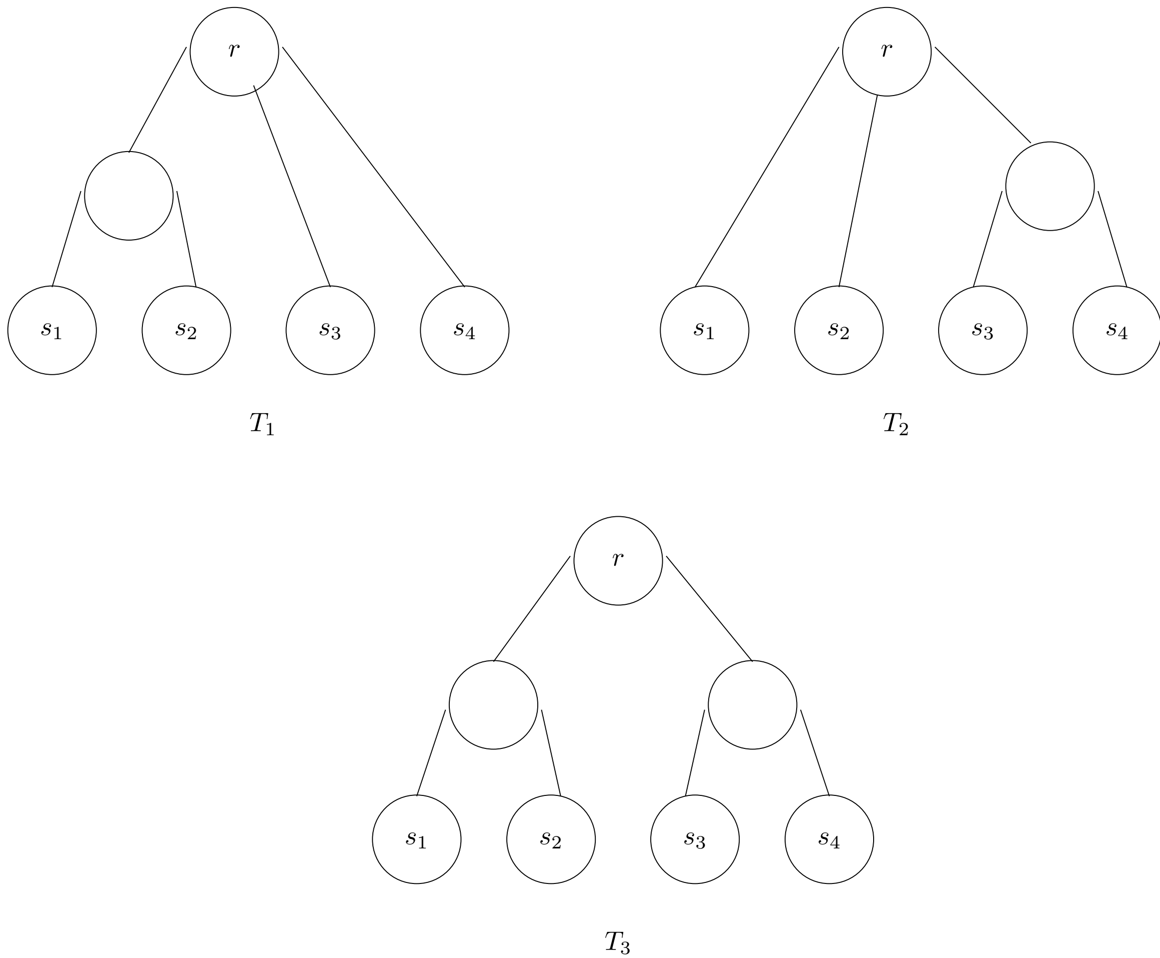


Figure 3.24: Refinement of T_1 and T_2 into T_3.

example, T_3 is obtained using the minimum number of refinement steps, hence, T_3 is the minimum common refinement of T_1 and T_2.

The posets given for each gene are used in the construction of one tree corresponding to each gene. The trees constructed can contain contradictory information, as the example shown in Figure 3.25. This contradiction is due to different information encoded in each tree, and the information is not only different, but it is contradictory, as defined in Section 3.2. To be able to identify HGT events, contradictory trees must be identified. This can be done by examining the number of ways leaves and the root in a tree can be partitioned. This is done by examining the cuts in edges that are not incident to leaf nodes. If two trees are contradictory, then there is evidence for HGT.

The *minimum common refinement* of two non-contradictory S-trees T_1 and T_2 is an S-tree T_3 that is a

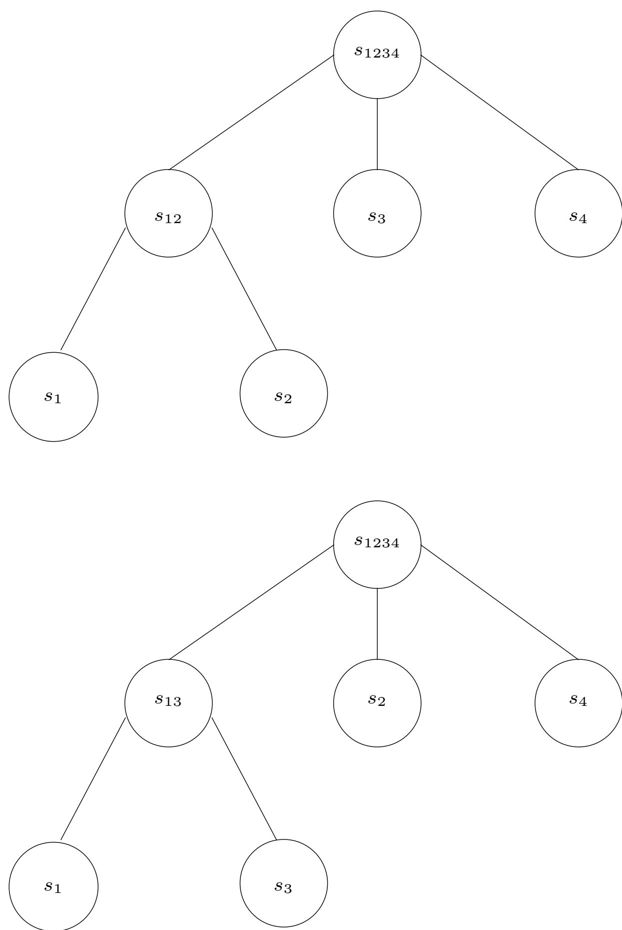


Figure 3.25: Two contradicting trees.

common refinement of T_1 and T_2 such that any other common refinement of T_1 and T_2 is a refinement of T_3.

Theorem 27. *Let T_1 and T_2 be S-trees that are non-contradictory. Let Q_1 and Q_2 be their respective sets of 2-partitions. Then there exists a unique tree T_3 that is their minimum common refinement. Furthermore, if Q_3 is the set of 2-partitions of T_3, then $Q_3 = Q_1 \cup Q_2$.*

Proof. Given two trees T_1 and T_2 that are non-contradictory, and their respective sets of 2-partitions, Q_1 and Q_2, $Q_3 = Q_1 \cup Q_2$. Therefore, Q_3 contains 2-partitions, where each 2-partition is obtained by cutting one edge of the tree T_3. Hence, the set Q_3 can be used to construct the tree T_3. This is done by checking each 2-partition, starting by the 2-partition with the minimum cardinality, siblings in T_3 are inferred and the set is reduced. This process is repeated until only 2-partitions with one of its elements having cardinality one are remaining. Since, $Q_3 = Q_1 \cup Q_2$, and since Q_1 already corresponds to a tree and also Q_2 corresponds

to a tree, therefore, all the 2-partitions in Q_1 and Q_2 already correspond to edges in a tree, therefore, using the two sets, a more refined tree can be constructed. Since, Q_1 and Q_2 both contain non-contradictory partitions, and since $Q_3 = Q_1 \cup Q_2$, therefore, Q_3 also contains non-contradictory partitions, and hence, there exists a tree T_3 that corresponds to Q_3. Using induction, we start by Q_1 and T_1 and add 2-partitions from Q_2 to Q_1. Let k be the number of 2-partitions added. If $k = 1$, then a 2-partition is added from Q_2 to Q_1. Since, T_1 and T_2 are non-contradictory, therefore, a 2-partition that exists in Q_2 but not in Q_1 only adds an internal node and an edge to T_1. Therefore, T_1 becomes a more refined tree. Hence, adding k 2-partitions to T_1 will further refine T_1 by adding more edges and internal nodes. Therefore, given Q_3, a set of non-contradictory 2-partitions, a tree T_3 can be constructed. □

Figure 3.26 shows an example to illustrate Theorem 27, where the tree T_3 is the minimum common refinement of the two trees T_1 and T_2, where T_3 is obtained using one refinement step, this refinement step is performed on T_1 by adding a parent for s_3 and s_4. The refined tree is the same tree as T_2.

An algorithm for finding the minimum common refinement of two given trees is shown in Figure 3.29. The algorithm finds all possible 2-partitions for the given two trees. A 2-partition is found by cutting one edge of the tree and finding the leaves in the two subtrees induced by cutting that edge. For example, cutting an edge (i, j), induces two subtrees, one with the root i and the other with the root j. Performing a depth-first search on the two subtrees finds the leaves in both subtrees. The species set for each subtree composes one of the 2-partitions, therefore, $S(i)$ composes one partition, and $S(j)$ composes the other. The subroutine FindTwoPartitions shown in Figure 3.30 finds the 2-partition set for a given tree. When the 2-partitions sets are found for both trees, a union is performed on these sets to obtain the minimum common refinement tree. Figure 3.27 shows an example to illustrate the algorithm. The node s_0 is added under the root to avoid having equivalent sets for a 2-partition, as these equivalent sets disappear when performing the union operation. In the example, T_1 has eight edges, including the edge connecting the s_0 to the root. Hence, there are eight 2-partitions sets for T_1. Similarly, their are eight 2-partitions sets for T_2. The 2-partitions

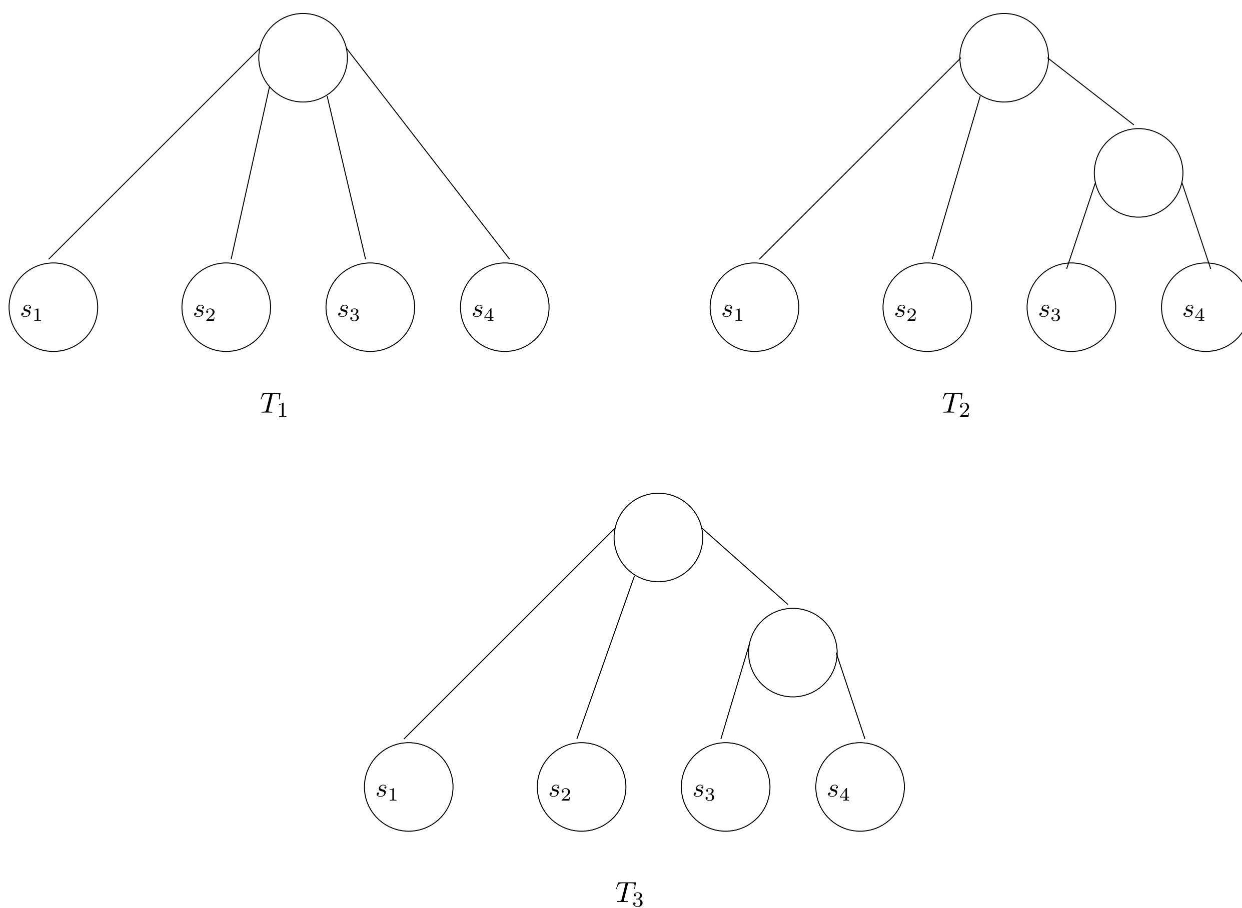


Figure 3.26: T_3 is the minimum common refinement of T_1 and T_2.

sets for T_1 are as follows:

$$
\begin{aligned}
Q_1 &= \{s_0\}, \{s_1, s_2, s_3, s_4, s_5\} \\
Q_2 &= \{s_1\}, \{s_0, s_2, s_3, s_4, s_5\} \\
Q_3 &= \{s_2\}, \{s_0, s_1, s_3, s_4, s_5\} \\
Q_4 &= \{s_3\}, \{s_0, s_1, s_2, s_4, s_5\} \\
Q_5 &= \{s_4\}, \{s_0, s_1, s_2, s_3, s_5\} \\
Q_6 &= \{s_5\}, \{s_0, s_1, s_2, s_3, s_4\} \\
Q_7 &= \{s_1, s_2, s_3\}, \{s_0, s_4, s_5\} \\
Q_8 &= \{s_4, s_5\}, \{s_0, s_1, s_2, s_3\}
\end{aligned}
$$

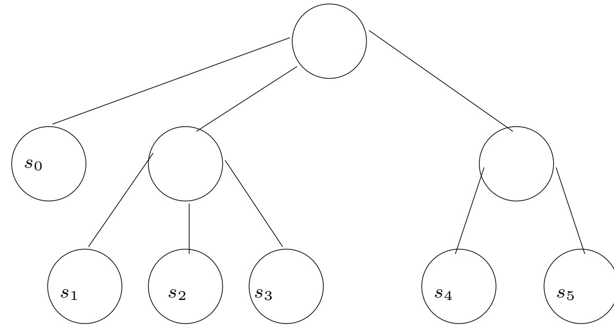
The 2-partitions sets for T_2 are as follows:

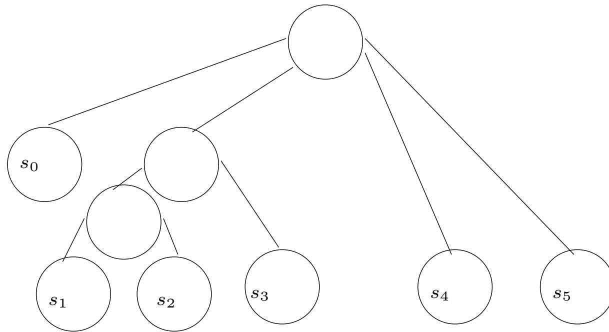
$$
\begin{aligned}
Q_1 &= \{s_0\}, \{s_1, s_2, s_3, s_4, s_5\} \\
Q_2 &= \{s_1\}, \{s_0, s_2, s_3, s_4, s_5\} \\
Q_3 &= \{s_2\}, \{s_0, s_1, s_3, s_4, s_5\} \\
Q_4 &= \{s_3\}, \{s_0, s_1, s_2, s_4, s_5\} \\
Q_5 &= \{s_4\}, \{s_0, s_1, s_2, s_3, s_5\} \\
Q_6 &= \{s_5\}, \{s_0, s_1, s_2, s_3, s_4\} \\
Q_7 &= \{s_1, s_2\}, \{s_0, s_3, s_4, s_5\} \\
Q_8 &= \{s_1, s_2, s_3\}, \{s_0, s_4, s_5\}
\end{aligned}
$$

The union of the two sets of partitions gives the following 2-partitions sets, which are the sets that give the tree T_3:

$$
\begin{aligned}
Q_1 &= \{s_0\}, \{s_1, s_2, s_3, s_4, s_5\} \\
Q_2 &= \{s_1\}, \{s_0, s_2, s_3, s_4, s_5\} \\
Q_3 &= \{s_2\}, \{s_0, s_1, s_3, s_4, s_5\} \\
Q_4 &= \{s_3\}, \{s_0, s_1, s_2, s_4, s_5\} \\
Q_5 &= \{s_4\}, \{s_0, s_1, s_2, s_3, s_5\} \\
Q_6 &= \{s_5\}, \{s_0, s_1, s_2, s_3, s_4\} \\
Q_7 &= \{s_1, s_2\}, \{s_0, s_3, s_4, s_5\} \\
Q_8 &= \{s_1, s_2, s_3\}, \{s_0, s_4, s_5\} \\
Q_9 &= \{s_0, s_1, s_2, s_3\}, \{s_4, s_5\}
\end{aligned}
$$

The algorithm that constructs a tree from its two-partition set is shown in Figure 3.31. Lets consider the





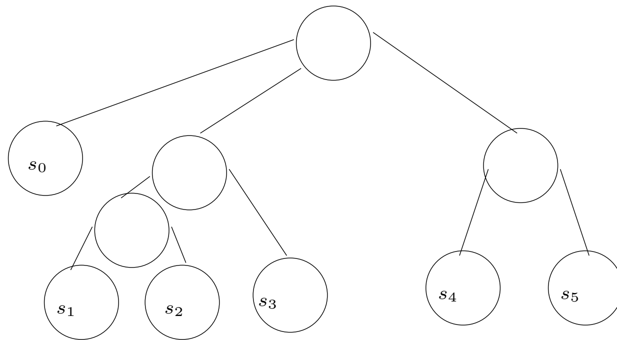


Figure 3.27: An example to illustrate the algorithm MinCommonRefine.

following two-partition set, Q, to illustrate the algorithm.

$$
\begin{aligned}
Q_1 &= \{s_0\}, \{s_1, s_2, s_3, s_4, s_5\} \\
Q_2 &= \{s_1\}, \{s_0, s_2, s_3, s_4, s_5\} \\
Q_3 &= \{s_2\}, \{s_0, s_1, s_3, s_4, s_5\} \\
Q_4 &= \{s_3\}, \{s_0, s_1, s_2, s_4, s_5\} \\
Q_5 &= \{s_4\}, \{s_0, s_1, s_2, s_3, s_5\} \\
Q_6 &= \{s_5\}, \{s_0, s_1, s_2, s_3, s_4\} \\
Q_7 &= \{s_1, s_2\}, \{s_0, s_3, s_4, s_5\} \\
Q_8 &= \{s_1, s_2, s_3\}, \{s_0, s_4, s_5\} \\
Q_9 &= \{s_0, s_1, s_2, s_3\}, \{s_4, s_5\}
\end{aligned}
$$

The algorithm starts by removing all sets with cardinality 1. So the set Q is reduced to the following:

$$
\begin{aligned}
Q_7 &= \{s_1, s_2\}, \{s_0, s_3, s_4, s_5\} \\
Q_8 &= \{s_1, s_2, s_3\}, \{s_0, s_4, s_5\} \\
Q_9 &= \{s_0, s_1, s_2, s_3\}, \{s_4, s_5\}
\end{aligned}
$$

The set with the minimum cardinality is in Q_7, therefore, the species s_1 and s_2 are detected as siblings and they are replaced by a parent node in all sets. Therefore, Q is modified to the following:

$$
\begin{aligned}
Q_8 &= \{u_1, s_3\}, \{s_0, s_4, s_5\} \\
Q_9 &= \{s_0, u_1, s_3\}, \{s_4, s_5\}
\end{aligned}
$$

The next step finds the minimum cardinality in both Q_8 and Q_9, where u_1 and s_3 are siblings, and s_4 and s_5 are siblings. When Q_8 and Q_9 are removed from Q, it becomes empty and the root connects the subtrees constructed. Figure 3.28 shows the tree constructed from the two-partition set Q.

Theorem 28. *The time complexity of the algorithm MinCommonRefine is $O(mn + n^2)$.*

Proof. Let n be the number of species. Let m be the number of edges in a tree T. The subroutine FindTwoPartitions on Lines 3 and 4 is $O(mn)$ Line 5 performs a union operation linear in the number of species. Line 6 constructs the tree from its two-partition set, ConstructTree2Partitions is $O(n^2)$. Therefore, the overall complexity of the algorithm MinCommonRefine is $O(mn + n^2)$. □

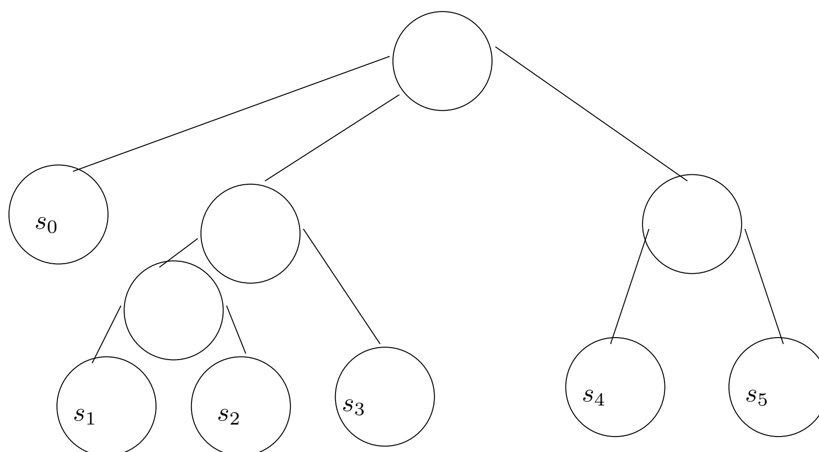


Figure 3.28: An example to illustrate the algorithm ConstructTree2Partitions.

1 **begin** MinCommonRefine(T_1, T_2)
2 Add node s_0 as a child of the root in T_1 and T_2
3 $Q_1 \leftarrow$ FindTwoPartitions(T_1)
4 $Q_2 \leftarrow$ FindTwoPartitions(T_2)
5 $Q_3 \leftarrow Q_1 \cup Q_2$
6 $T_3 \leftarrow$ ConstructTree2Partitions(Q_3)
7 **return** T_3

Figure 3.29: Algorithm to find the minimum common refinement of two trees.

3.6 Inferring HGT from Posets

In this section, we show how posets and trees are used to infer HGT.

The problem is defined as follows:

Inferring HGT From Posets

INSTANCE: Set $S = \{s_1, s_2, \ldots, s_n\}$ of n taxa; set $G = \{g_1, g_2, \ldots, g_m\}$ of m genes; mn individual posets $P_{ij} = (S, <_{ij})$, for $1 \leq i \leq m$ and $1 \leq j \leq n$.

SOLUTION: Sets of genes corresponding to contradictory trees.

A number of steps are followed to be able to infer HGT events. First, trees are constructed from posets, then the different trees are compared, where contradictory trees are identified. Trees that are contradictory

```
1  begin FindTwoPartitions(T)
2  Q ← {} ▷ This is the set of all 2-partitions.
3  partitionscount ← 1
4  for all edges e_ij ∈ T
5      do cut e_ij
6          speciesset_i ← all leaves in subtree with root i
7          speciesset_j ← all leaves in subtree with root j
8          Q_partitionscount ← {speciesset_i}, {speciesset_j}
9          Q = Q ∪ P_partitionscount
10         partitionscount = partitionscount + 1
11 return Q
```

Figure 3.30: Algorithm to find the 2-partitions set of a given tree.

with the majority of trees suggest HGT. Other events such as gene duplication, gene loss, and incomplete lineage sorting can cause the incongruence of trees [81]. In Section 3.4, we showed how trees are constructed from posets, in what follows, we show how contradictory trees are detected.

Two trees, T_1 and T_2 are said to be contradictory if there exists an edge e_1 in T_1 and an edge e_2 in T_2, that when cut creates contradictory 2-partitions. The algorithm DetectContradiction shown in Figure 3.32 takes two trees as an input and detects whether they are contradictory or not.

The process of identifying which genes are candidates of HGT proceeds as follows. Each two trees are tested for contradiction, for example, we start by T_1 and T_2, if they are contradictory, then they belong to two different sets, if not then they are placed in one set. And, the process continues, if the next tree to be tested is T_3, then it is compared with one tree from each set to test to which set the tree T_3 belongs. It is expected that the majority of the trees will be non-contradictory, with some trees contradicting this majority, so there will be one set with a higher cardinality, therefore, the other sets, which are the minority, are considered candidates of HGT, in other words, the genes corresponding to the contradictory trees are considered candidates of HGT.

The algorithm performs ideally when all the trees are completely refined (binary) trees, where the trees that are not identical are considered contradictory. It is expected that most of the trees will be non-contradictory, leading to a minority of contradictory trees that are expected to belong to genes that are candidates of HGT. In what follows, some real life HGT examples are shown to support the argument that the genes involved in HGT are a minority, and that there will always be a dominant tree. In [67], it is indicated that only 0.5% of

1 **begin** ConstructTree2Partitions(Q)
2 Let each $Q_i \in Q = \{X\}, \{Y\}$
3 **for all** $Q_i \in Q$
4   **do if** $|X| = 1$ or $|Y| = 1$
5   **then** $Q = Q - Q_i$
6 **while** Q not empty
7   **do** min $\leftarrow MaxInt$
8   **for all** $Q_i \in Q$
9   **do if** $|X| <$ min
10   **then** min $\leftarrow |X|$
11   index $\leftarrow i$
12   **else if** $|Y| <$ min
13   **then** min $\leftarrow |Y|$
14   index $\leftarrow i$
15   **for** Q_{index}
16   **do if** $|X| < |Y|$
17   **then** add $\{s_{x_1}, s_{x_2}, \ldots, s_{x_n}\} \in X$ as siblings in T
18   **for all** $Q_i \neq Q_{\text{index}} \in Q$
19   **do** replace $\{s_{x_1}, s_{x_2}, \ldots, s_{x_n}\}$ with their parent u_{index}
20   **else if** $|Y| < |X|$
21   **then** add $\{s_{y_1}, s_{y_2}, \ldots, s_{y_n}\} \in Y$ as siblings in T
22   **for all** $Q_i \neq Q_{\text{index}} \in Q$
23   **do** replace $\{s_{y_1}, s_{y_2}, \ldots, s_{y_n}\}$ with their parent u_{index}
24     $Q = Q - Q_{\text{index}}$
25 Add a root as a parent to all subtrees constructed.

Figure 3.31: Algorithm to construct a tree from its 2-partitions set.

```
1  begin DetectContradiction(T_1, T_2)
2  contradiction ← false
3  Q_1 ← FindTwoPartitions(T_1)
4  Q_2 ← FindTwoPartitions(T_2)
5  Let each 2-partition in Q_1 equal X_1, Y_1
6  Let each 2-partition in Q_2 equal X_2, Y_2
7  for each 2-partition u ∈ Q_1
8      do for each 2-partition v ∈ Q_2
9          do if  s_i, s_j ∈ X_1, s_k, s_l ∈ Y1 AND s_i, s_k ∈ X_2, s_j, s_l ∈ Y2
10             then contradiction ← true
11 return contradiction
```

Figure 3.32: Algorithm to detect contradiction between two trees.

all human genes were copied into the genome from bacteria by HGT. Rujan and Martin in [72] analyzed how many genes in *Arabidopsis* come from cyanobacteria, They used a sample of 3961 *Arabidopsis* nuclear protein-coding genes and compared those with the complete set of proteins from yeast and 17 reference prokaryotic genomes, including one cyanobacterium. In their analysis of 386 phylogenetic trees, they found that the number of genes horizontally transferred to *Arabidopsis* from *cyanobacteria* falls between approximately 400 genes and approximately 2200 genes. That is between 1.6% and 9.2% of nuclear genes.

The algorithm InferHGT is shown in Figure 3.33. The input to the algorithm is a set of trees $T = \{T_1, T_2, \ldots, T_n\}$, where n is the number of trees and also the number of genes. The number of species is m.

Theorem 29. *The algorithm InferHGT shown in Figure 3.33 has complexity* $max(O(n^2), O(m^2 n))$.

Proof. The two nested loops on lines 4 and 5 are $O(n^2)$, where n is the number of trees. The subroutine DetectContradiction on line 6 is $O(m^2 n)$, where m is the number of edges in a tree. □

Theorem 30. *Algorithm InferHGT has space complexity* $O(mn)$.

Proof. The algorithm InferHGT shown in Figure 3.33 stores n trees, one for each gene. For each tree, the upper bound on the number of nodes in the tree is $2m$, where m is the number of species. Therefore, the upper bound on the number of edges is $2m-1$. Therefore, the space complexity of InferHGT is $O(n(2m+2m-1))$, that is $O(mn)$. □

```
1  begin InferHGT(T)
2  Let Set = Set_1, Set_2, ..., Set_n
3  Set_1, Set_2, ..., Set_n ← T_1, T_2, ..., T_n
4  for i = 1 to n
5      do for j = 1 to n
6          do contradiction ← DetectContradiction(T_i, T_j)
7              if contradiction = 0
8              then Set_i = Set_i ∪ Set_j
9                     Set = Set − Set_j
10 return Set
```

Figure 3.33: Algorithm to infer HGT.

An example to illustrate the algorithm for inferring HGT is shown in Figure 3.34, where the trees T_1, T_2, and T_3 are non-contradictory, while the tree T_4 contradicts the three trees. In T_4 there is a 2-partition that places the two species $\{s_1, s_3\}$ in one partition, and $\{s_2, s_4\}$ in another partition. This 2-partition contradicts the other three trees. Therefore, the gene corresponding to T_4 is a candidate of HGT, where a horizontal transfer occurred between s_1 and s_3, or s_2 and s_4. The network in Figure 3.35 shows the possible horizontal transfers.

Figure 3.36 shows the complete steps for detecting HGT events.

3.7 Comparing the Presented Technique to Others

We first start by the categorization given in Section 3.3. According to [13], there are five categories of HGT detection methods. These methods are compositional methods, methods that use phyletic patterns, methods that use substitution rates, phylogenetic incongruence methods, and methods that scan for polymorphic sites. For details on these methods please refer to Section 3.3. Our method goes into the category of phylogenetic incongruence methods. Methods in this category overcome the drawback of working with the sequences, and the fact that genes adopt to new genomes making the detection of older transfers harder. It also avoids the errors that occur in statistical methods due to sampling errors.

In what follows, we compare our technique to other phylogenetic incongruence methods.

The first method to consider is the conditioned reconstruction method [41]. This technique uses a condi-

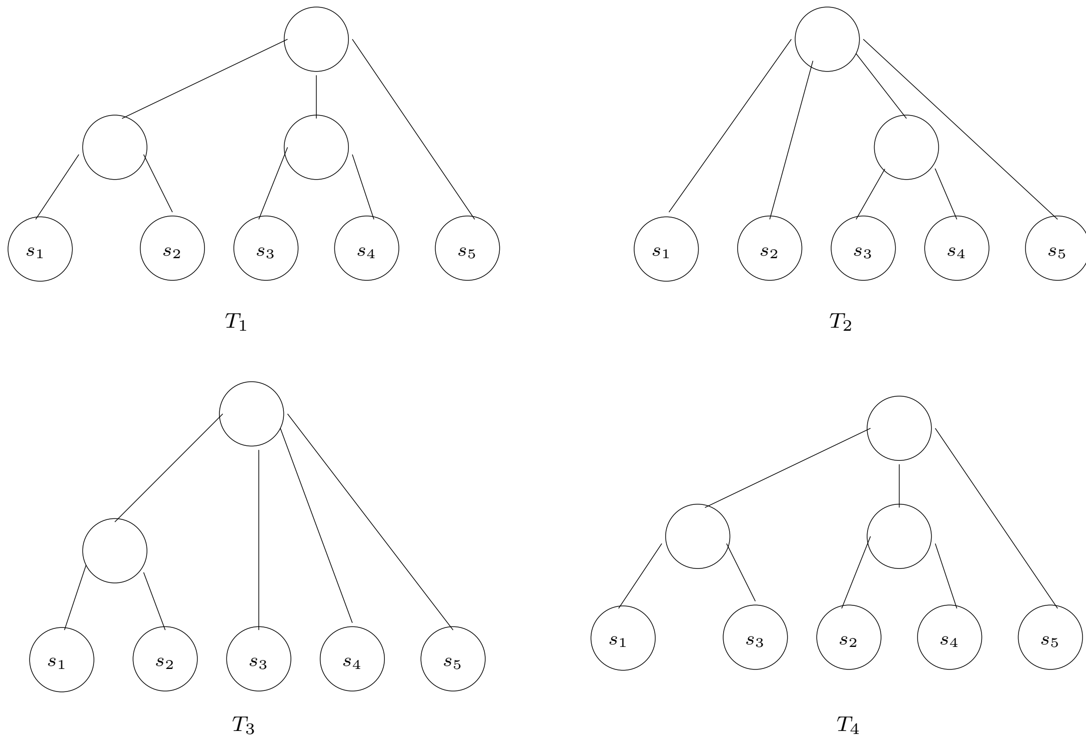


Figure 3.34: An example to illustrate the algorithm InferHGT.

tioning genome to define the set of genes to consider, however, in our approach this is avoided. Both our technique and CR make use of information about all genes in a genome. In analyzing CR, there is no evidence of its ability to detect horizontal transfers, it is more of a fusion detection method. Moreover, CR uses the coloring of fibers to compare the genomes under study to the conditioning genome, and in our technique, we use partial order sets to construct trees that are then compared to identify candidate horizontal transfers.

Nakhleh et al. [53] describe how to construct phylogenetic networks. They use inferred gene trees as an input. Their first method is polynomial time, however, they use only two gene trees in their comparison, allowing multiple reticulation events. Another method they present allows for errors in the trees, however, they allow for only one reticulation event. The method they use depends on comparing the trees and finding a common refinement between those trees, which is similar to our approach, however, in their future work, they stated that it would be best if one could find a common refinement for all trees, which we present in this work. Also, in our approach we do not need the input trees, it is enough to have the partial order sets to infer the HGT events, this overcomes the disadvantage of having problems in the constructed gene trees.

Macleod et al. [48] present a method for comparing a gene tree to a reference tree to detect horizontal gene transfers. Their approach makes use of consolidation and rearrangement, where consolidation involves

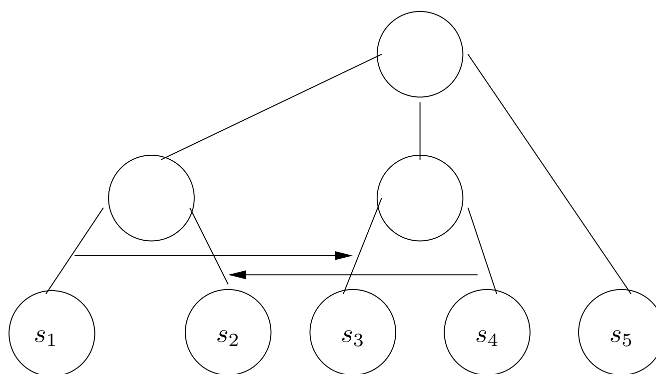


Figure 3.35: The possible HGT events for the example in Figure 3.34.

the collapsing of topologically identical features in the two trees to simplify the trees, then the leaves are rearranged until the two trees converge to identity. Rearrangements that make the two trees more similar are believed to be the undo of a horizontal gene transfer. This method relies on a reference tree, which we avoid in our approach.

Birin et al. [9] present a method for detecting rearrangements and horizontal gene transfers. Their method takes unaligned sequences as input and uses an alignment-free Average Common Substring (ACS) measure and pairwise alignment as distance measures between pairs of sequences. Birin et al. base their methods on the following assumptions:

- In phylogenetic networks each nucleotide evolves according to a tree which may be different from the organismal tree.
- Closely positioned nucleotides are more likely to have evolved according to the same tree than distantly positioned nucleotides.

The method they present infers different trees for different subsets of sequences and partitions the genomes into subsequences with the constraint that the nucleotides belonging to the same subsequence must have the same evolution. Given an organismal tree and a set of sequences, their method reconstructs the evolutionary history of the sequences by adding reticulation edges to the organismal tree, the reticulation edges that minimize the score they define as the minimum evolution criteria are the edges that are considered best to explain the evolutionary history of the input sequences. In our method, the input is just the posets, we do not need the whole sequences, and we do not need an input organismal tree that we base our results on. Moreover, we base our results on comparing all trees of genes together to obtain the ones that have different histories.

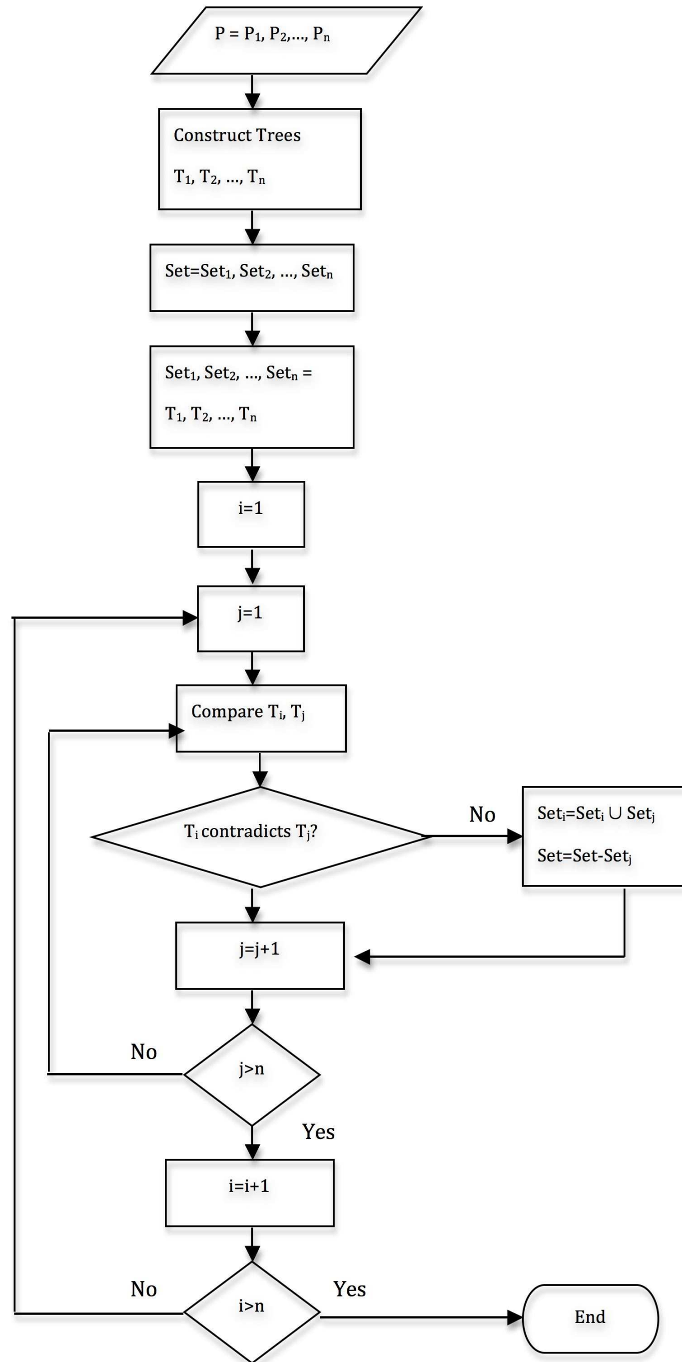


Figure 3.36: Flowchart for Detecting HGT

Abby et al. [1] present a phylogenetic detection method of lateral gene transfers. Their method relies on differences between a gene tree, and a reference species tree, where both trees are the input to their method. They assume that topological differences between the two trees can result from horizontal transfers or stochastic effects in the process of gene tree reconstruction, and they use an agreement function to decide whether the topological differences among trees are significant or not. If a species tree and a gene tree disagree, the method presented decomposes them into a maximal statistical agreement forest (MSAF). This forest is obtained as follows. Given a set of species S, the method decomposes S into a minimum number of subsets $S_1, ..., S_k$ such that for each subset S_i, the two trees, the species tree and the gene tree, agree, and are disjoint. Only one subset of $S_1, ..., S_k$ contains the root of the species tree. This is the non-transferred root as it is the most ancient node and can not acquire a gene from one of its descendants. The other subsets are considered horizontal gene transfers that occurred in the last common ancestor of S_i. The difference of this method from ours is that we use the posets to infer gene trees, and we do not require any input trees.

Chapter 4

Conclusions

This dissertation has presented two problems, namely whole genome alignment and inferring horizontal gene transfers using partial order sets. The two problems have been presented from a theoretical point of view.

In Chapter 2, a graph-based model was presented to represent different evolutionary events, and it was shown how each evolutionary event affects an alignment graph. The whole genome alignment problem was also defined, along with an NP-completeness proof for the general problem. A special class of sequences has been defined, that is breakable arrangements, with algorithms for handling such sequences. Also, a dynamic programming algorithm that has been proven optimal for breakable arrangements was presented and analyzed in the chapter. The chapter also presented a greedy algorithm, and that is another heuristic that solves the general whole genome alignment problem, however, in the greedy approach, reversals and duplications can be handled.

Chapter 3 introduced the problem of inferring horizontal gene transfers using partial order sets, where there is one poset for each gene per species. These posets have been used to construct S-trees for the genes corresponding to these posets, one tree is constructed for each gene. These trees are then compared, where the trees that contradict the majority of trees correspond to genes that are candidates for HGT. An algorithm for identifying contradiction is presented and then used in the algorithm to infer HGT. In Chapter 3, the concept of refinement is also presented, where it can also be used to identify contradiction among trees. An algorithm for finding a minimum common refinement for two trees is also presented. This algorithm finds the union of the 2-partition sets of two trees, and then uses this set to construct a third tree, which is their minimum common refinement. In the method presented, it is only necessary to know the relative distances rather than exact values, and the output of the algorithm is a collection of trees rather than a network,

and hence, there is no need to work with networks, but rather, the trees obtained are compared to infer horizontal gene transfers.

The graph-based model presented in Chapter 2 shows promising results for identifying many evolutionary events. The model can be generalized to allow the alignment of multiple sequences. Other algorithms can be devised to obtain other alignments and make it possible to use the alignment graph for better alignments. For example, a general algorithm for allowing all possible events. There also need to be heuristics that are not as costly as the dynamic programming and greedy algorithms. Also, practical analysis of the presented scoring mechanism and a comparison to other scoring mechanisms can help in obtaining more accurate alignments. Moreover, practical weights can be obtained and used in the presented scoring function. One way to estimate those weights is by analyzing real life alignments to find the frequency of different events. The presented dynamic programming alignment algorithm has been implemented, however, no empirical results have been obtained in this thesis; this gives space for future experiments on the methods described in this dissertation.

Regarding Chapter 3, the algorithms are well described and ready to be taken to the practical side for further investigation, to be studied from the practical point of view. We suggest using a programming language that supports sets in order to be able to implement the presented algorithms. Other points can be further studied in this problem. For example, more effort could be done to find solutions to the problem of incorrect or missing data in the input posets. This will be incredibly challenging, but, from a practical viewpoint, it would be most valuable. Another point is to develop algorithms that use the refinement of trees for identifying contradictory trees, where two contradictory trees do not have a common refinement. Faster algorithms could be developed to address this problem. One direction to also pursue is parallel programming, where each gene can be processed in parallel with the rest of the genes, then the comparison can be done. The input to the algorithm that infers horizontal gene transfers is a set of trees, one tree for each gene, those trees are constructed from posets in an earlier step, therefore, this step can be performed using parallel programming, where all the trees are constructed in parallel.

REFERENCES

[1] Sophie Abby, Eric Tannier, Manolo Gouy, and Vincent Daubin. Detecting lateral gene transfers by statistical reconciliation of phylogenetic forests. *BMC Bioinformatics*, 11(324):1–13, 2010.

[2] Mohamed Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. CoCoNUT: An efficient system for the comparison and analysis of genomes. *BMC Bioinformatics*, 9(476):17 pages, 2008.

[3] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Molecular Biology*, 215:403–410, 1990.

[4] Samuel V. Angiuoli and Steven L. Salzberg. Mugsy: Fast multiple alignment of closely related whole genomes. *Bioinformatics*, 27(3):334–342, 2011.

[5] C. Donovan Bailey, Mathew G. Fain, and Peter Houde. On conditioned reconstruction, gene content data, and the recovery of fusion genomes. *Molecular Phylogenetic and Evolution*, 39:263–270, 2006.

[6] Eric Bapteste and David A. Walsh. Does the ring of life ring true? *Trends in Microbiology*, 13(6):256–261, 2005.

[7] Nahla A. Belal and Lenwood S. Heath. Inferring horizontal gene transfers from posets. In *2nd International Conference on Computer Technology and Development, ICCTD 2010*, pages 32–36, 2010.

[8] Nahla A. Belal and Lenwood S. Heath. A theoretical model for whole genome alignment. *Journal of Computational Biology*, 18:1–24, 2011.

[9] Hadas Birin, Zohar Gal Or, Isaac Elias, and Tamir Tuller. Inferring horizontal transfers in the presence of rearrangements by the minimum evolution criterion. *Bioinformatics*, 24(6):826–832, 2008.

[10] Mathieu Blanchette. Computation and analysis of genomic multi-sequence alignments. *The Annual Review of Genomics and Human Genetics*, 8:193–213, 2007.

[11] Mathieu Blanchette, W. James Kent, Cathy Riemer, Laura Elnitski, Arian F.A. Smit, Krishna M. Roskin, Robert Baertsch, Kate Rosenbloom, Hiram Clawson, Eric D. Green, David Haussler, and Webb Miller. Aligning multiple genomic sequences with the threaded blockset aligner. *Genome Research*, 14:708–715, 2004.

[12] Tine Blomme, Klaas Vandepoele, Stefanie De Bodt, Cedric Simillion, Steven Maere, and Yves Van de Peer. The gain and loss of genes during 600 million years of vertebrate evolution. *Genome Biology*, 7(5):R43.1–R43.12, 2006.

[13] Maria Boekels, Johann Peter Gogarten, and Lorraine Olendzenski. *Horizontal Gene Transfer: Genomes in Flux*. Humana Press, 2009.

[14] Prosenjit Bose, Jonathan F. Buss, and Anna Lubiw. Pattern matching for permutations. *Information Processing Letters*, 65:277–283, 1998.

[15] R. Bowen. DNADot web-based dot plot tool, 1998.

[16] N. Bray, I. Dubchak, and L. Pachter. Avid: A global alignment program. *Genome Research*, 13:97–102, 2003.

[17] N. Bray and L. Pachter. MAVID: Constrained ancestral alignment of multiple sequences. *Genome Research*, 14:693–699, 2004.

[18] T. A. Brown. *Genomes 3*. Garland Science, third edition, 2007.

[19] M. Brudno, M. Chapman, B. Gottgens, S. Batzoglou, and B. Morgenstern. Fast and sensitive multiple alignment of large genomic sequences. *BMC Bioinformatics*, 4(66), 2003.

[20] M. Brudno, S. Malde, A. Poliakov, C. B. Do, and O. Couronne. Glocal alignment: Finding rearrangements during alignment. *Bioinformatics*, 19(Suppl. 1):i54–i62, 2003.

[21] Maria Carmela and Marcello Buiatti. The horizontal transfer of *Agrobacterium rhizogenes* genes and the evolution of the genus *Nicotiana*. *Molecular Phylogenetics and Evolution*, 20(1):100–110, 2001.

[22] Aaron C.E. Darling, Bob Mau, Frederick R. Blattner, and Nicole T. Perna. MAUVE: Multiple alignment of conserved genomic sequence with rearrangements. *Genome Research*, 14:1394–1403, 2004.

[23] Arthur Delcher, Simon Kasif, Robert Fleischmann, Jeremy Peterson, Owen White, and Steven Salzberg. Alignment of whole genomes. *Nucleic Acids Research*, 27(11):2369–2376, 1999.

[24] Inna Dubchak, Alexander Poliakov, Andrey Kislyuk, and M Brudno. Multiple whole-genome alignments without a reference organism. *Genome Research*, 19:682–689, 2009.

[25] Bernard Dujon, David Sherman, and et al. Genome evolution in yeasts. *Nature*, 430:35–44, 2004.

[26] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. UK: Cambridge University Press, 1998.

[27] Funda Ergun, S. Muthukrishnan, and S. Cenk Sahinalp. Comparing sequences with segment rearrangements. *Springer Berlin / Heidelberg*, 2914:183–194, 2004.

[28] Cdric Feschotte, Mark T. Osterlund Ryan Peeler, and Susan R. Wessler. Dna-binding specificity of rice mariner-like transposases and interactions with stowaway mites. *Nucleic Acids Research*, 33(7):2153–2165, 2005.

[29] M. R. Garey and D. S. Johnson. *Computer and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[30] Fan Ge, Li-San Wang, and Junhyong Kim. The cobweb of life revealed by genome-scale estimates of horizontal gene transfer. *PLoS Biology*, 3(10):e316, 2005.

[31] M. Hamady, M. D. Betterton, and R. Knight. Using the nucleotide substitution rate matrix to detect horizontal gene transfer. *BMC Bioinformatics*, 7:476, 2006.

[32] David Harper and Michael Benton. *Introduction to paleobiology and the fossil record*. Wiley-Blackwell, 2009.

[33] Daniel H. Huson and David Bryant. Application of phylogenetic networks in evolutionary studies. *Molecular Biology and Evolution*, 23(2):254–267, 2006.

[34] Guohua Jin, Luay Nakhleh, Sagi Snir, and Tamir Tuller. Inferring phylogenetic networks by the maximum parsimony criterion: A case study. *Molecular Biology and Evolution*, 24(1):324–337, 2007.

[35] Jijoy Joseph and Roschen Sasikumar. Chaos game representation for comparison of whole genomes. *BMC Bioinformatics*, 7(243):10 pages, 2006.

[36] Sampath K. Kannan and Tandy J. Warnow. Tree reconstruction from partial orders. *SIAM Journal of Computing*, 24(3):511–519, 1995.

[37] S. Karlin. Detecting anomalous gene clusters and pathogenicity islands in bacterial genomes. *Trends in Microbiology*, 9:335–343, 2001.

[38] Natsuko Kondo, Naruo Nikoh, Nobuyuki Ijichi, Masakazu Shimada, and Takema Fukatsu. Genome fragment of Wolbachia endosymbiont transferred to X chromosome of host insect. *Natural Academy of Science*, 99:14280–14285, 2002.

[39] L. B. Koski and G. B. Golding. The closest BLAST hit is often not the nearest neighbor. *Journal of Molecular Evolution*, 52:540–542, 2001.

[40] Stefan Kurtz, Adam Phillippy, Arthur L Delcher, Michael Smoot, Martin Shumway, Corina Antonescu, and Steven L Salzberg. Versatile and open software for comparing large genomes. *Genome Biology*, 5(R12):9 pages, 2004.

[41] James A. Lake and Maria C. Rivera. Deriving the genomic tree of life in the presence of horizontal gene transfer: Conditioned reconstruction. *Molecular Biology and Evolution*, 21(4):681–690, 2004.

[42] J. G. Lawrence and H. Ochman. Amelioration of bacterial genomes: Rates of change and exchange. *Journal of Molecular Evolution*, 44:383–397, 1997.

[43] J. G. Lawrence and H. Ochman. Molecular archaeology of the *Escherichia coli* genome. *Proceedings of National Academy of Sciences USA*, 95:9413–9417, 1998.

[44] B. Loftus, I. Anderson, and et al. The genome of the protist parasite *Entamoeba histolytica*. *Nature*, 433(7028):865–868, 2005.

[45] Manyuan Long, Esther Betrán, Kevin Thornton, and Wen Wang. The origin of new genes: Glimpses from the young and old. *Genetics*, 4:865–875, 2003.

[46] Guoqing Lu, Liying Jiang, Resa MK Helikar, Thaine W Rowley, Luwen Zhang, Xianfeng Chen, and Etsuko N Moriyama. GenomeBlast: A web tool for small genome comparison. *BMC Bioinformatics*, 7(S18):9 pages, 2006.

[47] Jian Ma, Aakrosh Ratan, Brian J. Raney, Bernard B. Suh, Webb Miller, and David Haussler. The infinite sites model of genome evolution. *Proceedings of National Academy of Sciences USA*, 105(38):14254–14261, 2008.

[48] D. Macleod, R. Charlebois, F. Doolittle, and E. Bapteste. Deduction of probable events of lateral gene transfer through comparison of phylogenetic trees by recursive consolidation and rearrangement. *BMC Evolutionary Biology*, 5(27):11 pages, 2005.

[49] Vladimir Makarenkov. T-REX: Reconstructing and visualizing phylogenetic trees and reticulation networks. *Bioinformatics*, 17:664–668, 2001.

[50] Paul Marjoram and Simon Tavar. Modern computational approaches for analysing molecular genetic variation data. *Genetics*, 7:759–770, 2006.

[51] B. Mau, J. Glasner, A. Darling, and N. Perna. Genome-wide detetction and analysis of homologous recombination among sequenced strains of *Escherichia coli*. *Genome Biology*, 7(R44):12 pages, 2006.

[52] B. Morgenstern, N. Werner, S. J. Prohaska, R. Steinkamp, and I. Schneider. Multiple sequence alignment with user-defined constraints at GOBICS. *Bioinformatics*, 21:1271–1273, 2005.

[53] Luay Nakhleh, Guohua Jin, Fengmei Zhao, and John Mellor-Crummey. Reconstructing phylogenetic networks using maximum parsimony. In *CSB '05: Proceedings of the 2005 IEEE Computational Systems Bioinformatics Conference*, pages 93–102, Washington, DC, USA, 2005. IEEE Computer Society.

[54] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Molecular Biology*, 48:443–453, 1970.

[55] CL Nesbo, S L'Haridon, KO Stetter, and WF Doolittle. Phylogenetic analyses of two "archaeal" genes in *Thermotoga maritima* reveal multiple transfers between archaea and bacteria. *Molecular Biology and Evolution*, 18(3):362–375, 2001.

[56] P. S. Novichkov, M. V. Omelchenko, M. S. Gelfand, A. A. Mironov, Y. I. Wolf, and E. V. Koonin. Genome-wide molecular clock and horizontal gene transfer in bacterial evolution. *Journal of Bacteriology*, 186:6575–6585, 2004.

[57] Artem S. Novozhilov, Georgy P. Karev, and Eugene V. Koonin. Mathematical modeling of evolution of horizontally transferred genes. *Molecular Biology and Evolution*, 22(8):1721–1732, 2005.

[58] Hasan H. Otu and Khalid Sayood. A new sequence distance measure for phylogenetic tree construction. *Bioinformatics*, 19(16):2122–2130, 2003.

[59] M. Pagni and T. Junier. DOTLET Java-based tool, 1998.

[60] Adams Palmer. Evolution of mitochondrial gene content: Gene loss and transfer to the nucleus. *Molecular Phylogenetics and Evolution*, 29:380–395, 2003.

[61] Benedict Paten, Javier Herrero, Kathryn Beal Stephen Fitzgerald, and Ewan Birney. Enredo and Pecan: Genome-wide mammalian consistency-based multiple alignment with paralogs. *Genome Research*, 18:1814–1828, 2008.

[62] J. Pei and N. Grishin. MUMMALS: Multiple sequence alignment improved by using hidden Markov models with local structural information. *Nucleic Acids Research*, 34:4364–4374, 2006.

[63] NT Perna and et al. Genome sequence of enterohaemorrhagic *Escherichia coli* o157:h7. *Nature*, 409(6819):529–533, 2001.

[64] Tu Minh Phuong, Chuong B. Do, Robert C. Edgar, and Serafim Batzoglou. Multiple alignment of protein sequences with repeats and rearrangements. *Nucleic Acids Research*, 34(24):5932–5942, 2006.

[65] Sheila Podell and Terry Gaasterland. DarkHorse: A method for genome-wide prediction of horizontal gene transfer. *Genome Biology*, 8(2):R16.1–R16.18, 2007.

[66] D. Pohler, N. Werner, R. Steinkamp, and B. Morgenstern. Multiple alignment of genomic sequences using CHAOS, DIALIGN and ABC. *Nucleic Acids Research*, 33:W532–W534, 2005.

[67] CP Ponting. Plagiarized bacterial genes in the human book of life. *Trends in Genetics*, 17(5):235–237, 2001.

[68] B. Raphael, D. Zhi, H. Tang, and P. Pevzner. A novel method for multiple alignment of sequences with repeated and shuffled elements. *Genome Research*, 14:2336–2346, 2004.

[69] Aaron O. Richardson and Jeffery D. Palmer. Horizontal gene transfer in plants. *Journal of Experimental Botany*, 58(1):1–9, 2006.

[70] Noah A. Rosenberg and Magnus Nordborg. Genealogical trees, coalescent theory and the analysis of genetic polymorphisms. *Genetics*, 3:380–390, 2002.

[71] Christian Roth, Shruti Rastogi, Lars Arvestad, Katharina Dittmar, Sara Light, Diana Ekman, and David A. Liberles. Evolution after gene duplication: Models, mechanisms, sequences, systems, and organisms. *Journal of Experimental Zoology*, 308:58–73, 2007.

[72] T Rujan and W Martin. How many genes in Arabidopsis come from cyanobacteria? An estimate from 386 protein phylogenies. *Trends in Genetics*, 17(3):113–120, 2001.

[73] S. Schwartz, W. J. Kent, A. Smit, Z. Zhang, and R. Baertsch. Human-mouse alignments with BLASTZ. *Genome Research*, 13:103–107, 2003.

[74] Louis Shapiro and A. B. Stephens. Bootstrap percolation, the Schröder numbers, and the n-kings problem. *SIAM Journal of Discrete Mathematics*, 4:275–280, 1991.

[75] Tom Simonite. Protists push animals aside in rule revamp. *Nature*, 438(7064):8–9, 2005.

[76] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Molecular Biology*, 147:195–197, 1981.

[77] Sagi Snir and Edward Trifonov. A novel technique for detecting putative horizontal gene transfer in the sequence space. *Journal of Computational Biology*, 17(11):1535–1548, 2010.

[78] Matthew Spencer, David Bryant, and Edward Susko. Conditioned genome reconstruction: How to avoid choosing the conditioning genome. *Systems Biology*, 56(1):25–43, 2007.

[79] M. Syvanen and C. I. Kado. *Horizontal gene transfer*. Academic Press, 2nd edition, 2002.

[80] R. Szklarczyk and J. Heringa. Aubergenena sensitive genome alignment tool. *Bioinformatics*, 22:1431–1436, 2006.

[81] Cuong V. Than and Noah A. Rosenberg. Consistency properties of species tree inference by minimizing deep coalescences. *Journal of Computational Biology*, 18(1):1–15, 2011.

[82] J. D. Thompson, D. G. Higgins, and T. J. Gibson. CLUSTAL W: Improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Research*, 22:4673–4680, 1994.

[83] Hale F. Trotter. Perm (Algorithm 115). *CACM*, 5(8):434–435, 1962.

[84] Aristotelis Tsirigos and Isidore Rigoutsos. A new computational method for the detection of horizontal gene transfer events. *Nucleic Acids Research*, 33(3):922–933, 2005.

[85] Jean-Stephane Varre, Jean-Paul Delahaye, and Eric Rivals. Transformation distances: A family of dissimilarity measures based on movements of segments. *Bioinformatics*, 15(3):194–202, 1999.

[86] Augusto F. Vellozo, Carlos E.R. Alves, and Alair Pereira do Lago. Alignment with non-overlapping inversions in $o(n^3)$-time. *Algorithms in Bioinformatics, 6th International Workshop, WABI 2006, Zurich, Switzerland, 2006 proceedings*, 4175:186–196, 2006.

[87] I. Wallace, O. Sullivan, D. Higgins, and C. Notredame. M-Coffee: Combining multiple sequence alignment methods with T-Coffee. *Nucleic Acids Research*, 34:1692–1699, 2006.

[88] Xintao Wei, Lenore Cowen, Carla E. Brodley, Arthur Brady, D. Sculley, and Donna K. Slonim. A distance-based method for detecting horizontal gene transfer in whole genomes. In *ISBRA*, pages 26–37, 2008.

[89] Eric W. Weisstein. *CRC Concise Encyclopedia of Mathematics*. CRC Press, 2 edition, 2003.

[90] Stephen J. Willson. Unique reconstruction of tree-like phylogenetic networks from distances between leaves. *Bulletin of Mathematical Biology*, 68:919–944, 2006.

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

[92] L. Ye and X. Huang. MAP2: Multiple alignment of syntenic genomic sequences. *Nucleic Acids Research*, 33:162–170, 2005.

[93] Y. Zhang and M. S. Waterman. An Eulerian path approach to local multiple alignment for DNA sequences. *Proceedings of the National Academy of Science*, 102:1285–1290, 2005.