ALGORITHMS AND HYPOTHESIS SELECTION IN DYNAMIC HOMOLOGY PHYLOGENETIC ANALYSIS

by

ANDRÉS VARÓN

A dissertation submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York

2010

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction for the dissertation requirement for the degree of Doctor of Philosophy.

Amotz Bar-Noy

Date — Chair of Examining Committee

Theodore Brown

Date — Executive Officer

Katherine St. John

Bud Mishra

Ward C. Wheeler

Supervision Committee

THE CITY UNIVERSITY OF NEW YORK

Abstract

ALGORITHMS AND HYPOTHESIS SELECTION IN DYNAMIC HOMOLOGY PHYLOGENETIC ANALYSIS

by

Andrés Varón

Adviser: Amotz Bar-Noy

Phylogeny and alignment estimation are two important, and closely related biological problems. In the typical alignment problem, insertions, deletions, and substitutions need to be inferred, to understand the evolutionary patterns of life. With the technological advances of the last 20 years, phylogenetic analyses will grow to include complete chromosomes and genomes. With these data sets, not only insertions, deletions, and substitutions, but also rearrangements such as duplications, translocations, transpositions, and inversions must be taken into consideration.

In this study, phylogenetic analysis is explored at three different levels. At the first level, new heuristic algorithms for the joint estimation of phylogenies and alignments under the Maximum Parsimony optimality criterion are described. Our experimental study showed that the new algorithms perform dramatically better when compared to previous heuristics. These new algorithms will allow biologists to analyze larger data sets in shorter periods of time. At the second level, new and existing algorithms where implemented in the computer program POY version 4. POY has had a significant impact in the biology community, and is used by hundreds of biologists around the world. POY will serve as a platform for long term research both in algorithm engineering, and biology. At the third level, the problem of parameter and model selection in complete chromosome analyses is explored. We propose and describe the

use of Kolmogorov Complexity (KC) as optimality criterion, as a unifying criterion for phylogenetic analysis. Our evaluation using simulations showed that KC correctly identifies phylogeny and model with high frequency. These results are evidence that KC is very well suited for the difficulties posed by the phylogenetic analysis of complete genomes.

Dedicado a Fabio y Gloria:
mis verdaderos maestros,
mis mejores amigos.

Acknowledgements

I would like to thank my co-advisor Ward C. Wheeler. His generosity, energy, loyalty, and impetus, can hardly be matched. Ward is a true believer in academy, fearless of passionate arguments. He offered me his time, the environment, infrastructure, and economic support that made my studies possible. Working with Ward has affected every aspect of my life positively. There are no words that could express my gratitude to him. I would also like to thank Amotz Bar-Noy, who has advised me these years. His directions lit the path of this dissertation, taught me the true meaning of Computer Science, and fueled the difficult task of merging Computer Science and Biology in my career. To my wife and life companion, Paola Pedraza, I thank her dedication, tireless support, and endless faith on me. I would also like to thank Katherine St. John and Bud Mishra for serving as members of my committee under peculiar circumstances, and Theodore Brown, Joe Driscoll, and Lina Garcia, for their support in matters related to the Computer Science program.

Finally, I thank all the people who tried POY 4 in all of its development stages. They gave me, and continue providing me with excellent comments, bug reports, data sets, and observations that helped me tuning the heuristic strategies. I would like to specially thank Illya Bomash, Megan Cevasco, Louise Crowley, Torsten Dikow, Julian Faivovich, Gonzalo Giribet, Taran Grant, Christian Kehlmaier, Frédéric Legendre, Kurt M. Pickett, Fernando Marques, Leo Smith, and Ilya Temkin for their particularly helpful comments.

This work was partially supported by the U.S. Army Research Laboratory and the U.S. Army Research Office [W911NF-05-1-0271], and the NSF-ITR grant "Building the tree of life: A national resource for phyloinformatics and computational phylogenetics" [NSF EF 03-31495].

Contents

List of Tables

List of Figures

Chapter 1

Introduction

A phylogeny postulates shared ancestry relationships among organisms in the form of a binary tree (e.g. Figure 1.1). Phylogenies attempt to answer an important question posed in biology: what are the ancestor-descendent relationships between organisms? At the core of every biological problem lies a phylogenetic component. The patterns that can be observed in nature are the product of complex interactions, constrained by the template that our ancestors provide. For example, the presence and structure of the human skull is mostly determined by its structure in our ancestors. The relationship between the features observed in different organisms can only be understood if the phylogenetic relationships can be hypothesized.

There are four schools to address this question in biology. The simplest, and probably most commonly used family, are the *distance methods*. These employ estimated distances between organisms to hypothesize their relationships by maximizing similarity of related taxa according to the rules of a particular algorithm (e.g. Neighbor-Joining [94]). Distance based phylogenies are typically polynomial time solvable, and known to be consistent (i.e. to produce the true phylogeny), provided the pairwise evolutionary distance between taxa has a small error [6].

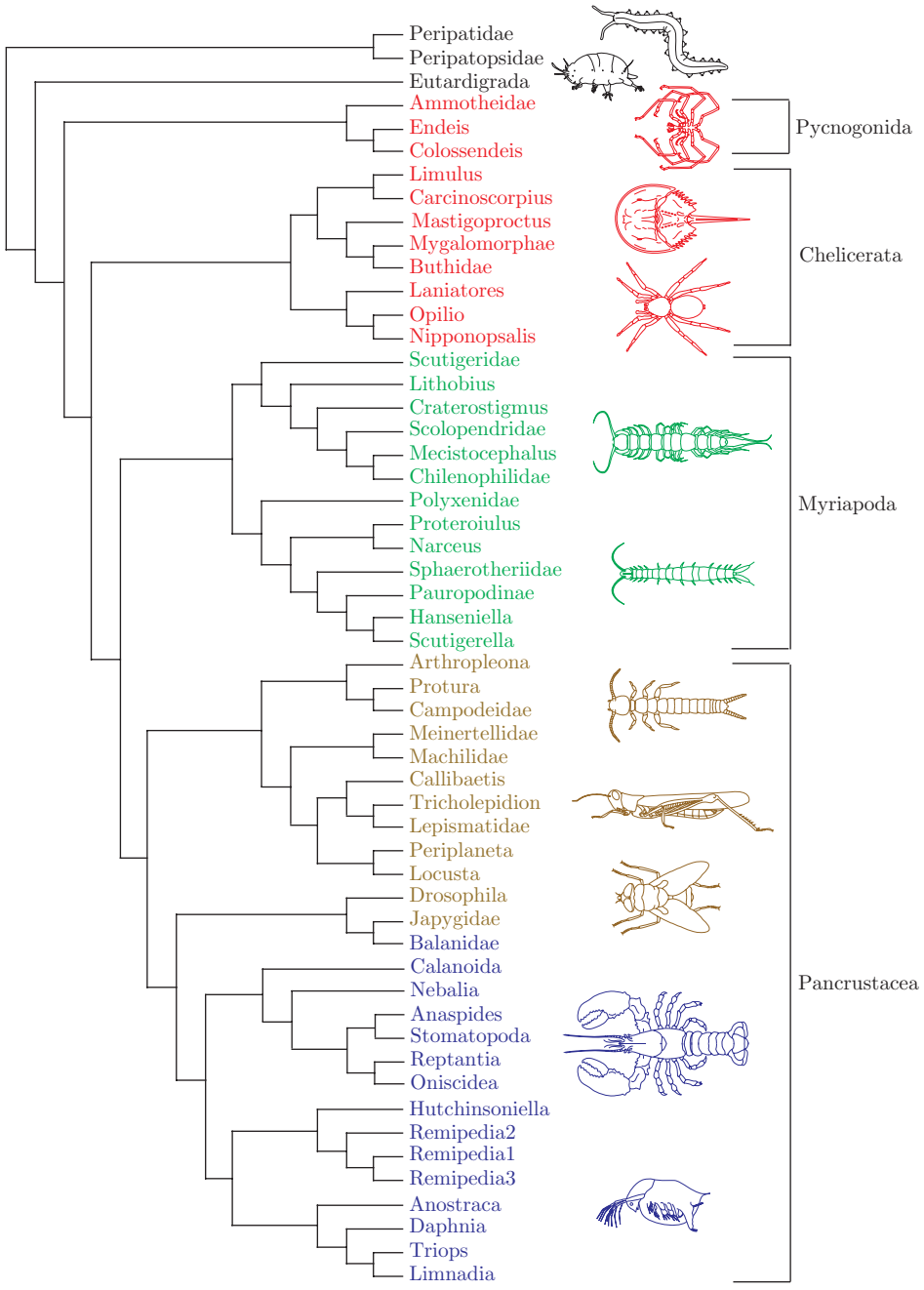


Figure 1.1: Arthropods phylogeny hypothesis example (from [43]).

A second method is *Maximum Parsimony* (MP). Under MP, the preferred hypothesis is the one that minimizes the overall number of evolutionary transformations required to explain the observed features [36]. This optimization problem is known in Computer Science as the Steiner Tree problem (see Definition 5), which is NP-Complete [41].

In *Maximum Likelihood* (ML) method, given a statistical evolutionary model, the best evolutionary hypothesis is the one that maximizes the likelihood of the observations given that model and tree. The corresponding optimization problem (minimizing the $-\log$ likelihood) is also NP-Hard [92]. ML is statistically consistent for some models of evolution, provided the model is the correct probabilistic source of the data [93]. A fourth method is *Maximum a posteriori*, which seeks to find a tree that maximizes its posterior probability given the data and priors [135]. This method is growing in popularity among practitioners in the field (for an overview see [38]).

A phylogenetic analysis selects a particular phylogeny using evidence collected by biologists (e.g. morphological, anatomical, molecular, developmental, behavioral). An individual feature that serves as evidence is known as a *character* (e.g. eye color, a gene), while its condition in an organism is the *state* (e.g. blue eye). Biologists are interested in a diverse set of characters. The following four examples illustrate this diversity.

Example 1 (Morphology)**.** *A morphological character could be the fruit color of a plant. The character states could be red, green, and yellow. Usually, such a set of valid states corresponds exactly to those observed in the organisms under study.*

Example 2 (Sequence of loci)**.** *The character is the set of sequences of loci from the mitochondrial chromosome. All the organisms of interest have the same set of loci, but each organism has a different permutation. We can also assume that the locus*

permutations in our sample do not constitute all the potential states, but a fraction of a much larger set, including all possible permutations (super-exponentially many, that is, $n!$ for n loci).

Example 3 (Nucleic acid sequence)**.** *In this example, a particular locus is the character (e.g. 18S rRNA). The states observed are RNA sequences, that is, words in the* $\{A, C, G, U\}$ *alphabet. Although we observe only a small fraction of the words, the states that could have occurred in nature include, in principle, all the possible words of this alphabet: an infinite number of states.*

Example 4 (Complete chromosome)**.** *In this example, we are interested in the analysis of a complete chromosome from a group of plants. Assume that we have one complete chromosome for each terminal that is believed to be homologous across the group. Moreover, we have annotated those chromosomes such that the limits of functional units are well established. We will further assume in the analysis that rearrangements, gain, and loss of functional units is possible, but restricted to our predefined limits (i.e. we consider impossible the rearrangement of the two halves of a functional unit). However, the correspondences between functional units is uncertain.*

Unlike the previous two examples, a chromosome state is defined by an infinitely large alphabet (i.e. the alphabet are the functional units, and a functional unit could be any DNA sequence). This character is the composition of the previous two examples, where DNA sequences are the elements comprising each character state. We are interested in the insertions, deletions, and substitutions occurring between corresponding functional units, and also in the higher level events that modify the order in which these units occur. Clearly, a huge number of valid states is not being observed.

We are ready now to formally define a character.

Definition 1 (Character)**.** *A character C is a set of states, where each state is an ordered set of elements from a predefined alphabet Σ.*

In our morphological example, $\Sigma = \{\text{red, yellow, green}\}$, and the valid states are ordered sets with only one element, that is, $C = \Sigma^1$ (i.e. $\{\langle\text{red}\rangle, \langle\text{yellow}\rangle, \langle\text{green}\rangle\}$). In the locus sequence example, the alphabet is the set of mitochondrial genes, that is $\Sigma = \{\text{CO1, CO2, CO3, ATP6, } \ldots\}$, while C includes all the permutations of the elements in Σ. In this case, every valid state must include all the genes (i.e. an exponential, but finite number of states). In the sequence character example, the alphabet is $\Sigma = \{\texttt{A,C,G,U}\}$, while the valid states are all the sequences that could be created with it, that is, $C = \Sigma^*$ (i.e. infinitely many states). In the chromosomal character example, the alphabet itself is $\Sigma = \{\texttt{A,C,G,T}\}^*$ (i.e. all the words that can be created with $\{\texttt{A,C,G,T}\}$), and the valid states are $C = \Sigma^*$. In this case, the alphabet itself has an infinite number of elements. We now define static homology and dynamic homology characters [112].

Let A and B be two states of a character C. A *correspondence* between the elements in A and B is a relation between them. A pair of elements correspond if there could be an (evolutionary) path that transformed one into the other. In the morphology example, every pair of elements of two states are corresponding elements. In the sequence example, equal elements of two permutations are corresponding elements. In the nucleic acid sequence, and complete chromosome examples, every element in a state corresponds to every other element in another state.

Definition 2 (Static homology characters)**.** *C is a* static homology character*, iff for every element in A there is at most one corresponding element in B, and the correspondence relations are transitive (i.e. let $a \in A$, $b \in B$, and $d \in D$ be elements of different states, where a corresponds to b, and b corresponds to d; Then a and d*

must also correspond to each other).

Definition 3 (Dynamic homology characters)**.** *C is a dynamic homology character [126] if it is* not *a static homology character. That is, if for some pair of states A and B, there exists an element $a \in A$ that has more than one corresponding element in B, or the correspondences are not transitive.*

A requisite for phylogenetic analysis is the specification of the character model to score transformations between states. This model is typically a distance function or a Markov chain, which is used to compare transformations and distinguish those with lower scores as better evolutionary hypotheses. As we have seen already, states can be modified in numerous ways, including insertion, deletion (both generically named indel), substitution, inversion, translocation, duplication, and horizontal gene transfer. Computationally, the length variation, and permutations occurring between states describe complex combinatorial optimization problems. Biologically, the vast number of possible transformations and parameters makes model selection uncertain.

A popular approach to analyze sequences of different lengths consists of two steps: an alignment is created using a multiple sequence alignment algorithm (MSA) (e.g. [108, 79, 63, 32, 40, 89, 31, 129]), followed by a phylogenetic analysis using as criterion either distance, maximum parsimony (MP), maximum likelihood (ML), or maximum posterior probability (MAP) (see [38] for an overview). In the two step approach, columns of the MSA are interpreted as independent characters, and the states are the elements in the sequence alphabet. Typically, the MSA is a heuristic step, and so, the model selection is limited to that required for transformations between elements in the alphabet. Nevertheless, an MSA is only a heuristic approach that treats a dynamic homology character (DNA sequences) as a set of static homology characters (individual nucleotides in an alignment column).

1.1 Contribution

In this study, I explore a number of algorithms and model selection criteria for some dynamic homology characters. In Chapters 3 and 4, I describe new algorithms for the Tree Alignment Problem (TAP), and local search in the Generalized Tree Alignment Problem (GTAP). The GTAP is the combinatorial problem associated with simultaneous tree and alignment reconstruction using as model the edit sequence distance function, under MP. I experimentally show that my new algorithms are faster by more than three orders of magnitude previously described methods.

In Chapter 5, I present the computer program POY version 4 [112], of which I'm the main architect. POY implements many algorithms for phylogenetic analysis, including those described in Chapters 3 and 4. POY has had a real impact in the biological community, (e.g. it is downloaded from more than 40 countries every month, and numerous research papers using it have already been submitted or published).

POY incorporates algorithms that can be used in the analysis of complete chromosomes under the MP optimality criterion. These algorithms heuristically detect rearrangement transformations such as translocations, and inversions. However, selecting the parameters for such analyses, and the set of significant transformations has remained open. In Chapter 6, I address the parameter selection problem by exploring the use of Kolmogorov Complexity (KC) as optimality criterion. My experiments show that KC presents desirable properties, that make it well suited to the difficulties inherent to dynamic homology characters, by inferring not only phylogenies, but also their character models.

Chapter 2

Model Description

In this study, I will concentrate on variations of phylogenetic analysis under the *Maximum Parsimony* (MP), and *Kolmogorov Complexity* (KC) optimality criteria. In both types of analysis, I will describe new algorithms, and use preexisting algorithms for variations of the Steiner Tree Problem and subproblems, under metric spaces. Here I define these problems precisely.

Definition 4 (Metric space)**.** *A space $m = (P, d)$ with point set P and distance function $d(a, b) \in \mathbb{R}, a, b \in P$ is* metric *iff the following three conditions hold:*

1. *For all $a, b \in P, d(a, b) = d(b, a)$ (symmetry).*
2. *$d(a, b) = 0 \iff a = b$ (identity).*
3. *For all a, b, c, $d(a, b) \leq d(a, c) + d(c, b)$ (triangle inequality).*

Let $m = (P, d)$ be a space with point set P and a metric distance function $d : P \times P \to \mathbb{R}$, and let $S \subseteq P$ be the problem set.

Definition 5 (Steiner Tree Problem [60])**.** *Find a binary tree $T = (V, E)$, and assignment $\chi(v) \in P, v \in V$, such that $S \subseteq \chi^{-1}$ and $\sum_{(u,v)\in E} d(\chi(u), \chi(v))$ is minimized.*

That is, find a tree that spans the set of points S with minimal length. The Steiner Tree Problem (STP) is different from a Minimum Spanning Tree (MST) [24] in that an MST can *only* span the points in S, whereas the STP may include other points to reduce the overall solution tree length; The extra points are known as *Steiner Points* (see Figure 2.1). Biologically, the set S are the character states observed in the organisms under study. Biologists define the distance function d according to the character's conditions. Depending on the distance function, particular combinatorial and geometry problems arise.

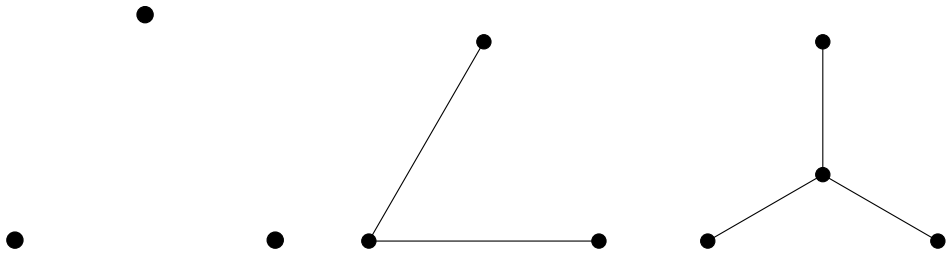


Figure 2.1: Difference between a Steiner and a Minimum Spanning Tree. (a) is the smallest, non–trivial problem, with three points at distance 1 between every pair. (b) is a Minimum Spanning tree of the instance problem; this solution has length 2. (c) is the optimal Steiner tree of the instance problem, of length 1.73. The center vertex is a Steiner point.

Morphological characters (Example 1) are typically assigned the hamming space.

Definition 6 (Hamming Space)**.** *Let $P = \{0,1\}^n$ for some fixed $n \in \mathbb{N}$, and let $A, B \in P$. The hamming distance $d(A,B)$ is defined as $d(A,B) = \sum_{i=1}^{n} |A_i - B_i|$.*

Definition 7 (The Phylogeny Problem [41])**.** *The Steiner Tree Problem where m is the Hamming Space is known as the phylogeny problem.*

When analyzing morphometric characters however, biologists may prefer the manhattan space.

Definition 8 (Manhattan Space)**.** *In the Manhattan space, $P = \mathbb{N}^n$ for some fixed n, and for $A, B \in P$, $d(A, B) = \sum_{i=1}^{n} |A_i - B_i|$.*

Definition 9 (The Manhattan Steiner Tree)**.** *The Steiner Tree Problem where m is the Manhattan Space is known as the manhattan steiner tree.*

Both the Manhattan Steiner Tree and the Phylogeny problem are typically used in the analysis of static homology characters. In the analysis of dynamic homology characters the most common metric of use is the sequence space.

Definition 10 (The Sequence Space)**.** *For some alphabet Σ, the sequence space is defined by $P = \Sigma^*$, and a sequence edit distance function (defined below).*

Definition 11 (Sequence Edit Distance [54])**.** *Two sequences A and B are aligned if and only if $|A| = |B|$. Let $d(x, y), x, y \in \Sigma$ be a distance between the elements in Σ. The cost of a pair of aligned sequences is defined as $cost(A, B) = \sum_{0 \leq i < |A|} d(A_i, B_i)$. Their affine cost is defined as $cost_{aff}(A, B) = (blocks \times a) + \sum_{0 \leq i < |A|} d(A_i, B_i)$, where $blocks$ is the number of maximal, consecutive sequences of indels in A and B, and a is the affine indel cost parameter.*

A sequence can be edited *by inserting indels anywhere in it. The edit distance between A and B is*

$$e(A, B) = \min_{A', B'} cost(A', B'),$$

where A' (B') is A (B) that has been edited to align it with B' (A'). Likewise, the affine edit distance is defined as

$$e_{aff}(A, B) = \min_{A', B'} cost_{aff}(A', B').$$

Although the sequence edit distance can be defined more succinctly using induction, the edit cost of a pair of aligned sequences will be used in Sections 3 and 4.

The sequence space composed with the STP, is known as the Generalized Tree Alignment Problem.

Definition 12 (The Generalized Tree Alignment Problem [95])**.** *The Steiner Tree Problem where d is the affine or non-affine distance functions is the Generalized Tree Alignment Problem (GTAP).*

A possible task in the GTAP is the Tree Alignment Problem (TAP) (Figure 2.2).

Definition 13 (The Tree Alignment Problem [95])**.** *Given a binary (phylogenetic) tree $T = (V, E)$ with leaf vertex set $L \subseteq V$, an assignment of sequences $\chi : L \to \Sigma^*$ for some alphabet Σ, and an edit distance function $e : \Sigma^* \times \Sigma^* \to \mathbb{R}$, the* Tree Alignment Problem *is to find an assignment of sequences $\chi' : V \to \Sigma^*$ such that for all $v \in L$, $\chi(v) = \chi'(v)$, and the total cost $\sum_{(u,v)\in E} e(\chi'(u), \chi'(v))$ is minimized [95].*

Typically $\Sigma = \{\text{A, C, G, T, } \textit{indel}\}$. The special *indel* element is used to represent insertions and deletions indistinctly, but is not allowed to occur simultaneously in an instance solution.

The edit distance (Definition 11) can be computed using dynamic programming [80], following the recursive function:

$$e(A_{1\ldots i}, B_{1\ldots j}) = \min \begin{cases} e(A_{1\ldots i-1}, B_{1\ldots j-1}) + d(A_i, B_j) \\ e(A_{1\ldots i-1}, B_{1\ldots j}) + d(A_i, \textit{indel}) \\ e(A_{1\ldots i}, B_{1\ldots j-1}) + d(B_j, \textit{indel}) \end{cases} \tag{2.1}$$

with base cases $e(\langle\rangle, \langle\rangle) = 0$, and $e(\langle\rangle, A) = e(A, \langle\rangle) = \sum_{1\leq i\leq |A|} d(A_i, \textit{indel})$. The affine case is more elaborate but possesses the same spirit and time complexity [50].

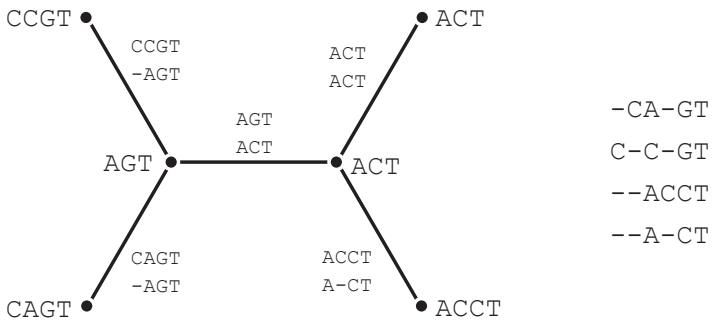


Figure 2.2: Tree alignment example. Given a tree, an assignment of sequences to the leaves, and a distance function, the problem is to find an assignment to the interior vertices such that the total tree length is minimized. In this example, if insertions, deletions, and substitutions have cost 1, then the total length of the tree is 5. Using the pairwise alignments defined on each edge, a multiple sequence alignment can be inferred for the solution.

Numerous other metric spaces are used in the the phylogenetic analysis of dynamic homology characters. One example of such space is the signed inversion space.

Definition 14 (Signed Inversion Space [98])**.** *Given a sequence* $\pi = \langle \pi_1, \ldots, \pi_n \rangle$*, a signed inversion of* π *starting in* $i, 1 \leq i \leq n$ *of length* l*, is*

$$\langle \pi_1, \ldots, -\pi_{i+l}, \ldots, -\pi_i, \pi_{l+i+1}, \ldots, \pi_n \rangle.$$

In the inversion space, P *is the set of* signed *permutations of the sequence* $\langle 1, \ldots, n \rangle$*, for some fixed* n*, and the distance* $d(A, B)$*,* $A, B \in P$ *is the minimum number of signed inversions required to transform* A *into* B*.*

In the remainder of this document, when I discuss the Steiner Tree Problem (STP), I am only referring to the Phylogeny Problem and Manhattan Steiner Tree Problem, which are extensively studied in the literature.

Chapter 3

The Tree Alignment Problem

The inference of homologies among DNA sequences, that is, positions in multiple genomes that share a common evolutionary origin, is a crucial, yet difficult task facing biologists. Its computational counterpart is known as the multiple sequence alignment problem. There are various criteria and methods available to perform multiple sequence alignments (e.g. [108, 79, 63, 32, 40, 89, 31, 129, 81]). Among these, given a distance function, *to minimize the overall cost of the alignment on a phylogenetic tree* [95, 97, 96, 57, 58, 124] is known in combinatorial optimization as the *Tree Alignment Problem* (TAP)(Definition 13, Figure 2.2). The TAP typically occurs as a subproblem of the *Generalized Tree Alignment Problem* (GTAP) [99], which looks for the tree with the lowest alignment cost among all possible trees [95] . The GTAP is equivalent to the Maximum Parsimony problem when the input sequences are not aligned, that is, when phylogeny and alignments are simultaneously inferred [95] (see Chapter 4).

An important element in sequence alignment and phylogenetic inference is the selection of the edit function, and in particular, the cost $G(k)$ of a sequence of k consecutive insertions or deletions, generically called indels (e.g. an insertion of 3 consecutive

`T` ($k = 3$) in the sequence `AA` could create the sequence `ATTTA`. The same operation in the opposite direction would be a deletion. The sequence alignment implied would be `A---A/ATTTA`, where `-` represents an indel). $G(k)$ could have a significant impact in the overall analysis [18, 76]. There are four plausible indel cost functions described in the literature: $G(k) = bk$ (*non-affine*) [122], $G(k) = a + bk$ (*affine*) [122], $G(k) = a + b \log k$ (*logarithmic*) [9, 53, 139, 20, 18], and $G(k) = a + bk + c \log k$ (*affine-logarithmic*) [18]. Simulations and theoretical work have found evidence that affine-logarithmic yields the most satisfactory results, but provide marginal benefits over the affine function, while its time complexity is much greater [18]. For this reason, many biologists adopt the affine indel cost function. (However, this topic is still a subject of controversy.)

For large data sets, a popular heuristic is *Direct Optimization* (DO) [124]. DO provides a good tradeoff between speed, scalability, and competitive scores, and is implemented in the computer program POY [132, 112]. For example, the alignment for the Sankoff *et al.* data set [97] produced by DO has cost 302.25, matching that of GESTALT [70] and SALSA [69]. Using an approximate iterative version of DO that has the same time complexity, POY finds a solution of cost 298.75, close to the best known cost of PRODALI (295.25) [102]. All other (competitive) algorithms have greater time complexities compared to DO (e.g [70, 69, 102]). An important limitation of DO, however, *is that it was not defined for affine edit distance functions.*

The properties of DO and the GTAP (DO+GTAP) for phylogenetic analysis were experimentally evaluated in [83]. The main conclusion of that study is that DO+GTAP could lead to phylogenies and alignments less accurate than those of the traditional methods (CLUSTALW + PAUP*). The initial work of Ogden and Rosenberg [83] raised a number of important questions: Do the conclusions hold if a better fit heuristic is used for the tree search in the GTAP? What would be the effect of

using an affine edit distance function? How do the hypothesis scores compare among the different methods? These questions have since been answered in various followup papers.

In [71], the author found that the opposite conclusion can be drawn when a better fit heuristic for the GTAP is used. That is, when the resulting tree is closer to the optimal solution, DO+GTAP is a superior method. Moreover, a good fit heuristic is a fundamental aspect in phylogenetic analysis that cannot be overlooked.

Although [83] performed simulations under affine gap costs, the study used the non-affine distance functions described for DO at the time of publication. Whether or not a different distance function could yield different conclusions was tackled in [76]. The authors found that when using the GTAP as phylogenetic analysis criterion under the affine gap cost function, the resulting phylogenies are competitive with the most accurate method for simulated studies (i.e. Probcons using a ML analysis under RaxML) [76]. It is important to note that [76] used an early implementation of the algorithms presented in this paper (available in POY version 4 beta).

A comparison of the tree scores of various methods was recently performed in [130] and is implicit in some of the conclusions of [76]. The authors concluded that when using a heuristic fit for the GTAP, the hypotheses have scores better than those produced by other methods. Therefore, without hindsight (i.e., when accuracy cannot be measured), biologists would prefer the hypotheses generated under the GTAP.

In this chapter, I introduce and present experiments using my new algorithm Affine-DO (Section 3.2). Affine-DO has the same time complexity of DO, but is correctly suited for the affine gap edit distance. I show its performance experimentally, as implemented in POY version 4, with more than 330,000 experimental tests (Section 3.3). These experiments show that the solutions of Affine-DO are close to the lower bound inferred from an LP solution. Moreover, iterating over a solution

produced using Affine-DO has very little impact in the overall solution, a positive sign of my algorithm's performance.

Although I build Affine-DO on top of the successful aspects of DO, DO has never been formally described, nor have its basic properties been demonstrated. To describe Affine-DO, I first formally define DO and demonstrate some of its properties (Section 3.1).

3.0.1 Related Work

The TAP is known to be NP-Hard [119]. Due to its difficulty, a number of heuristic methods are applied to produce reasonable (but most likely suboptimal) solutions. The first heuristic techniques [97, 96] consist of iteratively improving the assignment of each interior vertex as a median between the sequences assigned to its three neighbors. This method can be applied to any initial assignment of sequences and adjust them to improve the overall tree cost. In recent work, Yue *et al.* [136] used this algorithm in their computer program *MSAM* for the tree alignment problem, using as initial assignment the median computed between the 3 closest leaves to the interior vertex (ties arbitrarily resolved).

Hein [57, 58], designed a second heuristic solution which is implemented in the *TreeAlign* program. In TreeAlign, sets of sequences are represented by *alignment graphs*, which hold *all possible alignments* between a pair of sequences. The complete assignment can be performed in a post-order traversal of a rooted tree, where each vertex is assigned an alignment graph of the two closest sequences in the alignment graph of its two children vertices. The final assignment can be performed during a backtrack on the tree. Although this method is powerful, it is *not* scalable (e.g. using TreeAlign to evaluate one of the simulations used in this study did not finish

within 48 hours). Moreover, the TreeAlign program does not allow the user to fully specify the distance function. This algorithm was later improved by Schwikowski and Vingron, producing the best tree alignment known for the Sankoff data set [101].

The most important theoretical results for the TAP are several 2-approximation algorithms, and a Polynomial Time Approximation Scheme (PTAS) [121, 118, 88, 120]. These algorithms solve the TAP from a theoretical perspective, but the execution time of the PTAS renders it of no practical use. On the other hand, the 2-approximation algorithms have shown very poor performance when compared to heuristic methods such as that of TreeAlign.

*Direct Optimization*DO [124] is a heuristic implemented in the computer program POY [132, 131, 112], which yields good execution times and competitive alignment costs. Given that DNA sequences have a small alphabet (4 elements extended with an indel to represent insertions and deletions), DO represents a large number of sequences in a compact way by using an extended alphabet (potentially exponential in the sequence length). In the spirit of the TreeAlign method, DO heuristically assigns to each vertex, during a post order traversal, a set of sequences in an edit path connecting two of the closest sequences assigned to the children vertices. Later on, in a pre-order backtrack, a unique sequence is assigned to the interior vertices to produce the solution.

DO can be implemented with a time complexity of $O(n^2|V|)$, where n is the length of the longest sequences, and V is the vertex set of the tree. For larger alphabets the total time complexity is $O(n^2|V||\Sigma|)$, where $|\Sigma| \ll n$ is the alphabet.

Schwikowski and Vignon [102] published the best heuristic algorithm for the TAP, as implemented in the *PRODALI* software. Although powerful, this algorithm has a potentially exponential time and memory complexity, which in turn makes it non-scalable and difficult to use for the GTAP. Moreover, PRODALI is not publicly avail-

able. It is for these reasons that DO is the algorithm of choice for the GTAP, yielding slightly weaker tree cost approximations when compared to those of PRODALI, but suitable for better performance on larger data sets.

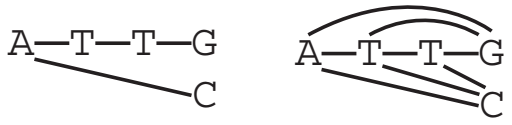
3.1 Direct Optimization

Direct Optimization (DO) has only been described informally in the literature [124, 131], and to build on it, we must first fill this gap. At the core of the algorithm is the use of an extended alphabet to represent sets of sequences. I begin by exploring the connection of this method with those using a tree alignment graph.

In TreeAlign and PRODALI, the set of optimal alignments between sequences are represented in an *alignment graph*. These graphs are aligned at each vertex in the tree to find their closest sequences. An alignment graph is then computed between these sequences, and assigned to a vertex of the tree. The alignment between a pair of such graphs, however, is an expensive computation, both algorithmically ($O(n^4)$), and in its implementation. PRODALI is more expensive in practice, as it not only stores the set of optimal, but also suboptimal alignments.

In DO, not all the possible alignments are stored, but only one. However, it stores all the possible sequences that can be produced from such alignment. I will call such set of sequences a *reduced alignment graph* (RAG). Thanks to their simplicity, DO use a more compact representation of a RAG, to permit greater scalability than that of TreeAlign or PRODALI. DO represents them as sequences in an extended alphabet by which we can then represent a complete RAG with an array.

It is then possible to align RAG's, find the closest sequences contained in them, and compute their RAG with time complexity $O(n^2)$. The following section formalizes these ideas.



(a) (b)

Figure 3.1: Graphs representing the alignment `ATTG`, `A--C`. **a.** A plain alignment graph. **b.** An alignment graph that contains more potential sequences.

3.1.1 Sets of Sequences, Edition Distance, and Medians

The first goal is to find a compact representation of sets of sequences produced in a pairwise alignment. For example, the alignment `ATTG A--C` is represented in an alignment graph shown in Figure 3.1. Such graph can then be extended to include intermediate sequences such as `ATG` or `ATC` (Figure 3.1).

The same information can be efficiently stores by using an extended alphabet $\Sigma_P = \mathscr{P}(\Sigma) \setminus \{\emptyset\}$ that includes all the subsets of Σ with the exception the empty set, as follows

$$\{\texttt{A}\}\{\texttt{T}, \text{indel}\}\{\texttt{T}, \text{indel}\}\{\texttt{G,C}\}.$$

I call such representation a *Reduced Alignment Graph* (RAG). Notice that all the intermediate sequences can be produced by selecting an element from each set in the RAG, and removing all the indels from the resulting sequence. If a sequence can be generated by following this procedure, then we say that the sequence is *included* in the RAG. The example then includes the sequences `ATTG`, `ATTC`, `ATC`, `ATG`, `AC`, and `AG`.

Observation 1. *Let $A \in \Sigma_P^*$ be a RAG. Then there are $\prod_{X \in A} |X|$ sequences contained in A.*

In the original problem definition we are given a distance d between the elements in Σ. Let $d_P(A, B) = \min_{a \in A, b \in B} d(a, b)$, be the distance between elements in Σ_P. The following observation is by definition:

Observation 2. *For all $A, B \in \Sigma_P$, there exists an $a \in A$ and $b \in B$ such that $d_P(A, B) = d(a, b)$.*

Define the RAG edit distance by setting $d = d_P$ in Equation 2.1.

I will show that we can efficiently find the closest sequences in a pair of RAGs, as well as their edit distance. Thanks to these properties, a RAG is used instead of an alignment graph, to bound the cost of a tree with lower time complexity.

Lemma 1. *For all RAGs A, B, there exists sequences U, V such that U is contained in A, V is contained in B, and $e(A, B) = e(U, V)$.*

Proof. I will define a procedure to produce U and V. Start with an empty U and V, and follow the backtrack of Equation 2.1. For each case, prepend the following to U and V:

case 1 Select an element $x \in X_i$ that holds Observation 2 and prepend it to U. Then find an element $y \in Y_k$ that is closest to x and prepend it to V. From Observation 2 we know that $d(x, y) = d_P(X_i, Y_j)$.

case 2 Select an element $x \in X_i$ closest to *indel* and prepend x to U and *indel* to V. Again from Observation 2 we know that $d(x, indel) = d_P(X_i, \{indel\})$.

case 3 Symmetric to case 2.

□

Observe that the overall time complexity remains $O(n^2)$ as in the original Needleman-Wunsch algorithm [80].

3.1.2 The DO Algorithm

DO (Algorithm 1) estimates the cost of a tree by proceeding in a post-order traversal on a *rooted tree*, starting at the root ρ, and assigning a RAG to each interior vertex.

Data: A binary tree T with root ρ
Data: An assignment $\chi : L(T) \rightarrow \Sigma$ of sequences to the leaves L of T
Data: $S(v)$ holds a set of sequences for vertex v, initially empty for every vertex
Result: *cost* holds an upper bound of the cost of T, χ
begin
  $cost \leftarrow 0$;
  foreach *level of* T*, with the bottom level first* **do**
    foreach *node* v *at the level* **do**
      if v *is a leaf (has no children)* **then**
        $S(v) \leftarrow \langle a_i, a_i = \{\chi(v)_i\}\rangle$;
      else
        Data: v has children u and w
        $cost \leftarrow cost + e_P(S(u), S(w)$;
        $U, W \leftarrow$ the alignment of $S(u)$ and $S(w))$ respectively;
        $S(v) \leftarrow m_P(U, W)$;
    return *cost*
end

Algorithm 1: $DO(T, \chi)$, Direct Optimization

I have not defined yet $m_P(U, W)$ to compute each RAG. Let $m(X, Y)$ be the set of elements in X and Y that realize the distance $d_P(X, Y)$. Let the RAG between two aligned RAGs $A, B \in \Sigma_P^*, |A| = |B| = n$ be

$$m_P(A, B) = \langle x_i = m(A_i, B_i)\rangle.$$

Without loss of generality, assume from now on that for all $x \in \Sigma\backslash\{indel\}$, $d(x, indel) = b$ for some constant b.

Lemma 2. *Let* $C = m_P(A, B)$*. Then for all* X *included in* C*, there exists* Y *and* Z

included in A and B respectively, such that $e_P(A, B) = e(Y, Z) = e(X, Y) + e(X, Z)$. Moreover, Y and Z are (some of) the closest sequences to C that are contained in A and B respectively.

Proof. Follows directly from the median definition and Lemma 1. □

Lemma 2 is important for the correctness of the DO algorithm. It shows that for every sequence contained in C, there are corresponding sequences in A and B of edit distance equal to $e_P(A, B)$. This lemma can then be used in the DO algorithm to delay the selection of a sequence from each RAG, and use e_P directly to calculate the overall cost of the tree. Without it, e_P cannot be used for this purpose directly.

Definition 15 (Compatible assignments)**.** *Two assignments $\chi : V \to \Sigma^*$ and $\chi' : V \to \Sigma^*$ are* compatible *if both assign the same sequences to corresponding leaves, that is, for all $v \in L$, $\chi(v) = \chi'(v)$.*

The following Theorem shows that the tree cost computed by DO is feasible:

Theorem 1. *There exists an assignment of sequences χ' compatible with χ such that*

$$DO(T, \chi) = \sum_{(u,v)\in E} e(\chi'(u), \chi'(v)).$$

Proof. Let T have root vertex ρ. Call χ' the final assignment of sequences to the vertices of T. Select any X included in $S(\rho)$ and set $\chi'(\rho) \leftarrow X$. Then for each other vertex v with parent p, following a pre-order traversal starting at ρ, let $\chi(v) \leftarrow X$ where $X \in \Sigma^*$ is included in $S(v)$ and is closest to $\chi'(p)$. From Lemma 2, we know that for any selection at p there exists a selection in its children that would yield the additional cost computed at p during the DO algorithm. Moreover, at each pre-order

traversal step, we assign to each vertex v the closest sequence to $\chi'(p)$ included in $S(v)$. Again from Lemma 2, we know that the total cost of the two edges connecting p with its children must be greater than or equal to the additional cost computed for vertex p in the DO algorithm. Therefore, $DO(T, \chi) \geq \sum_{(u,v)\in E(T)} e(\chi'(u), \chi'(v))$. $\square$

DO is weaker than the alignment graph algorithms [58, 101, 102], as the later techniques maintain the set optimal edit paths between sequences, or a superset including it. However, in these algorithms the overall execution time and memory consumption requirements could grow exponentially [102]. In contrast, DO maintains a polynomial memory and execution time, making it more scalable, with competitive tree scores. Moreover, DO can be efficiently implemented thanks to the simplicity of the data structures involved.

3.2 The Affine Gap Cost Case

In practice, biologists use DO because of its scalability and competitive costs. However, the DO algorithm was defined for the non-affine distance functions ($G(k) = bk$), and does not work correctly for the popular affine indel cost model [122] ($G(k) = a + bk$). Under many parameter sets, DO could produce worse tree cost estimations than those of the Lifted Assignment if used under the affine gap cost model (non published data). The fundamental reason for this problem is that Lemma 2 does not hold for the affine gap cost (e.g. Figure 3.2), and therefore, e_P cannot be directly used to correctly bound the cost of a tree.

To overcome this problem, I extend Gotoh's algorithm [50] to compute distances heuristically for sequences in Σ_P^*, and define a new median sequence. With these tools, I modify DO so that Lemma 2 still holds to compute tree cost bounds.

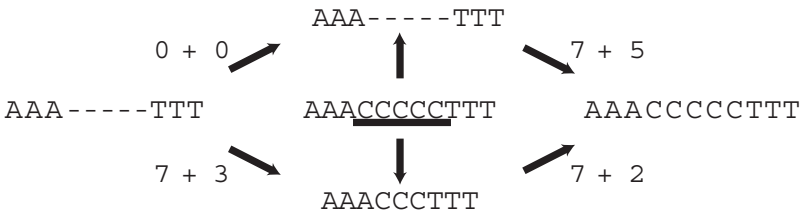


Figure 3.2: Let $G(k) = 7 + k$. The center sequence is the median for the alignment of the left and right sequences. (The underscored C represents {C, *indel*}.) Although the upper and lower sequences are included in the median, the lower one is not in an optimal edit path connecting left and right. This example shows Lemma 2 does not hold for affine gap costs. Therefore, there are sequences in this RAG that cannot be used directly in the DO algorithm without an extra cost, not computed by e_P. It follows that DO, if used directly for the affine gap cost case, can compute an incorrect cost for a given tree.

3.2.1 Heuristic Pairwise RAG Alignment

Let A and B be a pair of RAG's to be aligned. Define the *affine edit distance* function, analogous to e_P, using 4 auxiliary matrices (g, d, v, and h), as

$$e_{\text{aff}_P}(A_{1\ldots i}, B_{1\ldots j}) = \min\{g[i,j], d[i,j], v[i,j], h[i,j]\}.$$

The matrices g, d, v, and h will be filled recursively. Before defining them formally, let me explain the basic intuition of each. $g[i,j]$ is the cost of an alignment where A_i and B_j align elements other than an *indel*. $d[i,j]$ is the cost of an alignment using indel elements in A_i and B_j. $v[i,j]$ is the cost of an alignment where we use a "vertical" indel block by aligning B_j with an indel. Finally, $h[i,j]$ is the cost of an alignment where we use a "horizontal" indel block by aligning A_i with an indel.

To compute these values, I define a number of accessory functions. The cost of a pure substitution $subst(X,Y) = d_P(X \setminus \{indel\}, Y \setminus \{indel\})$. Symmetric to the

substitution cost, we need the cost of *extending* a gap when $indel \in A, B \subseteq \Sigma$:

$$diag(X,Y) \quad = \quad \begin{cases} 0 & \text{if } indel \in X \text{ and } indel \in Y \\ \infty & \text{otherwise.} \end{cases}$$

There are three remaining accessory functions required to compute the matrices g, h, v, and d. Each function handles various cases where a or b needs to be added. The first function, $go(A,i)$ evaluates whether or not it is necessary to add a gap opening value when aligning A_i with a gap:

$$go(A,i) \quad = \quad \begin{cases} 0 & \text{if } i = 1 \text{ and } indel \in A_i \\ 0 & \text{if } i > 1 \text{ and } indel \notin A_{i-1} \text{ and } indel \in A_i \\ a & \text{otherwise.} \end{cases}$$

The second function $go'(X,Y)$ calculates the extra cost incurred when *not* selecting an indel in one of the sequences means splitting an indel block:

$$go'(X,Y) \quad = \quad subst(X,Y) + \begin{cases} 0 & \text{if } indel \notin X \\ a & \text{otherwise.} \end{cases}$$

The third, and final accessory function, computes what would be the extra cost of *extending* an indel, that is:

$$ge(X) \quad = \quad \begin{cases} 0 & \text{if } indel \in X \\ b & \text{otherwise.} \end{cases}$$

Finally, the recursive functions for the cost matrices is defined as:

$$g[i,j] = \min \begin{cases} g[i-1,j-1] + subst(A_i, B_j) \\ d[i-1,j-1] + subst(A_i, B_j) + \\ \quad go(A,i) + go(B,j) \\ v[i-1,j-1] + go'(B_j, A_i) \\ h[i-1,j-1] + go'(A_i, B_j), \end{cases} \tag{3.1}$$

$$h[i,j] = \min \begin{cases} h[i,j-1] + ge(B_j) \\ d[i,j-1] + ge(B_j) + go(B,j), \end{cases} \tag{3.2}$$

$$v[i,j] = \min \begin{cases} v[i-1,j] + ge(A_i) \\ d[i-1,j] + ge(A_i) + go(A,i), \end{cases} \tag{3.3}$$

$$d[i,j] = diag(A_i, B_i) + \tag{3.4}$$

$$\min \begin{cases} d[i-1,j-1] \\ g[i-1,j-1] + go(A,i) + go(B,j), \end{cases} \tag{3.5}$$

with base cases $g[0,0] = 0$, $d[0,0] = \infty$, $v[0,0] = go(A,1)$, $h[0,0] = go(B,1)$, $g = [0,i] = d[0,i] = v[0,i] = \infty, h[0,i] = h[0,i-1] + ge(B_i)$, $1 \leq i \leq |B|$, $v[j,0] = v[j-1,0] + ge(A_j)$, and $g[j,0] = d[j,0] = h[j,0] = \infty, 1 \leq j \leq |A|$.

The following theorem shows that if we align a pair of sequences in A, B, then we can bound the cost of the closest pair of sequences included in them.

Theorem 2. *There exists a sequence X contained in A and a sequence Y contained in B such that $e_{aff_P}(A,B) \geq e_{aff}(X,Y)$.*

Proof. I am going to create a pair of sequences X and Y contained in A and B respectively that have edit cost at most $e_{\text{aff}_P}(A, B)$. To do so, follow the backtrack that yields $e_{\text{aff}_P}(A, B)$, and at each position i and j in the aligned A and B assign X_k and Y_k, where k is the alignment position corresponding to the aligned X_i and Y_j as follows:

1. $g[i, j]$ is the cost of aligning $A_{1\ldots i}$ and $B_{1\ldots j}$ when a non-indel element of A_i and B_j is aligned. If the backtrack uses $g[i, j]$ then assign to X_i and Y_j the closest elements in $A_i \setminus indel$ and $B_j \setminus indel$. Observe that all the cases in Equation 3.1 align a non-indel element from A_i and B_j, and add a cost that is always greater than or equal to $subst(A_i, B_j) = d(X_i, Y_j)$.

2. $h[i, j]$ is the cost of extending an indel in the horizontal direction. Therefore, select $X_k = indel$, and

$$Y_k = \begin{cases} indel & \text{if } indel \in B_j \\ y, y \in B_j & \text{otherwise.} \end{cases}$$

   If $Y_k = indel$, then the alignment of X_k and Y_k causes no additional cost in the particular alignment being built between X and Y. Otherwise, then there is an extra cost, of at least the b parameter, which both cases of Equation 3.2 account for. Additionally, if the previous pair of aligned elements are a pair of indels (second case in 3.2, see below for the treatment of this option), then an extra indel opening cost is added.

3. $v[i, j]$ is the cost of extending an indel block in the vertical direction. The

treatment is symmetric to that of h, with $Y_k = indel$ and

$$X_k = \begin{cases} indel & \text{if } indel \in A_i \\ x, x \in A_i & \text{otherwise.} \end{cases}$$

4. $d[i, j]$ is the cost of extending an indel in the *diagonal direction*, that is, when both A and B hold indels, and those indels are being selected during the backtrack. Equation 3.5 ensures that this choice is only possible by assigning ∞ whenever at least one of A_i or B_j does not contain an indel. Otherwise, if this option is selected, then simply assign *indel* to both X_k and Y_k with no extra cost for the alignment of X and Y.

□

3.2.2 The Main Algorithm: Affine-DO

I will now use $e_{\text{aff}_P}(A, B)$ to bound the cost of a tree using a post-order traversal, in the same way we did with DO (Algorithm 1). In order to do so a RAG to be assigned on each step must be defined (i.e. the function m_P in Algorithm 1). To create the RAG M (initially empty), do as follows in each of the 4 items described in the proof of Theorem 2:

1. If we selected two indels in X_k and Y_k, don't change M.
2. If $X_k = indel$ and $Y_k \neq indel$, then prepend $\{indel\} \cup B_j$ to M.
3. If $X_k \neq indel$ and $Y_k = indel$, then prepend $\{indel\} \cup A_i$ to M.
4. If $X_k \neq indel$ and $Y_k \neq indel$, then prepend $\{x \in A_i, \text{for some } y \in B_j, d(x, y) = d(X_k, Y_k)\} + \{y \in B_j, \text{for some } x \in A_i, d(x, y) = d(X_k, Y_k)\}$ to M.

5. Once the complete M' is created, remove all the elements $M_i = \{indel\}$ to create the indel-less RAG M. I call M the RAG produced by $m_{\text{aff}_P}(A, B)$.

Definition 16 (Affine-DO)**.** Affine-DO *is Algorithm 1, modified by replacing* m_P *with* m_{aff_P}*, and* e_P *with* e_{aff_P}*.*

It is now possible to use the Affine-DO algorithm to bound heuristically the cost of an instance of the TAP.

Theorem 3. *Given a rooted tree* T *with root* ρ*, and an Affine-DO assignment* $S : V(T) \to \Sigma_P^*$*, there exists an assignment* $\chi' : V(T) \to \Sigma^*$ *such that* $X = \chi'(\rho)$ *and the cost computed by Affine-DO equals that implied by* χ'*.*

Proof. If there are no indels involved in the tree alignment, then the arguments of Theorem 1 would suffice. Hence, let me now concentrate on the cases that involve indels.

To prove those remaining cases, we will use induction on the vertices of the tree. To do so, we will count the *credits* that each vertex adds to the subtree it roots as added by the Affine-DO algorithm. The credits represent the maximum total cost of the indels involved in a particular subtree; we will compare them with the *debits* incurred by a set of indels, and verify that the *credits* are always greater than or equal to the *debits*. To simplify the description, we will call type A subsequences of maximal size holding only indels, and type B subsequences of maximal size holding sets that include, but are not limited to, indels, and type C maximal subsequences holding sets with no indel. I will count without loss of generality the *credits* and *debits* within those subsequences. In Figures 3.3 and 3.4, Type A is represented as a line, type B as a box with a center line, and type C as an empty box.

For the inductive step, consider the leaves of the tree. By definition, for all $v \in L$,

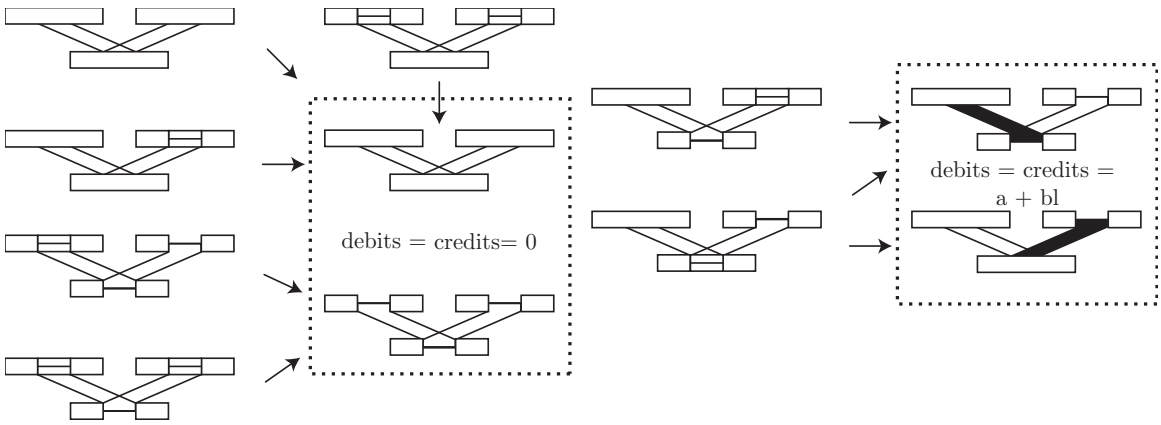


Figure 3.3: *credits* and *debits* incurred by the different possible arrangements of subsegments with matching limits in $S(p)$, $S(u)$, and $S(v)$. The only cases with $credits = debits > 0$ (in the right box) represents with filled boxes the assignments that would yield an indel block.

$S(v)$ can contain subsequences of neither type A nor B, as there are no indels allowed. Therefore, the theorem holds true, with a $credits = debits = 0$.

Consider now the interior vertex v, with children u and v. In Figure 3.3 all the simple cases where the limits of the subsequences in $S(u)$ and $S(v)$ match those of $S(p)$. It is straightforward to see that in all those cases $credits = debits$.

Consider now the more difficult case when the blocks do not have exact limits. Assume without loss of generality that $S(u)$ and $S(v)$ have a segment of type B, and $S(p)$ has in the corresponding segment a series of blocks of type A and C (Figure 3.4). (There can be no subsequences of type B in $S(p)$ aligned with those of type B in $S(u)$ and $S(v)$ as m_{aff_P} does not allow it.)

The total credit granted by Equation 3.1 is $c \geq 2ma + 2b\sum_{i=1}^{m} s_i$. I can transfer $c/2$ to u (v), so that in one edge rooted by u (v), a series of insertions corresponding to the subsequences $s_1, s_2, \ldots, s_m$ can occur (Figure 3.4, lower, solid boxes), while the other branch supports a single deletion of length $l - \sum_{i=1}^{m} s_i$ (Figure 3.4 lower, upper

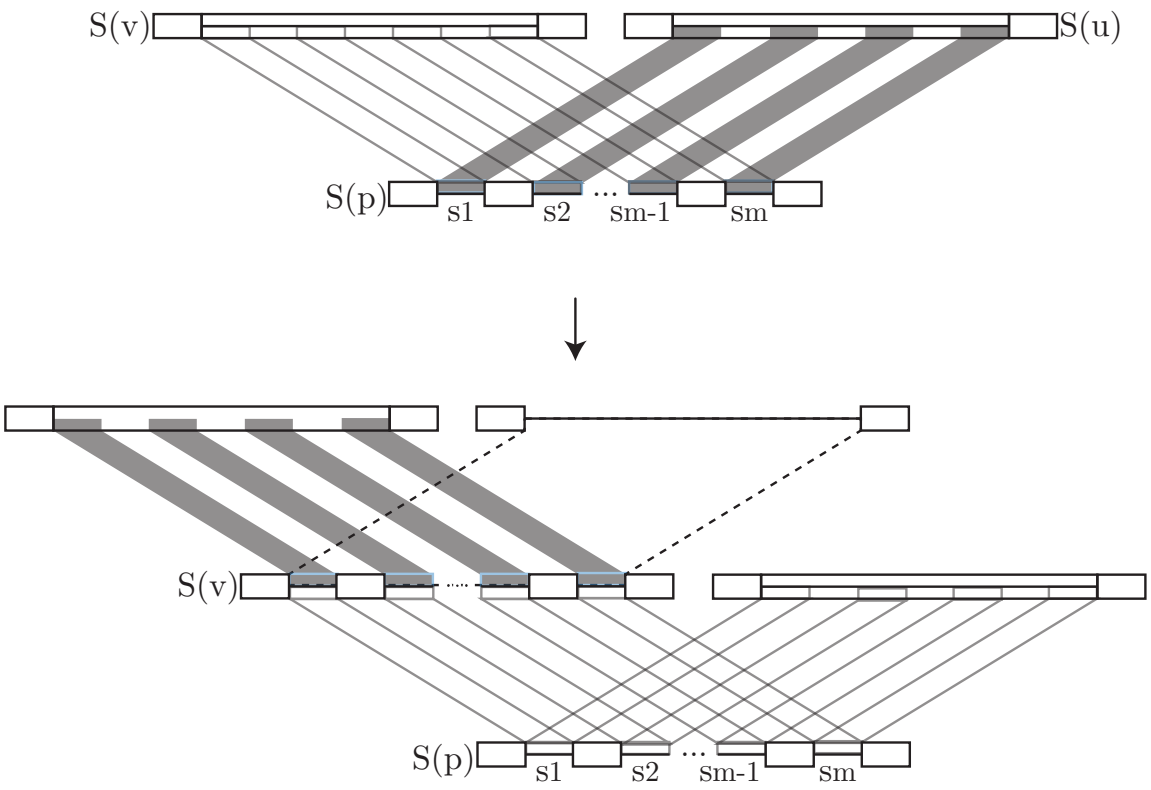


Figure 3.4: In the upper part, overlapping blocks of type B in $S(u)$ and $S(v)$, with a complex pattern of insertions and deletions in $S(p)$. The total *credits* added at $S(p)$ by Affine-DO can be transferred to u and v. In the lower, the credits transferred to v can be assigned to m individual insertion blocks (solid boxes), and one deletion block (dashed empty box) which maintain *debits* > *credits*.

dashed box). The total debit of these events now rooted in u would be

$$a(m+1) + b\sum_{i=1}^{m} s_i + b(l - \sum_{i=1}^{m} s_i) \leq c/2 + a + bl. \tag{3.6}$$

By the inductive hypothesis, the subtree rooted by u (v) has *credits* $\geq$ *debits*, and from Equation 3.6 we also have that *credits* > *debits* in p, therefore the theorem holds, and the overall tree rooted by p has a sequence assignment of cost at most that computed by the Affine-DO algorithm. □

Theorem 4. *If Σ is small, then Affine-DO has time complexity $O(n^2|V|)$ time, otherwise the time complexity is $O(n^2|V||\Sigma|)$.*

Proof. If the alphabet is small, then m_{aff_P} and d_P can be pre-computed in a lookup table for constant time comparison of the sets. For large alphabets the maximum size of the sets contained in Σ_P can be made constant. Otherwise, a binary tree representation of the sets would be necessary, adding a $|\Sigma|$ factor to the set comparison. Each heuristic alignment can be performed using dynamic programming, with time complexity $O(n^2)$ where n is the maximum sequence length (Ukkonen's [110] algorithm makes no obvious improvement as insertions and deletions could have cost 0 when aligning sequences in Σ_P^*) . Each alignment must be repeated for $|V|$ vertices during the post-order traversal, yielding the claimed time complexity. □

3.3 Experimental Evaluation

In this section, I describe the methods used to generate the instance problems (Section 3.3.1), assess the solutions generated by each algorithm (Section 3.3.2), and compare the algorithms (Section 3.3.3). This allow me to assess the performance of each algorithm (Section 3.3.4), Affine-DO in greater detail (Section 3.3.5), and an evaluation of Affine-DO using exact solutions for trees with only 3 leaves (Section 3.3.6).

3.3.1 Data Sets

To generate the instance problems, I simulated a number of sequences using DAWG 1.1.1 [17] with insertions and deletions following a power law distribution. The simulations followed random binary trees of 50 leaves comprising all the combinations

of the parameters listed in Table 3.1. These produced a total of 96,000 independent simulations. For each data set I collected the true sequence assignment. This information allowed me to compare the cost calculated by Affine-DO with the cost implied by the true sequence of events. My expectation was to produce costs lower than those using the true sequence assignment.

Parameter	Values Evaluated
Substitution Rate	1.5
Average Branch Length	$0.05, 0.1, 0.2, 0.3, \infty$
Max. Gap	$1, 2, 5, 10, 15$
Root Sequence Length	$70, 100, 150, 200,$ $300, 400, 500, 1000$

Table 3.1: Simulation parameters. All combinations of parameters were employed to generate the test data sets. The branch length variation equals the average branch length.

3.3.2 Solution Assessment

The sequences assigned by the simulation can be far from the optimal solution. To evaluate Affine-DO, I used two algorithms: the standard Fixed States algorithm, which is known to be a 2-approximation, and the cost calculated by the solution of an LP instance of the problem. A good heuristic solution should always be located between these two bounds. As a comparison measure for each solution, the ratio between the solution cost and the LP bound was computed. The closer the ratio to 1.0, the better is the solution.

This form of evaluation has the main advantage (but also disadvantage), of being overly pessimistic. Most likely, the LP solution is unachievable, and therefore, the

approximation ratio inferred for the solution produced by Affine-DO will most likely be an overestimate. To assess how over-negative the LP bound is, we produced 2100 random sequences divided in triplets of lengths between 70 and 1000. For each triplet, the Affine-DO, the LP bound, and the exact solution were computed. These three solutions were compared to provide an experimental overview of the potential performance of our algorithm. I selected random sequences because preliminary experiments showed evidence that these produce the most difficult instances for Affine-DO.

3.3.3 Algorithms compared

I implemented a number of algorithms to approximate the tree alignment problem. My implementation can be divided in two groups: initial assignment, and iterative improvement.

Initial Assignment includes the *Fixed States* (a stronger version of the Lifted Assignment [121, 125]), Direct Optimization [124], and Affine-DO. Each of these algorithms starts with a function χ and creates a χ' compatible with χ which is an instance solution. DO and Affine-DO have already been described. The Fixed States [125] is a simple algorithm were the interior vertices are optimally assigned one of the leaf sequences of the input tree, yielding a 2-approximation solution [121].

Iterative Improvement modifies an existing χ' by readjusting each interior vertex using its three neighbors. This procedure is repeated iteratively, until a (user provided) maximum number of iterations is reached, or no further tree cost improvements can be achieved. The adjustment itself can be done using an *approximated* or an *exact* three dimensional alignment, which I call the *Approximate Iterative* and *Exact Iterative* algorithms. Approximate Iterative (Figure 3.5), uses DO or Affine-DO

(the selection depends on which kind of edit distance function is used) to solve the TAP on the three possible rooted trees formed by the three neighbors of the vertex used as leaves. The assignment yielding the best cost is selected as the new center. The exact three dimensional alignment has time complexity $O(n^3)$ [87]. My implementation uses the low memory algorithms implemented by Powell [87], though they can be improved to $O(n^2)$ memory consumption [137].

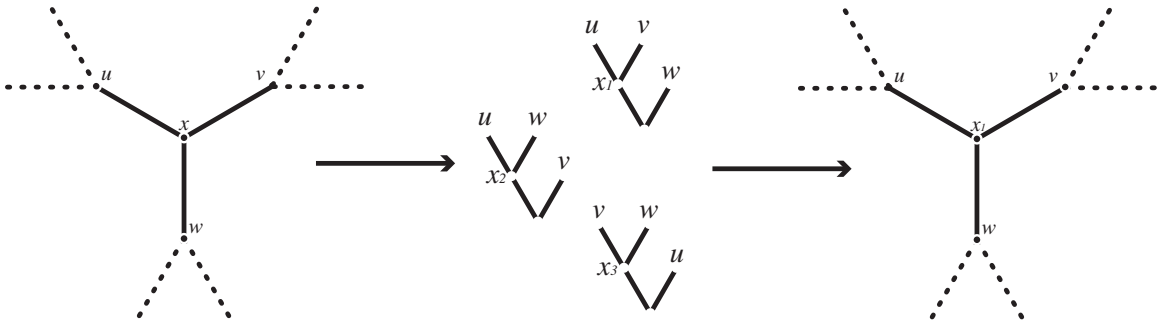


Figure 3.5: An iteration of the approximated iterative improvement. To improve x, Affine-DO is used to produce x_1, x_2, and x_3 in the three possible rooted trees with leaves u, v, and w. If the best assignment x_1 yields better cost than the original x, then it is replaced, otherwise no change is made.

I compared MSAM [136], Affine-DO, Approximate Iterative, Exact Iterative, and Fixed States, using a lower bound computed with an LP solution. I do not include DO in the comparisons because *it could not solve this problem* [114]. It is therefore impossible to compare it directly with our algorithm. GESTALT, SALSA, and PRODALI where unavailable, and so, could not be used in our comparative evaluation. TreeAlign did not produce a solution for the simulations within 48 hours of execution time, and therefore, was not included in the comparisons.

In total, more than 330,000 solutions were evaluated. I only present those results that show significant differences, and represent the overall patterns detected. The

Exact Iterative algorithm was only evaluated for the short sequences (70 to 100 bases), due to the tremendous execution time it requires. Fixed States followed by iterative improvement is not included because its execution time is prohibitive for this number of tests (POY version 4 supports this type of analysis). Nevertheless, my preliminary analyses showed that this combination of algorithms produce results in between Fixed States and Affine-DO, but not competitive with Affine-DO.

3.3.4 Algorithm Comparison

The most important patterns observed between the evaluated algorithms are presented in Figure 3.6 and Table 3.2. In general, Affine-DO yields a better approximation than Fixed States. According to the density histograms (data not shown), the expected approximation ratio of 1.1 (versus 1.5 for Fixed States) in the best parameter combination, and 1.5 (versus 1.7) for the worst. Iterative improvement (both in exact and approximated forms) has a small overall impact in the approximation ratio (with a maximal decrease of 0.05 when compared with the solution inferred by Affine-DO alone). In all cases, Affine-DO found better solutions than the simulations (Table 3.2).

Although the combination of Affine-DO and Iterative improvement produces better solutions, its execution time is dramatically higher. In the current implementation, running on a 3.0 Ghz, 64 bit Intel Xeon 5160 CPU with 32 GB of RAM, Affine-DO evaluates each tree in less than 1 second in the worst case, while Affine-DO + Iterative improvement may take more than 1 hour per tree. For this reason, Affine-DO is well suited for heuristics that require a very large number of tree evaluations such as the GTAP, where millions of trees are evaluated during a heuristic search.

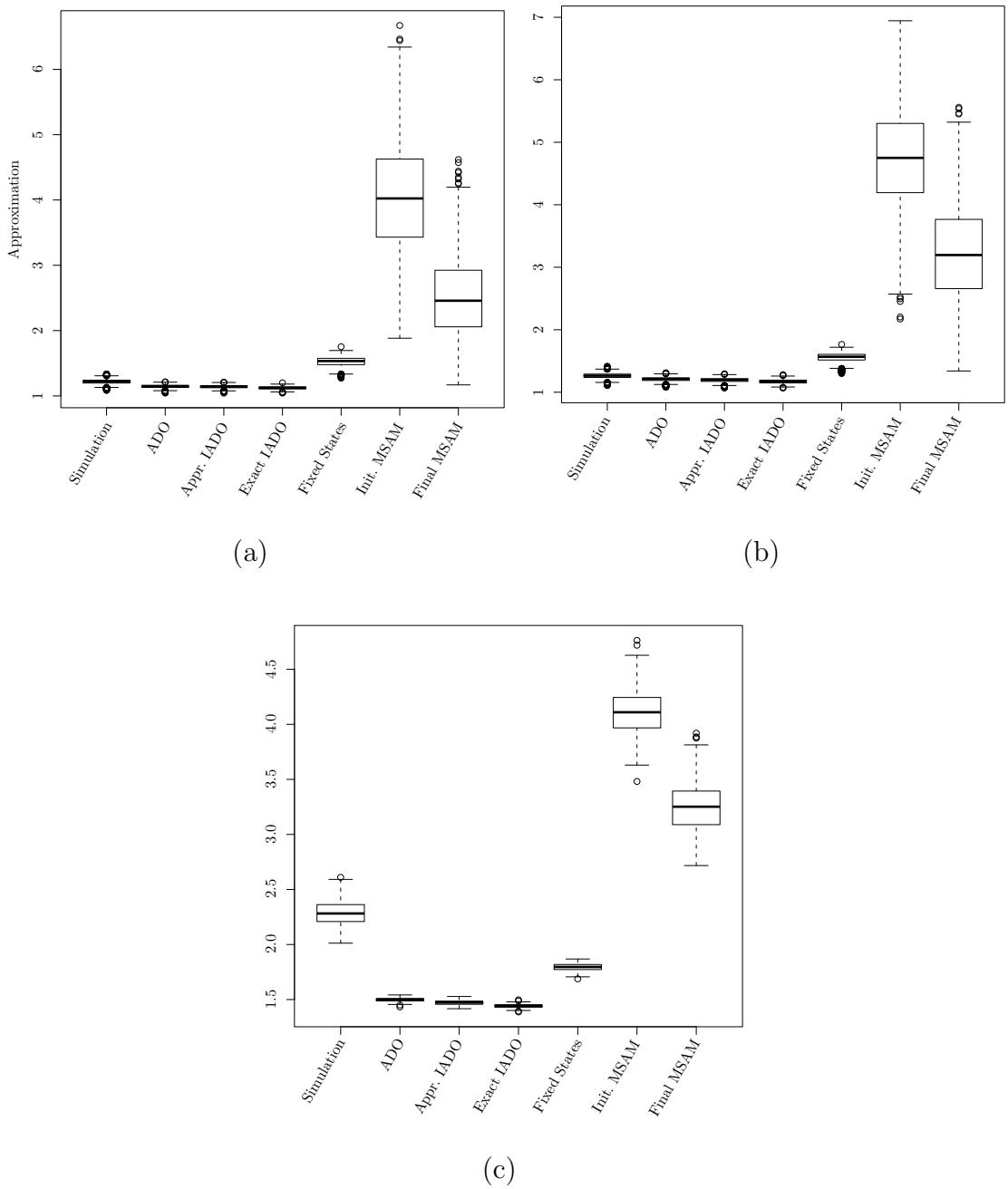


Figure 3.6: General patterns observed in the approximation ratio of the different algorithms. Simulation is the simulated data, ADO is Affine-DO, Approx. and Exact IADO are the approximated and the exact iterative Affine-DO algorithms respectively, initial and final MSAM are the initial and final estimations of the MSAM algorithm. **a.** substitutions = 1, $a = 0$, $b = 1$, branch length=0.05. **b.** substitutions = 4, $a = 3$, $b = 1$, branch length=0.05. **c.** substitutions = 4, $a = 3$, $b = 1$, branch length=0.3.

Subst.	Indel	Gap Op.	Branch Len.	Algorithm	Min.	Median	Max
1	1	0	0.05	Simulated	1.088	1.218	1.337
				Fixed States	1.275	1.534	1.755
				ADO	1.044	1.148	1.215
				ADO + Iter.	1.044	1.123	1.202
1	1	0	0.3	Simulated	1.731	2.022	2.396
				Fixed States	1.621	1.725	1.816
				ADO	1.314	1.398	1.453
				ADO + Iter.	1.300	1.377	1.393
4	1	3	0.05	Simulated	1.108	1.262	1.415
				Fixed States	1.302	1.557	1.766
				ADO	1.084	1.208	1.312
				ADO + Iter.	1.067	1.171	1.283
4	1	3	0.3	Simulated	2.012	2.284	2.611
				Fixed States	1.688	1.795	1.868
				ADO	1.433	1.500	1.542
				ADO + Iter.	1.388	1.442	1.453

Table 3.2: Numerical comparison of a pair of parameter combinations that represents the variation observed between the different algorithms.

3.3.5 Approximation of Affine-DO

Figure 3.7 shows the density histogram of the guaranteed approximation of the Affine-DO algorithm when compared with the LP theoretical solution for a representative set of parameters. The results show that Affine-DO has a guaranteed approximation of less than 60% in every case.

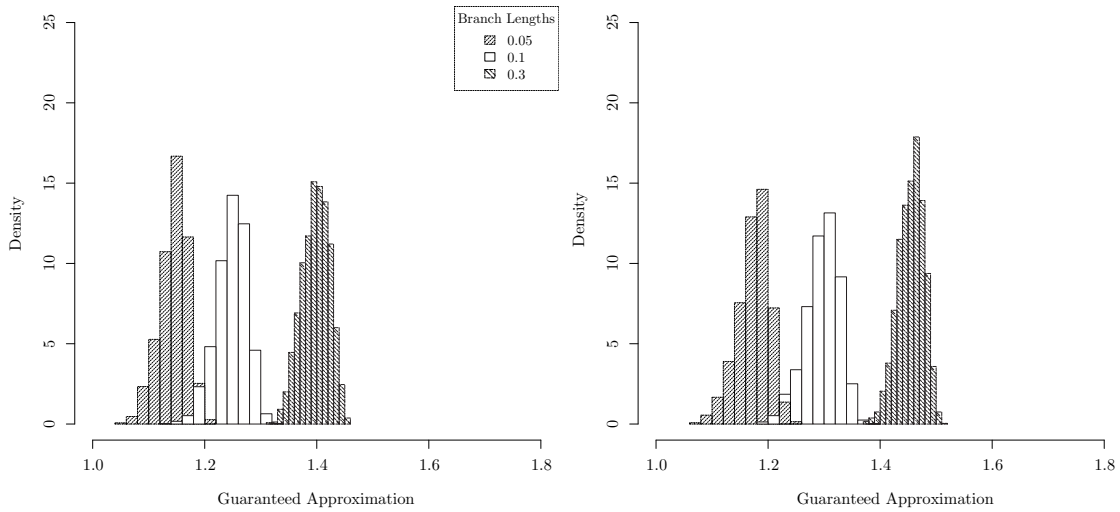
Typically, the larger the sequence divergence, the larger is the approximation degree of Affine-DO. The same pattern is observed for larger a. To test an extreme case, were the branch length is maximal, we evaluated the behavior of random sequences in the same set of trees. Figure 3.8 shows the results of this experiment.

The worst case is observed with an average approximation slightly over 1.5. This variation, however, could have been caused by a more relaxed LP bound, which could be producing an overly pessimistic evaluation of the algorithm. To assess the importance of this factor, we evaluated its tightness experimentally.

3.3.6 Comparison with an exact solution

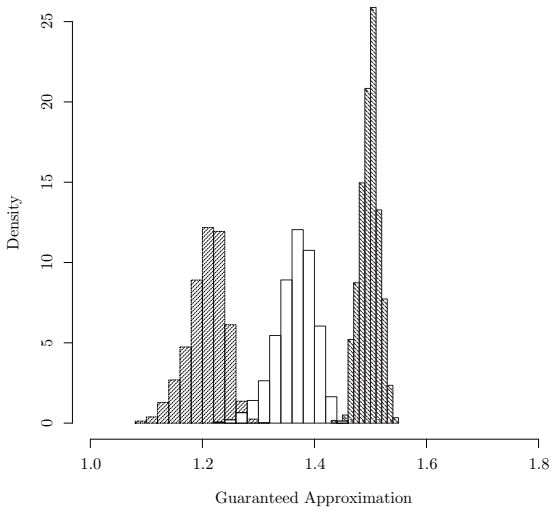
To assess Affine-DO and the tightness of the LP bound, we computed the exact solution for 700 unrooted trees consisting of 3 leaves with random sequences assigned to their leaves, under all the parameter sets tested. Figure 3.9 shows the density histograms for the results obtained.

Note that the LP-inferred bound is overly negative even for these small test data sets, with the inferred approximation expected at around 1.15, while in reality Affine-DO finds solutions that are expected to approximate within 1.05 of the optimal solution, a 10% difference for trees consisting of only 3 sequences.



(a)

(b)



(c)

Figure 3.7: Guaranteed approximation ratio of Affine-DO compared with the theoretical LP bound, for different cost and sequence generation parameters. **a.** substitutions $= 1$, $a = 0$, $b = 1$. **b.** substitutions $= 2$, $a = 1$, $b = 1$. **c.** substitutions $= 4$, $a = 1$, $b = 3$.

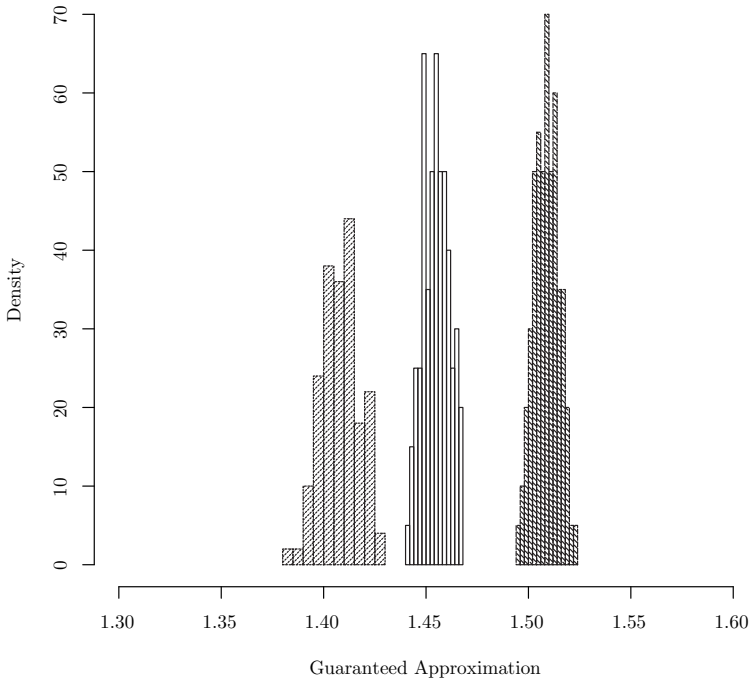


Figure 3.8: Guaranteed approximation of Affine-DO for random sequences. In the left substitutions=1, $a = 0$, $b = 2$, in the center substitutions=1, $a = 0$, $b = 1$, and in the right substitutions=2, $a = 1$, $b = 1$. These are representative of the distributions observed in the experiments.

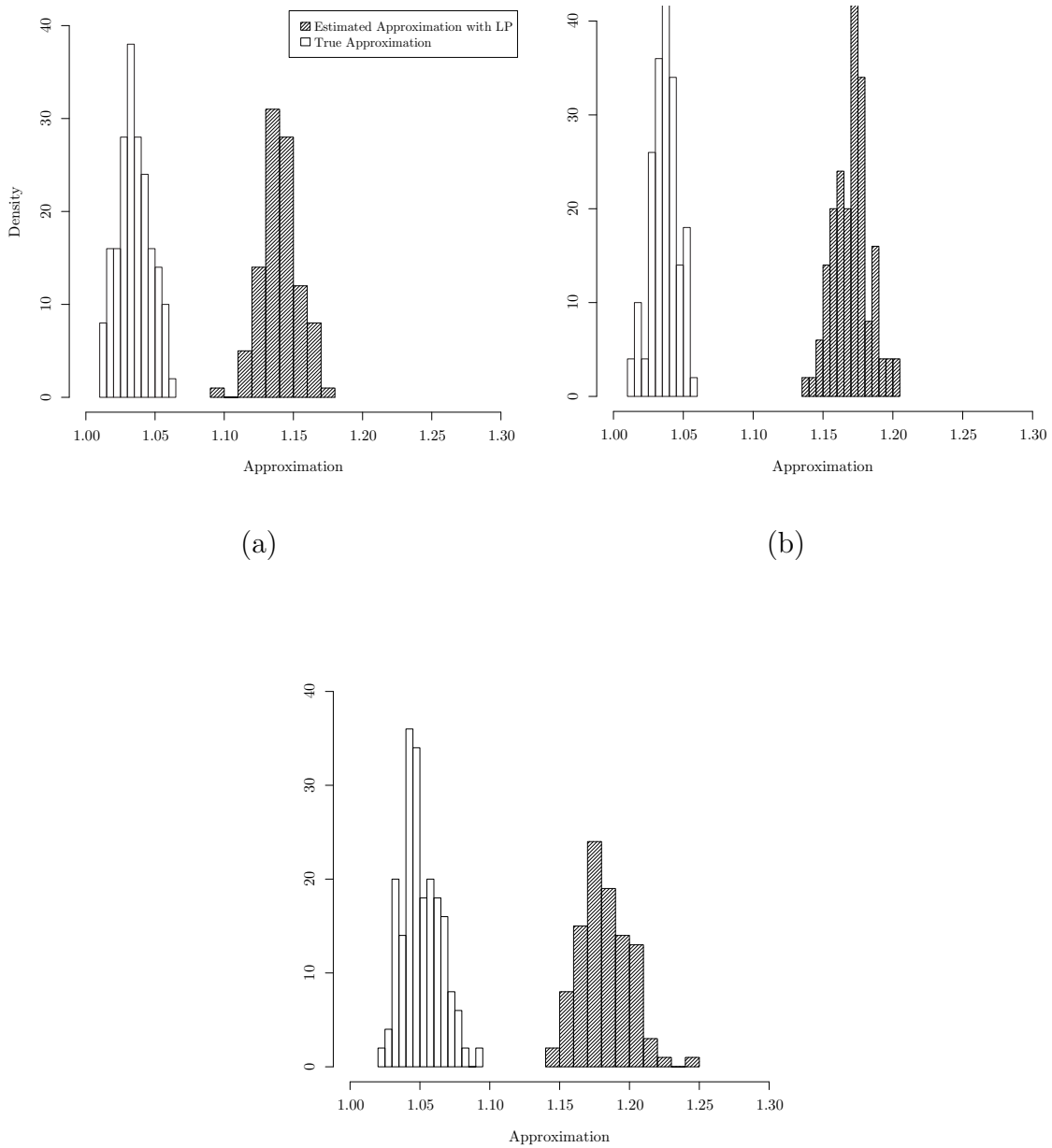


Figure 3.9: Tightness of the Affine-DO solution according to the LP bound compared to the exact approximation. Observe that even for a very small data set, the LP bound is not realistic, and Affine-DO is close to the optimal solution. **a.** substitutions = 1, $a = 0$, $b = 1$. **b.** substitutions = 2, $a = 1$, $b = 1$. **c.** substitutions = 4, $a = 1$, $b = 3$.

Chapter 4

Local Search for the Generalized Tree Alignment

In Chapter 3, I explored the problem of estimating an alignment given a tree under the MP criterion. The problem of *simultaneous tree and alignment estimation* under MP is known in combinatorial optimization as the *Generalized Tree Alignment Problem* (GTAP) [95]. The GTAP is the Steiner Tree Problem, for the sequence edit distance, and like many biologically interesting problems, the GTAP is NP-Hard [41]. A subproblem of the GTAP is the TAP, see Chapter 3. Typically the Steiner Tree is presented under the Manhattan or the Hamming distances. We will refer to these two forms generically as the STP.

In the previous chapter I presented experiments comparing a number of algorithms, and found that Affine-DO yields the best tradeoff between runtime and solutions' optimality (tightness) for the most common edit distance functions in the TAP. The only theoretical results on the GTAP are the general 2-approximation results applicable from the STP [23]. Experimentally, the accuracy of the GTAP has been subject to evaluation [83, 71, 76]. The most recent results have shown evidence

that phylogenies selected using the GTAP from unaligned sequences are competitive with the best two step methods [71, 76].

Due to its hardness, biologists interested in the GTAP rely on heuristic procedures to find good solutions. The simplest, and arguably the most important heuristic for the GTAP is a *local search*. A local search, iteratively evaluates trees similar to a current solution T, where similar trees constitute the neighborhood of T. If a shorter tree S is found in the neighborhood, then T is replaced by S, and the search continues. Otherwise, T is the final solution. Local search is the working horse of most phylogenetic analysis procedures of practical use, and the core search procedures to solve the GTAP in the computer programs MSAM [136], and POY [131, 112]. It is known that the quality of a GTAP analysis is heavily dependent on the fit of the local search heuristics used [71], but the question of which heuristics are a better fit under what conditions remains unanswered.

In this chapter, I discuss, implement, and explore experimentally existing and new local search heuristics for the GTAP using simulated data. My methods improve by more than three orders of magnitude the best local search heuristics existing to date with real data. I begin by formally explainint the existing heuristics (Section 4.1.1), and new heuristics for the GTAP (Section 4.1.2). Following the results of [112], we use the Affine-DO algorithm to compute the tree length heuristically.

4.1 The Algorithms

A subproblem of the GTAP is the Tree Alignment Problem (TAP) (Definition 13). Heuristically solving the TAP with Affine-DO requires a $O(n^2|V|)$ computation (Theorem 4), where n is the maximum sequence length, and typically $n \gg |V|$. To simplify notation, in this section we assume that calculating the assignment of a vertex in a

tree is a constant time operation (i.e. the score of a tree is computed in $O(|V|)$ time).

4.1.1 Existing Heuristics

A local search consists of two steps: initial tree construction, and refinement (defined below). Given an initial tree T, refinement evaluates trees similar to T, in the search for a better solution. Those trees similar to T are its neighborhood. The most commonly used neighborhood function is known as *Tree Bisection and Reconnection* (TBR) [103]. TBR is based on two simple tree modifications: breaking an unrooted tree in two components, and joining two separate trees in one (Figure 4.1):

Tree Breaking. Given a tree T, remove an edge (u, v) to produce two connected components, one with u, the other with v. If u (v) is not a leaf, then collapse it.

Tree Joining. Let $T = (V, E)$ and $S = (V', E')$ be two binary trees. T and S can be joined by selecting a pair of edges $(u, v) \in E$ and $(u', v') \in E'$, create subdivision vertices x in the edge (u, v) and x' in (u', v'), and add the edge (x, x'). If T (S) does not have edges, but only one vertex v, then take v as x (x').

The TBR neighborhood of T is the set of trees that can be produced by breaking T at any edge to produce two trees U and V, and then joining U and V. This neighborhood is used in the local search step of the GTAP solver programs POY [131, 112] and MSAM [136].

The most popular strategy for the initial tree construction is the *Wagner algorithm* [35], a randomized, greedy strategy, of time complexity $O(|V|^2)$, used in most software packages for phylogenetic analysis under MP (e.g. [106]), including POY [131, 112]. MSAM takes a different approach, by using a Neighbor Joining

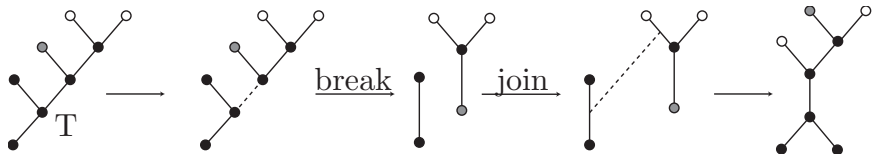


Figure 4.1: Breaking a tree in two connected components, and joining them again with a different edge. The resulting tree is part of T's TBR neighborhood.

tree, with time complexity $O(|V|^3)$ [136]. Deterministic algorithms are not typically used in the tree building step: for non trivial data sets, a good randomized method can be used repeatedly to initiate independent refinements resulting in different solutions. Their shared properties can give insights into the problem's structure, and help discovering better solutions.

Depending on the distance function, different procedures are used to compute the score of the trees in the TBR neighborhood efficiently [138, 133, 47, 46, 48, 124, 128, 44]. In particular, for the Hamming and Manhattan distance, to calculate all of the tree scores in the TBR neighborhood has time complexity $O(|V|^3)$ [103]. For the GTAP however, it has time complexity $O(|V|^4)$ [57, 124, 96, 127], or $O(|V|^3)$ by increasing the hidden factor from $O(n^2)$ to $O(n^3)$ (remember that typically $s \gg |V|$) [96, 128].

Exploring a neighborhood requires two additional criteria: the stopping rule, and the selection of the next candidate solution. Depending on their properties, a number of local search strategies can be described. A classic heuristic that specifies the stopping and selection criteria is *simulated annealing* (SA) [66, 8, 140]. Contradictory conclusions about the applicability of SA to phylogenetic analysis can be found in the literature [86, 107, 48, 8, 140]. A form of simulated annealing with better performance under the Hamming and Euclidean distance is known as *Tree-Drifting* [48]. However, its Metropolis and stopping criteria make Tree-Drifting inapplicable to the GTAP.

The potential of Simulated Annealing for the GTAP has remained unexplored.

Sectorial search [48] is a heuristic that restricts or extends the TBR neighborhood by only breaking and joining selected subtrees (i.e. connected subgraphs), or exhaustively solving such subtrees. Two variations of this scheme have been proposed: in the Random-Based SS, subtrees are selected uniformly at random. In the second variation, the Consensus-Based SS, given a parameter $0 \leq n \leq 1$, only rearrange (or evaluate exhaustively) subtrees occurring in at least $n * m$ solutions found in m previous searches (n typically set to 0.85) [48].

Other strategies (e.g. Parsimony Ratchet, Tree Fusing, the Genetic Algorithm, DCM), do not strictly belong to the set of local search heuristics. Given that local search is part of all these strategies, then all of them would be more efficient if a good local search is in place.

4.1.2 New Heuristics for the GTAP

In this section, I describe four ideas to improve the local search strategies in the GTAP: efficient tree length calculation during the search, better tree cost bounding, a smarter local search strategy, and initial tree building algorithms.

4.1.2.1 Efficient Tree Updates

To apply the selection and stopping rules during TBR, it is necessary to calculate the tree length after every break, and join. Affine-DO requires a *directed* tree as induced by its root (Algorithm 1). If the sequence edit distance function is metric, the true tree length is independent from the root location. Given that metric distances are a common requirement under MP we assume from now on that the edit distance is metric. It follows that, although Affine-DO can produce a different tree length for

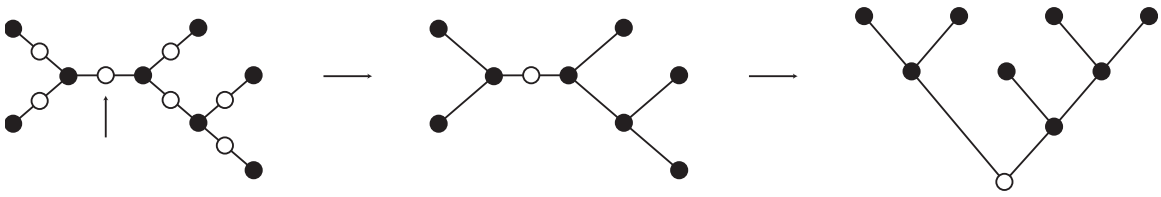


Figure 4.2: All possible roots of the unrooted tree correspond to the subdivision vertices of its edges (empty circles).

each possible root, there is no constraint to maintain one.

To efficiently update a tree, we do not maintain a unique rooted representation, but rather take its unrooted representation and keep all the potential roots assigned to every edge of the tree (Figure 4.2). We call this a *three directional assignment*. Although we describe it for its application for the GTAP, it is applicable to any algorithm that requires pre-order traversal to compute the tree length. (I have used it successfully under the breakpoint [98], inversion [64], and double cut and join [134] distances.)

Three Directional Assignment For an unrooted binary tree, we assign to each edge (u, v) a sequence. This sequence is the Affine-DO assignment to the subdivision vertex w of (u, v) (i.e. Affine-DO(w), Algorithm 1). Computing Affine-DO(w) is dependent on the assignment to its neighbors (Figure 4.3, center). In a binary tree, each interior vertex has three incident edges. Therefore, there are three possible Affine-DO assignments for every interior vertex (i.e. vertex v in Figure 4.3). Each assignment is required to compute some subdivision vertices. Hence, I maintain the three possible assignments for each interior vertex. These assignments can be computed with time complexity $O(|V|)$, using first a pre-order traversal then followed by a post-order traversal, starting on any edge.

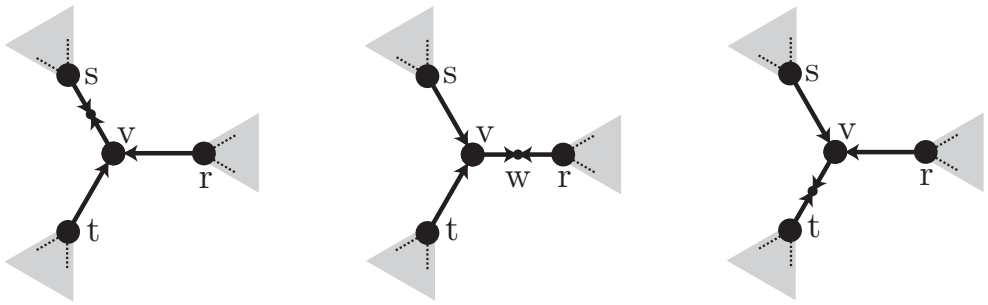


Figure 4.3: Three possible assignments to interior vertices of an unrooted tree. Left: computing the subdivision vertex of (s, v), or any edge rooted by s (grey triangle on s), would require to compute the assignment to v using those of t and r. Center and right: similarly, the assignment of v could be computed using s and t, or t and r. Each direction is needed for some subdivision vertices.

Observation 3. *A tree with a three directional assignment computes the length of every tree that can be produced by breaking any one edge with time complexity* $O(|V|)$.

Observation 4. *Given two separate trees* S *and* T *with the three directional assignment, computing the length of all the trees produced by joining every pair of edges in* S *and* T *has time complexity* $O(|V|^2)$.

The simplest implementation of the three directions is to eagerly compute all the assignments in preparation for the first tree break, and join. However, such an algorithm would entail overhead for greedy heuristics such as simulated annealing, where the first acceptable tree should be choosen to continue with the local search.

We solve this problem by using lazy evaluation and memoization [84] as follows: eagerly assign a lazy function to each vertex and edge of the tree, but only compute its value (and the values it depends on) upon request, while memoizing the result. In this way, we only spend time computing each vertex if used. This technique has greater value if the tree break, and join order is carefully chosen (e.g. Section 4.1.2.3).

In the following section, we will see how the three directional assignment can also be used to improve the estimation of each tree cost with no additional time complexity.

4.1.2.2 Multiple Heuristic TAP Solutions

The Affine-DO algorithm may calculate different tree length bounds depending on the root location (i.e. one per subdivison vertex). Nevertheless, the best of all the assignments is preferable for each tree. Computing all of the Affine-DO tree lengths, however, would add a $O(n)$ time complexity multiplicative factor to each tree break and join. I avoid such factor and still produce better bounds for the tree cost during the search by using Algorithm 2 on each break, and Algorithm 3 on each join of the local search.

Data: A tree T with assigned length l
Data: A lazy Affine-DO assignment to all subdivision vertices of $E(T)$
Data: An edge $(u, v) \in E(T)$ to break, with subdivision vertex w
if *Affine-DO*$(w) < l$ **then**
    $l \leftarrow$ Affine-DO(w);
proceed to break (u, v);

Algorithm 2: Improving the bound of a tree on each edge break.

For a fixed n, the join procedure adds only a constant multiplicative factor, without increasing the time complexity. Note that if all the edges of a tree T are broken during a local search, then $2n - 3$ alignments are evaluated for the final tree, with no additional time complexity. I call this variation of the TBR *Exhaustive-TBR*.

4.1.2.3 Smarter local searches

Affine-DO defined a compact representation of sets of sequences: the Restricted Alignment Graphs (RAGs) (Section 3.1). Ultimately, Affine-DO is a method to compute

Data: A tree T created by joining two separate trees S and R.
Data: S and R have a three directional assignment
Data: The new edge $(u, v) \in E(T)$ created to join S and R.
Data: $n \in Nr$ is the maximum distance parameter
Result: the estimated length of T using Affine-DO
$l \leftarrow \infty$;
foreach $(s, t) \in E(T)$ *at distance less than n from* (u, v) *following a BFS that starts in* (u, v) **do**
    Assume that t is closer to (u, v);
    $w \leftarrow$ subdivision vertex of (s, t);
    if *Two directions of t have not been updated after the join* **then**
        Update the new two directions of t;
    Assign the corresponding sequence computed in Affine-DO(w) to w;
    if *Affine-DO*$(s) < l$ **then**
        $l \leftarrow$ Affine-DO(s);
end
return l;

Algorithm 3: Improving the bound of a tree on each join.

the distance between the closest sequences contained in a pair of RAGs efficiently.

RAGs can be used to guide a local search. If the union of a pair of RAGs A and B can be efficiently computed in a new RAG C, then C can be used to bound the distance between any other RAG D and A or B simultaneously. Therefore, it is possible to use the union of multiple RAGs assigned to multiple vertices in a tree, to compute a lower bound of the closest pair of sequences contained in a pair of vertex sets (Figure 4.4).

Theorem 1. *Let* $R = merge(|X|, |Y|, |M|, \langle\rangle)$ *(Algorithm 4). All the sequences contained in* X*,* Y*, and* M *are contained in* R*.*

Proof. At each step, either X_i, Y_j, M_k, $\{indel\}$, or any of their combinations is prepended to the result. Therefore, no element appearing in X, Y, or M is missing in R. Moreover, for all $0 < e, f \leq |X|$, X_e is prepended before X_f if and only if $e < f$. Hence, the relative order of the elements in X is maintained in R. Finally, for

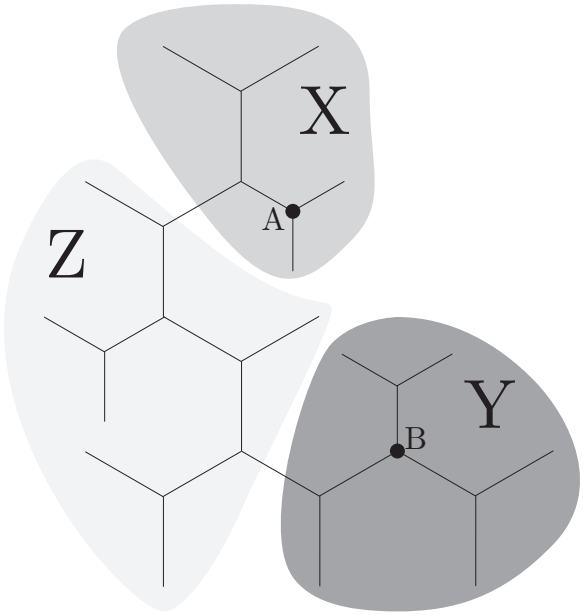


Figure 4.4: Use of unions to bound the cost during a local search. Shade areas enclose disjoint sets of vertices in the tree. Suppose that we merge all the RAG's of each vertex set using Algorithm 4 to produce the unions X, Y, and Z. Then we can heuristically bound $d(A, B)$ as $d(X, Y) \leq \min_{A \in C, B \in D} d(A, B)$, where d is the distance as calculated using the Affine-DO alignment algorithm.

Data: The operation $(a, b) : r$ prepends the pair (a, b) to the list r.
Data: A and B are a pair of aligned RAGs with median M.
Data: X and Y are the unions associated with A and B.
Result: $merge(|X|, |Y|, |A|, \langle\rangle)$ computes the union Z associated with M
if $i > 0$ *and* X_i *flag is false* **then**
  $merge(i - 1, j, k, ((X_i, \text{false}) : result))$;
else if $j > 0$ *and* Y_j *flag is false* **then**
  $merge(i, j - 1, k, ((Y_j, \text{false}) : result))$;
else if $k > 0$ **then**
  $flag \leftarrow M_k \neq \{indel\}$;
  if $A_k \neq \{indel\}$ *and* $B_k \neq \{indel\}$ **then**
    $merge(i - 1, j - 1, k - 1, ((X_i \cup Y_j \cup M_k, flag) : result))$;
  else if $A_k \neq \{indel\}$ **then**
    $u \leftarrow X_i \cup M_k \cup \{indel\}, flag$;
    $merge(i - 1, j, k - 1, (u : result))$;
  else
    $u \leftarrow Y_j \cup M_k \cup \{indel\}, flag$;
    $merge(i, j - 1, k - 1, (u : result))$;
  end
else return $result$

Algorithm 4: Algorithm to compute $merge(i, j, k, result)$. The union of a single RAG A is $\langle A_i, true\rangle$.

Data: A pair of trees S and R produced by breaking a tree T
Data: A set of unions U containing the assignment of the vertices of T
Result: A tree which is a heuristic TBR local optimum
0 **foreach** $Y \in U$ **do**
  foreach $Z \in U$ **do**
    if *There exists vertices y and z such that* $y \in Y, y \in V(S), z \in Z, z \in V(R)$ **then**
      if *The distance between Y and Z is less than* $1.17 \times (length(T) - length(S) - length(R))$ **then**
        Attempt all the TBR joins on edges incident in vertices of Y and Z;
        if *A better tree T' is found* **then**
          $T \leftarrow T'$;
          Update U with the assignment of T';
          Goto line 0;
  end
end
return T

Algorithm 5: Heuristic Union-pruning TBR. The threshold 1.17 parameter was experimentally tuned.

all the cases where X_i is not prepended, then the *indel* element is included in R. It follows that that we can recover X by removing those elements in R where such indels where inserted and no element of X was. By the definition of sequences contained in a RAG [115], it follows that every sequence in X is contained in R.

The analysis of Y and M is symmetric. □

Theorem 2. *Algorithm 4 computes the union of X, Y, and M with time complexity $O(|X|)$ where X is the longest union.*

Proof. The algorithm stops when $i, j, k < 1$. At each recursive step, either i or j is reduced by one, with initial values $i = |X|$ and $j = |Y|$. □

The union of RAGs can be executed in $O(n|V|)$, on each vertex, during the Affine-DO computation. Affine-DO is $O(n^2)$, therefore, this method entails a small additive

factor to the time complexity of Affine-DO. In our implementation, we have fixed the size of the vertex sets to 12 vertices on all data sets experimentally.

Using unions during a local search Let T be the current candidate solution during a local search, and U the set of unions of T by applying Algorithm 4 while traversing the tree in Affine-DO. If a new candidate tree S is accepted during the local search, then update U using the direction for the best subdivision vertex computed for S (i.e. the one that bounds S with the lowest length). By maintaining this set of unions, I can modify the TBR local search as in Algorithm 5, to join only edges that are incident in unions at short distance. I call this method Union–pruning.

4.1.2.4 Building the initial trees

The Wagner algorithm is a basic procedure to compute an initial tree (Algorithm 6). I modify this procedure in two ways.

Data: A sequence $L = \langle l_1, \ldots, l_n \rangle$ of trivial trees corresponding to the leaves.
Data: A tree T, initially empty
Result: A tree such that every element in L is a leaf.
for $i = 1$ *to* n **do**
  $c \leftarrow \infty$;
  foreach T' *produced by joining* l_i *and* T **do**
    if $c >$ *length of* T' **then**
      $T \leftarrow T'$;
      $c \leftarrow$ length of T';
  end
end

Algorithm 6: The Wagner algorithm for initial tree building.

Union–pruning Unions can be used to efficiently prune candidate trees during the wagner algorithm by maintaining the union set of the tree T in Algorithm 6, and

treat each leaf to be added as a union of its own. Then use Algorithm 5 to guide the join step in Algorithm 6.

Addition sequence The initial sequence L in Algorithm 6 is typically randomized, assigning equal probability to every permutation. This algorithm is known as Random Addition Sequence (RAS). The randomization of L is used to obtain multiple starting points for local searches. We have explored the following variation successfully:

1. Compute a Minimum Spanning Tree (MST) of L (i.e. the set of leaves).
2. Traverse L using a BFS. The order in which we visit the elements of L is our initial addition sequence $Q(0)$.
3. To produce the n'th tree, produce the sequence $Q(n)$ by flipping consecutive elements in $Q(n-1)$ with probability 0.5.

I call this procedure *MST-Wagner*.

4.2 Experimental Evaluation

I experimentally evaluated a number of algorithms for local searches under the GTAP. An experimental evaluation of this kind has three fundamental components: a selection of heuristics, implementation, and selection of data sets. The overall performance is compared with the score of the trees found by each method.

4.2.1 Algorithms compared

We compared the following heuristic local searches, in all meaningful combinations. *TAP Computation*: Using Affine-DO in two variations, Exhaustive, and Non-exhaustive.

Building: Wagner algorithm using RAS and MST addition sequences, and the Neighbor Joining (NJ) algorithm. The Wagner algorithm was executed with lookahead parameters of 1, 2, 4, and 10. *Neighborhood*: TBR and SPR (a subset of TBR). *Edge breaking order*: randomized, or in length decreasing order. *Join order*: randomized, or in ascending order based on the distance of the union that each edge belongs to. In the second case, the Union-pruning strategy was used to filter candidates. *Sector and reroot diameters*: 2, 3, 5, and infinity (i.e. no sector). The rerooting order followed a breadth first search (BFS) order, around the broken edge. The sector and reroot diameters were selected to match the simulation size (50 leaves). *Simulated annealing*: using initial temperatures of 2, 5, and 10, and coefficients of 12, 50, 250, and 500. The values were selected experimentally as a good sample of the performance variation observed by the authors in real GTAP problems.

For the edit distance parameters we tested the following combinations of substitution, indel, and gap opening parameters: (1, 1, 0), (1, 2, 0), (2, 1, 1), (3, 1, 2). In our experience, these parameters encompass enough variation in the GTAP, while maintaining a limited number of combinations with the algorithms. In total, 34 combinations of build algorithms and distance functions were tested. For the refinement step, a total of 208 combinations of algorithms and edit distance functions were tested.

4.2.2 Implementation

I implemented the algorithms under comparison in the Objective CAML and C programming languages. All the algorithms are available in the author's computer program POY version 4 [112]. The functions are highly optimized for performance.

4.2.3 Data Sets

To generate the instance problems, I simulated sequences using DAWG 1.1.1 [17] with insertions and deletions following a power law distribution. The simulations followed random binary trees of 50 leaves comprising all the combinations of the parameters listed in Table 4.1. These produced a total of 30 independent simulations. Each simulation was analyzed independently with 100 repetitions for each randomized algorithm. NJ was tested only once, as our implementation is deterministic. An initial exploration with 300 repetitions showed no significantly difference compared to 100 repetitions. In total, 102000 builds, and 624000 refinements were performed. Due to the large number of simulations and local searches performed, we will concentrate on a minimal set of cases that represent the overall patterns observed. (The complete results can be found in Appendix B.)

Parameter	Values Evaluated
Substitution Rate	1.5
Average Branch Length	$0.1, 0.2, 0.3, \infty$
Max. Gap	$1, 2, 5, 10, 15$
Root Sequence Length	500

Table 4.1: Simulation parameters. All combinations of parameters were employed to generate the test data sets. The branch length variation equals the average branch length.

4.2.4 Results and Discussion

This section begins with the difference in performance between the Exhaustive (E) and the Non-exhaustive (NE) algorithms, which can be applied in conjunction with any other search strategy (Section 4.2.4.1). It continues with a comparison of the

build algorithms (Section 4.2.4.2), and the refinement algorithms (Section 4.2.4.3). Finally I compose the results in a simple local search heuristic which we compared with the previous best heuristic on a real dataset (Section 4.2.4.3).

4.2.4.1 Exhaustive and Non-exhaustive algorithms

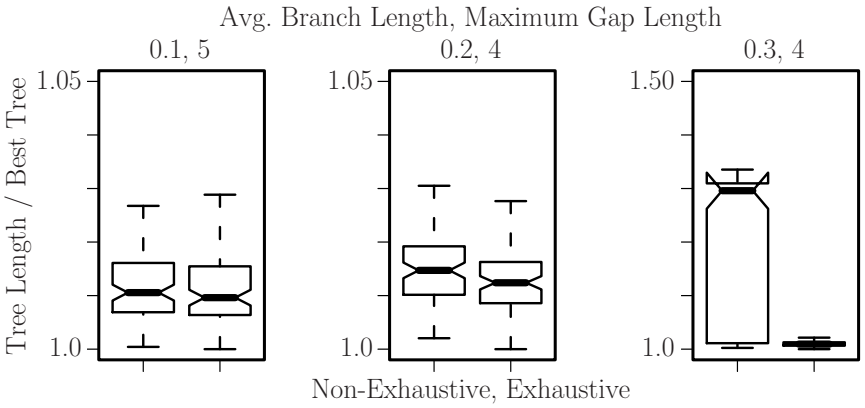
In the build step (Figure 4.5a), the difference between E and NE is small for all equivalent algorithms with branch lengths of 0.1 and 0.2 (Figure 4.5a, left and center). The most striking difference, however, occurs for branch length 0.3 (Figure 4.5a, right), where NE shows an expected tree length 50% higher than that of E. Such extreme variation shows a strong dependence on the root location when branch lengths make sequences close to random relative to each other.

For the TBR step, E significantly outperforms NE, with better minimum and expected scores (Figure 4.5b). This pattern was observed for every combination of algorithm, simulation, and edit distance parameters. In the following two sections, we concentrate on the results obtained using the E algorithm. The same general patterns were observed with NE, but with less competitive tree scores.

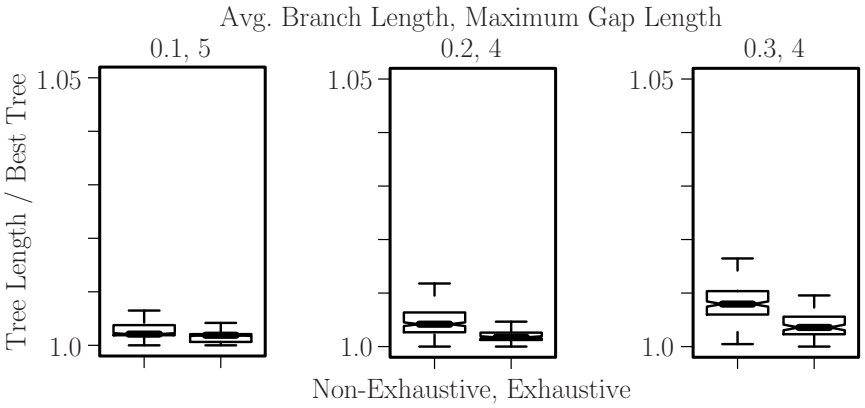
4.2.4.2 Initial Tree Building

The initial tree building algorithms fall into two main groups: algorithms with RAS, and algorithms using MST. In all cases, MST produced significantly shorter trees (Figure 4.6). The use of higher lookahead parameters did not produce consistent improvements in the resulting trees, while the use of the Union-pruning algorithm did significantly improve the expectation, and the minimum tree cost for branch lengths 0.1 and 0.2. For long branch lengths, however, no significant improvement was observed.

Neighbor joining produced trees of highest score among all the algorithms for all



(a)



(b)

Figure 4.5: Comparison of the Non-Exhaustive (NE), and Exhaustive (E) TAP approximation algorithms in tree building (Figure a), and TBR (Figure b). The patterns showed were observed in most of the combinations of simulation, algorithm, and edit distance parameters. **a.** Tree building using the Wagner algorithm. In every case, E outperformed NE, but the difference is not significant. However, as the branch lengths increased, the performance of the NE algorithm showed high variability (right), making E highly competitive for all distance functions with average branch length 0.3. **b.** Refinement using Union-pruning with NE and E. In this case, for almost every combination of algorithm, simulation, and distance function, E produce significantly shorter trees.

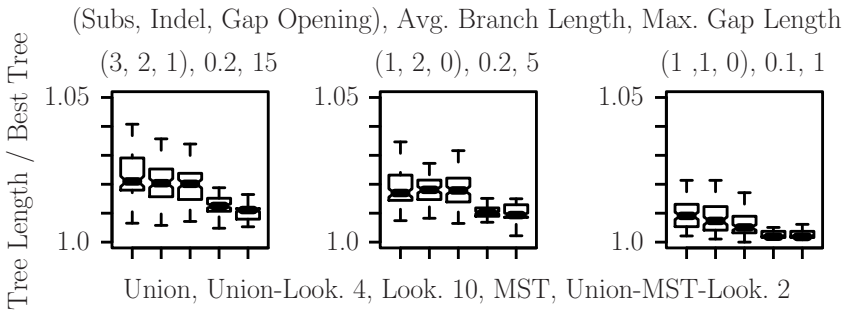


Figure 4.6: Comparison of initial tree build algorithms. *Union* is the Wagner algorithm + RAS + Union-pruning. *Union - Look. 4* is the Wagner algorithm + RAS + Union-pruning + Lookahead of at most 4 trees. *Look. 10* is the Wagner algorithm + RAS + Lookahead of at most 10 trees. *MST* is the Wagner algorithm + MST sequence, but no Union-pruning. *Union-MST-Look. 2* is the Wagner algorithm + MST sequence + Union-pruning + Lookahead of at most 2 trees.

parameters (i.e. the worst, between 10% and 20% higher). We do not present it in the graphs as it would make the more subtle differences between other algorithms difficult to observe. Overall, the most important improvement occurs with the MST addition sequence in first place, followed by the use of the Union-pruning strategy in second. Nevertheless, we will see in the next section that the use of the MST algorithm remains limited.

4.2.4.3 Refinement

To evaluate the TBR refinement experimentally, we must produce an initial tree. Although MST showed better results than RAS, we found that in almost every instance TBR failed to improve the MST trees. At the end, RAS + TBR would always find better trees than MST + TBR. For this reason, we used the second best method to construct the initial trees: RAS using Union-pruning.

The refinement comparison can be divided in two groups: 1.) a comparison be-

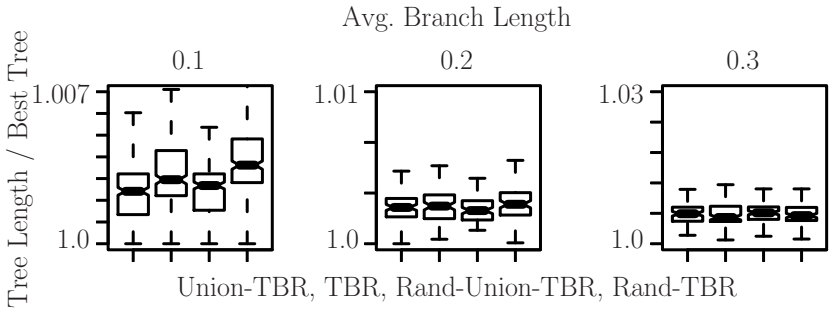


Figure 4.7: Comparative performance of Union-pruning, and branch length sorting, with randomized algorithms in TBR. *Union-TBR* is the length sorted edge break + Union-pruning. *TBR* is length sorted edge break + randomized edge break and edge join ordering. *Rand-Union-TBR* is a randomized edge break + Union-pruning. *Rand-TBR* is randomized edge break and edge join.

tween basic TBR using Union-pruning, and branch length sorting, and 2.) the comparison of different algorithms using the best combination among those in 1.

Union-pruning and branch length sorting The behavior of TBR with Union-pruning and branch length sorting is presented in Figure 4.7. For short branch lengths, the Union-pruning algorithm produced significantly better trees, both in the minimum and expected scores. This advantage disappears as sequences diverge to close to random (branch length of 0.3) (Figure 4.7 left to right). Branch length sorting had a small positive impact, but not significative.

The results match my expectation: the Union-pruning algorithm can positively guide the search with better taxon sampling. I have observed this behavior in real data sets, where new terminals some times *speedup* the local search.

Local search strategy Beyond the use of Union-pruning, and Exhaustive TAP estimation, the differences among the algorithms compared are not significant (Table 4.2). Although in general Sectorial finds with highest frequency the shortest tree,

the difference is typically less than two length units, compared to the second best algorithm. In general, the algorithm with the best mean is BFS, but again, not significative. However, due to the algorithm design, BFS is the fastest of all.

Gap Len.	Edition Distance			TBR		Sectorial		BFS		Annealing	
	Subst.	Indel	GO	Min.	Avg.	Min.	Avg.	Min.	Avg.	Min.	Avg.
1	1	1	0	7190	7222.75	**7186**	7221.188	7190	**7220.969**	7198	7230.802
	1	2	0	8410	8437.76	8405	**8429.812**	**8406**	8436.865	8416	8457.24
	2	1	1	**14022**	14111.76	14032	14107.58	**14022**	**14096.88**	14031	14144.56
	3	1	2	20089	20236.07	20118	20303.64	**20062**	**20221.83**	20172	20373.85
2	1	1	0	6680	6702.115	**6674**	**6697.76**	6676	6699.719	6687	6713.854
	1	2	0	7969	7992.562	7963	**7989.333**	7969	7990.583	**7967**	8005.479
	2	1	1	12994	13040.67	**12978**	13034.80	12981	**13030.21**	13001	13074.80
	3	1	2	18603	18690.26	**18588**	18716.78	18589	**18678.82**	18629	18785.17
4	1	1	0	**7164**	7190.719	**7164**	**7186.323**	7166	7188.062	7176	7208.594
	1	2	0	**8684**	8719.552	**8684**	**8714.406**	8682	8716.677	8698	8751.26
	2	1	1	**13586**	13652.25	13590	13658.08	13592	**13646.89**	13601	13694.72
	3	1	2	19148	19291.41	19149	19344.61	**19113**	**19283.66**	19209	19448.12
5	1	1	0	7049	7077.542	**7043**	7074.229	7049	**7073.729**	7057	7092
	1	2	0	8692	8716.01	**8683**	8715.5	8688	**8711.104**	8690	8730.646
	2	1	1	**13329**	13389.48	13334	13394.16	13336	**13387.41**	13363	13429.17
	3	1	2	18876	18983.53	**18861**	19027.35	18870	**18974.93**	18930	19091.70
10	1	1	0	7149	7181.74	**7141**	**7174.938**	7145	7176.719	7163	7200.5
	1	2	0	8965	9002.677	**8944**	8993.438	8948	**8992.656**	8979	9020.635
	2	1	1	13200	13271.72	13199	13277.82	**13195**	**13266.54**	13235	13320.24
	3	1	2	**18395**	18557.96	18423	18630.5	18402	**18549.86**	18470	18648.79
15	1	1	0	7162	7194.01	7160	7194.531	**7159**	**7190.542**	7182	7216.719
	1	2	0	9151	9196.552	**9142**	9192.125	9147	**9191.344**	9151	9228.146
	2	1	1	13168	13230.11	13164	13231.83	**13155**	**13217.84**	13186	13271.46
	3	1	2	18194	18350.44	18234	18415.64	**18166**	**18335**	18290	18484.11

Table 4.2: Minimum and average tree score comparison among algorithms using Union-pruning and Exhaustive TAP estimation. The differences observed are not significant. All the simulations shown have branch length 0.3, but similar patterns were observed for branch lengths 0.1 and 0.2. The minima across each row is in bold.

4.2.5 Overall performance

Based on the previous experiments, I prefer a heuristic local search strategy that consists of the following steps: build initial trees using RAS guided by Union-pruning, followed by a refinement step consisting of TBR using the three directional heuristics, Exhaustive TAP, Union-pruning, and cutting edges according to descending lengths. We compared this algorithm (implemented in POY version 4), with that of POY version 3 which uses a one directional algorithm, with randomized TBR steps [124]. Due to limitations in POY version 3's implementation, I only compare an edition

distance with substitution parameter 1, indel parameter 1, and gap opening parameter 0. Based on the results of Chapter 3, and because of the implementation limitation, MSAM was not included in the comparison.

For this comparison, a random subset of 100 published anurans [34] was analyzed. The data set includes 12S rRNA, tRNA valine, 16S rRNA, and fragments of cytochrome b, rhodopsin, tyrosinase, 28S rRNA, and RAG 1, and a small set of 38 morphological, non-additive characters (i.e. Hamming distance model).

To compare the performance of POY version 3 and version 4, we executed 1000 independent repetitions consisting of 1 build, followed by refinement, and reported the resulting tree score. This procedure can be executed in POY 3 with the command: `poy -replicates 1 -seed -1 -maxtrees 1 -nooneasis -minterminals 0 -terminalsfile ranNamesPH.txt *.fas *.ss`. The score of the trees found by each program were plotted in a density histogram (Figure 4.8). The results show that one repetition of our new heuristic in POY version 4 outputs a tree which is expected to belong to the top 15% of the best trees found by this very simple search strategy. To expect a tree within the same percentile using the old heuristic, it would be necessary to run more than 2000 local searches. It follows that the new heuristic is more than 2000 times faster than the previous heuristic of POY 3.

4.2.6 Discussion

I described and implemented new heuristics for the GTAP. I have shown that they find better solutions than previous approaches. I found that a number of conditions affect the fit of the heuristic to the problem: long branch–length data sets can be better analyzed with Sectorial Search instead of the Union-pruning, while Union-pruning yields excellent results in medium, and short branch lengths. Exhaustive-TBR yields

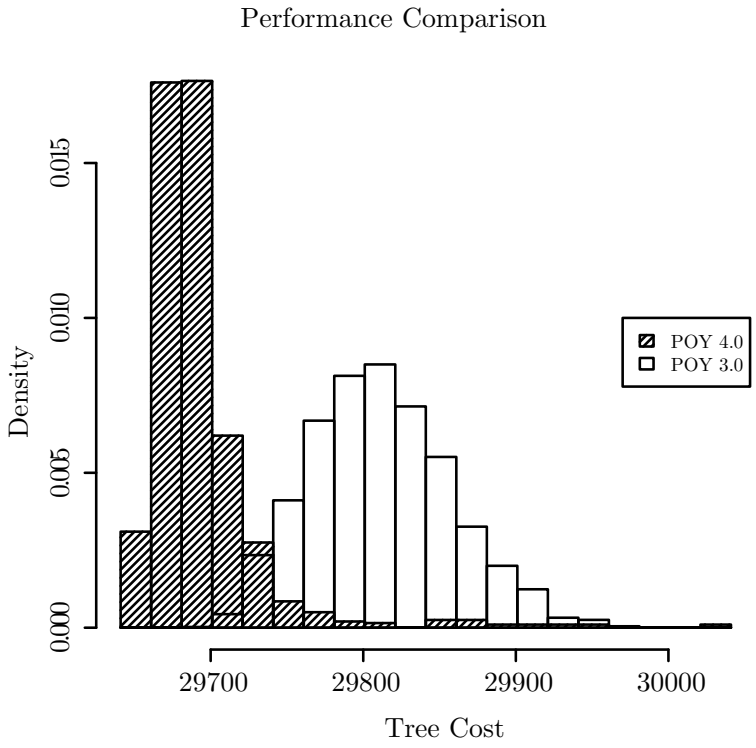


Figure 4.8: Density histogram of the frequency of occurrence of different tree scores in POY version 3 and version 4 for the example data set.

the best results overall and should always be preferred. Although the MST algorithm yields better initial results than RAS, it is not preferable in the long run, and a small number of local searches should never be used to produce reliable results. It remains to be explored the quality of the numerous meta–heuristics available in the literature. It is now possible to explore them using a more efficient local search strategy.

Chapter 5

POY version 4: A Phylogenetic Analysis Program

POY is an open source, phylogenetic analysis program for molecular and morphological data. Version 3.0.11 was released in September 2004 [132]. After more than one year of public beta testing which started early in 2007, versions 4.0, and 4.1 have now been released.

Version 4 supports Maximum Parsimony as its optimality criterion. Like most software of this class, POY analyzes the standard non-additive, additive, and matrix characters, commonly found in other phylogenetic analysis programs [106, 45, 49]. Most importantly, POY supports the analysis of *dynamic homology characters* (DH) which allow the use of unaligned sequences as characters [131]. In DH characters, POY can infer substitutions, insertions, deletions, inversions, and translocations, at the locus, chromosomal, and genomic level, as the phylogenetic analysis goes on. This makes POY a unique application, providing the broadest range of characters for its users.

The main goals of version 4 are to increase the application flexibility (e.g. POY

3.0 only supported one set of parameters for all sequences), increase performance, reduce the learning curve for new users, improve quality control, and maximize the maintainability and extensibility of the source code. Version 4 has been completely rewritten, and does not share code base with POY version 3.0.

In this chapter, I describe the basic features of the program. We begin with POY 4's most important phylogenetic analysis features (Section 5.1), followed by the script execution in sequential and parallel environments (Section 5.2), a number of other relevant application features as well as limitations (Section 5.3), and a list of available resources for current and new users (Section 5.4).

5.1 Phylogenetic analysis features

As with most phylogenetic analysis software, the features in POY can be divided into three groups: calculating the evolutionary distance between a pair of vectors of states, computing the score of a tree given an assignment of character states to its terminals, and searching for a tree of minimal cost. More complex functions are performed by composing elements of these three groups (e.g. support calculation), while others belong to the basic input and output functionality (e.g. printing a consensus tree).

For the most common types of static homology analyses, the first two groups (i.e. distance between vectors of states, and tree score) have well known algorithms, for which efficient polynomial time solutions exist and have been implemented in POY 4. For dynamic homology characters, however, computing a distance and the cost of a tree can be major computational tasks by themselves.

Following these main groups, I describe the phylogenetic analysis features available in POY in a bottom up fashion: first the character types that are supported, then the algorithms for the tree cost calculation (informally), and finally the search strategies.

5.1.1 Supported character types

A character is defined with two components: its valid states, and the function to compute the evolutionary distance between states. Considering the properties of the valid states, two main groups of characters are supported in POY 4: static homology, and dynamic homology. To define them, I must first clarify the notion of state.

5.1.1.1 Character states

We are interested in characters that encompass multiple sources of variation. The following four examples are not exhaustive, but illustrate this diversity:

Morphology A typical character could be the fruit color of a plant. The character states could be red, green, and yellow. Usually, such a set of valid states corresponds exactly to those observed in the taxa of interest. Consider now two possible encoding schemes: non-additive, and additive.

- As a non-additive character (i.e. Hamming distance), the transformation cost between any pair of different states is equal. States that could occur in nature, but where not observed (such as orange), do not have any effect on the score of the phylogenetic hypotheses: if included in the list of acceptable states, it would be ignored throughout the tree cost evaluation.
- As an additive character (i.e. Manhattan distance), the interpretation is different. Suppose now that the systematist chooses to treat the states as ordered conditions in a continuum, for example by coding red as one, yellow as two, and green as three. If orange were later found occurring in the group of interest, it might be preferable to encode the states of the character with red as one, orange as two, yellow as three, and green as

four, producing an alternate cost regime. If not observed, it still would be included in the character coding scheme.

Sequence of loci Suppose now that we are analyzing sequences of loci from the mitochondrial chromosome. For the sake of argument, we assume that all the species in the analysis have exactly the same set of loci. The character is the chromosome itself, and the states are represented by the order of loci; it is not the elements included in each state, but their particular order, which is phylogenetically informative. We can also assume that the locus permutations in our sample do not constitute all the potential states, but a fraction of a much larger set, including all possible permutations (super-exponentially many, that is, $n!$ for n loci). Unlike the morphology example, the mechanisms that could explain such permutations do not include substitutions per se. Instead, the distance between a pair of permutations could be computed using very different mechanisms (e.g. inversions, tandem duplication - random loss). For such a character, the homologies between loci are not tested, but the order in which they occur.

Nucleic acid sequence In this example, a particular locus is the character (e.g. 18S rRNA). The states observed are RNA sequences, that is, words in the {`A,C,G,U`} alphabet. Although we observe only a small fraction of the words, the states that could have occurred in nature include, in principle, all the possible words of this alphabet: an infinite number of states.

Complete chromosome Suppose now that we are interested in the analysis of a complete chromosome from a group of plants. Assume that we have one complete chromosome for each terminal that is believed to be homologous across the group. Moreover, we have annotated those chromosomes such that the limits of

functional units are well established. We will further assume in the analysis that rearrangements, gain, and loss of functional units is possible, but restricted to our predefined limits (i.e. we consider impossible the rearrangement of the two halves of a functional unit). However, the correspondences between functional units is uncertain, and we would like to generate them for each phylogenetic analysis.

Unlike the previous two examples, a chromosome state is not defined by a small, but an infinitely large alphabet. Each functional unit could be, potentially, any DNA sequence. This character is the composition of the previous two examples, where DNA sequences are the elements comprising each character state. We are interested in the insertions, deletions, and substitutions occurring between corresponding functional units, and also in the higher level events that modify the order in which these units occur. Clearly, a huge number of possible states is not being observed, yet must be considered in the character coding scheme if we want to produce a meaningful analysis.

Two characteristics should be highlighted from the previous examples:

1. Not all the states need to be observed to be relevant on the analysis. Depending on the conditions, states that have not been observed may have no (e.g. as in non-additive characters), or a fundamental effect (e.g. additive, DNA genes as described above).

2. A character could have infinitely many states, describing complex entities, such as the order of the elements composing it. Moreover, there could also be infinitely many possible elements.

We say that *a character C is a set of states, where each state is an ordered set of elements from a predefined alphabet* Σ. In our morphological example, $\Sigma = \{$red,

yellow, green}, and the valid states are ordered sets with only one element, that is, $C = \Sigma^1$ ($\langle$red$\rangle$, $\langle$yellow$\rangle$, $\langle$green$\rangle$; a terminal could have multiple states). In the locus sequence example, the alphabet is the set of mitochondrial genes, that is $\Sigma = \{$CO1, CO2, CO3, ATP6, ...$\}$, while C includes all the permutations of the elements in Σ. In this case, every valid state must include all the genes (i.e. an exponential, but finite number of states). In the sequence character example, the alphabet is $\Sigma = \{\texttt{A,C,G,U}\}$, while the valid states are all the sequences that could be created with it, that is, $C = \Sigma^*$ (i.e. infinitely many states). In the chromosomal character example, the alphabet itself is $\Sigma = \{\texttt{A,C,G,T}\}^*$ (i.e. all the words that can be created with $\{\texttt{A,C,G,T}\}$), and the valid states are $C = \Sigma^*$. In this case, the alphabet itself has an infinite number of elements.

We are ready to define static homology and dynamic homology characters:

Static homology characters Let A and B be two states of a character. A *correspondence* between the elements in A and B is a relation between them. We define *static homology characters* as those in which, for every element in A there is at most one corresponding element in B, and the correspondence relations are transitive (i.e. let $a \in A$, $b \in B$, and $c \in C$ be elements of different states, where a corresponds to b, and b corresponds to c; Then a and c must also correspond to each other). Corresponding elements with the same value match the notion of primary homology [30].

Dynamic homology characters We define as dynamic homology characters [126] the complement of their static homology counterparts: for some pair of states A and B, there exists an element $a \in A$ that has more than one corresponding element in B, or the correspondences are not transitive. Dynamic homology characters typically have states that may have different cardinalities, and no

putative homology statements among the state elements. These characters formalize the multiple possibilities in the assignment of correspondences (primary homologies) between the elements in a pair of states, which can only be inferred from a transformation series linking the states, and the edition distance function of choice. A subset of correspondences from dynamic homology sequence character that matches the conditions of statc homology characters (i.e. at most one corresponding element, and transitivity), is what [29] has called comparable bases. (See the definition of sequence characters below.)

In the first two examples, the correspondences are hypothesized *a priori*, and tested in the phylogeny. To illustrate this, in the morphology example, the element red in the state ⟨red⟩ corresponds only to the element yellow in the state ⟨yellow⟩; in the sequence of loci example, the occurrence of the subsequence ⟨CO1,ATP6,CO2⟩ in a state, can only correspond to a subsequence containing exactly those three elements in another state (e.g. ⟨CO2,CO1,ATP6⟩).

In the later two examples, a hypothesis of correspondence between the elements of a state is based on a particular sequence of intermediate states spanning them. In a phylogenetic context, such intermediate conditions are only sound if defined as hypothetical ancestral states of a tree. To illustrate this case, consider the nucleic acid sequence example. Assume that the following pair of sequences are homologous: `AGAGAGAG` and `GA`. To simplify the example, suppose that only insertions, and deletions, could have occurred in the transformation from one sequence into the other. It would be difficult then to define with certainty a set of correspondences between these two sequences prior to a phylogenetic analysis: there are 14 possible correspondence relations between the elements of this pair of states. In static homologies, only one set of correspondences can be selected for the analysis, while under dynamic homologies,

multiple are considered.

5.1.1.2 Static homology characters

POY 4 recognizes five types of static homology characters: Sankoff, Additive, Non-additive, Breakpoint, and Inversion:

Sankoff characters have n valid states, and an $n \times n$ metric distance matrix m such that $m_{i,j}$ holds the distance between state i and state j. The maximum number of states accepted is limited only by the memory constraints of the computer executing POY. Sankoff characters can be loaded from `dpread` files [131], prealigned molecular files, or generated from an implied alignment (see Section 5.3.1). The distance computation between a pair of vectors of states has time complexity $O(n^2)$.

The following two static homology characters (additive, and non additive), are common special cases of Sankoff characters, for which the distance between two vectors of states can be computed in constant time ($O(1)$).

Additive characters allow each state $i \in \mathbb{N}, 0 \leq i \leq 255$, with distance matrix $m_{i,j} = |i - j|$. Additive characters can be loaded from Nona/TNT matrices, or NEXUS files.

Non-additive characters are also known as unordered characters [39]. POY supports up to 30 states in 32 bit architectures, and 62 states in 64 bit architectures. The distance matrix is the hamming distance [55], that is

$$m_{i,j} = \begin{cases} 1 & \text{if } i \neq j \\ 0 & \text{otherwise.} \end{cases}$$

Non-additive characters can be loaded from Nona/TNT, NEXUS files, prealigned molecular files, or automatically generated from the implied alignment of dynamic homology characters when the cost of all substitutions is some constant a, and that of all indels is some constant b (see Section 5.1.1.3).

Breakpoint characters consist of sequences in any user-defined alphabet (known in the POY 4 user interface as custom alphabets). Typically each element in the alphabet corresponds to a homologous locus. The evolutionary distance between these sequences is computed as the breakpoint distance [10]. Formally, given two permutations $A = \langle a_1 \dots a_n \rangle$ and $B = \langle b_1 \dots b_n \rangle$ of elements in some alphabet Σ, we say that every a_i and a_{i+1} are adjacent elements in A (a_1 and a_n are also considered adjacent in circular chromosomes). A pair $x, y \in \Sigma$ is a breakpoint if x and y are adjacent in A but not in B. Given a breakpoint cost c, the breakpoint distance between two sequences A and B is $c\beta(A, B)$, where $\beta(A, B)$ is the number of breakpoints in A (and symmetrically in B). Breakpoint characters can be loaded from *custom alphabet* files [111]. The time complexity to compute the distance between a pair of states is $O(n)$.

Inversion characters consist of sequences in any user-defined alphabet extended with the tilde sign ($\sim$) to represent "inverted" characters, that is, their reverse complement. Typically, each element is a locus, where loci with the same name are homologous. In this notation, $\sim A$ is the inversion of A (i.e. the reverse complement of A) and *vice versa*. The evolutionary distance between these sequences is the inversion distance [15]. Formally, let $A = \langle a_1 \dots a_n \rangle$ and $B = \langle b_1 \dots b_n \rangle$ be a pair of permutations of the same set of elements. An inversion of a subsequence $a_i, a_{i+1}, \dots, a_j$ is $\sim a_j, \dots, \sim a_{i+1}, \sim a_i$, such that $\sim\sim x = x$. Given an inversion cost c, the inversion

distance between the permutations A and B is $c\iota(A, B)$, where $\iota(A, B)$ is the minimum number of inversions required to transform A into B. Inversion distances in POY are computed using the high performance functions of GRAPPA [78]. Inversion characters can be loaded from custom alphabet files [111].

5.1.1.3 Dynamic homology characters

Dynamic homology characters are generically referred to as "molecular" in the POY 4 user interface. Such naming is due to their more common usage with molecular sequences, but the input data need not represent molecular characters. The following dynamic homology character types are supported:

Sequence characters support as valid states any word in Σ^*, from a predefined alphabet Σ (typically $\Sigma = \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$). Sequence characters allow the occurrence of insertion, deletion, and substitution events in the calculation of the edit distance and correspondences of elements implied by each tree (Section 3.

An important difference between POY version 3 and version 4 is the way a non-metric *tcm* is handled. A non-metric *tcm* was *not* supported in POY version 3, and would produce incorrect results and tree lengths. POY 4 supports non-metricity, provided it is caused by a low (but greater than zero) indel cost. The application issues a warning when non-metric *tcm*'s are being used. This feature, however, does not imply that POY 4 somehow avoids trivial alignments when the indel cost is too low (e.g. `AAA---` and `---TAA`). Its main usage is to define a very low indel cost in conjunction with a gap opening parameter (i.e. affine gap costs).

POY 4 also accepts any alphabet: nucleotide (using the complete IUPAC codes, see [75]), amino-acid (a subset of the IUPAC codes, see [75]), and user-defined custom alphabets [111]. Sequence characters can be loaded from FASTA files, NEXUS files

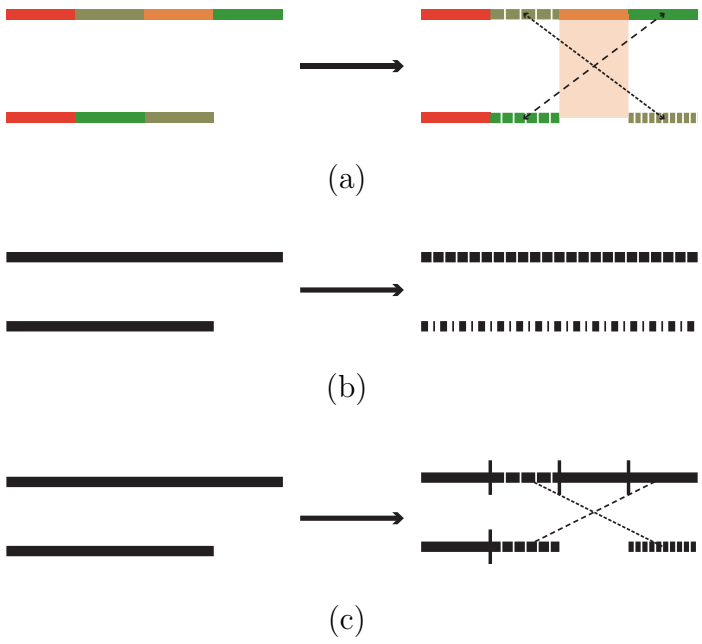
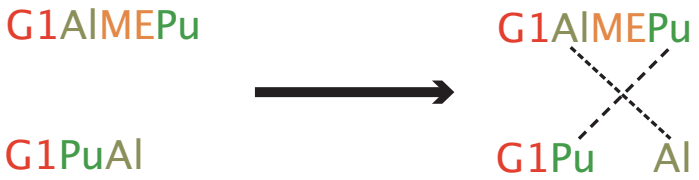


Figure 5.1: Homologies potentially inferred by the different classes of dynamic homology characters, compared to a reference set of transformations. a. Input sequences on the left, and expected homology statements on the right. The sequences present four (upper sequence) and three loci (lower sequence), with indels occurring in the green loci, as well as a locus rearrangement. The orange locus shows an indel event between the two sequences. b. As sequence characters. Insertions, deletions, and substitutions are inferred. For sufficiently complex sequences, the alignment will expand, trespassing the locus "limits". c. As raw chromosome characters. With no user provided limits, POY 4 attempts to infer rearrangements, locus indels, in addition to sequence insertions, deletions, and substitutions. The program attempts to establish locus limits based on conserved segments.

(a)



(b)

Figure 5.2: Homologies potentially inferred by the different classes of dynamic homology characters (excepting genomes), compared to a reference set of transformations. a. As chromosome characters. With user provided limits between loci, POY 4 attempts to infer rearrangements, locus indels, as well as sequence insertions, deletions, and substitutions. The program will not attempt to modify the user provided locus limits. b. As annotated chromosome characters, employing the user provided alphabet to represent homologous loci. Only rearrangements, locus indels, and locus substitutions can be inferred directly by the application.

with the unaligned block, custom alphabet files, and most file formats produced by GenBank. The time complexity to compute the distance between a pair of states of cardinality m and n is $O(mn)$.

Chromosomal characters have as valid states any word in Σ^*, with $\Sigma = \{\texttt{A},\texttt{C},\texttt{G},\texttt{T}\}$. Each element of a state represents a chromosomal fragment, each fragment a nucleotide sequence character itself. Chromosomal characters can detect fragment inversions, fragment rearrangements, and fragment indels, along with the familiar sequence-level insertions, deletions, and substitutions within the segment. The distance computation is done in two steps: a pairwise alignment at the fragment level, under the user-provided parameters, followed by a rearrangement distance computation using the functions provided by GRAPPA [78]. The selection of homologous segments is collaborative work that has been described elsewhere [116].

Genome characters are defined as sets of chromosomes. For this type of characters, there is no implied order for the chromosomes, and therefore, the user input order is irrelevant. POY automatically detects homologous chromosomes, and considers chromosomal insertions and deletions, along with those events occurring within a chromosomal character as described in the previous section. Genome characters can be loaded from FASTA files, where each chromosome is delimited with the @ sign.

5.1.2 Tree cost calculation

Well known algorithms are used for the three most commonly used static homology characters: the cost of trees with Non-additive [39] and Additive [37] characters is computed in $O(nm)$ time complexity, where n is the number of nodes in the tree, and m is the number of characters. The cost calculation for trees with Sankoff charac-

ters [99] has time complexity $O(nms^2)$, where s is the maximum number of character states. These algorithms yield exact tree costs and an optimal assignment to the interior nodes. For Breakpoint, and Inversion characters, the tree cost calculation is heuristically approximated, with an overall time complexity of $O(nm)$, where n is the number of nodes in the tree and m is the cardinality of the breakpoint or inversion states.

The tree cost calculation for dynamic homology characters, that is, sequence, chromosome, and genome characters is at least NP-Hard (e.g. [119]). POY 4 implements a number of heuristic algorithms to bound the tree cost. These algorithms can be divided in two classes: initial assignment to the interior nodes of the tree, and iterative improvement to refine the total cost calculated for that tree.

5.1.2.1 Initial assignment

The initial assignment is similar in spirit to the *down-pass* in static homology algorithms (e.g. [39]). During the diagnosis of an input tree with n terminals, POY 4 computes $2n - 3$ implied alignments, one for each possible root (i.e. the alignments inferred for each possible rooted tree from the initial unrooted tree). From these, the best alignment (i.e. the one yielding the lowest tree cost), is assigned to the tree. Each tree can only have one alignment assigned.

Sequence characters There are three basic algorithms for an initial tree cost calculation in POY 4: Fixed States [125] (similar but stronger than the Lifted Assignment of[121]), Direct Optimization [124], and Affine-DO 3. The first is a 2-approximation method of time complexity $O(n^3)$. As currently implemented, fixed states yields tighter results (i.e. better tree costs) for molecular characters with amino-acid or large user defined alphabets (more than 6 elements), and therefore, is the recommended

heuristic for those character types.

Direct Optimization and Affine-DO have time complexity $O(nms^2)$, where s is the maximum state cardinality. These algorithms yield tighter results for nucleotide alphabets or small user defined alphabets (fewer than 7 elements). Direct Optimization is used when the gap opening parameter is 0, otherwise Affine-DO is employed.

Chromosomal and genome characters Within the chromosomal types, a set of k medians is heuristically selected and maintained at each node, where k is a user provided parameter. With larger k, more medians are maintained. Each median is created using a randomized greedy algorithm, and improved using a local search, rearranging each median to produce a new one of lower cost, until no better can be found [116].

5.1.2.2 Iterative improvement

Once an initial character assignment is performed, POY can iteratively improve the overall tree cost by adjusting the characters of each interior node, based on the corresponding characters assigned to its three neighbors. The adjustment of the characters on each node can occur with two possible methods: using the same techniques of the initial assignment, or an exact three dimensional alignment.

Approximate uses the initial assignment algorithm of each character to pick a better median for each interior node. On every iteration, POY produces three potential medians, corresponding to the three possible directions to compute the initial assignment algorithm [113] (Figure 3.5). This method is supported in all the dynamic homology characters.

Exact performs a complete three dimensional alignment of the three neighbor sequences of an interior node, and creates an optimal median which is the new sequence of the node [97, 127]. This method is supported only in nucleic acid sequence characters.

Both methods can be applied until one of the following two conditions occur: no further tree cost improvement can be made, or a user specified maximum number of iterations is reached. The selected method is applied to all the dynamic homology nucleotide sequences.

5.1.3 Phylogenetic tree search

POY 4 provides numerous algorithms for heuristic searches of the most parsimonious tree. To simplify the exposition, in the time complexity description of the following algorithms we will assume that the computation of the character distances and interior nodes of the trees takes constant time. Due to implementation details, most of the algorithms mentioned below have a $O(\log n)$ overhead factor, where n is the number of terminals. In a modern analysis, this factor is typically small compared to the number of characters and sequence lengths.

5.1.3.1 Initial tree building

Every heuristic search algorithm requires a method to generate the initial set of trees. POY 4 includes three main methods: branch and bound [59], Wagner tree building [37], and minimum spanning tree guided.

Branch and bound This method of tree building provides, in principle, an exact solution to the phylogeny problem [59]. Unfortunately this is only true if the calculation of the tree cost is exact, something that cannot be guaranteed for some character

types. Therefore, if a user builds a tree using branch and bound, the solution is exact up to the goodness of the tree cost algorithm. The overall time complexity of branch and bound remains exponential in the number of terminals, and therefore, it is only recommended for data sets with a very small number of terminals.

Wagner tree The Wagner algorithm [37] uses a greedy strategy to create an initial tree, by iteratively connecting a terminal to the tree in the best position. Due to its greedy nature, the algorithm is sensitive to the order in which the terminals are added. This order-dependency is used as a heuristic to visit a larger portion of the tree space, limited to "sound" trees. By default, when using this algorithm, POY randomizes the terminal addition sequence. The overall time complexity of the implementation of this algorithm is $O(n^2)$.

Minimum spanning tree guided A third strategy available in the application is the use of a minimum spanning tree (MST) [25]. An MST generates a sequence of terminals that can produce better results compared to a single, randomized, Wagner tree algorithm. Unfortunately this method has limited use in real data sets, where the distance between terminals is usually not metric due to polymorphisms and sample errors, and randomization is used with a larger number of repetitions to improve the overall search results. The overall time complexity of the algorithm is $O(n^2)$.

Additionally, POY 4 provides methods to build trees with positive constraints, that is, build trees where certain clades are required to exist. These methods can be applied together with any of the Wagner tree or the minimum spanning tree building strategies previously described. Negative constraints will be supported in a future version.

5.1.3.2 Local search strategies

The local search consists of the iterative modification of a current tree, in an attempt to find a similar tree of better score. POY supports a number of algorithms, classified in the various components that they involve for a local search: neighborhood, trajectory, branch break order, and join method. Additionally, the trees visited during the search can be *sampled* (e.g. to collect trees for Bremer support [14]).

Neighborhood The neighborhood describes those trees that can be evaluated, given the current best tree. These are known as the neighbors of the current best, hence the name. POY supports nearest neighbor interchange (NNI), subtree pruning and regrafting (SPR), and tree bisection and reconnection (TBR) (Section 4). These sets can be limited further using a positive constraint (an unresolved tree that shows clades that must be present in a neighbor). Every neighborhood in POY 4 consists of successive branch breaks, joins, reroots (in TBR), and the trajectory of the search (i.e. the tree that is selected for the next iteration). Each can be fine tuned, as follows:

Branch break order POY includes algorithms to break the branches in decreasing length order (`distance`), fully randomized breaking order (`randomized`), to break only once, and never again, even if the local optimum has changed (`once`). By default, the `distance` method is employed.

Join specifies those branches that can be joined and in what order. The options available include `constraint` to specify either a sectorial search or a tree that constrains possible solutions to the problem, `all` to turn off all the heuristics used by the program to reduce the number of trees evaluated during a local search, and `sectorial` to specify sectorial searches constrained by the subtree size.

Rerooting specifies the roots that can be used during TBR. By default, the order in which roots are visited follows a breadth-first search algorithm on the branches [24], starting at the nodes incident in the broken branch. The number of trees evaluated at this step can be limited with the `bfs` argument, specifying the maximum distance allowed for each new root from the initial root. The distance is defined as the number of branches in the path connecting the new with the original root.

Trajectory specifies how the program selects the next neighboring tree to be evaluated. The default algorithm is a greedy *first best*, which selects the first tree found that has better score than the current best, `around` to completely evaluate the neighborhood before selecting the next local optimum, simulated annealing (`annealing`) [66], which uses a probabilistic function to choose a tree, and tree drifting (`drift`) (a modified version from that described by Goloboff [48, 111].

Samplers As the local search is executed, POY 4 provides various *sampler* methods, to allow users to collect information, either for error recovery, support calculations, or analytical purposes. For instance, all trees that have been visited during a search can be printed out with the `visited` argument.

5.1.3.3 Escaping local optima

Local searches are often not sufficient to generate satisfactory solutions. A number of algorithms exist to escape locally optimum solutions; POY 4 supports two main classes: tree fusing, and search space perturbation.

Tree fusing is described by [48], to find better trees in complex data sets. The basic algorithm consists of selecting pairs of trees uniformly at random, the first is

considered the source, and the second the target. These trees are compared, and for all pairs of compatible subtrees, the subtree in the source replaces the corresponding subtree in the target. (A pair of subtrees is compatible if both contain the same set of terminals, but their topologies differ.) If the best tree resulting from this exchange has a lower score than the target, then this new tree replaces the target. This procedure is repeated for a user-determined number of iterations. The algorithm can be tuned, by selecting a local search strategy to follow the new subtree selection, as well as the number, and algorithm to select trees that are maintained between iterations.

Perturbation is a basic strategy that allows the user to perform a local search (or a series of local searches) on a modified set of characters. The tree space (i.e the space representing the cost of each tree) is therefore "perturbed," and depending on the perturbation method, could help the search by escaping locally optimum trees and finding better solutions. The most notable form of perturbation is the parsimony ratchet [82]. The basic ratchet algorithm consists of perturbing the tree space by reweighting a random set of characters, according to user-provided parameters, followed by a local search, and the resulting tree is used in a new iteration. When the user-selected number of iterations is completed, the search space is restored, and a new local search proceeds. The original tree is replaced with the final only if better. Along with the parsimony ratchet, all the transformations (including those described in Section 5.3.1) are supported as perturbation methods.

5.1.3.4 Search command

POY 4 introduces a new command: `search`. It is intended as a default search strategy for most users. This strategy includes tree building using the Wagner algorithm [37], swapping using TBR, swapping using Exhaustive Direct Optimization [111], Nixon's

Parsimony Ratchet [82], and Tree Fusing [48]. The command supports arguments to specify the maximum or minimum execution time, minimum number of hits before stopping, and the maximum number of trees to be held (measured in memory). The function takes care of removing duplicated trees and reducing repeated effort. Upon completion, it reports the number of trees built, the number of rounds of tree fuse, the best tree cost found, and the number of times that cost was found (hits).

Search is *a recommended way to execute an analysis*. It does not eliminate the user responsibility to ensure that a reasonable tree search is performed for the input data set. It is important to verify that several searches converge to the minimum cost (i.e. maximize the "hits"), and a reasonable number (in the order of hundreds) of replicates is performed (each tree fuse can be considered a separate replicate).

5.2 Script execution

POY 4 accepts files containing scripts for non-interactive execution. A script is a sequence of valid POY 4 commands. The execution of scripts in POY 4 does not necessarily follow exactly the input order specified by the user. Instead, a script is analyzed and modified to achieve the same analytical effort (measured in number of trees evaluated, randomized procedures executed, etc.), while reducing memory consumption, and limiting the amount of information exchanged between processes when executing in parallel.

To understand the script execution better, we must first describe the parallelization strategy used in POY 4, followed by the description of the script analysis and optimization methods employed in the application.

5.2.1 Parallel model

POY 4 supports parallel execution using any implementation of the Message Passing Interface (MPI) version 1.0. MPI has become the most important standard for parallel execution using Message Passing. By using MPI, POY 4 can be executed in parallel under virtually any architecture, from laptops with multiple cores, to computer clusters running Linux, Windows, or Mac OS X.

The parallelization model used in POY 3 consisted of a master-slave model of computation, where one process (the master), directed other processes (the slaves) to perform certain calculations upon request. For instance, if 10 trees were to be built using the Wagner algorithm, and 11 processes were available, then the master would order each of the 10 slaves to perform one of the builds. During most of the computation, however, the master would remain idle, waiting for requests from the slave processes.

The parallel model of POY 3 posed significant scalability difficulties. Even for fast networks, if sufficient processes attempted to communicate concurrently, the master process was a bottleneck, producing sub-linear scalability and even reduced performance under a number of circumstances [61, 132]. To solve this problem, POY 3 included "controller" processes, which could serve as intermediate relays, responsible for managing a smaller number of slaves [61]. Although the scalability limitations could be reduced in this way, the problem remained at a larger scale, while increasing the number of idle processes overall.

POY 4 is fundamentally different in that there is no process directing the computation of any other process. Instead, upon receiving the input script, each process independently decides what tasks it should perform. There exists a master process, which performs the same operations that other processes would, but also central-

izes access to input files when other processes cannot directly (as in some computer clusters), and generates the desired program output (e.g. printing the trees in a file).

The fundamental advantage of this parallelization model is the increased scalability and the reduced volume of communications. Moreover, resources are better exploited, by eliminating an idle process (the master), which can instead spend resources on the analysis itself. It follows that POY 4 can scale even in computers with two cores, as both processes are responsible for part of the complete analysis.

There are two fundamental limitations in POY 4's model: fault tolerance has been eliminated, as have the parallelization of the operations within a tree (e.g. parallel building of a single tree). The former has a lower priority, but the latter will be included in future releases of the application.

5.2.2 Script analysis

A script analysis consists of three steps: dependency analysis, memory optimization, and parallel execution division.

5.2.2.1 Dependency analysis

In the first step, POY 4 analyzes the data dependencies between different components of a script. For example, the calculation of the jackknife support value information is independent from the search for the most parsimonious tree (but not assigning the support values to the shortest tree found). POY 4 evaluates mutual dependencies in input files, output files, trees, jackknife frequencies, bootstrap frequencies, and bremer supports, to produce a dependency graph that describes how commands relate to each other.

5.2.2.2 Memory optimization

Once the dependency analysis is completed, POY 4 classifies each command in the script into one of four classes that allow the application to optimize their execution:

Parallelizable is a command that can be executed in parallel. Examples of commands of this class are `build` and `swap`.

Composable is a command that can be applied composed over intermediate results, yielding exactly the same output as if it was applied once over all the results directly. For example, selecting the shortest tree among 10 trees has the same effect as selecting the best tree among the first two, then select the best between the result of the previous selection and the third tree, and so on until all the trees are evaluated. `select (best)` is an example from this class.

Linearizable is a command that can be applied independently with subsets of results, yielding the same effect as applying it to

Non-Composable are commands that cannot be parallelized, and sets hard limits in the way a script is executed. An example of this class of commands is `report (treestats)`.

Script execution is modified in the following manner: parallelizable, composable, and linearizable commands can be modified to improve performance, conforming to *pipelines*, while non-composable commands break the pipelines. To understand how these pipelines are formed, we will illustrate them using an example.

Figure 5.3 shows a script that can be described as follows: read an input file, build 1000 trees, swap each until its local optimum is found, redraw the screen, select the best trees and filter out duplications, report the remaining trees to the screen in

graphical format, and quit the application. If executed in this way, at peak memory consumption, POY 4 would require enough memory to hold 1000 trees.

```
read ("file")         (* Non-Composable *)
build (1000)          (* Parallelizable *)
swap ()               (* Parallelizable *)
redraw ()             (* Linnearizable  *)
select ()             (* Composable     *)
report (graphtrees)   (* Non-Composable *)
quit ()               (* Non-Composable *)
```

Figure 5.3: A POY 4 script, with comments showing the type of each command.

If we look at the same script considering the class each command belongs to, a different picture emerges. The core of the script is parallelizable, linearizable, and composable. It follows that this script could be executed more efficiently in the following way: read the input file, and repeat 1000 times the following three steps: build one tree, swap, redraw the screen, and select the best trees in memory. Upon concluding the 1000 repetitions, report the remaining trees in memory on screen in graphical format, and quit the application. Overall, POY 4 will only use as much memory as the maximum number of shortest trees found at the same time. For most real data sets, this will tend to be a small number.

Notice that the 1000 iterations involve a sequence of four commands. Each sequence is the "pipeline" mentioned two paragraphs above. The user interface updates the overall script execution progress, and estimates termination time for the set of pipelines instead of individual commands.

5.2.2.3 Parallel execution division

Notice that in the previous example, each pipeline can be executed independently from the others, with the results being merged by the composable elements of the pipeline. Pipelines are the script components that are parallelized by POY 4.

If the previous script is executed in parallel with 1000 processors, each processor would have taken care of a single pipeline, and the selection of the shortest trees would have followed with only 11 ($\lceil \log_2 1000 \rceil$) rounds of messages between processors.

The general rules for parallelization are as follows:

1. Only the master process can print to files or screen.

2. Pipelines and support calculation pseudo-replicates are divided among all processes. If there are m processes, and n pipelines, each process does at most $\lceil n/m \rceil$ pipelines, to complete exactly n.

3. All processes synchronize execution at the end of each pipeline.

Using this strategy, the application shows linear scalability in the number of processors and number of trees evaluated (Figure 5.5). The exact execution strategy of a particular script can be verified using the `report (script_analysis:"script.poy")` command (Figure 5.4).

5.3 Other features

There are many other new features in the program. The following are several highlighted functions.

```
beginning of the program
read an input file
I will calculate the following in separate processors (if available)
processor group 1:
  in parallel:
    build some trees from scratch
    swap the trees in memory
    while keeping the following invariant:
      eliminate repeated trees
      select the optimal trees
  eliminate repeated trees
  select the optimal trees
processor group 2:
  redraw the screen
  output the trees in memory
  close POY
```

Figure 5.4: Analysis of input script in Figure 5.3 as generated by POY 4 using the `script_analysis` report.

5.3.1 Transformation between character types

POY 4 supports functions for the easy transformation of character types. For example, suppose a user would like to run an analysis for one complete day, select the best tree, fix the alignment of the dynamic homology sequences implied by the best tree found, and compute the jackknife support values using the characters inferred by that alignment. In this example, we will assume that the program found only one tree during the analysis. Another example would show the effect that different alignment parameters may have on the implied alignment of a particular tree (Figure 5.6).

An important effect of the `static_approx` transformation shown in Figure 5.6 is the way it is performed for other distance functions. For instance, assume the user has defined an affine distance function, with cost 2 for every substitution, 4 for gap opening, and 1 for gap extension. (The total cost of an indel of length 3 would be 7 in this example.) After performing `transform (static_approx)`, the program will

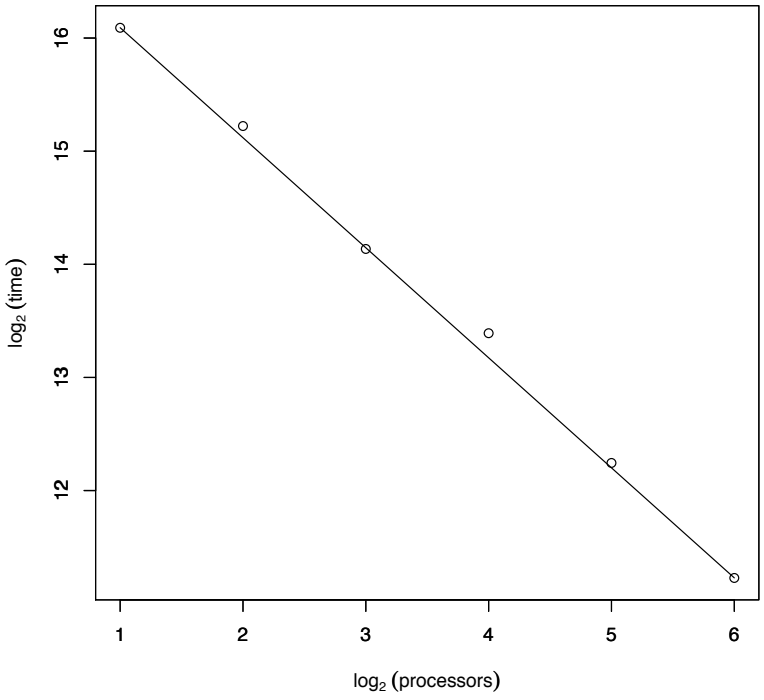


Figure 5.5: POY 4 scales linearly in parallel execution. In this example, 64 RAS+TBR where tested with 1 to 64 processors in parallel. The speedup is linear in the number of processors, with a slope of ≈ 0.9.

```
(* We read three genes from one fasta file, and a tree  *)
read ("18S.fasta", "input.tree")
(* We will use substitutions and indels with cost 1 *)
transform (tcm:(1,1))
(* Output the implied alignment on screen *)
report (implied_alignments)
(* Repeat the procedure with various alignment parameters *)
transform (tcm:(2,1))
report (implied_alignments)
transform (tcm:(3,1), gap_opening:2)
report (implied_alignments)
transform (tcm:(6,1), gap_opening:5)
report (implied_alignments)
exit ()
```

Figure 5.6: Comparing the effect of various alignment parameters in the alignment implied by the same phylogenetic tree and the same locus.

```
swap ()
redraw ()
exit ()
```

Figure 5.7: The redraw command to refresh the screen contents. It would have the same effect as executing it once after all the trees have been swapped, or each time a tree is swapped. This type of command yields a greater execution order flexibility.

create characters corresponding to the individual columns of the implied alignment, representing the substitution events, and characters of different weight representing the individual indel blocks as inferred from the best tree in memory. For each indel block character, the program stores as the state names the sequence block that was inserted (or deleted) (e.g. an inferred insertion of the block `AACTTG` will have state names `AACTTG` and - representing the presence of the block, and its absence). All the characters can then be exported in Nona/TNT format for use in other programs, and generate the apomorphy list (using the command `report (phastwinclad)`).

5.3.2 Input file formats

POY 4 supports the data and tree input and specification of Nona/TNT files, NEXUS files [77], all GenBank sequence formats (the most commonly used is FASTA), as well as CLUSTAL file formats [108]. The program honors character and state names in the input, and all reports use them accordingly. This ensures that the user will be able to employ the application output more efficiently for publication.

5.3.3 Graphical output

Along with newick formatted trees, POY 4 can output graphical trees in PDF format, allowing modification in vector image editors, for screen and print layout.

5.3.4 Support calculation

POY 4 supports Bootstrap, Bremer, and Jackknife support calculations. It is important to note that Bootstrap and Jackknife will resample characters as listed by the command `report (data)`: that is, if dynamic homology characters are loaded, the characters themselves are sampled (e.g. the sequences), and not the elements within

each state (e.g. the bases).

5.3.5 What the program cannot do

Although flexible, POY 4 has a number of limitations. The following are the most important functions that the program cannot do, yet users may assume.

Automatically detect non-homologous sequences At the input level, the user provides a set of states that are believed to be homologous, (e.g. fragments from the 5S subregions for all the species). For this reason, *in dynamic homology characters, any pair of elements from homologous states can be hypothesized homologous.* This assumption has two main implications:

1. The program cannot detect incomplete sequences and treat them as such. If the sequences are incomplete, the user may follow one of the procedures suggested by [131] to treat sections as missing data, or simply accept that this will be a source of noise.

2. If a pair of sequences are random relative to each other (for example when reverse complements included in the analysis), the program will do something with them, no matter how little sense that may make. The software is designed to help, but it is not a goal keeper.

Automatically select alignment parameters Parameter selection is an important problem in phylogenetic analysis. POY 4 provides functions to apply different parameters to each character, and assign default parameters in every analysis, but users must be careful in this respect.

Detect inversions, and breakpoints in the same character The functions provided in version 4 can detect either inversions, or translocations, or rearrangements, in a particular character, but neither two nor three can be detected simultaneously on the same character. However, POY 4 can simultaneously analyze multiple characters, each of a different type, and using different parameters. It is possible then to detect inversions in one character, while rearrangement, or translocations are detected in others during the execution of a combined analysis (see Section 5.1.1.3).

Maximum likelihood POY 3 supported phylogenetic analyses using Maximum Likelihood (ML) as optimality criterion. This optimality criterion is not supported in POY versions 4.0 to 4.1.2. However, ML is currently in development and will be supported again soon in a future release of the software.

5.4 Program resources, availability, distribution, and license terms

5.4.1 Obtaining the program

The released versions of POY 4 can be freely downloaded from `http://research.amnh.org/scicomp/projects/poy.php` as binaries and source code. The bleeding edge development version and bug tracking system can be found at `http://code.google.com/p/poy4/`.

Binaries for sequential and parallel execution are available for Microsoft Windows XP, Vista, and Mac OS X Tiger and Leopard (Universal binaries). Binaries for sequential execution in Linux x86 are also available for download. For all other architectures, users can download the source code and compile themselves. POY 4 is

highly portable, and has been successfully compiled in Linux AMD-64 and Itanium2, AIX, Sun OS, and Solaris, both for sequential and parallel execution. For parallel environments of individual workstations or computers clusters, it is necessary to use any of the available implementations of the Message Passing Interface Standard v 1.0 (MPI 1.0).

5.4.2 License

POY 4 is open source, distributed under GPL v. 2, written in OCaml and C. Inversion and Breakpoint distance functions provided by GRAPPA [78] (`http://www.cs.unm.edu/~moret/GRAPPA/`). PDF generation provided by Camlpdf of Coherent Graphics (`www.coherentgraphics.co.uk`). The three dimensional alignment functions of versions 4.0 and 4.1 are provided using software of David R. Powell [87], (`ftp://ftp.csse.monash.edu.au/software/powell/README.html`).

Chapter 6

Kolmogorov complexity phylogenetic analysis

Morphological, anatomical, molecular, and developmental characters are some of the rich set of character types used in biology to postulate phylogenies. Each kind of character has opened a growing number of philosophical, mathematical, and computational questions. One of these questions is: how to select among models that can explain differently the evolution of a character? This problem can be illustrated with the use of chromosomes as phylogenetic characters.

In its simplest interpretation, a chromosome is a sequence of genes (e.g. $\langle 1, 2, 3 \rangle$, where 1, 2, and 3 represent different genes), such that each gene has a unique function, and is easily distinguishable from other genes (not an uncommon assumption, e.g. [26]). In some chromosomes, genes are highly conserved, while the gene order is variable (e.g. mitochondrial chromosomes [98]). According to these observations, a pair of homologous chromosomes A and B can be described as a permutation of each other. Suppose for the sake of argument that we would like to test the hypothesis that chromosome A is an ancestor of B. To do so, we must select a model to hypothesize

and score the transformations from A to B, versus the hypothesis where A is not B's ancestor. There are many possible models to explain such variation, including (but not limited to) inversion [98], signed inversion [56], double cut and join [134], and tandem duplication random loss [11]. For each model, a different hypothesis could arise. However, these models (and their respective hypotheses), cannot be directly compared. Hence, a biologist cannot objectively prefer any one based on A and B only. Model selection remains problematic even in simpler settings (e.g. when only substitutions are allowed; see [91] for an overview).

To compare models in MP, the only method proposed is a sensitivity analysis [123]. In ML, scores computed for different models are non-comparable, although external criteria have been proposed [42, 1, 100]. Bayesian methods provides a possible solution, by using the BIC [100]. The situation is exacerbated in published research, where multiple methods are typically used, and their relative advantages and disadvantages cannot be quantified. In this section, I propose a different optimality criterion using the Kolmogorov Complexity criterion (KC) [74], defined by the function K.

Definition 17 (Kolmogorov Complexity [74])**.** *Let $x \in \{0,1\}^*$ (a binary string). The Kolmogorov Complexity of x, $K(x)$, is the length in bits of the shortest Turing machine that, with no input, outputs the binary string x.*

Suffice to say that a Turing machine is a universal computing device, i.e. the theoretical foundation of multipurpose languages and processors [28]. Under the KC optimality criterion, we are interested in the hypothesis H to explain the data D, such that

$$K(D|H) + K(H) \tag{6.1}$$

is minimized. In the approach to phylogenetic analysis proposed in this chapter, D is the set of observed character states (e.g. the sequence of a particular gene on each

organism), while H is the model and underlying phylogeny (e.g. affine indels).

This section begins with an overview of Kolmogorov Complexity inductive inference (Section 6.1), followed by a description of its use in phylogenetics, including models applicable in the analysis of complete chromosomes (Section 6.2). Then an experimental evaluation is described (Section 6.3), followed by a discussion of the results and future work (Section 6.4).

6.1 Compression as hypothesis selection criterion

Let D be the set of possible observations in some experiment. Assume without loss of generality that $D \in \{0,1\}^*$ (i.e. D is a binary string). Suppose that we wish to compress optimally (hitherto unknown) outcomes of this experiment. A simple counting argument shows that, if the goal is to efficiently encode a binary sequence of length n, then no matter what description method is used, only a fraction of at most 2^{-k} can be compressed by more than k bits [52]. This observation is generalized by the following theorem:

Theorem 3 (No-Hypercompression Inequality [52])**.** *Let P^* be the probabilistic source of data D. The probability that a code* not *corresponding to P^* compresses D substantially more than K bits than the code corresponding to P^* is less than 2^{-K} (i.e. negligible).*

Example 5. *The number π can be estimated to any precision level by using Monte Carlo methods. A circle of radius one and a square of side two are drawn with center in the same coordinates of a Cartesian space. Dots are randomly drawn in the square, and those which fall in the circle, are counted. The total area of the circle is πr^2 that is π when $r = 1$. Therefore the ratio of dots falling in the circle should be approximately*

$\frac{4}{\pi}$. *The more dots used, the more precise the ratio, and the estimation of* π. *Note that* π *passes any statistical test of randomness. Nevertheless, it has a clear pattern that allow us to describe it and compute it succinctly.*

Theorem 3 says that only very peculiar binary strings can be significatively compressed. Information theory tells us that the expected compression of a binary string is optimal if the distribution that produced the sequence matches the distribution used to compress it [27].

Theorem 5 (Entropy [104])**.** *Suppose we have a set of possible events whose probability of occurrence are* $P = \{p_1, p_2, \ldots, p_n\}$. *Then the information content of* $p_1, p_2, \ldots, p_n$ *(i.e. the amount of uncertainty in the outcome of an event) is*

$$H = -K \sum_{i=1}^{n} p_i \log p_i,$$

where K *amounts to a choice of a unit of measure. We call* $H(P)$ *the* Entropy *of* P.

Example 6. *Let* $K = 1/\log 2$ *(i.e. bit are units). Suppose now that we wished to transmit such message in some alphabet* $\Sigma = \{A, B, C, D\}$, *where* $P(A) = 0.1, P(B) = 0.2, P(C) = 0.3$, *and* $P(D) = 0.4$. *Therefore, in order to optimally transmit a message, we expect to use* $H(P) = 1.846$ *bits. If all the elements in* Σ *had equal probability, then* 2 *bits would be expected (i.e. there would be more uncertainty).*

Intuitively, D is not peculiar relative to P^*, but random. Hence, it cannot be described more succinctly based on any peculiarities. Observe that there is a fundamental relation between compression and expectation. The closer a hypothesis H is to the true source P^*, the better will H compress observations from D on average [104].

Compression can be used to objectively measure the fitness of a hypothesis to explain and predict observations [74]. In historical sciences (e.g. phylogenetics), ob-

servations always occur before an analysis, and so, can be arbitrarily compressed with hindsight. If some $x \in D$ has already been observed, then it is possible to assign it the shortest code, attaining maximum (but meaningless) compression. Furthermore, it is not possible to evaluate a hypothesis by repeated experimentation and resampling.

One of the great ideas of Solomonoff, Kolmogorov, and Chaitin, was to use a universal computing device, as an objective language to describe patterns in the form of *a pair of code and decoder* [105, 19, 67]. By including the decoder, an arbitrarily short code cannot be used to describe x. The joint complexity of code and decoder becomes an objective measure of x's compressibility. In Solomonoff's inductive theory, the *complexity* of a number is proportional to its *compressibility*: the more complex, the less compressible the number is. Rewording Definition 17, the Kolmogorov complexity $K(x) = \ell(x^*)$ is the *length in bits of the shortest program x^* that outputs the sequence x when executed in a Turing Machine.* The more x can be compressed, the more is known about x's properties.

Let $P(H) = 2^{-K(H)}$ be the prior probability of some function H (e.g. a probability generating function). P defines the *universal probability distribution* $\mathbf{m}$ over all possible H. Turns out that $\mathbf{m}$ is consistent, and inversely proportional to the degrees of freedom of H [74]. These two properties make $\mathbf{m}$ ideal for inductive inference. Unfortunately, $\mathbf{m}$ has limited use in practice, as the following theorem shows.

Theorem 6 (Halting Problem [28])**.** *There is a Turing machine with alphabet $\{0, 1\}$ that has an unsolvable halting problem.*

That is, $\mathbf{m}$ is non-computable. Due to this limitation, $\mathbf{m}$ can only be approximated, either by eliminating constant factors (as proposed in the Minimum Description Length (MDL) principle, see [51]), or by explicitly using a Turing complete language to describe and bound the complexity of H [74].

Definition 18 (Turing complete language)**.** *A language is Turing complete if it can simulate a non-deterministic Turing Machine.*

That is, any language at least as powerful as a Turing machine can be used. A natural question is: what is the effect of selecting a particular language to describe a hypothesis? The following theorem shows that it is an additive term.

Theorem 4 (Invariance theorem)**.** *For all Turing complete languages a and b,* $K_a(x) = K_b(x) + c$*, where c is independent of x.*

Phylogenetic analysis under the MDL principle has been proposed in the past [22, 90]. Due to the complexity of the hypotheses under consideration, and the limited application that MDL has for the kind of distributions of interest in phylogenetics, I am interested in exploring the second approach only. Similar approaches using compression have been proposed for sequence comparison and alignment [2, 3, 4], compression as a distance function coupled with distance method (such as neighbor joining) [73, 85], and also as an information–theoretic formulation of the MP optimality criterion [5]. In this chapter, I explore the use of KC in numerous models of phylogenetic analysis in a broader sense, and explore its performance with simulated data.

6.2 Kolmogorov Complexity Phylogenetic Analysis

Suppose that a biologist collects a set of character states S from a number of organisms (e.g. the sequence of genes in the mitochondrial chromosome of each organism). As specified in Equation 6.1, the KC hypothesis selection criterion consists of a two part code: H, and a binary string S encoding D. In my approach, H includes decoders built according to the model (e.g. inversion as transformation mechanism),

and underlying structure (e.g. a binary phylogenetic tree), necessary to compute D (i.e. H compresses the patterns inferred from D). On the other hand, S encodes the random components of D, using the encoding that can be interpreted by the decoders in H.

Example 7. *Suppose that a hypothesis is a particular sequence of signed inversions (defined below) between two chromosome permutations A and B. In a KC hypothesis, D is the pair A, B. H contains an initial chromosome decoder X (to reproduce A), and a signed inversion decoder Y (to generate B given A and a sequence of signed inversions). Finally, S encodes the necessary instructions for X and Y to generate D (i.e. the pair A, B).*

6.2.1 Description Language

To compute K, it is necessary to select a Turing complete language (Definition 18). There are an infinite number of Turing complete languages that could be used for with this purpose. Although Theorem 4 tells us that the language is asymptotically irrelevant, hypotheses are constructed with relatively small samples, making this selection important in practice. I recognize two desirable conditions on the language choosen:

Small. It must have a minimal number of symbols. A large number of symbols, or operations, can easily introduce bias in preference for certain computations.

Machine independent. Languages could be attached to a particular machine representation, which is clearly irrelevant in biological hypotheses. For example, the C programming language requires a memory layout, as well as mutable variables. For this reason, I prefer a *functional* language [7] to estimate the complexity of our hypotheses.

One Turing complete language that fills both of properties is the SK computation model [7]. SK supports only two operations: S, and K defined as follows:

$$\begin{aligned} \mathrm{K}xy &\rightarrow x \\ \mathrm{S}xyz &\rightarrow xz(yz). \end{aligned}$$

Only three symbols are required: S, K, and the grouping symbol '('.

Example 8 (Booleans and Conditional Branching)**.** *True and false can be represented using K and SK respectively. In this way,*

$$KAB \rightarrow A,$$

and

$$SKAB \rightarrow KB(AB) \rightarrow B.$$

Example 9 (Recursive Functions: Fixedpoint Theorem)**.** *Althought not obvious, the SK language supports recursion through the Fixedpoint Theorem. Let*

$$F = SSK(S(K(SS(S(SSK))))K).$$

Then

$$Fg \rightarrow g(Fg).$$

In the KC phylogenetic criterion, the two part code can be laid out by concatenating H and S (Figure 6.1). The SK interpreter is left as a constant factor that will be ignored in the remainder of this document.

Each machine will be specified using a subset of the Objective Caml language [72]. To calculate the complexity of an SK machine in bits, I implemented an SK compiler that takes each specification, and produces an equivalent SK program whose length can be counted. In each SK machine, I represent each '(' with 0, S with 10 and K with 11.

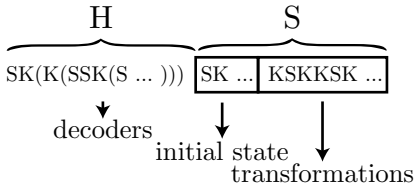


Figure 6.1: Structure of a KC hypothesis.

6.2.2 Complexity of H

A typical phylogenetic hypothesis in MP, ML, and MAP, consists of a phylogenetic tree, and properties associated with each tree edge (e.g. branch lengths, or a sequence of transformations). The model is not evaluated as part of the hypothesis, the mechanism is rarely under discussion, and the underlying binary–tree structure is not tested. Under KC, all these components are included in H. Therefore, the complexity of H is part of the overall hypothesis score.

6.2.2.1 Structure of the Decoders in H

In the SK language, there are no references or pointers (i.e. there is no internal memory representation). For this reason, all the computation is *purely functional* [7], where an input string is read and processed, to recursively compute D.

I will approximate $K(H)$ with an SK machine construction that is realizable using

zeroes and ones (e.g. if $\Sigma = \{A, B\}$, and $P(A) = 0.25$ and $P(B) = 0.75$, then both would use exactly one bit in the machine realization, regardless the difference in probabilities). In this way, I can bound the model's complexity from above, while maintaining comparability between models. To compute $K(D|H)$, I will use a real valued encoding complexity, that is not realizable (e.g. in the previous example, I will count $-\log_2 0.25 = 2$ bits to encode A, and $-\log_2 0.75 = 0.42$ bits to encode B). Such counting scheme will allow me to have a finer analysis of the data itself, with an unknown fixed error between models. To simplify notation, I define $\lg x = \log_2 x$.

One constraint of the SK machine scheme shown in Figure 6.1, is that the decoders must be able to process S as an input stream. Therefore, every decoder must output a function that, by itself, can continue processing the remainder of S. Instead of simply computing a value, functions should pass control to another function upon termination. This form of computation is known as *continuation passing style* (CPS), which I use in a simplified form (e.g. Figure 6.2.2.1).

To fill the CPS requirement, each model implementation is defined in the form $fcx_1x_2\ldots$ where f is the function, c is a continuation function, and $x_1x_2\ldots$ are arguments. The function f will typically be recursive, such that, the fixed point of the recursion calls back c, with the input arguments unmodified.

In Section 6.2.2.2, the underlying structure of the data (i.e. binary trees or phylogenetic networks) is described. Section 6.2.3, I describe a number of possible models to generate D (e.g. affine indels, inversions). For each model, a proper encoding for S, will be described. This encoding is used to compute $K(D|H)$, and in the experimental evaluation of the method (Section 6.3).

```
let rec add c x y =
    if x = 0 then
        c y
    else
        add c (predecessor x) (successor y)
```

Figure 6.2: Recursive definition of an addition function, using the a simplified form CPS. The function adds recursively $x + y$, and passes control to the function c, which continues with the computation. It is not CPS because the functions `predecessor` and `successor` are not written in CPS. In the final recursive call, I apply `c y`, to continue with the computation.

6.2.2.2 Binary Tree Hypotheses

KC would not be an interesting phylogenetic analysis criterion if it were unable to represent phylogenetic trees. Binary trees are typically *assumed* to be the underlying structure to explain an evolutionary hypothesis. In KC, H can interpret an input string, and reproduce D by resembling inheritance by descent and modification. In this way, H *includes* a model to explain the observed patterns: a binary tree decoder. The precise binary tree needed to reproduce D is encoded in S.

Let `tree_hypothesis` be the name of an SK machine that can process a stream representing a binary tree $T = (V, E)$. The machine `tree_hypthesis` can use a stack to keep V, and compute the observed states of the leaves in a depth first search fashion [25]. Therefore, we need only to distinguish interior vertices (that should be added to the stack, to continue computing D), and leaf vertices (that should be added to the machine's output). We can optimally do this by using the only two symbols available to represent each: an S represents an interior vertex, and a K represents a leaf. Figure 6.3 shows an example of this encoding scheme for binary trees.

Observation 5. $K(\texttt{tree\_hypothesis}) < 1628$ *bits.*

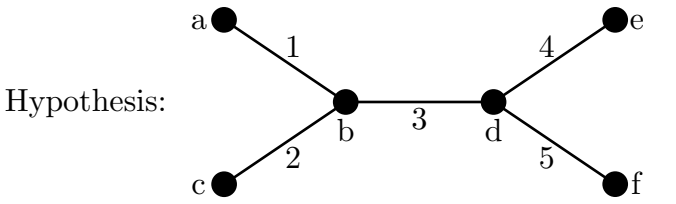


Figure 6.3: Tree hypothesis encoded in S. Each interior vertex is distinguished from leaves with the S and K symbols, respectively. Between each mark, the mechanism transforms the data on the edge, to reproduce the observations in the leaves of the tree.

Proof. See Section C.1.2.1. □

The following theorem shows that we can ignore the complexity of the decoder `tree_hypothesis` in a particular analysis, provided that all the hypotheses under consideration follow a binary tree structure.

Observation 6. *The complexity of the binary tree hypothesis is constant for a fixed D.*

Proof. An unrooted binary tree with n leaves has $n-2$ interior vertices. However, one of the leaves is the starting sequence for `tree_hypothesis` (a in Figure 6.3). Therefore, the total complexity added by the binary pattern is $K(\texttt{tree\_hypothesis})+2n-3$ bits. But we are only interested in comparing hypotheses for a fixed D. Therefore, n is constant, and so, the complexity of the binary tree hypothesis is constant across all comparable hypotheses. □

Nevertheless, binary phylogenetic trees are not the only possible underlying structure of an evolutionary hypothesis. Phylogenetic networks are, perhaps, the most important alternative.

6.2.2.3 Phylogenetic Networks

Genetic information is exchanged constantly among organisms: hybrids, viruses, and symbionts, are all good examples of associations where multiple lineages exchange DNA, creating new traits derived from multiple most recent ancestors. This form of genetic material exchange is known as Horizontal Gene Transfer (HGT). Tree hypotheses (like those supported by the model described in the previous section), cannot produce phylogenies reflecting HGT's.

A phylogeny represents historical events: its vertices hypothesize time dependencies among ancestor and descendants. Accordingly, a phylogenetic network must represent a partial ordering of its vertices (Figure 6.4). To compute D, it is necessary to traverse the vertices of the network in a particular order. The vertex order guarantees that all the necessary information exists prior to the application of the model over an edge. To decode the vertices in the correct order, it would be enough to know, for each vertex, the level at which it can be computed (Figure 6.5). Effectively, each level corresponds to ancestors that *could* have coexisted, without contradicting the network's partial order. A possible encoding of this model is shown in Figure 6.5.

Observation 7. *Let* `network` *be an SK machine capable of decoding the scheme described in Figure 6.5. Then* $K(\texttt{network}) < 10100$ *bits.*

Proof. See Section C.1.2.2. □

Unlike tree hypotheses, the complexity of the network pattern is not constant across all the hypotheses (Figure 6.6). Using the encoding described in Figure 6.5, a

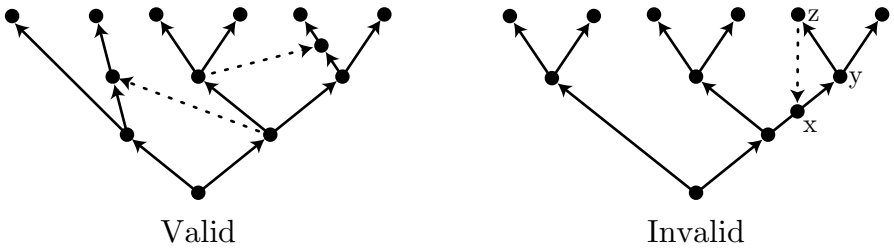


Figure 6.4: Biological constraint of Horizontal Gene Transfer (HGT) hypotheses. The arrows represent ancestor-descendant relations between vertices. An ancestor must occur *before* the descendant. The dotted arrows represent a HGT event. Hence, arrows represent a precedence relation ($\prec$). To be biologically consistent, the network must produce a valid partial ordering. The figure on the left is valid, while the one on the right is invalid; $x \prec y \prec z \prec x$, which is a contradiction.

network with l levels, n vertices, and maximum offset o, has complexity $2 \lg l + (n + 2) \lg o + 10100$.

Comparing two hypotheses, e.g. where HGT is and is not allowed, becomes easy in KC: add their complexity to the hypotheses tested to make them comparable. Under the KC optimality criterion, to allow HGT is a natural extension, permitting the comparison of single-ancestor versus multiple-ancestor hypotheses transparently. In the following sections, I will concentrate on a number of possible models for the analysis of complete chromosomes or genomes under KC.

6.2.3 Model Complexity

In the phylogenetic analysis of genomic and chromosomal sequences, numerous possible transformation mechanisms have been proposed. The mechanisms include (but are not limited to) substitutions, insertions, deletions, tandem duplications (e.g. [33]), tandem duplication – random loss (TDRL) [11], inversions [98], double cut and join (DCJ) [134], and inversions – fission – fusion – translocations (IFFT) [98]. Never-

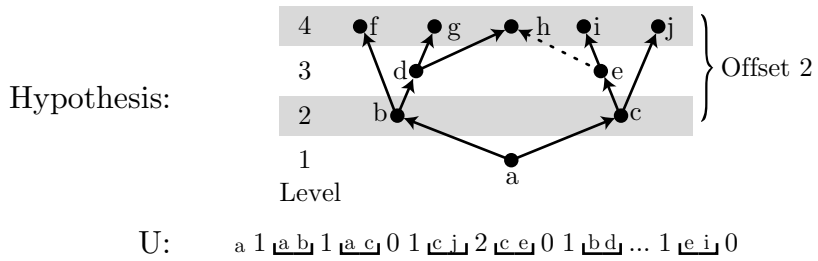


Figure 6.5: Encoding of a network hypothesis. The phylogenetic network define precedence relations between vertices in a partial ordering. The levels define equivalency relations provided no equivalency contradicts the precedence relations (i.e. vertices belonging to the same level are equivalent), and all vertices with outdegree 0 belong to the same level. The union of equivalency and precendence relations implies a total ordering of the levels. The difference between two levels is their offset. Each black edge in the network is encoded as an offset from its ancestor's level, and a sequence of transformations using the hypothesis model. If the offset is 0, then no more black edges are connected to the current ancestor, and the computation continues with the next ancestor. Upon reaching the last edge of the before–to–last level, D has been computed. In this example, U starts with the root of the tree a. Section C.1.2.2 has an implementation of an SK decoder of this encoding scheme.

theless, it is not generally known how important each of these transformations is. Moreover, combinations of all mechanism constitute valid models themselves! To make the exposition simpler, I will describe models that use only one one mechanism at a time. Nevertheless, defining models that would accept combinations of mechanisms is lengthy, and yet a feasible task.

Every model M in Sections 6.2.3.1 to 6.2.3.6 can be composed with the phylogenetic tree model of Section 6.2.2.2, by replacing M's encoding in the segments of Figure 6.3.

6.2.3.1 Substitutions

The simplest of all models of evolution is the substitution-only model. Given an initial sequence $s = \langle s_1, s_2, \ldots, s_n \rangle, s \in \{A, C, G, T\}^n$, at each position $s_i \in s$, with

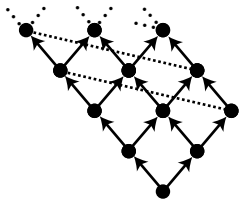


Figure 6.6: Construction showing a network with an unbounded number of interior vertices and levels, yet a constant number of leaves. Given that there are uncountably many networks, although D is fixed, different networks may have different complexities.

probability $P(s_i)$, we replace s_i with one of the elements in $\{A, C, G, T\} - \{s_i\}$. In the simplest model, assume that $P(A) = P(C) = P(G) = P(T) = \alpha$, for some fixed time t. This is the Jukes-Cantor (JC) model of evolution [62].

I define the `jc_substitution` function, which recursively processes a stream encoding substitution events as described in Figure 6.7.

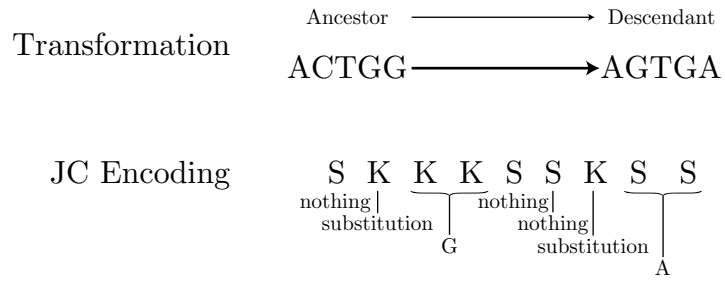


Figure 6.7: Encoding in S of a transformation from the ancestor `ACTGG` into the descendant `AGTGA` using only substitutions. For each element in the ancestor sequence, either nothing happens (S), or a substitution occurs (K). If a substitution occurs, then the corresponding base is specified (e.g. guanine is KK, adenine is SS). The SK machine specified in Figure C.1.4 can process an input matching this specification.

Observation 8. *The probability of n substitutions in a sequence of length $|s|$ using*

the JC model is $\alpha^n(1-\alpha)^{|s|-n}$*, and so, requires* $-\log \alpha^n(1-\alpha)^{|s|-n} = -n\log\alpha - (|s|-n)\log(1-\alpha)$ *bits.*

Observation 9. *Let* `jc_substitution` *be an SK machine for the JC substitution model as described in Figure C.1.4. Then* $K($`jc_substitution`$) \leq 1136$ *bits.*

Proof. See Figure C.1.4. □

We have the following analogous results for the Kimura Two Parameter [65] and the General Time Reversible [68] models:

Theorem 7. *Let* `k2p_substitution` *be an SK decoder for substitutions encoded using the K2P substitution model. Then* $K($`k2p_substitution`$) \leq 2123$ *bits.*

Proof. See Section C.1.4. □

Theorem 8. *Let* `gtr_substitution` *be an SK machine for substitutions encoded using the GTR substitution model. Then* $K($`gtr_substitution`$) \leq 3566$ *bits.*

Proof. See Section C.1.4. □

To apply `jc_substitution`, `k2p_substition`, or `gtr_substitution`, an initial sequence must be computed. Unless more complex models are included, we can only assume that the initial sequence is random. Let $P(x), x \in \{A, C, G, T\}$ be the probability of observing x in a sequence. Then a sequence $s = s_1, \ldots, s_n$ is optimally encoded with $H(s)$ bits.

Observation 10. *Let* `nucleotide_sequence` *be an SK function to decode an initial sequence of nucleotides. Then* $K($`nucleotide_sequence`$) \leq 1600$ *bits.*

Proof. See Section C.1.4. □

6.2.3.2 Substitutions and Indels

Substitutions and indels have been described in Sections 3 and 4. Three variations of the substitution model where described in the previous section, and two indel models where described in Section 3 (affine and atomic). In general, the probability of an indel of length k is $\alpha\beta^{k-1}$. For affine indels set $\beta > 0$, and for atomic indels set $\beta = 0$. An encoding scheme for both atomic, and affine models is shown in Figure 6.8.

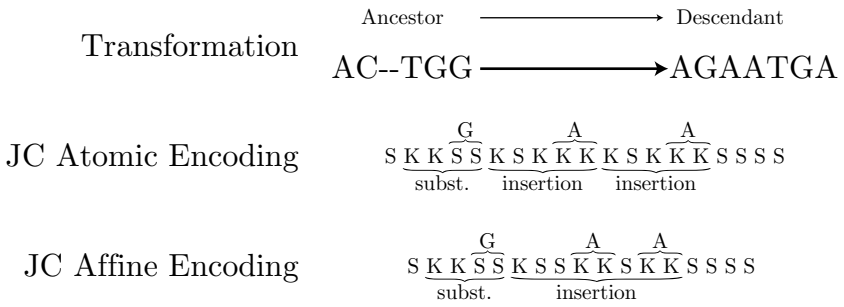


Figure 6.8: Encoding for atomic and affine insertions, deletions, and substitutions.

New models can be defined by combining the various indel and substitution models. Table 6.1 shows K for each combination (see Section C.1.4 for a complete specification of each).

Subst.\Indel	Non-Affine	Affine
JC	3152	4241
K2P	3584	7019
GTR	6926	10361

Table 6.1: Kolmogorov Complexity K in bits, of different combinations of Substitution and Indel models. Section C.1.4 shows a complete specification of each. Slightly lower complexity values can be achieved with more complex function definitions. These complexities where preferred to improve the Appendix readability.

6.2.3.3 Tandem Duplication – Random Loss

The models described in Sections 6.2.3.3 to 6.2.3.6, can be used in the analysis of rearrangement processes occurring in chromosomes and genomes. Like insertions, deletions, and substitutions, they occur on the same A,C,G,T alphabet of nucleotide sequences. However, to ease the exposition, and simplify the connection between the models and the experiments of Section 6.3, I will describe them as rearrangements processes occurring on sequences of *genes*. Formally, let $G = \{1, 2, \ldots, n\}$ be the gene set. A chromosome is a permutation of a subset of genes. A genome is a set of chromosomes, such that every gene in G occurs in exactly one chromosome. A genome could have only one chromosome.

The simplest of all rearrangement models is the Tandem Duplication – Random Loss model (TDRL) [11]. Under TDRL, each transformation consists of a tandem duplication followed by the loss of exactly one copy of the resulting genes (Figure 6.9). An immediate limitation of the TDRL is that it cannot explain the occurrence chromosomal inversions, nor transformations that involve more than one chromosome at a time. Nevertheless, it has been suggested as an important source of variation among organisms [11].

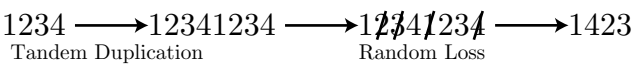


Figure 6.9: Example of Tandem Duplication Random Loss (TDRL). Initially, the chromosome is duplicated in tandem. In a second step, due to strong selective pressure, one copy of each gene is lost, producing a new permutation containing exactly the initial set of genes.

In the simplest setup, equal probability can be assigned to the occurrence of every TDRL.

Theorem 9. *If all TDRL's have equal probability, then the optimal encoding of a TDRL require at most n bits.*

Proof. There are $2^n - (n+1)$ possible TDRL (Figure 6.10). As every one of them has equal probability, then $\lg(2^n - (n+1)) < \lg 2^n = n$ bits are required to optimally encode one of them. □

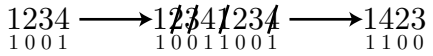


Figure 6.10: There are $2^n-(n+1)$ possible TDRL patterns. After the duplication step, one of the gene copies is deleted. The copies that are deleted in the first duplication can be marked with a 1, while those that are deleted in the second copy are marked with a 0. The only cases that would map back to the original permutation are the $n+1$ patterns of the form 0^*1^*. Clearly, the remaining cases produce a one-to-one mapping of permutations and binary patterns.

Under this model, we can compute the distance between two permutations efficiently.

Theorem 10 (TDRL distance diameter [21])**.** *Define $\rho(\pi)$ to be the number of maximal increasing substrings in a permutation, (e.g. $\rho(142563) = 3$). Then $d(\pi) = \lceil \lg \rho(\pi) \rceil$.*

The next observation follows directly from Theorems 9 and 10.

Observation 11. *A sequence of TDRL events can be encoded with at most $n\lceil \lg n \rceil$ bits.*

Observation 12. *Assume that every distinguishable TDRL has equal probability. Let* `equal_tdrl` *be an SK function that processes a stream of n bits specifying what genes*

are deleted from the first, and what genes are deleted from the second tandem repeat. Then $K(\texttt{equal\_tdrl}) \leq 1547$ *bits.*

Proof. See Section C.1.6. □

A more elaborated case is the geometric model,which assigns probability $\beta\alpha^{k-1}$ to a TDRL of length k, where β is its probability to start in any given position of the chromosome (Figure 6.11). It is easy to see that the following observation holds.

Observation 13. *Encoding a TDRL under the geometric model requires*

$$\lg \beta + \lg n + (k-1)\lg \alpha + \lg(1-\alpha)$$

bits.

Note that, unlike the substitution and indel models, a sequence of TDRL may involve the same genes several times. For this reason, we cannot simply use offsets to mark the location of each TDRL.

Observation 14. *Let* `gtdrl` *be an SK decoder function for the geometric TDRL model. Then* $K(\texttt{gtdrl}) \leq 4370$ *bits.*

Proof. See Section C.1.7. □

To apply the function `equal_tdrl`, an initial chromosome must be generated. Let a be the initial chromosome (i.e. a in Figure 6.3). Given that each gene is assigned a unique code by convention, I can set it to minimize the information required to compute a. Therefore, I will say that $a = \langle 1, \ldots, n \rangle$, which can be computed for a given n.

Observation 15. *Let* `identity` *be the SK machine that decodes* $\langle 1, \ldots, n \rangle$ *from a given* n*. Then* $K(\texttt{identity}) \leq 3452$ *bits.*

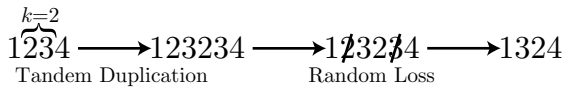


Figure 6.11: A TDRL of length 2 ($k = 2$), with probability $\beta\alpha^{k-1} = \beta\alpha$.

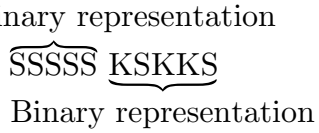


Figure 6.12: Universal integer encoder. $\lceil \lg n \rceil$ bits are prepended to the binary representation of n, allowing a decoder to preprocess the number of bits needed to decode n. This encoding is nearly optimal to decode an integer, when all integers are equally likely [74].

Proof. See Section C.1.3.1 □

The integer n can be encoded using the universal integer code (Figure 6.12), with overall complexity $2\lceil \lg(n) \rceil$.

6.2.3.4 Inversions

Let $\pi = \langle \pi_1, \ldots, \pi_n \rangle$ be a permutation of $\{1, \ldots, n\}$. An unsigned inversion ρ of the interval i, j in π is defined as

$$\pi\rho = \langle \pi_1, \ldots, \pi_{i-1}, \pi_j, \pi_{j-1}, \ldots, \pi_{i+1}, \pi_i, \pi_{j+1}, \ldots, \pi_n \rangle.$$

A signed inversion ρ of the interval i, j in π is defined as

$$\pi\rho = \langle \pi_1, \ldots, \pi_{i-1}, -\pi_j, -\pi_{j-1}, \ldots, -\pi_{i+1}, -\pi_i, \pi_{j+1}, \ldots, \pi_n \rangle.$$

An example illustrating unsigned and signed inversions can be seen in Figure 6.13.

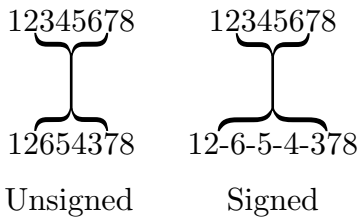


Figure 6.13: Examples of unsigned and signed inversions, in a pair of chromosomes with 8 genes. In reality, chromosomes are signed, and therefore, all inversions are signed. However, unsigned inversions could occur in other kinds of characters (e.g. developmental sequences).

Theorem 11. *If all inversions have equal proability of occurrence, an optimal encoding of an inversion requires* $2\lg(n+1)-1$ *bits.*

Proof. All possible inversion intervals can be described with the upper and lower limits of the inversion. There are $n+1$ possible limits. Assume that empty inversions cannot occur, then there are

$$\binom{n+1}{2}$$

possible intervals. Taking its logarithm, the result follows. □

Theorem 12. *Let the function* `inversion` *perform the m'th signed inversion in a given sequence of fixed length. Then* $K($`inversion`$) < 4460$ *bits.*

Proof. See Section C.1.8 □

The diameter of the (signed) inversion function is n [56]. The next observation follows directly.

Observation 16. *If all inversions have equal probability, then a sequence of inversions will require at most* $2n\lg(n+1)-1$ *bits.*

The TDRL mechanism collapsed the distance between permutations logarithmically. Inversion distance remained linear. However, the difference between signed inversions and TDRL is very small in KC units.

Encoding the initial unsigned sequence a for the inversion model, has the same constraints as in the TDRL (Section 6.2.3.3), and so, has its complexity. The only difference in the case of signed inversion is the specification of the root sequence sign. As the sign is convention, we can assume that every gene in a has a positive sign. Given that the representation of a sequence must include the sign of each gene, the decoders are slightly different.

Theorem 13. *Let* `signed_identity` *be the SK machine that decodes the signed sequence* $\langle 1, ..., n \rangle$ *from a given* n*. Then* $K(\texttt{signed\_identity}) < 3518$ *bits.*

Proof. See Section C.1.3.2. □

6.2.3.5 Double Cut and Join

In the Double Cut and Join (DCJ) mechanism, an edition consists of a pair of cuts in one (or two chromosomes), followed by the join of the resulting tips in one of two possible rearrangements (Figure 6.14). A peculiarity of the DCJ mechanism is its pervasive use of *circular chromosomes*. Circular chromosomes also allow *transpositions* in two DCJ steps (Figure 6.15). In the DCJ model all inversions are signed.

Theorem 14. *Assume that all DCJ's are equally likely. Then the optimal encoding of a DCJ in a genome with* n *genes, and* l *linear chromosomes, requires at most* $1 + 2\lceil \lg(n + l) \rceil$ *bits.*

Proof. A double cut may occur in any adjacency, either within a chromosome, or in one of the chromosome limits when the chromosome is linear. Under the DCJ

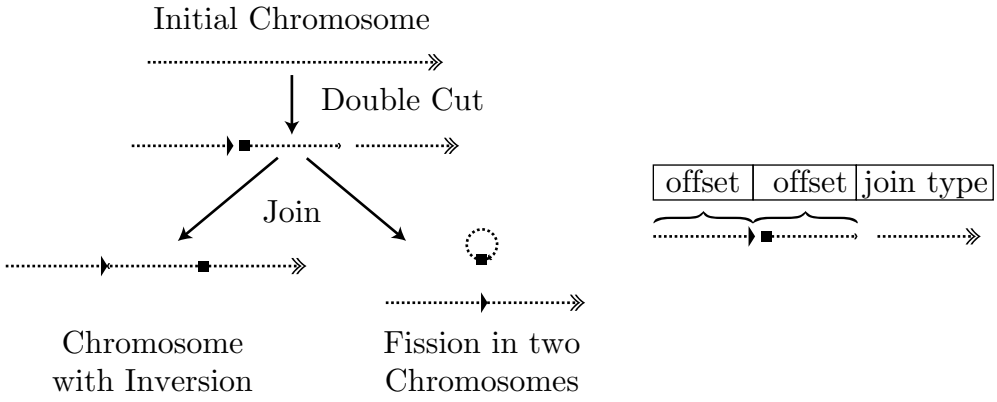


Figure 6.14: Example of the Double Cut and Join (DCJ) mechanism (left), and its encoding (right). Under DCJ, a genome can be cut in two positions and joined back in one of two rearrangements. In this example, I show the effect of DCJ when both cuts occur in the same chromosome. To encode these operations, it is enough to include the two cut location offsets using $\lg n$ bits, and the type of join in one extra bit.

mechanism, empty cuts are allowed. Therefore, for a chromosome of length m, if it is circular, then there are m possible cut positions; if linear, then there are $m + 1$ possible cut locations. Hence, there are $(n + l)^2$ cut combinations. For each double cut, there are two possible join operations, for a total of $2(n + l)^2$ rearrangements. If all are equally likely, then encoding them requires $\lg 2(n + l)^2 \leq 2\lceil \lg(n + l) \rceil + 1$ bits. □

Theorem 15. *Let* `dcj` *be an SK machine that decodes DCJ operations, where every operation is equally likely. Then* $K(\texttt{dcj}) \leq 26276$ *bits.*

Proof. See Section C.1.9. □

DCJ supports both circular, and linear chromosomes, of varying lengths. A common assumption under DCJ is that each gene occurs in exactly one chromosome. Therefore, just as I did for the signed inversion mechanism, I will assume that a will

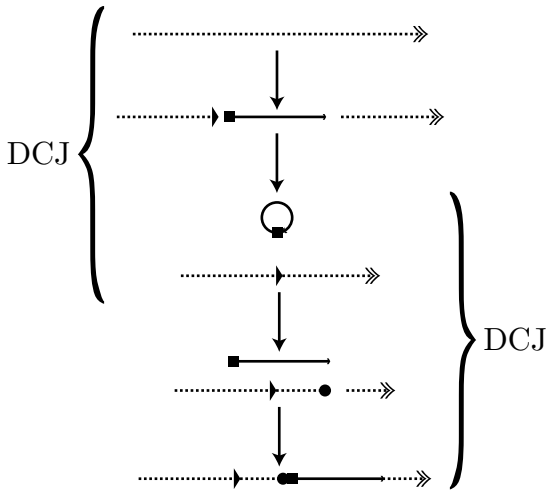


Figure 6.15: Example of transposition under the DCJ mechanism using two successive DCJ operations. In the first operation, a temporary circular chromosome is created. In the second operation, the temporary chromosome is inserted in a different position of the original chromosome it belonged to.

consist of $\langle 1, \ldots, n \rangle$ for n genes of positive sign. The limits of the chromosomes, and their type (circular or linear) would still have to be encoded.

Theorem 16. *Assume that all genomes with n genes are equally likely. Then encoding the chromosome limits under the DCJ mechanism requires at most $1 + 2(n-1)$ bits.*

Proof. All chromosomes can be described by taking the sequence $G = \langle 1, \ldots, n \rangle$, and mark adjacencies as either the beginning of a circular chromosome, the beginning of a linear chromosome, or an internal chromosome adjacency, that is, 3 possible states in possible $n - 1$ adjacencies. The beginning of G is a chromosome limit that only requires distinguishing between circular and linear states. Hence, for a sequence of n genes, there are $2 \times 3^{n-1}$ possible chromosome arrangements, requiring $\lg(2 \times 3^{n-1}) < 1 + (n-1)\lceil \lg 3 \rceil = 1 + 2(n-1)$ bits. □

Theorem 17. *Let* `dcj_identity` *be the SK machine that decodes the signed genome by decoding the type of each gene adjacency. Then $K($*`dcj_identity`*$) \leq 2912$ bits.*

Proof. See Section C.1.9.1. □

6.2.3.6 Inversions, Fusions, Fissions, and Translocations

Another model of *genome* evolution allows the occurrence of signed inversions (Figure 6.13), fusions, fissions, and translocations (IFFT, Figure 6.16) [12]. Moreover, under this model, complete chromosomal inversions are indistinguishable (i.e. $\langle 1, 2, 3, 4\rangle = \langle \sim 4, \sim 3, \sim 2, \sim 1\rangle$).

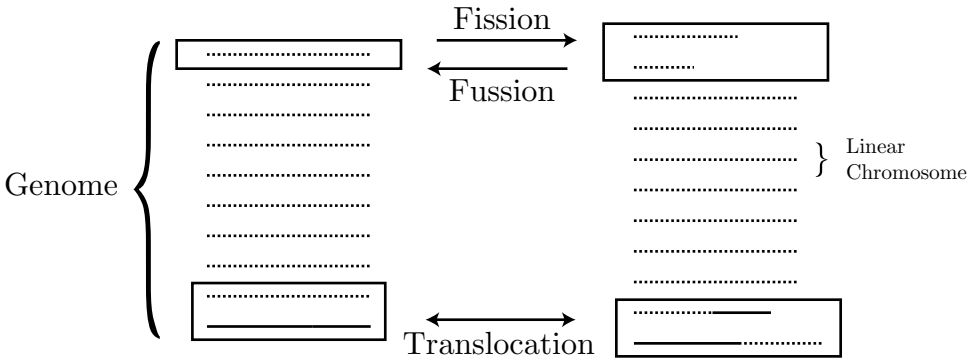


Figure 6.16: Example of fusion, fission, and translocation, under the IFFT mechanism. A fission splits a linear chromosome in two linear chromosomes. A fusion is symmetric to a fission, by merging two linear chromosomes into one. A translocation exchanges tips between chromosomes. In the IFFT mechanism, all the chromosomes are linear.

Theorem 18. *Let G be a genome with n genes and l chromosomes. If all inversions, translocations, fusions, and fissions have equal probability, then an optimal encoding of an IFFT requires at most $1 + 2\lceil \lg(n + l + 1)\rceil$ bits.*

Proof. Every gene-gene adjacency and chromosome tip could be the position to perform the cut under the IFFT mechanism. There are $n + l$ possible positions. We can also represent fissions by adding an extra chromosome limit (Figure 6.17), adding an

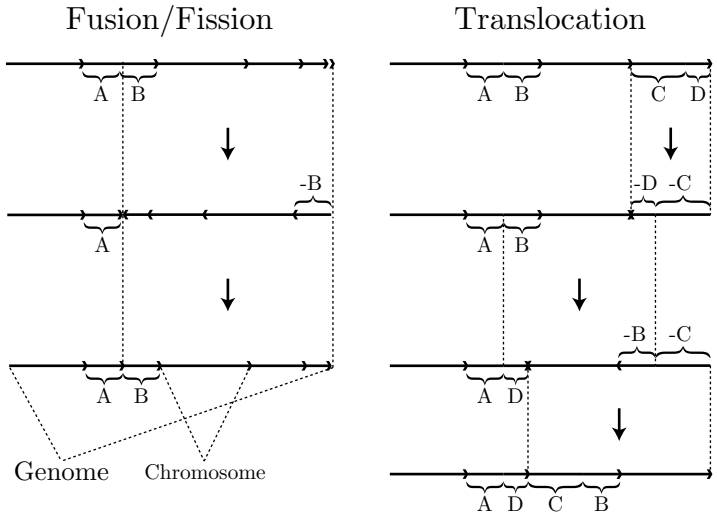


Figure 6.17: Emulating fusions, fissions, and translocations using inversions. Each arrow represents a chromosome, and its direction. Fusion and fission use the empty chromosome (rightmost chromosome in upper left genome). Note that the order of the chromosomes in a genome is irrelevant under the IFFT model.

extra possible position. Therefore, there are 2^{n+l} position patterns under the IFFT mechanism. When doing translocations, we must specify which of the two possible forms of translocations should be performed (i.e. the translocation of the two halves AB of one chromosome, and CD of a second chromosome could produce AD and CB, or A-C, -BD), adding an extra bit. The result follows. □

All transformations of the IFFT model can be performed using inversions only with the additional requirement of correcting translocations, fusions, and fissions that span more than one chromosome (Figure 6.17).

Theorem 19. *Let* $\mathtt{ifft}$ *be an SK machine that performs inversions, fusions, and fissions as specified by a pair of offset positions in the genome and the type of translocation if applicable (i.e. as shown in Figure 6.14 for DCJ). Then* $K(\mathtt{ifft}) < 22877$ *bits.*

Proof. See Section C.1.10. □

As discussed in Section 6.2.3.4, to compute the initial genome a, the codes and signs of the genes can be calculated using n. We are left only with the size of each individual chromosome. Just like the DCJ mechanism, in the IFFT, the maximum number of chromosomes is the number of genes in the genomes. However, unlike the DCJ mechanism, all chromosomes are assumed to be linear (i.e. there are no empty chromosomes in the leaves).

Theorem 5. *If all chromosomal arrangements are equally likely, then the optimal encoding of one arrangement requires* $n - 1$ *bits.*

Proof. Every gene adjacency in the sequence $\langle 1, \ldots, n \rangle$ is a potential chromosome limit. Assuming that the genes 1 and n are at one end of (potentially different) chromosomes, then we are left with exactly $n - 1$ adjacencies. There are 2^{n-1} arrangements in which we can select chromosome limits among the adjacencies. If every arrangement is equally likely, then we can encode them with $\lg 2^{n-1} = n - 1$ bits. □

In the following Section, I will evaluate these mechanisms and the KC hypothesis selection criterion in phylogenetic analysis using simulations.

6.3 Experimental Evaluation

Evaluating a new optimality criterion for phylogenetic analysis is a difficult task. This is particularly true if our interest is to explore the use of complete genomes as evidence. In my experience, most simulations are clearly oversimplified even for the simplest mechanism of all: insertions and deletions.

This study cannot pretend to overcome all of these limitations. Instead, KC is intended to be used as an inductive tool to *recognize* a potentially meaningful mechanism. The mechanism may not occur in reality, but would reflect those patterns that can be inferred from the data alone, and could help in the understanding of diversity.

In this section, I explore experimentally, using simulations, how KC can help us understanding patterns occurring at the chromosome level in phylogenetic analysis. With this goal, I simulated data using atomic and affine indels, TDRL, Inversion, IFFT, and a simplified version of DCJ, and compared the inferences made based on each case using KC.

6.3.1 Simulations

Three aspects were varied in the simulation step: tree topology and branch lengths, mechanisms, and rates.

Tree topology. For each mechanism under evaluation, I generated 200 random trees of average branch length 0.2 and standard deviation 0.1. All the trees had 50 leaves. The branch length distribution parameters where selected based on preliminary results showing that these maintained enough phylogenetic signal to produce meaningful analyses that where difficult to select. I used 50 leaves in the simulations to make the analyses computationally feasible, and interestingly large at the same time.

Mechanism. For the simulation mechanism, we are limited by the biological and computational components. Biologically, very few models of indels and rearrangements have been proposed, and most have been proposed in the computational lit-

erature [134, 98]. Computationally, each mechanism pose a complex combinatorial optimization problem. For indels, the algorithms described in Chapters 3 and 4 can be adapted for KC. However, in the case of rearrangements, even computing the transposition distance between permutations remain an open problem, of unknown hardness. I limited the simulations to the TDRL, Signed Inversion, Transposition , IFFT, and a simplified form of DCJ where every circular chromosome creation was followed by a fusion that eliminated it. For each mechanism, I assigned equal probability to all possible rearrangements. Although this may be considered unrealistic, for each of the selected models M, $K_M(A|B)$ can be computed in polynomial time, making these experiments feasible. For the indel models, the root's random sequence was set to length 400. For each rearrangement mechanism, roots with 50, 100, 150, and 200 genes were generated. For multichromosomal genomes, a chromosomal arrangement was selected uniformly at random.

Rates. For every mechanism, and every phylogenetic tree, the following rates were evaluated: 0.2, 0.1, 0.05, 0.04, 0.02, 0.01, and 0.009 to 0.001 in steps of 0.001. The rate was maintained constant for every tree. Therefore, for each tree and rate combination, the number of transformations was only dependent on the branch length.

6.3.2 Analyses

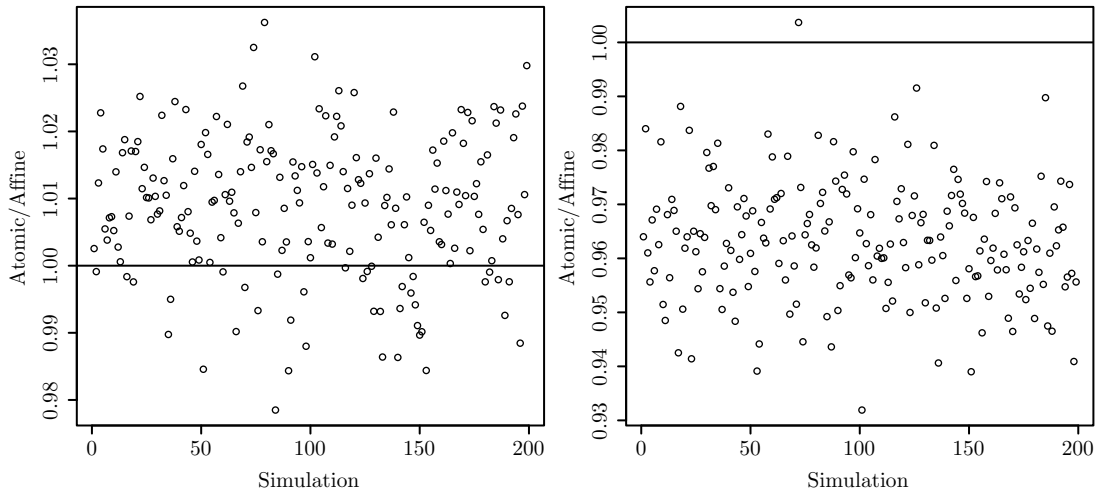
My goal is to compute the KC of an observed data set. Being non-computable, an attempt to find an exact solution is impossible. To compute results in a reasonable time, I limited the possible solutions to tree hypotheses (as described in Section 6.2.2.2). Although network hypotheses could be considered, the computational problems become rapidly intractable, and so, were not included in the analyses.

Atomic and Affine Indels. For the atomic and affine indel simulations, I used the algorithms described in Sections 3, and 4. Each tree was subjected to a local search strategy as described in Section 4, with a time limit of 24 hours. The analyses where performed with $\alpha = 0.2$, and $\beta = 0$ for the atomic, and $\beta = 0.7$ for the affine gap cost models. No further parameters where explored due to the long computation time. The model selected, and number of false positive clades where computed for the preferred hypothesis. Note that my implementation always return a binary tree, whereas the simulated phylogeny need not to be binary. Hence, the number of false positives is biased, overestimating the hypotheses error.

Rearrangements. I implemented a number of exact algorithms for the tree cost calculation under TDRL, Signed Inversion, and DCJ distance functions. However, these algorithms proved to be too slow to be used in a complete local search. Although heuristics exist for Signed inversion [16], and IFFT [13], they couldn't be used for the purpose of this study, which required a comparison including TDRL, and DCJ to be meaningful (IFFT is Signed inversion if the genomes have only one chromosome). For these reasons, I took the simulation tree and used the exact algorithms to optimize such tree under each model the iterative algorithm (see Section 3.3.3). The different hypotheses where then compared.

6.3.3 Results

Insertions and Deletions. In the atomic indel simulations, the KC criterion correctly selected the atomic model in 84% of the cases; for the affine gap simulations the KC criterion correctly selected the affine model in 99% of the cases (Figure 6.18). In all cases, the great majority of the recovered trees have more than 95% of the groups found in the true phylogeny.



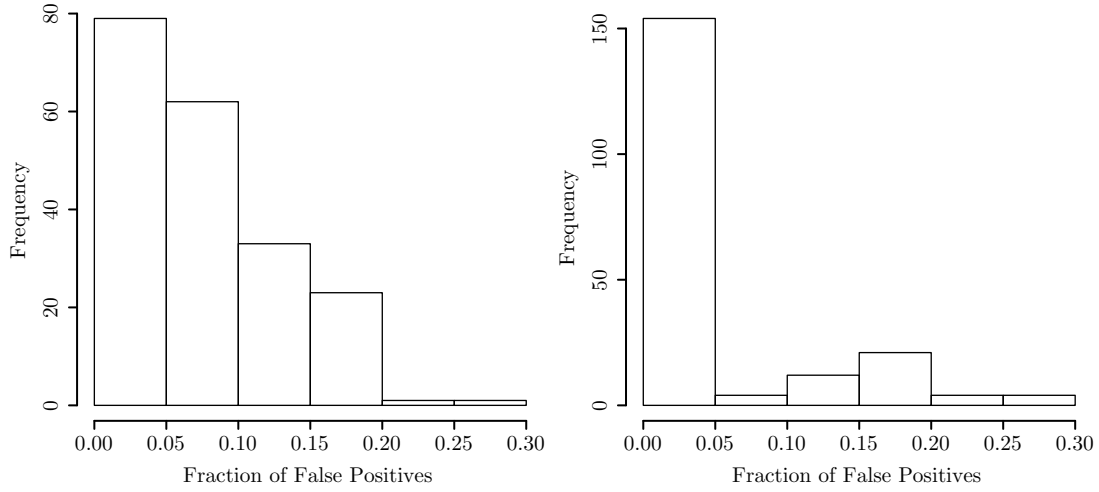
(a) (b)

Figure 6.18: Model selection for indel model. Points above the line are analyses where atomic indels would be the preferred hypothesis, points below would prefer the affine indel model. a. atomic gap cost. b. affine gap cost.

Rearrangements. The model was correctly identified in all the rearrangement simulations with two exceptions: the simplified DCJ mechanism was never correctly preferred, and TDRL was preferred for short branch lengths under the transposition simulation. In the first case, DCJ appears to be overly complex when compared to the IFFT if no mixture of circular and linear chromosomes occur in a genome. In the second case, TDRL is a very simple model, that can be preferable over the Transposition model for very short branch lengths.

6.4 Discussion

The simulations showed that, by using KC, I can consistently identify the model of each simulation. Moreover, in the analyses using insertions and deletions, the



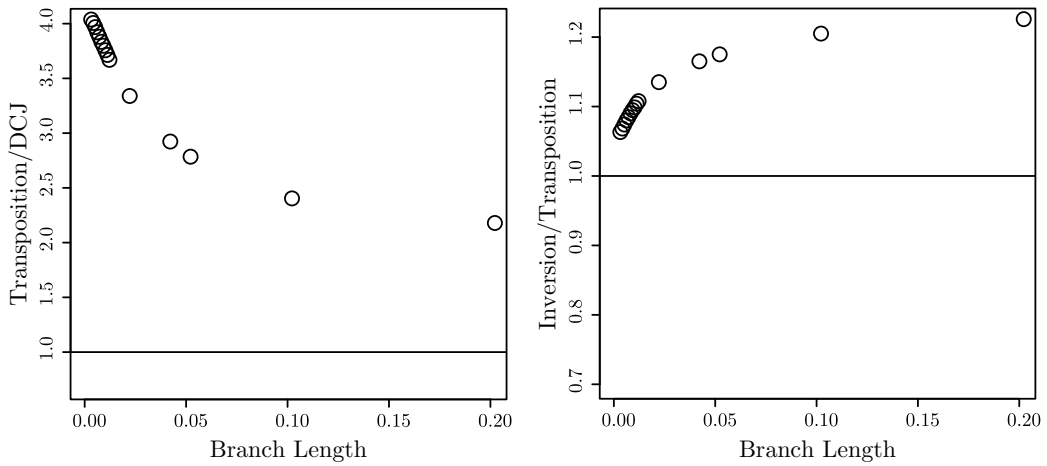
(a) (b)

Figure 6.19: Phylogenetic tree fraction of false positives. a. atomic gap cost. b. affine gap cost.

error rate was small, and produced reasonable phylogenies with low false positive frequencies.

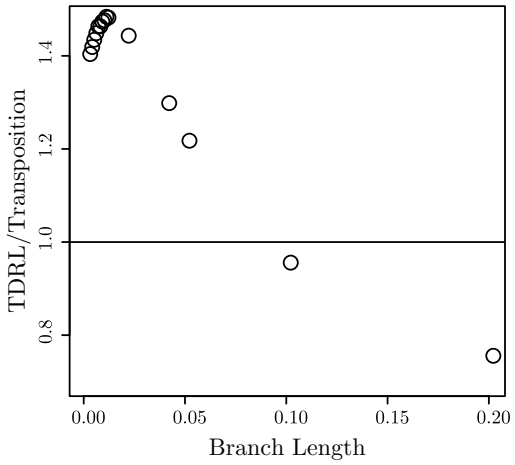
Although DCJ has been a major player in recent computational literature, according to KC, it has a very limited application in phylogenetics. To my knowledge, genomes consisting of mixtures of circular and linear chromosomes are rarely observed. In the absence of such mixtures, the higher complexity of DCJ makes it a less likely hypothesis compared to those produced using IFFT. Nevertheless, exploring DJ will increase our understanding of the combinatorics of the analysis of rearrangements.

A major aspect that I did not explore in this study is to adjust the model parameters on each branch (i.e. to incorporate branch length). Better heuristic methods to simultaneously estimate alignments and phylogenies in a form that can be used under the KC criterion are needed. The algorithms in Chapters 3 and 4 are steps in



(a)

(b)



(c)

Figure 6.20: Model selection under KC, for the transposition simulation. In most cases (a and b), the model is correctly selected as the transposition. Note that for very short branch lengths, the TDRL model is preferred over the Transposition model, due to low complexity of the TDRL machine.

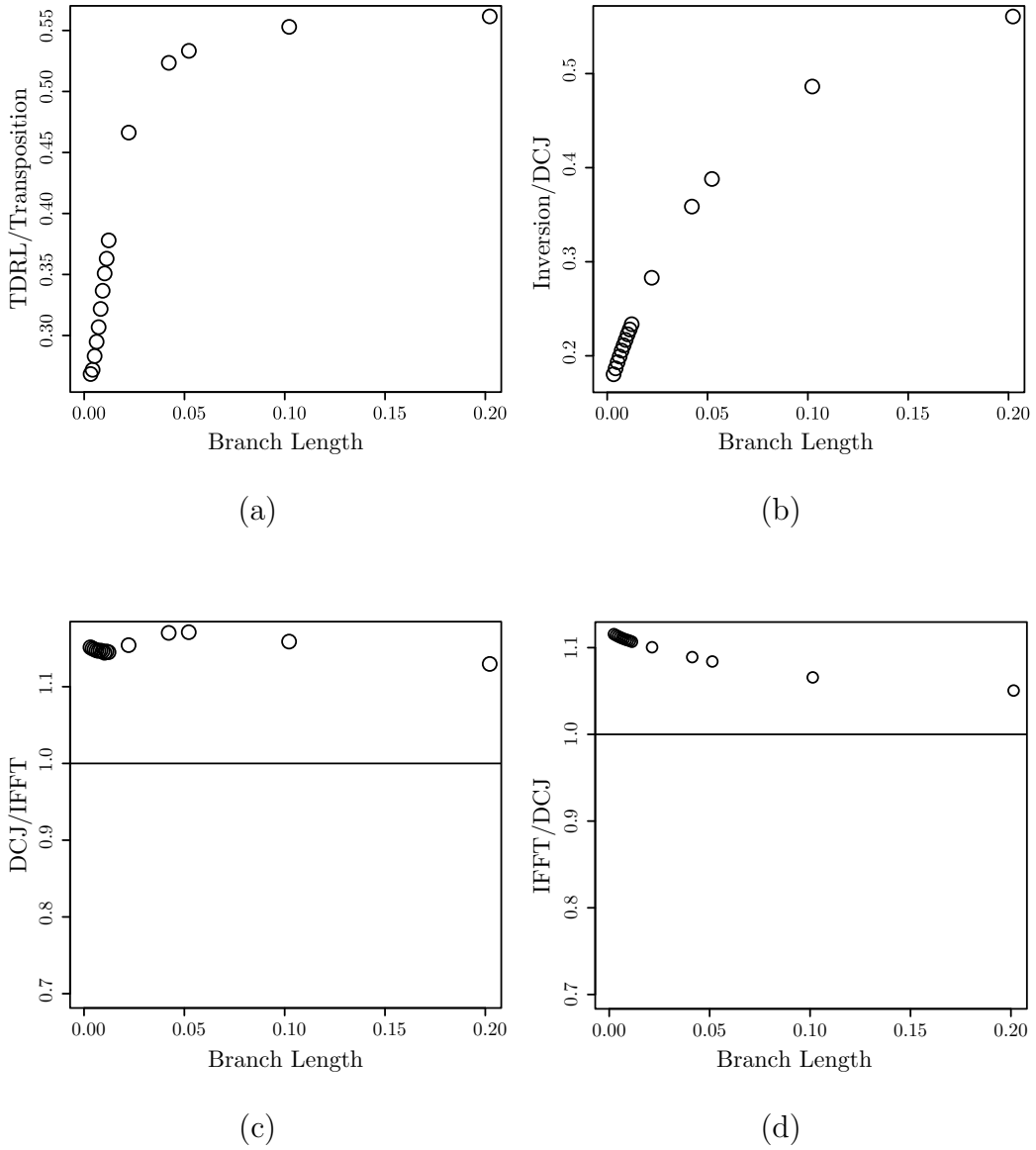


Figure 6.21: Model selection under KC for various simulations. a. TDRL simulation; the TDRL is correctly selected. b. Inversion simulation; the inversion mechanism is correctly selected. c. IFFT simulation; the IFFT model is correctly selected. d. Simplified DCJ simulation; the DCJ model was not preferred in any of the simulations performed.

this direction.

A number of other interesting directions for KC remain to be explored. For instance, the analysis of morphometric characters (geometric information describing morphological features), developmental sequences, and metabolic pathways, appear extremely well suited for the KC criterion. Regardless the limitations of this study, and available computational methods, KC shows promising results for philosophically sound phylogenetic inference.

6.4.1 Connection with other Methods

Phylogenetic analysis has been dominated for a few decades by four competing methods: Distance, Maximum Parsimony (MP), Maximum Likelihood (ML), and Bayesianism (see [38] for an overview). KC provides a general framework that connects existing criteria in a uniform theory of phylogenetic inference.

6.4.2 Connection with Maximum Parsimony

Under MP, each character is not comparable to any other one (i.e. columns are independent from each other). It follows that the state encoding is irrelevant (e.g. if we assign one state the representation A, 1, or 20, the analysis remains unaffected). Therefore, there is an encoding of the characters and their states such that at least one of the leaves of the tree consists of only 1's. Then we can reproduce such leaf by simply encoding the total number of characters n. Therefore, the complexity of such leaf is constant for all theories.

Moreover, for all theories in MP, the tree must be binary. The number of edges in a binary, unrooted tree is $2l-3$, where l is the number of leaves. In MP, we only attach to each branch the transformations inferred in the most parsimonious hypothesis,

therefore, we can assume that the representation of branches and vertices is constant. Hence, the complexity of the tree is only dependent on l. But l is constant for all the theories, and so, the complexity of the tree is fixed for all theories explaining D.

It follows that the most parsimonious tree is exactly the shortest program that explains D under the restrictions posed above: characters are independent, and the only valid theory is a binary tree. It follows that if both restrictions are enforced in the KC analysis (by using a unique alphabet for each character, and forcing a particular kind of program), then KC and MP would select the same hypothesis.

6.4.3 Maximum Likelihood

Suppose some tree T is the theory (tree and model) maximizing the likelihood of D. It follows from information theory that D can be compressed optimally using T. Therefore, if the complexity of the theory T itself is ignored, then ML and KC would prefer the same hypothesis. A corollary of this observation is that ML and MP converge to the same solution when there is no common mechanism between characters (i.e. if the alphabet used for each character is unique). (A complex proof of this theorem was discovered by [109].)

6.4.4 Bayesianism

If the hypotheses under consideration are finite sets of binary strings of finite length, then KC and the phylogenetic tree that maximizes the posterior probability using the universal distribution $\mathbf{m}(\cdot)$ as prior distribution select the same hypothesis [117]. Given that phylogenetic hypotheses fill this requirement, then bayesian methods with such priors, and KC phylogenetic analysis converge to the same hypotheses. (The interpretation of the hypothesis in Bayesianism and KC, is very different, see [52].)

The universal distribution is not computable, and therefore, in practice, the criteria diverge [117].

Chapter 7

Conclusions

I have presented a novel algorithm that I have called Affine-DO for the TAP under affine gap costs (Section **??**). My experimental evaluation, the largest performed for this kind of problem, shows that Affine-DO performs better than Fixed States. However, I observed that the LP bound is too pessimistic, producing unfeasible solutions 10% worse, even for the smallest non-trivial tree consisting of 3 leaves. Based on these observations, I believe that Affine-DO is producing near-optimal solutions, with approximations within 10% for sequences with small divergence, and within 30% for random sequences, for which Affine-DO produced the worst solutions.

Affine-DO is well suited for the GTAP under affine sequence edit distances, and yields significantly better results when augmented with iterative methods. The main open question is whether or not there exists a guaranteed bound for DO or Affine-DO. Then, if the answer is positive, whether or not it is possible to improve the PTAS using these ideas. Additionally, many of these ideas can be applied for true simultaneous tree and alignment estimation under other optimality criteria such as ML and MAP. Their use under these different optimality cirteria remains to be explored.

Based on the positive results of Affine-DO, I described new strategies that can be

composed to produce a powerful local search strategy for the Tree Alignment Problem (Section 4). The results showed that my methods improve on the best existing local search heuristics by more than three orders of magnitude.

In general, the Exhaustive–TBR refinement strategy should always be used, while my Union-pruning should only be preferred if good taxon sampling or short branch lengths are expected. Moreover, although the MST build strategy yields better results than the traditional Wagner build, the former should not be preferred in real analyses since it tends to produce less competitive trees after the refinement step.

It is difficult to predict the performance of other high level heuristics applied to the GTAP. Strategies such as Sectorial Search, and Tree Fusing should be effective. However, Divide and Conquer techniques such as DCM-3 may have a more limited application, unless used in the spirit of Sectorial Search. Given that phylogenetic analysis under MP shows a simplified setting compared to other optimality criteria, it is my opinion that metaheuristics such as Simulated Annealing have limited applicability in the join estimation of tree and alignments for all optimality criteria, and novel strategies are needed to successfully scale to larger problem sizes. Nevertheless (unless $P = NP$), all these strategies will belong to the heuristic realm, and further experimental efforts will be required.

Affine-DO, Union–pruning, and Exhaustive–TBR are some of the algorithms that I have implemented in the computer program POY version 4 (Section 5). The algorithms and their implementation have had a significant impact in the biology community interested in different approaches to joint tree and phylogeny reconstruction. By using better algorithms, algorithm engineering, and better parallel strategies, POY version 4 is three orders of magnitude faster than its predecessor. The concepts, and desirable properties of this implementation should be extended to other phylogenetic inference criteria, to broaden its usability, and better serve the research purposes that

set for the software package.

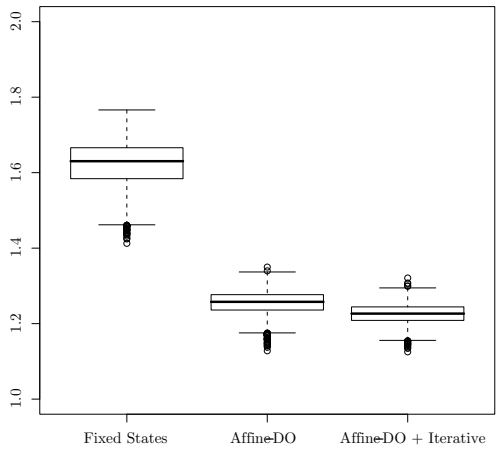
The KC optimality criterion showed promising results (Section 6). It consistently preferred the correct model and hypothesis in a number of settings. However, it also increases the uncertainty in a number of aspects. What is the effect of the language in reality? Data sets are finite, therefore the language selected could have a significant impact in the hypothesis preferred. I considered only biologically sound underlying structures (phylogenetic trees and networks). Are there other structures, less meaningful, that could render the method unsound? How competitive is the method when many models are combined?

Algorithms for the analysis of rearrangements are in their early stages. I have no doubt that better methods will be discovered in the following years. In their current form, however (chromosomes as sequences of genes), the algorithms are of limited real use. Biologists need algorithms that can efficiently analyze the raw DNA sequences, to obtain phylogenies that have more objective support. In this way, meaningless chromosome annotations can be avoided.

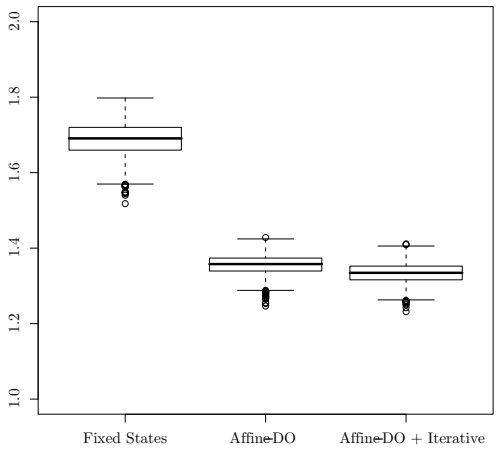
The experimental evaluations gave me first hand experience with the limitations of phylogenetic analysis programs, commonly advertised for chromosomal rearrangement analysis. Most programs in this group (e.g. GRAPPA), use software engineering strategies that may produce very fast implementations, at the price of unsafe, error prone, and hard coded limits. These properties make them of limited use for biologists, who need computer scientists to get closer to their needs. Biological justifications are not just a tool for publication: the algorithms and software packages in those publications also need to be a toolbox that biologists can use with experimental data.

Appendix A

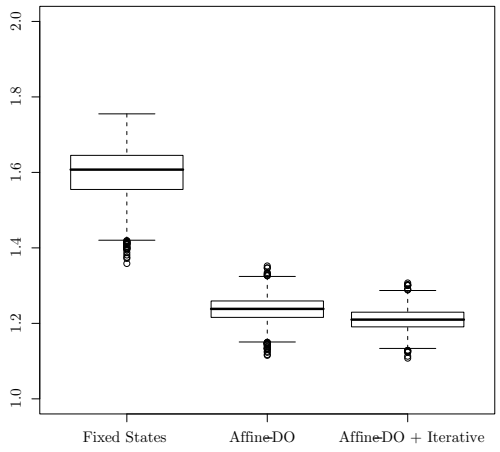
TAP Results



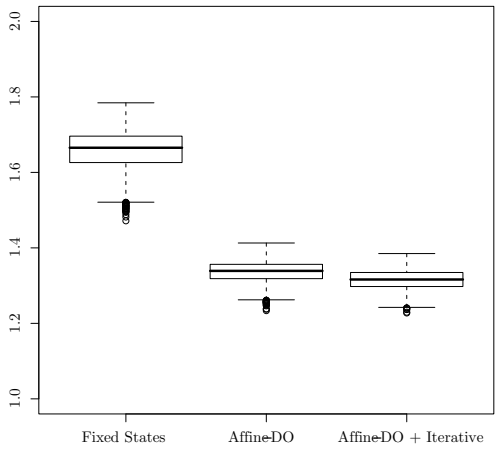
(a) 1,1,0,0.1



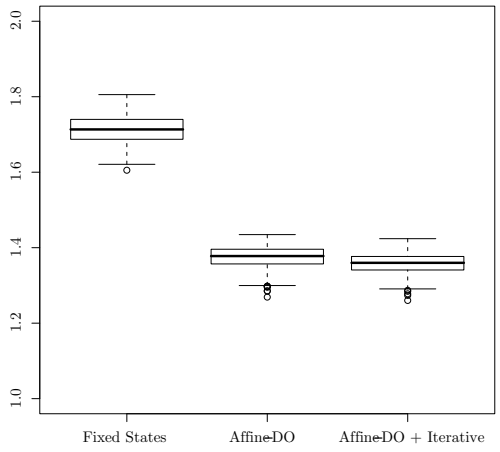
(b) 1,1,0,0.2



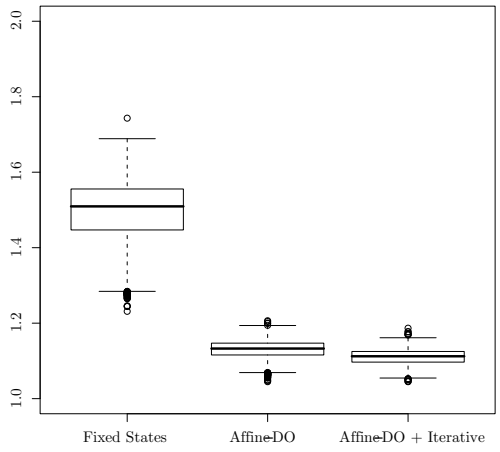
(c) 1,2,0,0.1



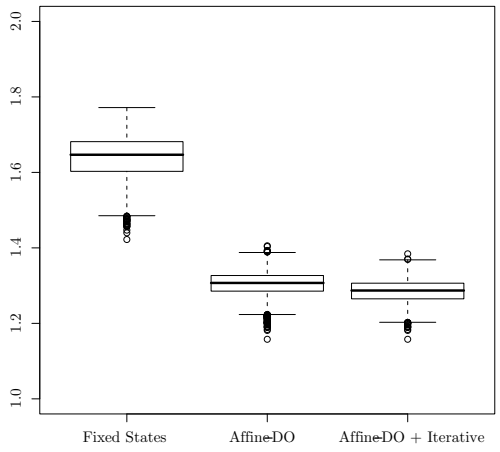
(d) 1,2,0,0.2



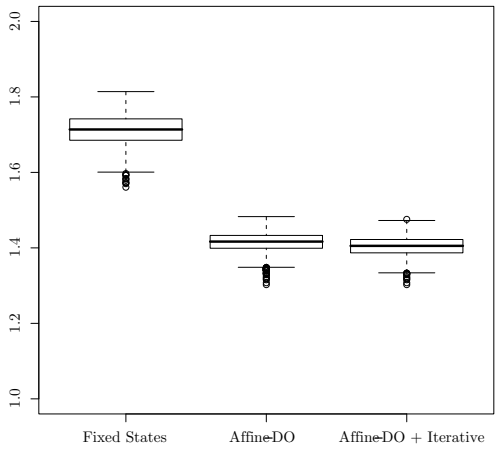
(e) 1,2,0,0.3



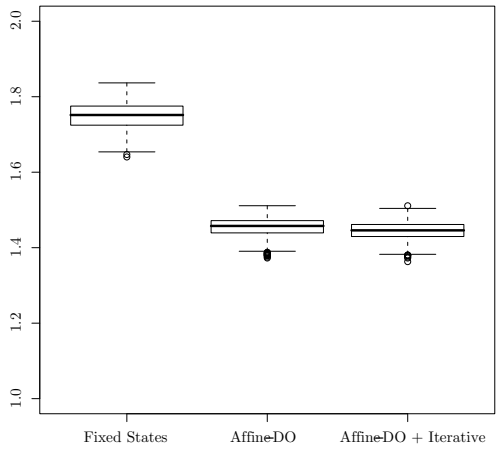
(f) 1,2,0,0.05



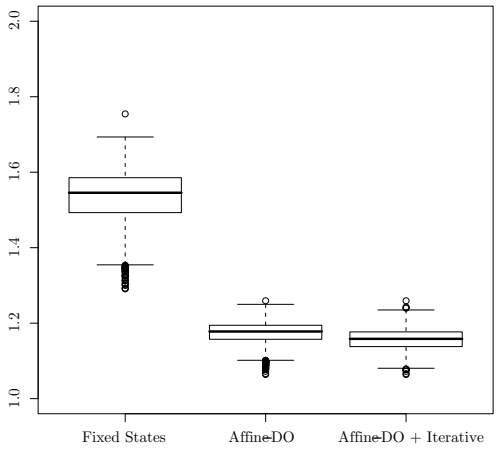
(g) 2,1,1,0.1



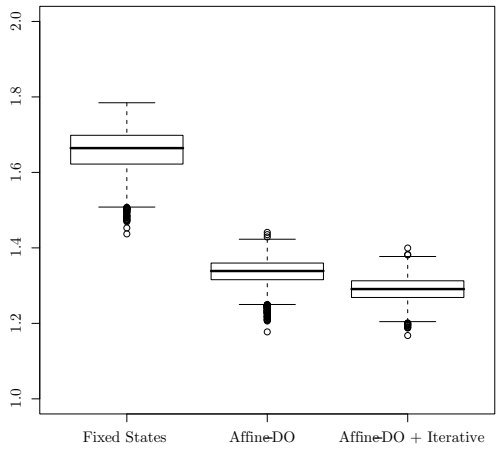
(h) 2,1,1,0.2



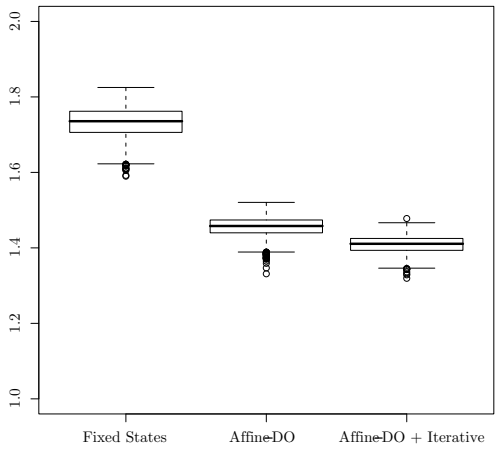
(i) 2,1,1,0.3



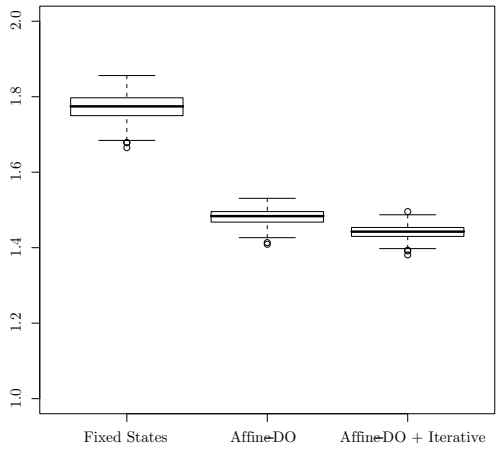
(j) 2,1,1,0.05



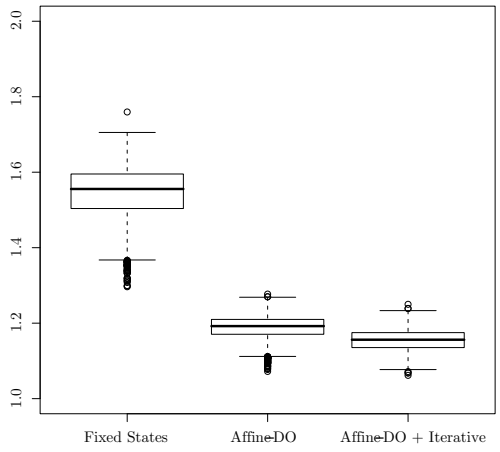
(k) 3,1,2,0.1



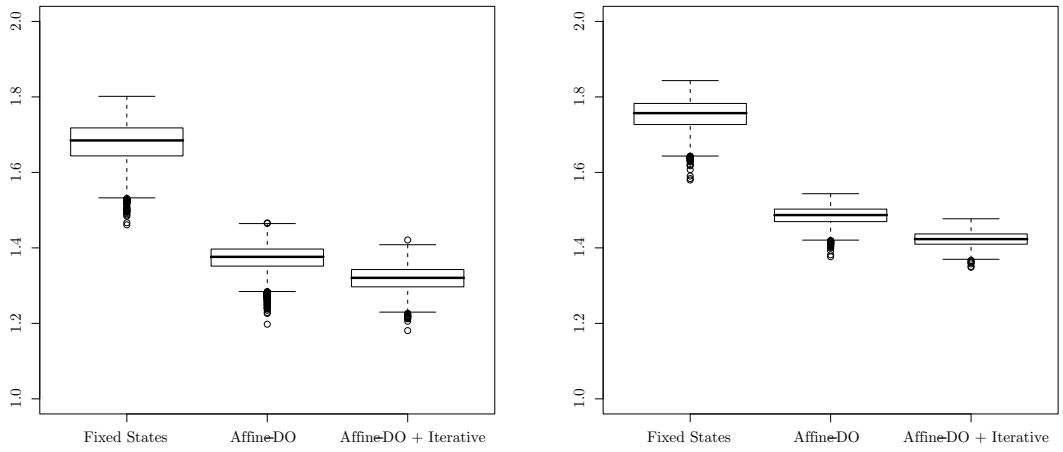
(l) 3,1,2,0.2



(m) 3,1,2,0.3



(n) 3,1,2,0.05



(o) 4,1,3,0.1

(p) 4,1,3,0.2

Figure A.1: Comparison of the three main Affine-DO algorithms under study: Fixed States, Affine-DO, and Affine-DO followed by iterative improvement. (Substitution, Indel, Gap Opening, Branch Length)

Appendix B

GTAP Results

B.1 Build Comparison

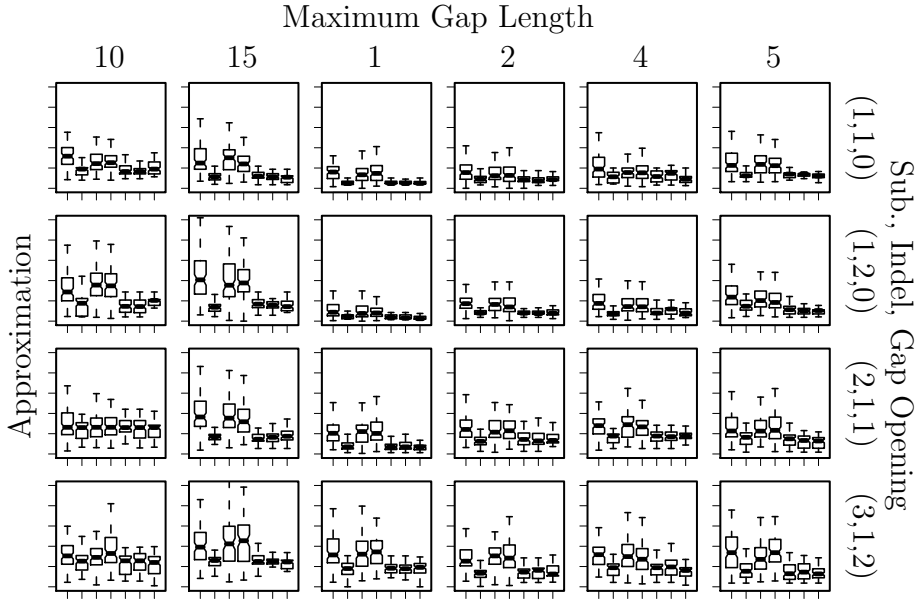


Figure B.1: Build algorithm comparison under Normal Affine-DO. Average branch length 0.1

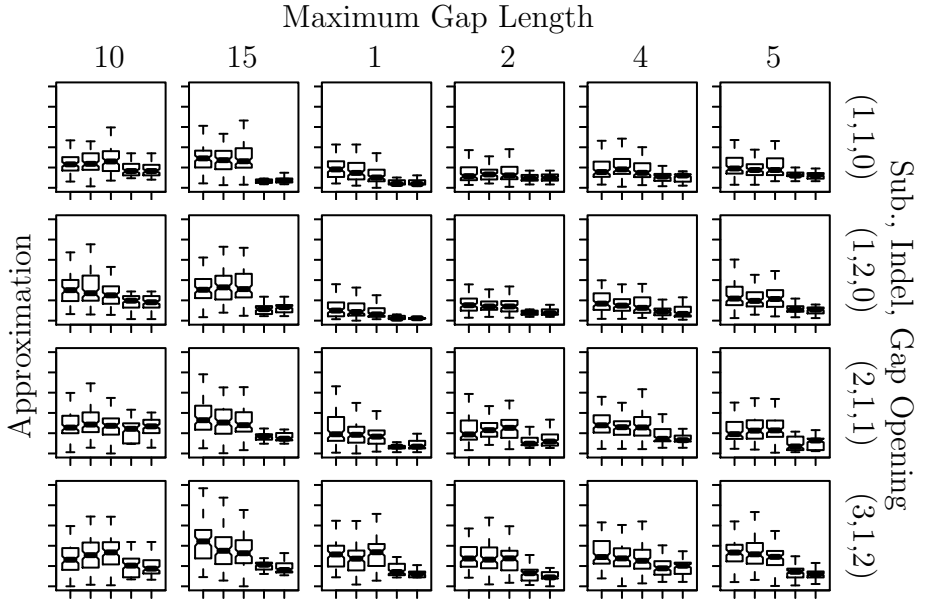


Figure B.2: Build algorithm comparison under Exhaustive Affine-DO. Average branch length 0.1

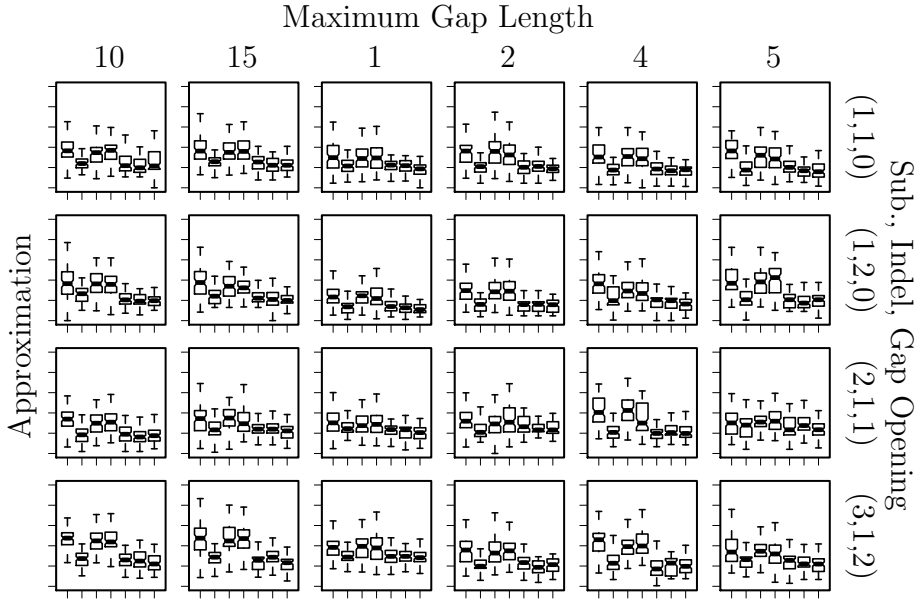


Figure B.3: Build algorithm comparison under Normal Affine-DO. Average branch length 0.2

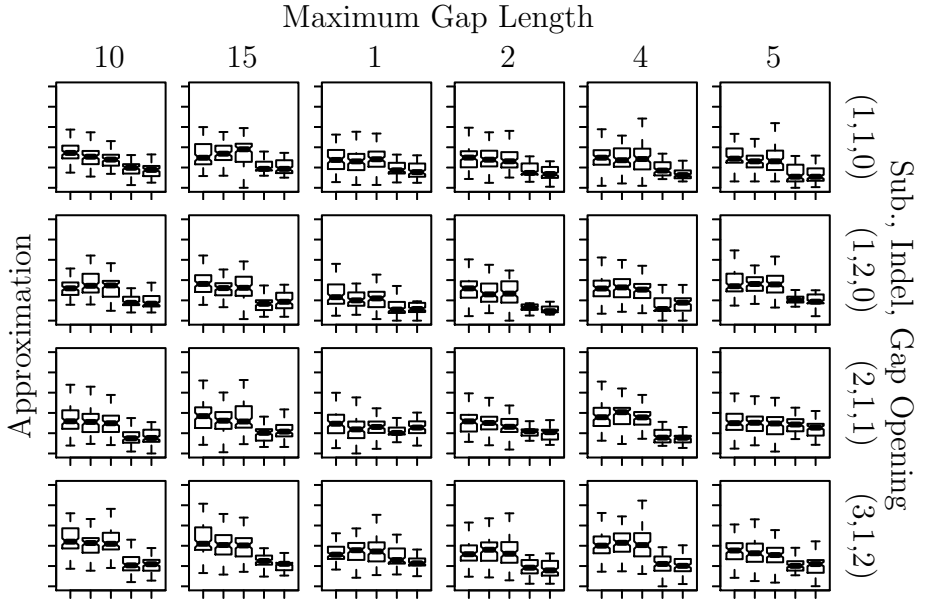


Figure B.4: Build algorithm comparison under Exhaustive Affine-DO. Average branch length 0.2

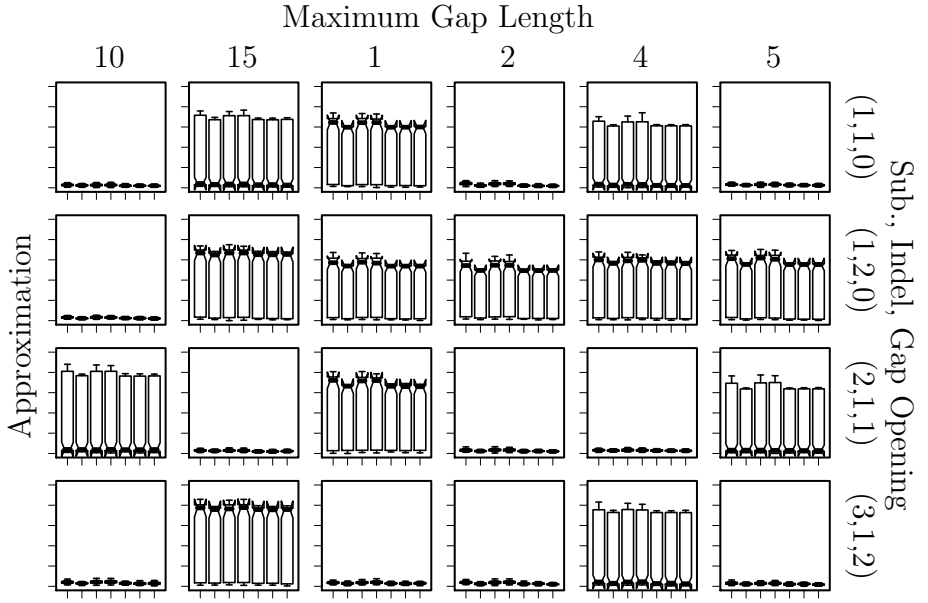


Figure B.5: Build algorithm comparison under Normal Affine-DO. Average branch length 0.3

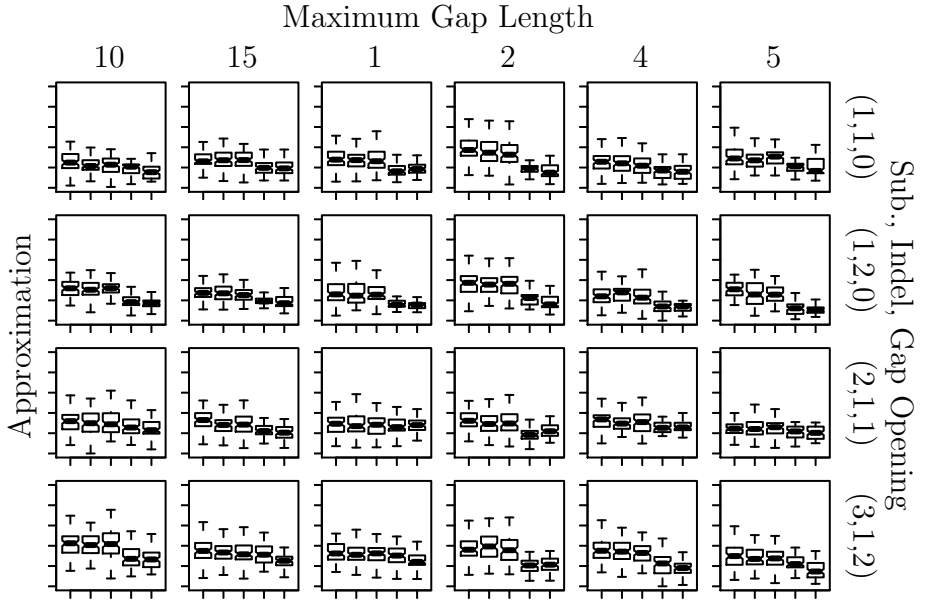


Figure B.6: Build algorithm comparison under Exhaustive Affine-DO. Average branch length 0.3

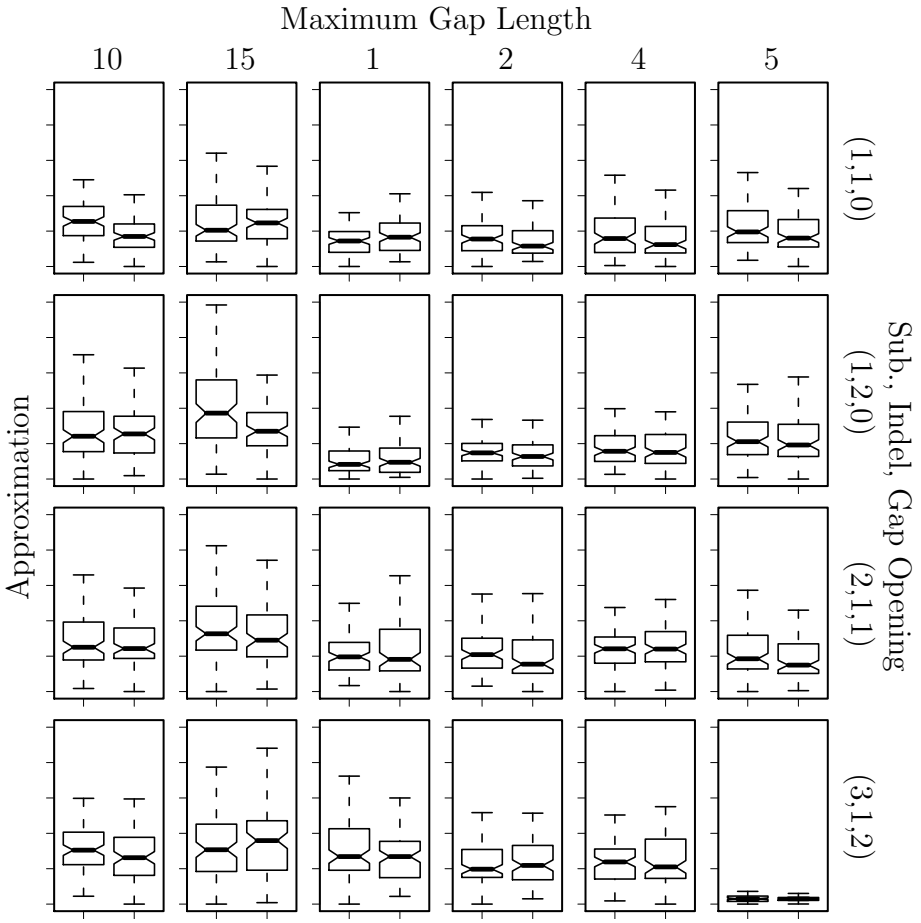


Figure B.7: Comparison between Normal and Exhaustive Affine-DO build. Average branch length 0.1.

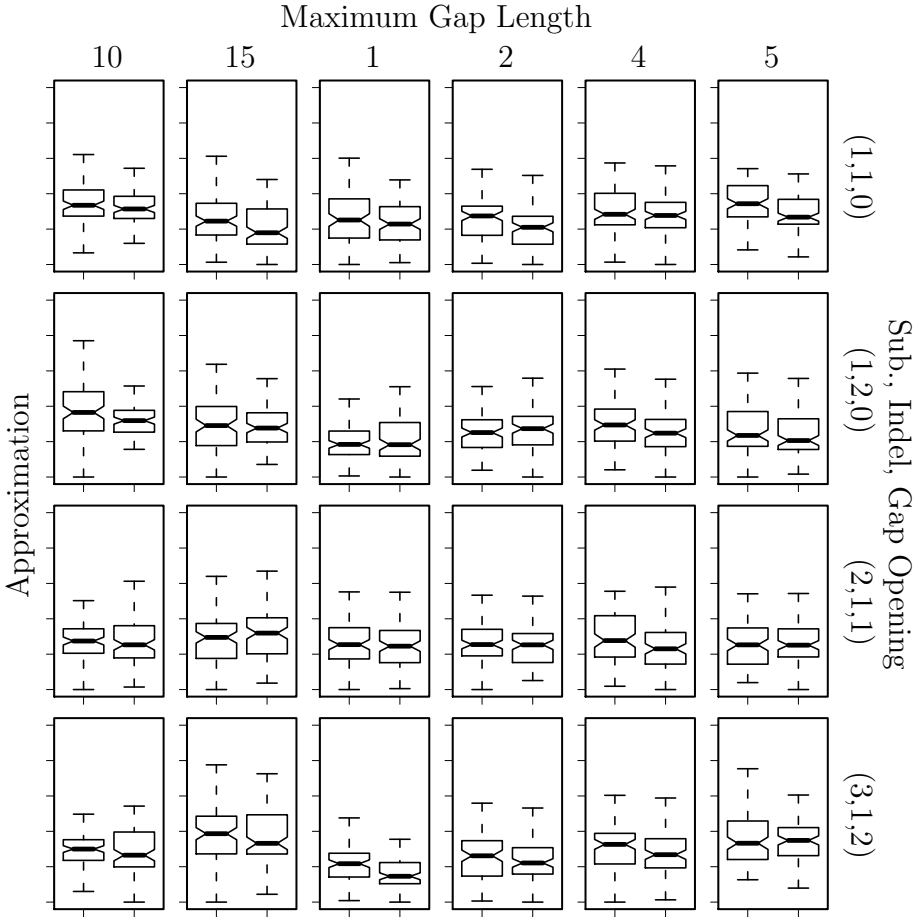


Figure B.8: Comparison between Normal and Exhaustive Affine-DO build. Average branch length 0.2.

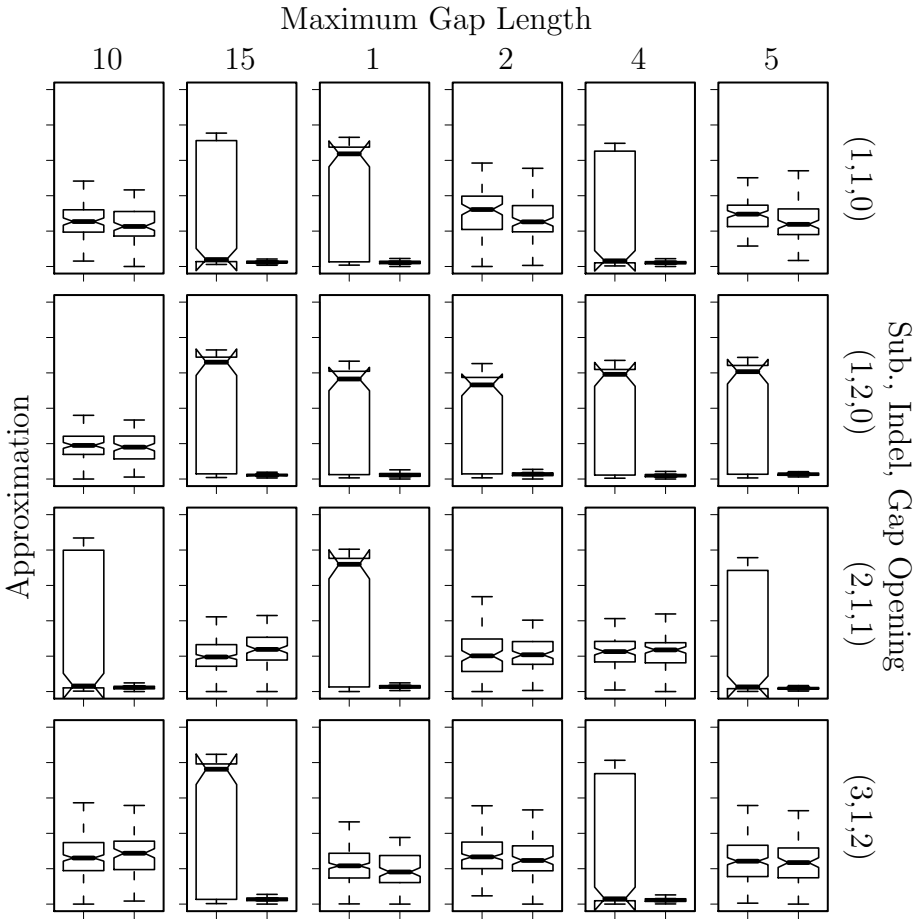


Figure B.9: Comparison between Normal and Exhaustive Affine-DO build. Average branch length 0.3.

B.2 Local Search Comparison

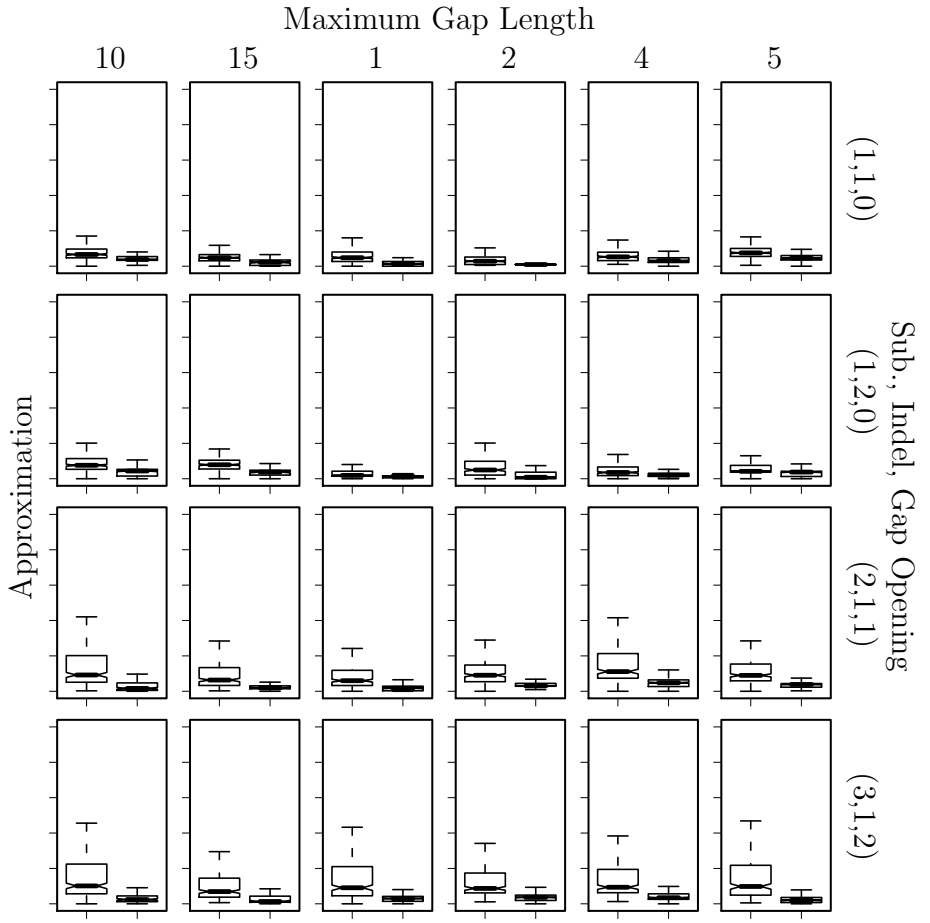


Figure B.10: Comparison between Normal and Exhaustive Affine-DO local search. Average branch length 0.1.

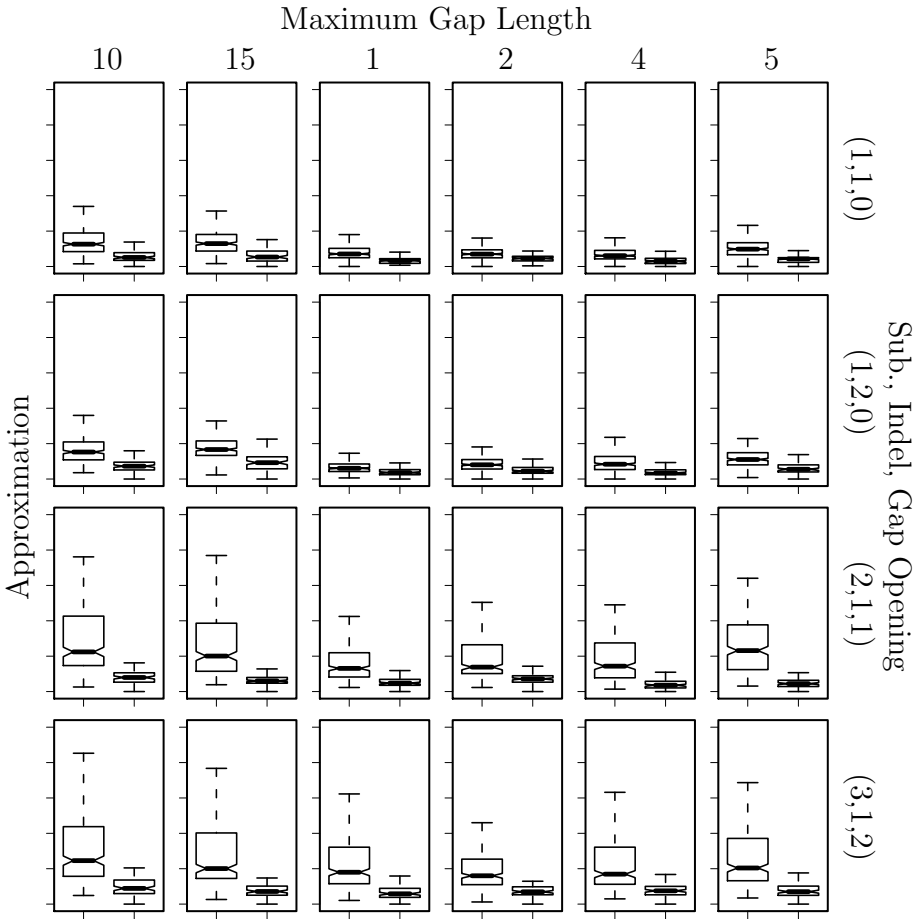


Figure B.11: Comparison between Normal and Exhaustive Affine-DO local search. Average branch length 0.2.

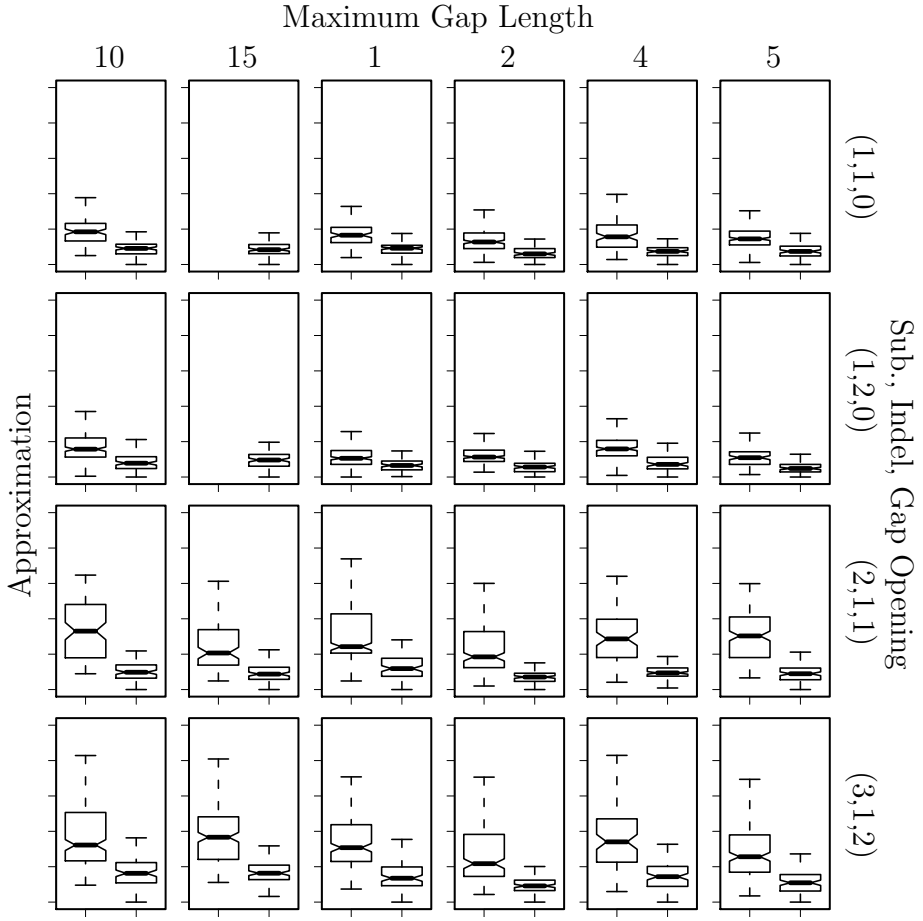


Figure B.12: Comparison between Normal and Exhaustive Affine-DO local search. Average branch length 0.3.

B.3 Annealing Parameter Comparison

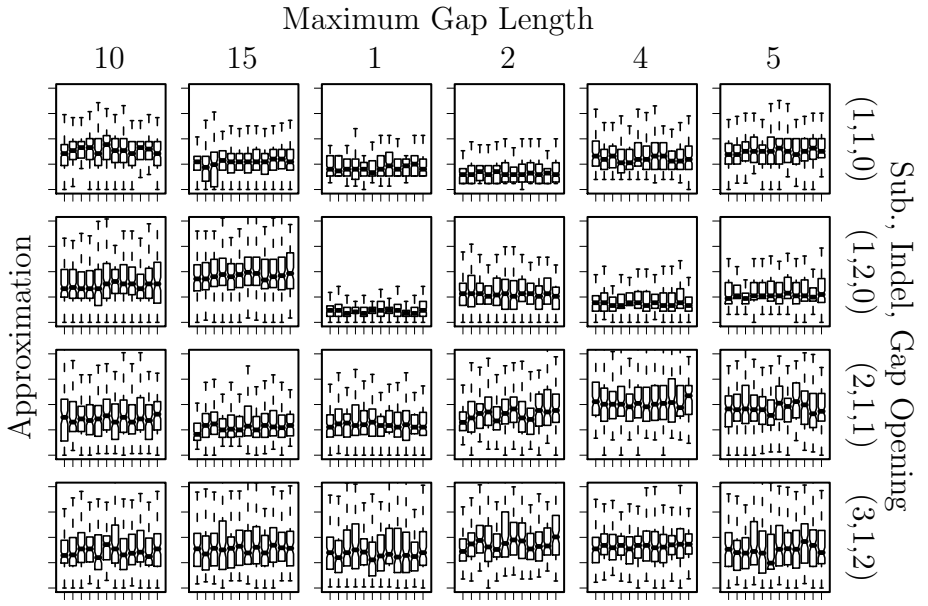


Figure B.13: Comparison various simulated annealing parameters. Average branch length 0.1.

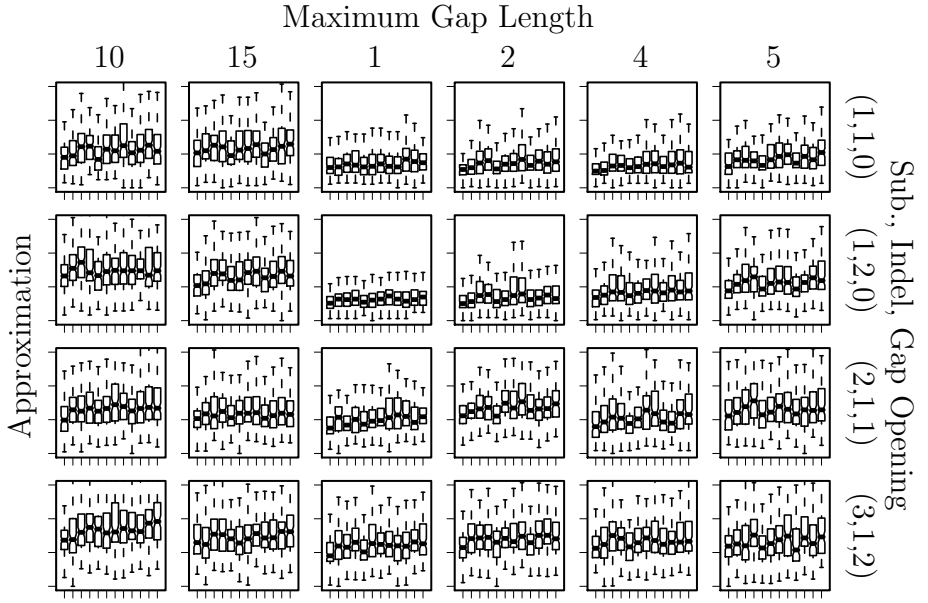


Figure B.14: Comparison various simulated annealing parameters. Average branch length 0.2.

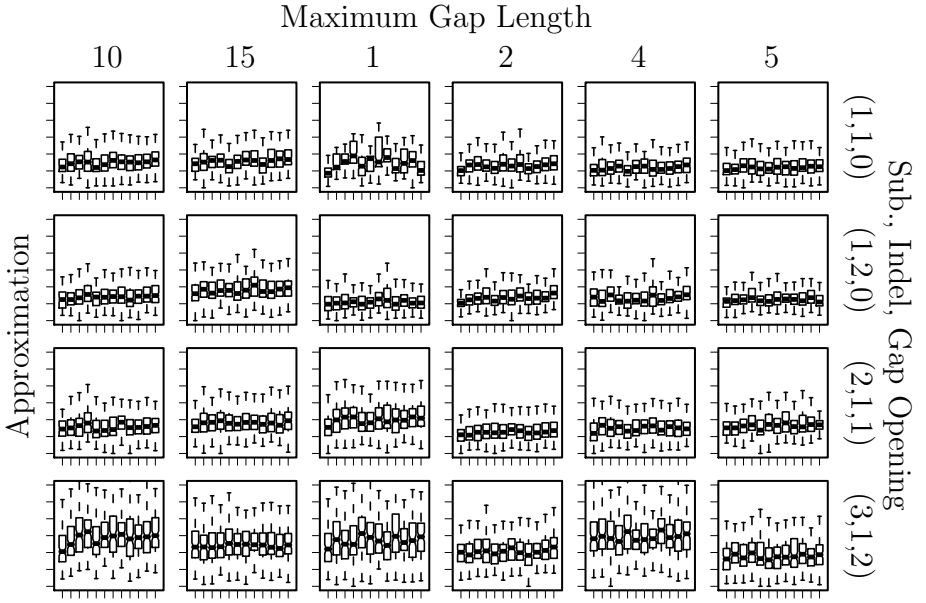


Figure B.15: Comparison various simulated annealing parameters. Average branch length 0.3.

B.4 TBR and Union Comparison

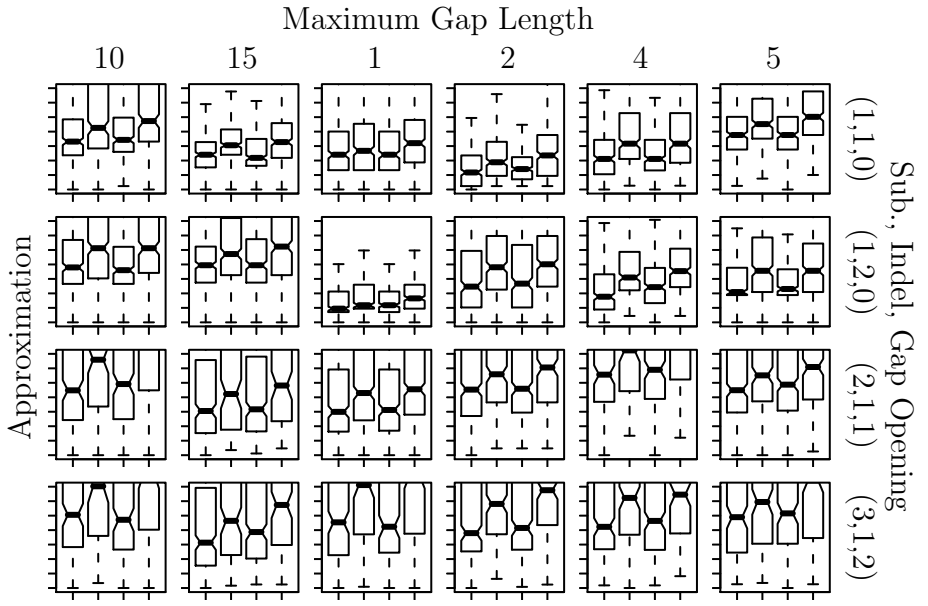


Figure B.16: Randomized versus Union-prunning Affine-DO TBR local search. Average branch length 0.1.

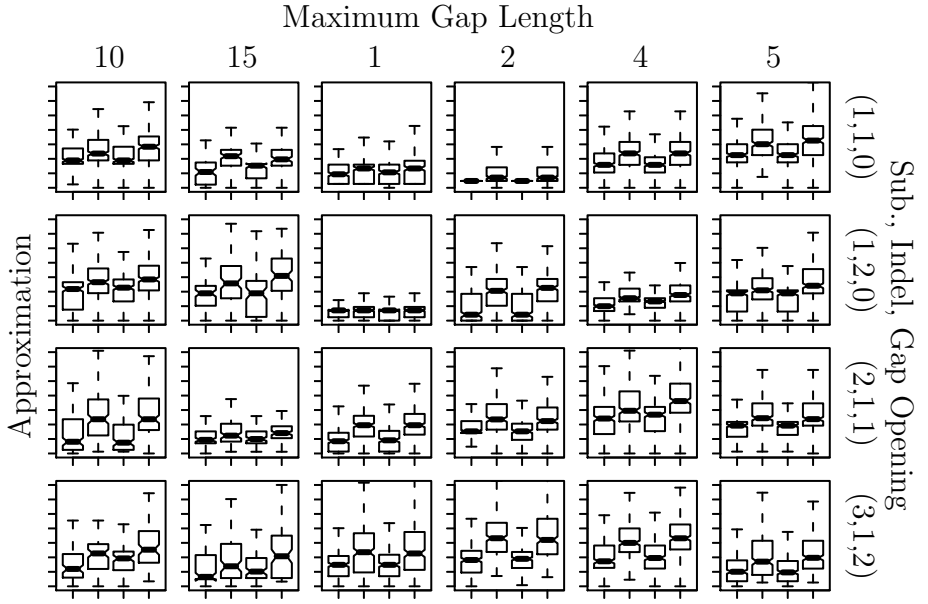


Figure B.17: Randomized versus Union-prunning Exhaustive-Affine-DO TBR local search. Average branch length 0.2.

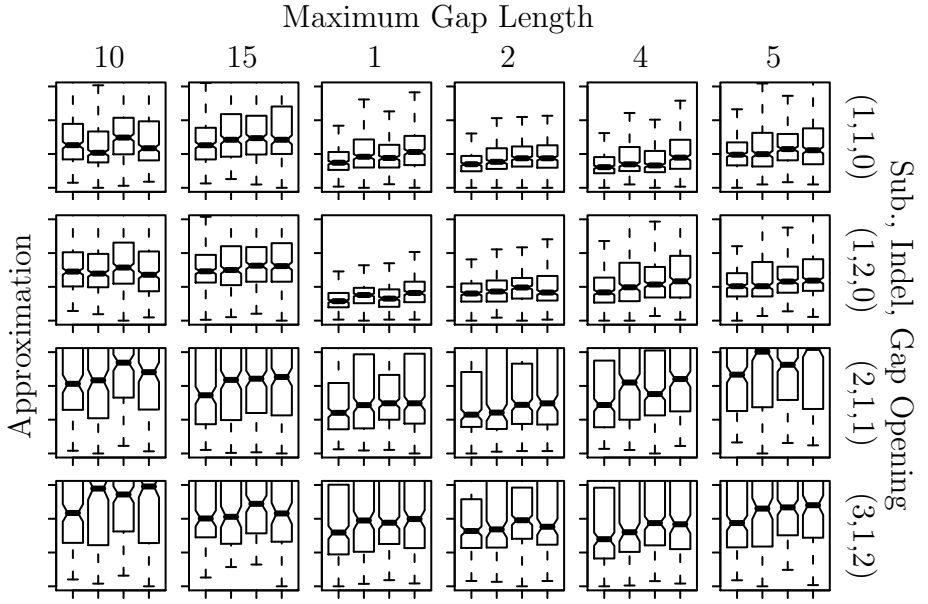


Figure B.18: Randomized versus Union-prunning Affine-DO TBR local search. Average branch length 0.2.

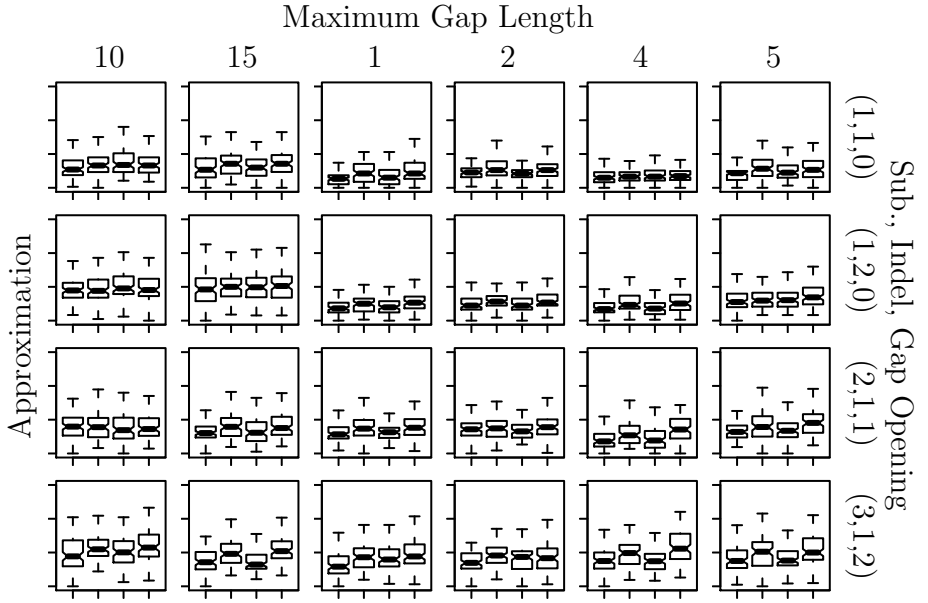


Figure B.19: Randomized versus Union-prunning Exhaustive-Affine-DO TBR local search. Average branch length 0.2.

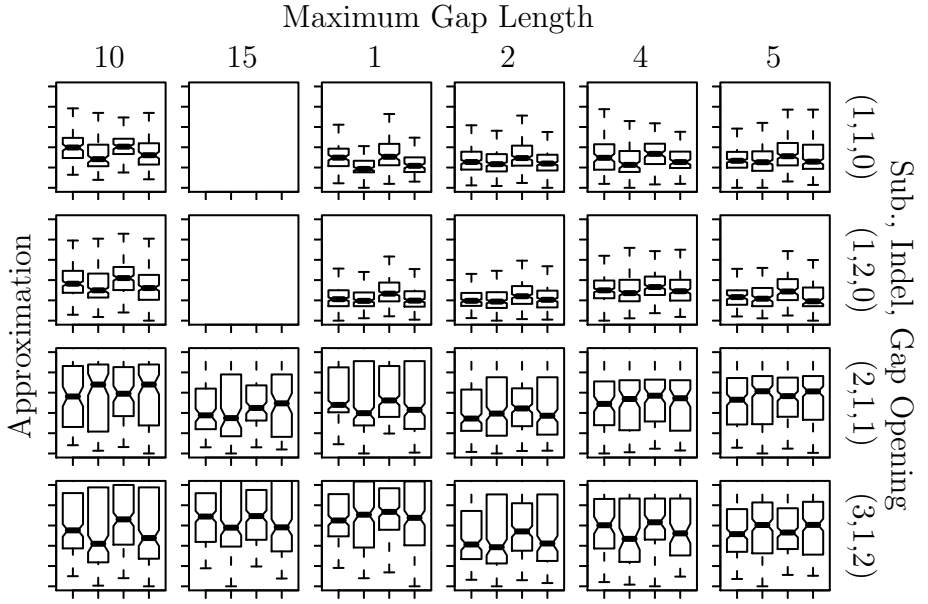


Figure B.20: Randomized versus Union-prunning Affine-DO TBR local search. Average branch length 0.3.

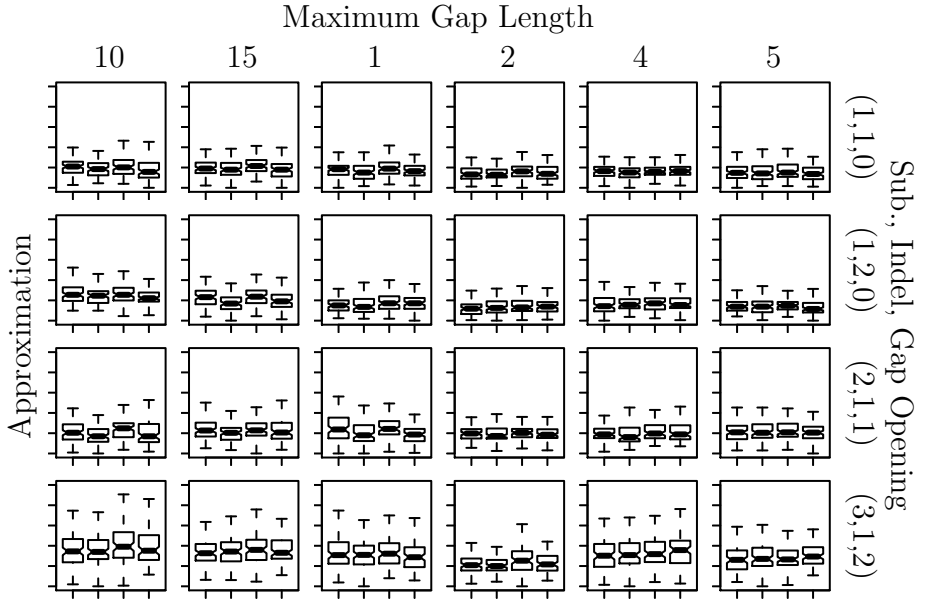


Figure B.21: Randomized versus Union-prunning Exhaustive-Affine-DO TBR local search. Average branch length 0.3.

B.5 Union Only

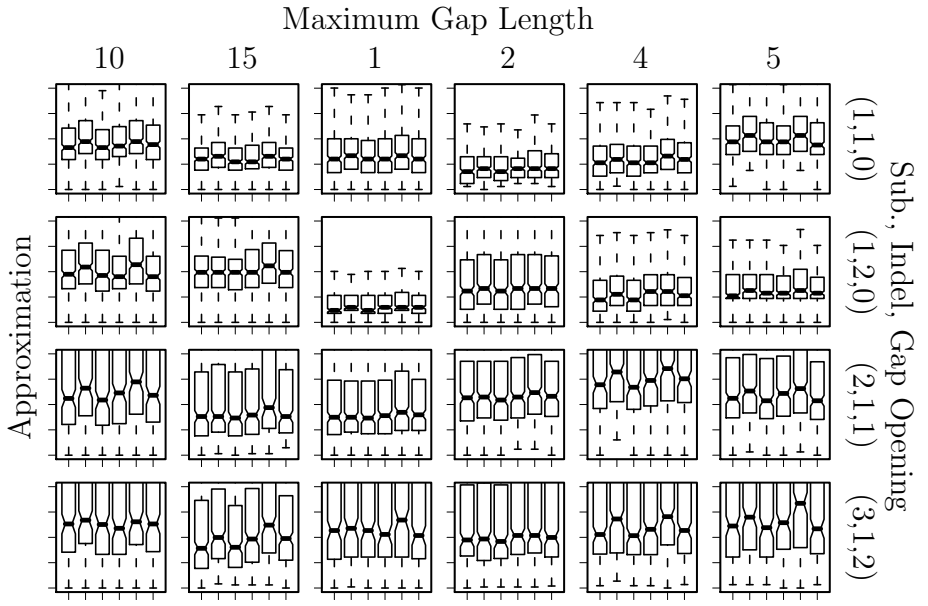


Figure B.22: Union-prunning Affine-DO TBR local search variations comparison. Average branch length 0.1.

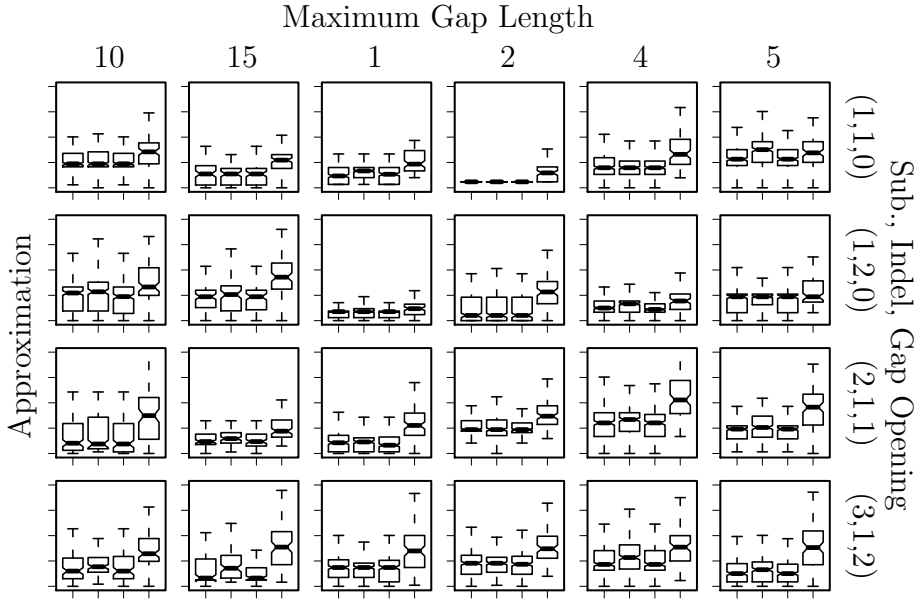


Figure B.23: Union-prunning Exhaustive-Affine-DO TBR local search variations comparison. Average branch length 0.1.

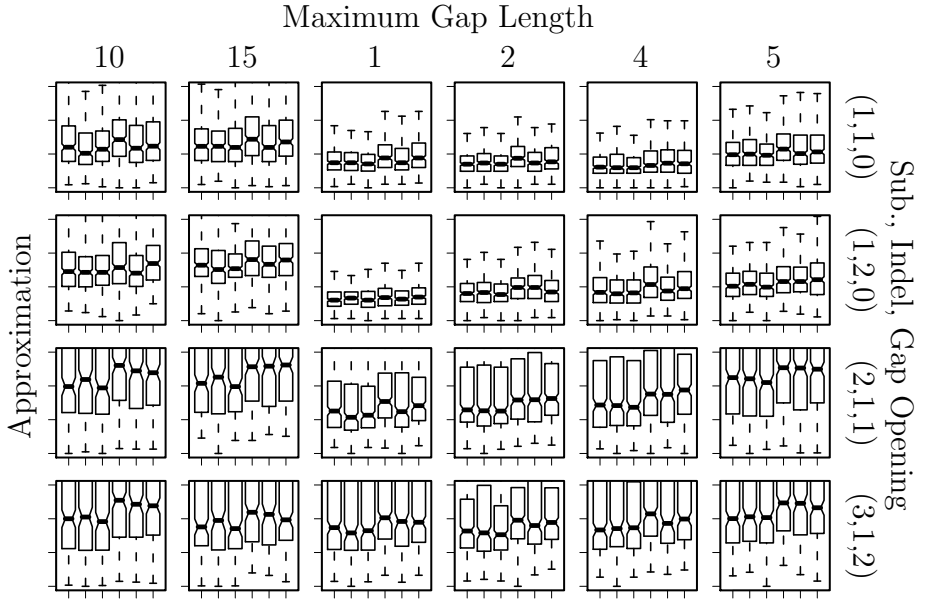


Figure B.24: Union-prunning Affine-DO TBR local search variations comparison. Average branch length 0.2.

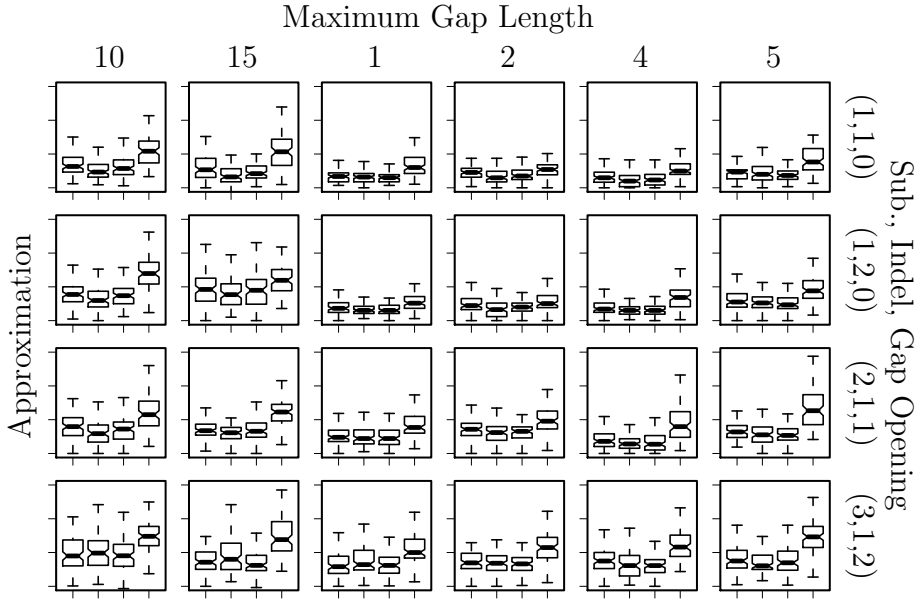


Figure B.25: Union-prunning Exhaustive-Affine-DO TBR local search variations comparison. Average branch length 0.2.

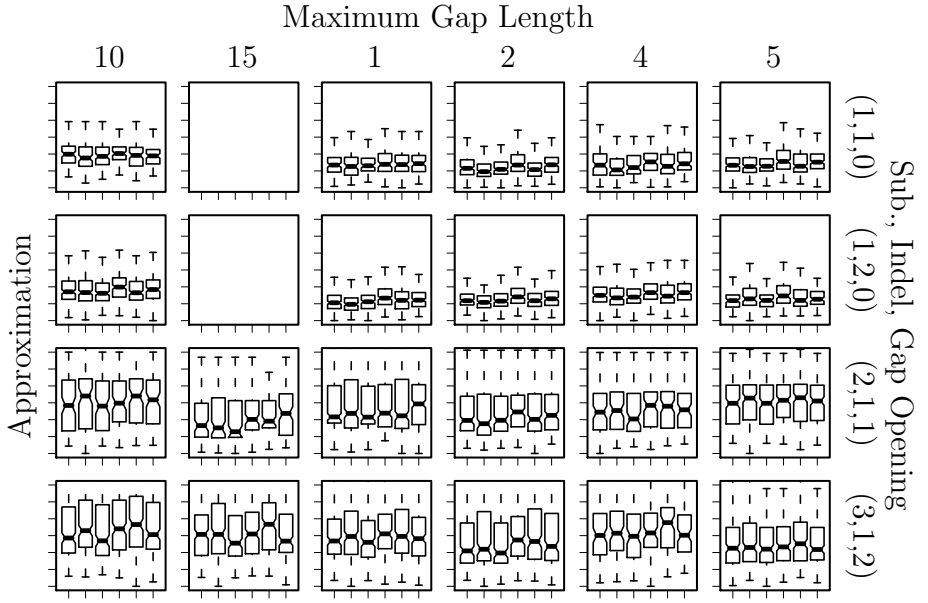


Figure B.26: Union-prunning Affine-DO TBR local search variations comparison. Average branch length 0.3.

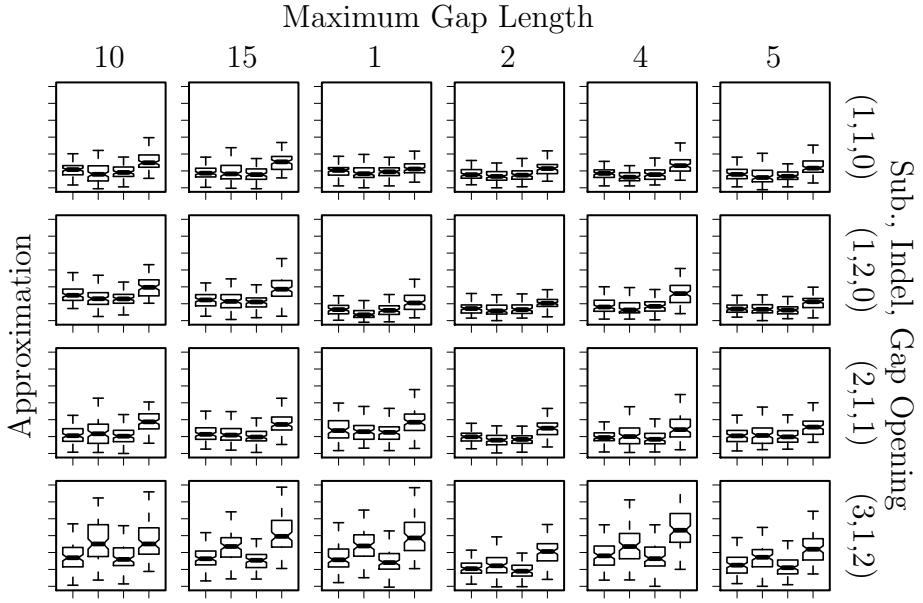


Figure B.27: Union-prunning Exhaustive-Affine-DO TBR local search variations comparison. Average branch length 0.3.

Appendix C

KC

C.1 Model Specifications

In this Section, the complete specification of the functions and SK representation of the models described in Section 6 are listed. The functions are defined in purely functional OCaml [72]. Each SK machine is represented as a black and white figure, where each white square represents a 0 and and a black square a 1. '(' is 0, S is 10, and K is 11. Read from top to bottom and left to right.

C.1.1 General Functions

This section specify a number of generic functions that will be used in the model definitions.

C.1.1.1 Booleans

Functions that can be used for conditional execution in SK.

([m_true] is defined as follows K x y -> x *)*
let m_true = K

```
(* [m_false] is the opposite to true S K x y -> K y (x y) -> y *)
let m_false = S K

(** [m_or x y] is the boolean [x] OR [y]. The argument order in the function
 * reduces the size. *)
let m_or x y = if x then x else y

(** [m_not x] is the negation of [x]. *)
let m_not x = if x then m_false else m_true

module Stream = struct
    (* A function that maps S -> SK, and K -> K. It is used to treat S and K
    as the representation of false and true in a n input stream. *)
    let to_bool x = x S K K m_not m_true
end
```

C.1.1.2 Tuples

Functions to represent pairs of of elements and extract one at a time. These are the basic functions needed to represent any data structure like stacks or lists (e.g. a DNA sequence).

```
(** [pair a b c] is used to represent pairs of elements [a] and [b] by holding
 * their computation. The idea is as follows, [z <- pair a b], then we can use
 * [m_true] and [m_false] to extract either [a] or [b] by applying it to z, as
 * follows [z m_true] -> [m_true a b] -> [a], and [z m_false] -> [m_false a b] ->
 * [b]. *)
let pair a b c = c a b

(* From the description of [pair], it is easy to see that [first] and [second]
  * are [m_true] and [m_false] respectively . *)
let first  = m_true
let second = m_false
```

C.1.1.3 Integer Representation

The simplest of all integer representations are the Church Integers, where S represents 0, and Kx is $x + 1$. Each integer is therefore a list of K's finished with an S.

```
module Church = struct

    (** We represent zero as a pair of false and true *)
    let zero = pair m_false m_true
    let successor x = pair m_true x
    let predecessor x = x second
    let not_zero x = x first
    let is_zero x = m_not (not_zero x)

    let rec equal a b =
        if not_zero a then
            if not_zero b then equal (predecessor a) (predecessor b)
            else m_false
        else equal b a

    let rec gt a b =
        if not_zero a then
            if not_zero b then gt (predecessor a) (predecessor b)
            else m_true
        else m_false

    let rec gt a b =
        if not_zero a then
            if not_zero b then gt (predecessor a) (predecessor b)
            else m_true
        else m_false

    let lt a b = gt b a

    let rec add x y =
        if not_zero x then add (predecessor x) (successor y)
        else y

    let rec substract x y =
        if not_zero y then substract (predecessor x) (predecessor y)
        else x
```

```
    let rec multiply x y =
        if not_zero y then
            add x (multiply x (predecessor y))
        else 0

    let log2 x =
        let rec log2 acc cur x =
            if m_or (gt cur x) (equal cur x) then acc
            else log2 (successor acc) (add cur cur) x
        in
        log2 1 1 x

end
```

Using Church representation is ready, we can take more compact integer representations and decode them into the corresponding Church integer which requires shorter functions. The following functions decode integers that are represented as binary numbers using S as 0 and K as 1 (e.g. 5 is represented KSK). All the decoders are written in continuation passing style as all the decoding will occur in the S stream of the KC hypothesis.

```
module IntegerDecoder = struct
    (* Decode a sequence of [n] [S] symbols to represent the number [n].
    The next symbol must be a $K$ which is passed to the continuation
    function. This decoder is used in the universal integer decoders, which
    need to preprocess the number of bits that are needed to represent
    some integer. *)
    let church_stream continuation =
        let rec _church_stream continuation acc next =
            if Stream.to_bool next then
                continuation acc next
            else
                _church_stream continuation (Church.successor acc)
        in
        _church_stream continuation 0

    (* The basic decoder for integers with uniform probability. *)
    let rec _uniform_max continuation acc bits next =
```

```
        let nacc =
            Church.add (Church.add acc acc)
            (if (Stream.to_bool next) then 1 else 0)
        in
        if Church.equal bits 1 then continuation nacc
        else _uniform_max continuation nacc (Church.predecessor bits)

    (* Decoder for integers with a known number of bits *)
    let uniform_max continuation = _uniform_max continuation 0

    (* Decoder for unbounded integers, all equally likely *)
    let uniform continuation = church_stream (uniform_max continuation)

    (* Decoder for integers with equal probability, in a fixed range. *)
    let uniform_min_max continuation min max =
        let my_continuation decoded_integer =
            continuation (Church.add min decoded_integer)
        in
        _uniform_max my_continuation max

end
```

C.1.1.4 Data Structures

Only two (purely functional) data structures will be used: stacks and an indexed stack (aka arrays).

```
module Stack = struct
    let push x stack = pair [SK S] (pair x stack)
    let empty = pair [SK K] [SK K]
    let pop stack = stack second first
    let rest stack = stack second
    let is_empty stack = Stream.to_bool (stack first)

    let rec inv_merge a b =
        if is_empty a then b
        else inv_merge (rest a) (push (pop a) b)

    let rec merge a b =
        if is_empty a then b
```

```
        else push (pop a) (merge (rest a) b)

    let rec length stack =
        if is_empty stack then Church.zero
        else Church.successor (length (rest stack))
end

module Array = struct
    let rec create n =
        if Church.not_zero n then
            Stack.push Stack.empty (create (Church.predecessor n))
        else Stack.empty

    let rec insert position element arr =
        if Church.not_zero position then
            Stack.push (Stack.pop arr)
                (insert (Church.predecessor position) element (Stack.rest arr))
        else
            Stack.push (Stack.push element (Stack.pop arr)) (Stack.rest arr)
end
```

C.1.2 Hypothesis Shape: Trees and Networks

In this section, decoders for trees and networks are specified. These define the underlying structure of the phylogenetic hypothesis.

C.1.2.1 Tree Hypotheses

The phylogenetic tree decoders assume a binary tree structure. One of the organism states should be the initial `sequence` of the function `tree_hypothesis`. The function `tree_hypothesis` should be the entry point for the decoder.

```
module Tree = struct

    let rec process_tree mechanism stack results sequence next_node_is_leaf =
        let continuation sequence =
            if Stream.to_bool next_node_is_leaf then
```

```
            if Stack.is_empty stack then (Stack.push sequence results)
            else
                process_tree mechanism (Stack.rest stack) results
                    (Stack.pop stack)
        else
            process_tree mechanism (Stack.push sequence stack) results
            sequence
    in
    mechanism continuation

let tree_hypothesis mechanism sequence =
    process_tree mechanism (Stack.push sequence Stack.empty)
    Stack.empty sequence

end
```

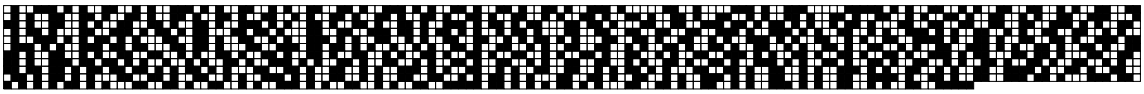


Figure C.1: SK machine of `tree_hypothesis`.

C.1.2.2 Network Hypotheses

The phylogenetic network hypothesis decoder follows the encoding pattern described in Figure 6.5. The `sequence` argument of the `network_hypothesis` is the state assigned to the *root* of the network, that is, the only vertex at level 0 and has indegree 0.

```
module Network = struct

    let rec item_processor _uniform_max max_offset max_levels
    continuation mechanism levels current_level previous_levels
    seq offset =
        if Church.not_zero offset then
            let process_output_of_mechanism result =
                item_processor _uniform_max max_offset max_levels
```

```
            continuation
            mechanism
            (Array.insert (Church.predecessor offset) result levels)
            current_level
            previous_levels seq
        in
        mechanism process_output_of_mechanism seq
    else
        (* We are done with this seq, time to move on *)
        if Stack.is_empty current_level then
            (* We are done with this level, see if there is any more
            * available in the next level *)
            if Church.not_zero max_levels then
                let new_current_level = Stack.pop levels in
                let new_previous_levels =
                    Stack.push new_current_level previous_levels
                in
                let new_levels = Stack.rest levels in
                item_processor _uniform_max
                (Church.predecessor max_levels)
                continuation mechanism
                new_levels (Stack.rest new_current_level)
                new_previous_levels (Stack.pop new_current_level)
            else
                continuation (Stack.pop levels)
        else
            item_processor _uniform_max max_levels continuation
            mechanism levels (Stack.rest current_level) previous_levels
            (Stack.pop current_level)

let identity x = x

let start_processing church_stream _uniform_max mechanism sequence =
    let start max_levels max_offset =
        item_processor _uniform_max max_offset
        (Church.predecessor max_levels) identity mechanism
        (Array.create max_levels) Stack.empty Stack.empty sequence
    in
    let decode_offset max_levels =
        church_stream (_uniform_max (start max_levels) 0)
    in
    church_stream (_uniform_max decode_offset 0)
```

```
    let network_hypothesis mechanism sequence =
        start_processing  IntegerDecoder.church_stream
        IntegerDecoder._uniform_max

end
```

Figure C.2: SK machine of `network_hypothesis`.

C.1.3 Initial sequence generation

This section has the description of various machines that compute an initial state. This state is needed to apply the model on the first edges of the underlying phylogeny (tree or network).

C.1.3.1 Unsigned Chromosomes

Unsigned chromosomes with n genes consist of the sequence $\langle 1, 2, \ldots, n \rangle$. The decoder uses a universal integer decoder to compute n and produce the desired list with Church integers in a stack.

```
let rec aux_identity n stack =
    if Church.not_zero n then
        aux_identity (Church.predecessor n) (Stack.push n stack)
    else stack

let chromosome_identity mechanism =
    let create_initial n =
        mechanism (aux_identity n Stack.empty)
    in
    IntegerDecoder.uniform create_initial
```

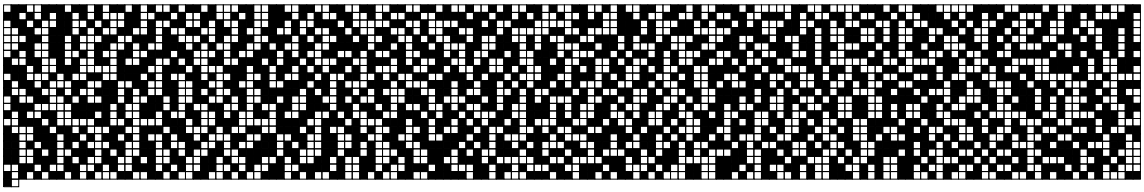


Figure C.3: Function to compute an initial chromosome representation given its length n with the universal integer code, and the corresponding visual representation of its SK machine.

C.1.3.2 Signed Chromosomes

Signed chromosomes follow the same encoding scheme as the unsigned chromosomes of the previous section. The only difference is that each gene is a pair of the gene number, and the sign (which is positive in the initial sequence).

```
let rec aux_signed_identity n stack =
    if Church.not_zero n then
        aux_signed_identity (Church.predecessor n)
        (Stack.push (pair m_true n) stack)
    else stack

let chromosome_signed_identity mechanism =
    let create_initial n =
        mechanism (aux_signed_identity n Stack.empty)
    in
    IntegerDecoder.uniform create_initial
```

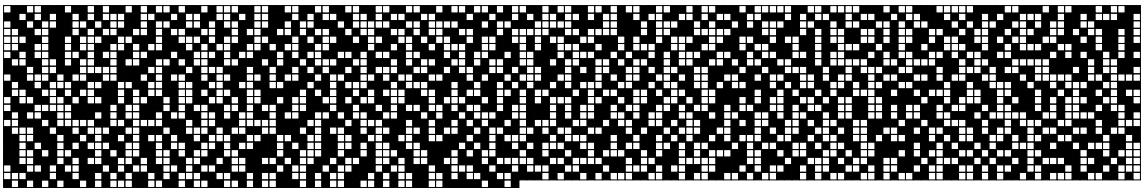


Figure C.4: Function to compute an initial *signed* chromosome representation given its length n with the universal integer code, and the corresponding visual representation of its SK machine.

C.1.4 Insertions, Deletions, and Substitutions

We begin by defining a generic function `substitution` that takes a `decode_base` function as an argument, that can be used to decode the base that will be used in the substitution, according to the model of choice.

```
module IndelSub = struct

let prepend c x y = c (Stack.push x y)

let rec substitution decode_base continuation seq =
    if Stack.is_empty seq then
        let do_substitution something_happened =
            if Stream.to_bool something_happened then
                (decode_base substitution (prepend continuation)
                    seq (Stack.pop seq))
            else
                substitution decode_base
                    (prepend continuation (Stack.pop seq))
                        (Stack.rest seq)
        in
        do_substitution
    else continuation seq
```

As a first case, I would like to define the Jukes-Cantor model decoder, which, regardles the base that is being substitued, decodes a pair of bits to read the new base.

```
let jc_decode_base next prepend seq previous a b =
    next
    (prepend
        (pair (Stream.to_bool a) (Stream.to_bool b)))
        (Stack.rest seq)
```

Now I can apply the function `substitution` with the `jc_decode_base` to produce a substitution machine corresponding to the JC model.

```
let jc_substitution = substitution jc_decode_base
```

I define in the same way a machine for the Kimura Two Parameter model (K2P). In K2P, I decode differently transitions and transversions.

```
let k2p_decode_base next prepend seq previous is_transition =
    if Stream.to_bool is_transition then
        next
            (prepend
                (pair ( first previous)
                    (m_not (second previous))))
            (Stack.rest seq)
    else
        let decode_other d =
            next
                (prepend
                    (pair (m_not (first previous))
                        (Stream.to_bool d)))
                (Stack.rest seq)
        in
        decode_other

let k2p_substitution = substitution k2p_decode_base
```

Finally, I define the GTR model machine, in the decoding pattern is particular to each base that is being substituted.

```
let gtr_decode_base next prepend seq previous =
        let decode_another opt1 opt2 d =
            next
                (prepend
```

```
            (if Stream.to_bool d then opt1
            else opt2))
        (Stack.rest seq)
in
let continue base = next (prepend base) in
let decode_base a b c =
    if a then
        if b then
            (* We are dealing with Adenine *)
            if Stream.to_bool c then
                continue
                    (pair [SK (S K)] [SK K]) (* Guanine *)
            else
                decode_another
                    (pair [SK (S K)] [SK (S K)]) (* Timine *)
                    (pair [SK K] [SK (S K)]) (* Citosine *)
        else
            (* We are dealing with Citosine *)
            if Stream.to_bool c then
                continue
                    (pair [SK (S K)] [SK (S K)]) (* Timine *)
            else
                decode_another
                    (pair [SK K] [SK K]) (* Adenine *)
                    (pair [SK (S K)] [SK K]) (* Guanine *)
    else
        if b then
            (* We are dealing with Guanine *)
            if Stream.to_bool c then
                continue
                    (pair [SK K] [SK K]) (* Adenine *)
            else
                decode_another
                    (pair [SK K] [SK (S K)]) (* Guanine *)
                    (pair [SK (S K)] [SK (S K)]) (* Timine *)
        else
            (* We are dealing with Timine *)
            if Stream.to_bool c then
                continue
                    (pair [SK K] [SK (S K)]) (* Citosine *)
            else
                decode_another
```

```
                    (pair [SK K] [SK K]) (* Adenine *)
                    (pair [SK (S K)] [SK K]) (* Guanine *)
    in
    decode_base (first previous) (second previous)

let gtr_substitution = substitution gtr_decode_base
```

In the same way, I define the generic `indelsub` SK machine that takes the indel model function `indel` and one of the substitution functions defined above.

```
let rec indelsub indel substitution continuation seq =
  if Stack.is_empty seq then (* Only an insertion could occur *)
    let check_insertion do_insertion =
      if Stream.to_bool do_insertion then
        let curry1 a b = a b in
        indel curry1 continuation seq [SK S]
      else continuation seq
    in
    check_insertion
  else
    let do_operation nothing_happened =
      if Stream.to_bool nothing_happened then
        indelsub indel substitution
          (prepend continuation (Stack.pop seq))
            (Stack.rest seq)
      else (* Indel or substitution *)
        let do_indelsub is_substitution =
          if Stream.to_bool is_substitution then
            substitution (indelsub indel substitution)
              (prepend continuation) seq (Stack.pop seq)
          else (* Insertion or deletion *)
            indel (indelsub indel substitution) continuation seq
        in
        do_indelsub
    in
    do_operation
```

Now I define the simplest form of indel model, where insertions and deletions always have length 1, and occur uniformly at random along the sequence.

```
let atomic_indel next continuation seq is_deletion =
```

```
    if Stream.to_bool is_deletion then
        next continuation (Stack.rest seq)
    else (* Is insertion *)
        let decode_base a b =
            next
                (prepend
                    (pair (Stream.to_bool a) (Stream.to_bool b)))
                (Stack.rest seq)
        in
        decode_base
```

We can now plug all the combinations of models that I have defined so far, for substitutions, and the atomic indels. (The corresponding SK machines are shown in Figures C.5, C.6, C.7.)

```
let jc_atomic_indelsub = indelsub atomic_indel jc_decode_base

let k2p_atomic_indelsub = indelsub atomic_indel k2p_decode_base

let gtr_atomic_indelsub = indelsub atomic_indel gtr_decode_base
```

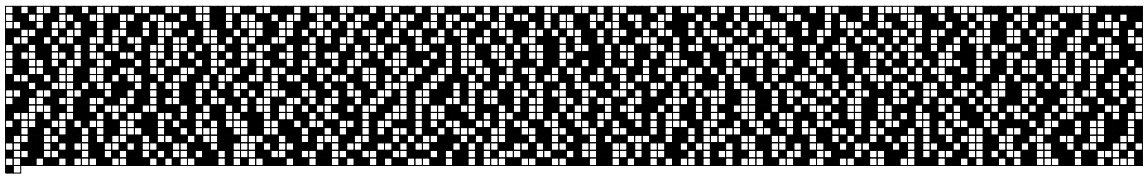


Figure C.5: SK machine for `jc_atomic_indelsub`, atomic Insertions and Deletions with the Jukes-Cantor substitution model.

In the same way, I define the `affine_indel` model, for affine insertions and deletions, as follows.

```
let affine_indel substitution next continuation seq is_deletion =
    if Stream.to_bool is_deletion then
        let rec extend_deletion seq should_continue =
            if Stream.to_bool should_continue then
```

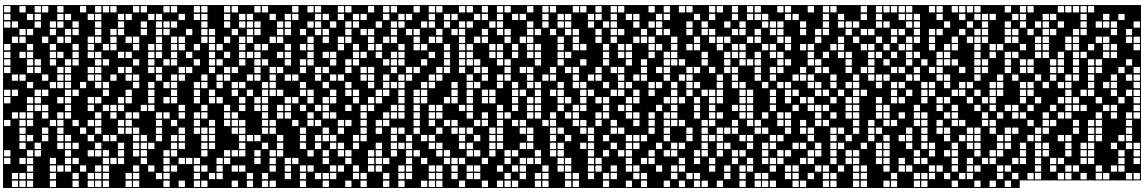


Figure C.6: SK machine for `k2p_atomic_indelsub`, atomic Insertions and Deletions with the K2P substitution model.

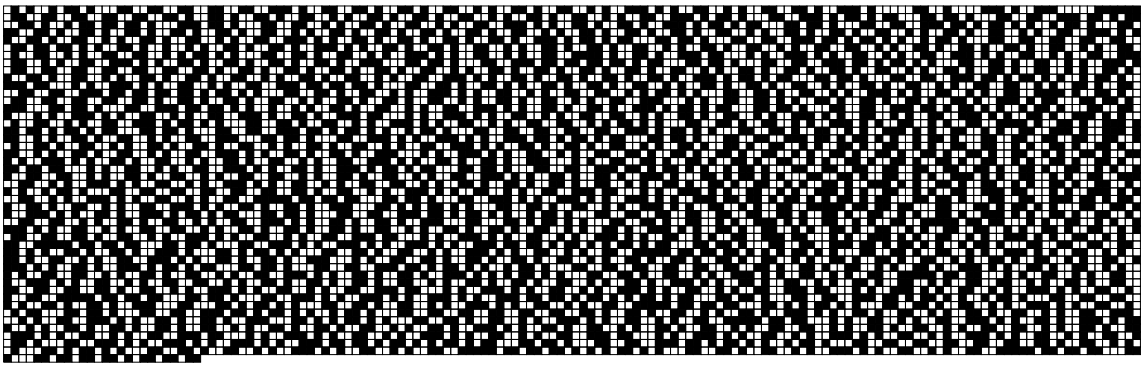


Figure C.7: SK machine for `gtr_atomic_indelsub`, atomic Insertions and Deletions with the GTR substitution model.

```
                extend_deletion (Stack.rest seq)
            else next continuation (Stack.rest seq)
        in
        extend_deletion
    else
        let rec extend_insertion continuation seq should_continue =
            if Stream.to_bool should_continue then
                substitution extend_insertion (prepend continuation)
                    (Stack.push [SK K] seq)
                    [SK K]
            else
                substitution
                    next
                    (prepend continuation)
                    (Stack.push [SK K] seq)
```

```
                    [SK K]
    in
    extend_insertion continuation seq [SK K]
```

And just like before, I create all the model combinations for each substitution model (Figures C.8, C.9, C.10), and we have finished all the insertion, deletion, and substitution models.

```
let affine_indelsub decode_base =
    indelsub ( affine_indel decode_base) decode_base

let jc_affine_indelsub = affine_indelsub jc_decode_base

let k2p_affine_indelsub =
    indelsub ( affine_indelsub jc_decode_base) k2p_decode_base

let gtr_affine_indelsub =
    indelsub ( affine_indelsub jc_decode_base) gtr_decode_base
end
```

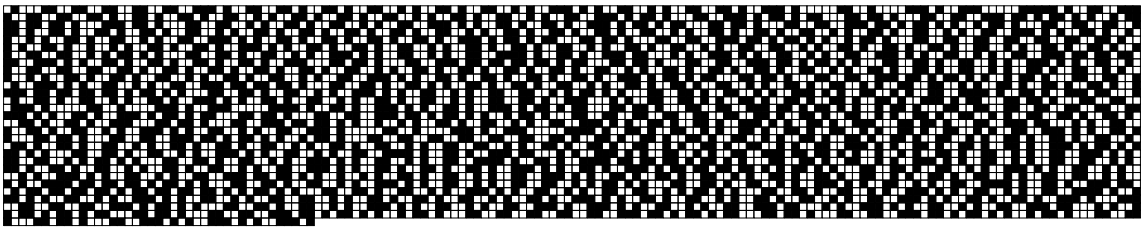


Figure C.8: SK machine for `jc_affine_indelsub`, affine Insertions and Deletions with the Jukes-Cantor substitution model.

C.1.5 Tandem Duplication

The tandem duplication model takes a pair of sequence offsets and duplicates their contents. This model works fine for any sequence that can be represented as a stack.

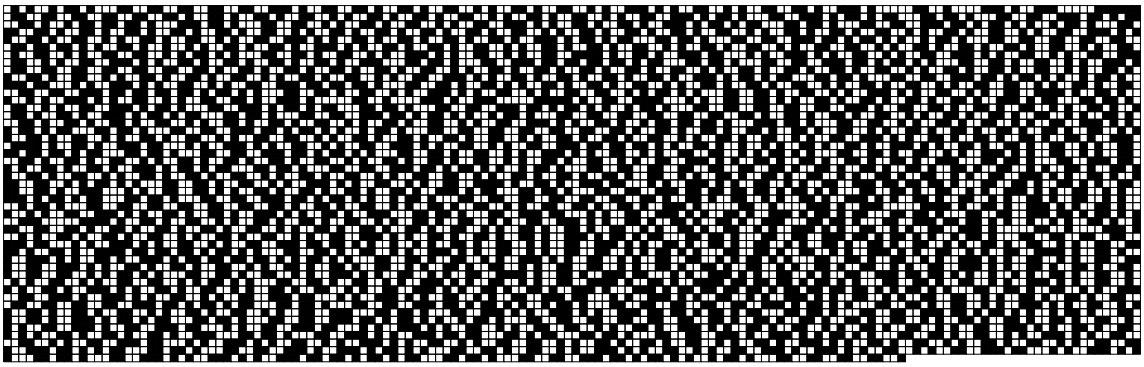


Figure C.9: SK machine for `k2p_affine_indelsub`, affine Insertions and Deletions with the K2P substitution model.

```
module Tandem = struct

    let rec do_duplicate toduplicate seq offset1 offset2 =
        if Church.not_zero offset1 then
            Stack.push (Stack.pop seq)
            (do_duplicate toduplicate (Stack.rest seq) offset1
                (Church.predecessor offset2))
        else if Church.not_zero offset2 then
            Stack.push (Stack.pop seq)
            (do_duplicate (Stack.push (Stack.pop seq) toduplicate)
                (Stack.rest seq) offset1 (Church.predecessor offset2))
        else Stack.inv_merge toduplicate seq

    let decode_positions_and_duplicate continuation decoder bits seq =
        let apply_offsets offset1 offset2 =
            continuation (do_duplicate Stack.empty seq offset1 offset2)
        in
        let decode_offset2 offset1 =
            decoder (apply_offsets offset1) 0 bits
        in
        let decode_offset1 =
            decoder decode_offset2 0 bits
        in
        decode_offset1

    let rec duplicate continuation bits seq another_duplication =
```

Figure C.10: SK machine for `gtr_affine_indelsub`, affine Insertions and Deletions with the GTR substitution model.

```
        if Stream.to_bool another_duplication then
            decode_positions_and_duplicate (duplicate continuation bits)
            IntegerDecoder._uniform_max bits seq
        else continuation bits seq

end
```

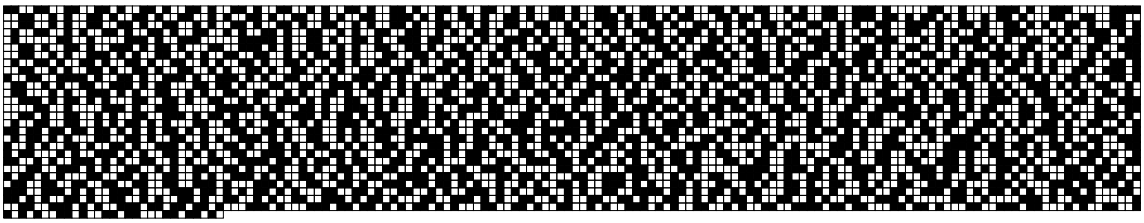


Figure C.11: SK machine `duplicate`, for tandem duplication of sequences model.

C.1.6 Tandem Duplication - Random Loss

This Section specifies the classic TDRL model of genome rearrangement [11], where all TDRL have equal probability.

```
module TDRL = struct

    let prepend c x y = c (Stack.push x y)
    let identity x = x

    let rec process fh sh seq =
        if Stack.is_empty seq then sh (fh (Stack.empty))
        else
            let process_next item =
                (if (Stream.to_bool item) then
                    (process (prepend fh (Stack.pop seq)) sh
                    (Stack.rest seq))
                else
                    (process fh (prepend sh (Stack.pop seq))
                    (Stack.rest seq)))

            in
            process_next

    let rec equal_tdrl continuation seq another_tdrl =
        if Stream.to_bool another_tdrl then
            process identity (equal_tdrl continuation) seq
        else continuation seq

end
```

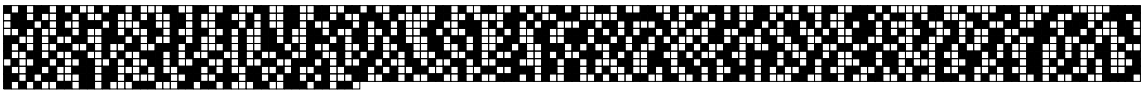


Figure C.12: SK machine `equal_tdrl` for Tandem Duplication – Random Loss model.

C.1.7 Geometric Tandem Duplication - Random Loss

In this Section, the TDRL model is modified to follow a geometric distribution in the length of the TDRL.

```
module GTDRL = struct

    let prepend c x y = c (Stack.push x y)

    let identity x = x

    let curry1 f x = f x

    let rec process fh sh seq position len =
        if Church.not_zero position then
            process (prepend fh (Stack.pop seq)) sh
            (Stack.rest seq) (Church.predecessor position)
            len
        else
            if Church.not_zero len then
                let process_next item =
                    curry1
                    (if Stream.to_bool item then
                        process (prepend fh (Stack.pop seq)) sh
                        (Stack.rest seq)
                    else
                        process fh (prepend sh (Stack.pop seq))
                        (Stack.rest seq)) (Church.predecessor len)

                in
                process_next
            else sh (fh seq)

    let rec gtdrl continuation loglen seq another_tdrl =
        if Stream.to_bool another_tdrl then
            let decode_len position =
                IntegerDecoder.church_stream
                    (process identity (gtdrl continuation loglen) seq
                        position)
            in
            IntegerDecoder._uniform_max decode_len 0 loglen
```

else continuation seq

end

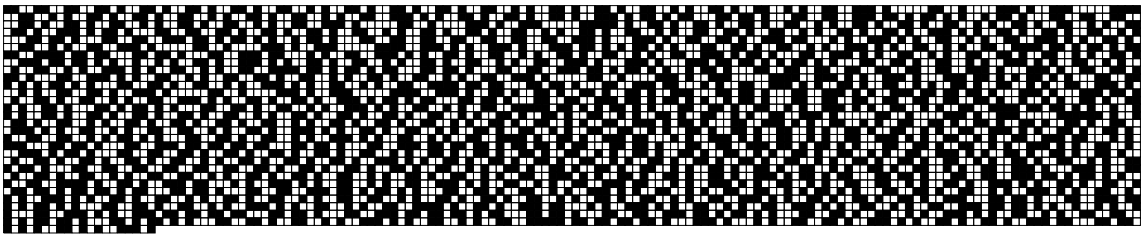


Figure C.13: SK machine `gtdrl` for the Geometric Tandem Duplication – Random Loss model.

C.1.8 Inversion

The signed inversion model, decodes offsets in the universal integer code, to find the beginning and end of the inverted segment.

```
module Inversion = struct
    let rec invert_and_merge stack1 stack2 =
        if Stack.is_empty stack1 then stack2
        else
            let a_head = Stack.pop stack1 in
            let a_sign = m_not (a_head second first) in
            let a_base = a_sign second second in
            let new_a_head = pair m_false (pair a_sign a_base) in
            Stack.push new_a_head (invert_and_merge (Stack.rest stack1) stack2)


    let rec process genome stack offset1 offset2 =
        if Church.not_zero offset1 then
            Stack.push (Stack.pop genome)
            (process (Stack.rest genome) stack (Church.predecessor offset1)
             offset2 )
        else
            if Church.not_zero offset2 then
                process
```

```
                (Stack.rest genome)
                (Stack.push (Stack.pop genome) stack)
                offset1
                (Church.predecessor offset2)
            else
                invert_and_merge stack genome

    let rec my_process _uniform_max continuation genome length do_nothing =
        if do_nothing then continuation genome length
        else
            let decode_operation offset1 offset2 =
                my_process _uniform_max continuation
                (process genome Stack.empty offset1 offset2) length
            in
            let decode_offset2 offset1 =
                _uniform_max (decode_operation offset1) 0
                length
            in
            let decode_offset1 =
                _uniform_max decode_offset2 0 length
            in
            decode_offset1

    let inversion continuation genome length =
        my_process IntegerDecoder._uniform_max continuation genome length
        do_nothing
end
```

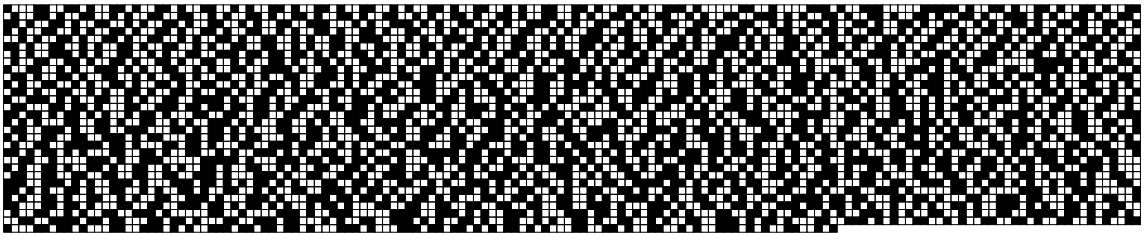


Figure C.14: SK machine `inversion` for the Inversion model.

C.1.9 Double Cut and Join

C.1.9.1 Initial Sequence Decoding

```
let dcj_identity mechanism =
    let rec build_genome genome cnt =
        if Church.not_zero cnt then
            let check_if_internal_limit is_internal =
                if Stream.to_bool is_internal then
                    build_genome
                    (Stack.push
                        (pair m_false (pair m_true cnt)) genome)
                    (Church.predecessor cnt)
                else
                    let choose_class circular_mark =
                        build_genome
                        (Stack.push (marker (Stream.to_bool circular_mark))
                            genome)
                        cnt
                    in
                    choose_class
            in
             check_if_internal_limit
        else mechanism genome
    in
    IntegerDecoder.church_stream
        (IntegerDecoder._uniform_max (build_genome Stack.empty) 0)
```

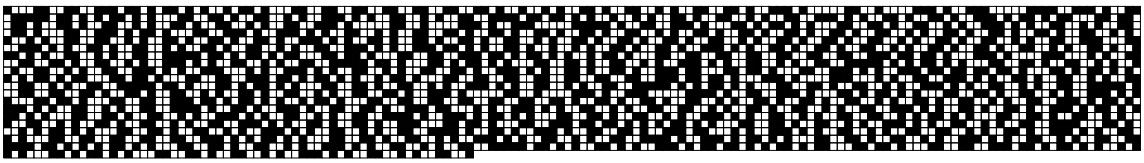


Figure C.15: Identity function creation for the DCJ mechanism. The function decodes an integer n using the universal decoder, and continues with a stream of at most $2n$ specifying the limits and kind (i.e. circular or linear), of the chromosomes in the initial genome.

C.1.9.2 Transformations

```
module DCJ = struct

    let rec invert_and_merge stack1 stack2 =
        if Stack.is_empty stack1 then stack2
        else
            let a_head = Stack.pop stack1 in
            let a_sign = m_not (a_head second first) in
            let a_base = a_head second second in
            let new_a_head = pair m_false (pair a_sign a_base) in
            Stack.push new_a_head (invert_and_merge (Stack.rest stack1) stack2)


    let is_chromosome_limit marker = marker first

    let rec are_in_same_chromosome offset1 offset2 genome =
        let first_base = Stack.pop genome in
        let genome = Stack.rest genome in
        if Church.not_zero offset1 then
            let offset1 = Church.predecessor offset1 in
            are_in_same_chromosome
             offset1  offset2
            genome
        else
            if Church.not_zero offset2 then
                if is_chromosome_limit first_base then
                    m_false
                else
                    are_in_same_chromosome offset1
                    (Church.predecessor offset2)
                    genome
            else m_true

    let apply1 f x = f x
    let marker x = pair m_true x
    let circular_mark = marker m_true
    let linear_mark = marker m_false

    let invert_in_place inv_merge stack1 stack2 genome =
        invert_and_merge stack1
            (inv_merge stack2 genome)
```

```
let make_circular_in_place inv_merge stack1 stack2 genome =
    inv_merge stack2
        (Stack.push circular_mark
            (inv_merge stack1 genome))

let rec in_one_chromosome function_to_apply stack1 stack2 offset1
  offset2 genome =
    let first_base = Stack.pop genome in
    let genome = Stack.rest genome in
    if Church.not_zero offset1 then
        let offset1 = Church.predecessor offset1 in
        Stack.push first_base
        (in_one_chromosome function_to_apply stack1 stack2 offset1
            offset2 genome)
    else
        if Church.not_zero offset2 then
            in_one_chromosome
            function_to_apply
            (Stack.push first_base stack1)
            stack2
            offset1
            (Church.predecessor offset2)
            genome
        else
            if is_chromosome_limit first_base then
                function_to_apply Stack.inv_merge stack1 stack2
                (Stack.push first_base genome)
            else
                (in_one_chromosome function_to_apply stack1
                (Stack.push first_base stack2)
                offset1 offset2 genome)

let in_one_chromosome f x offset1 offset2 offset3 genome =
    in_one_chromosome f x x x offset1 offset2 offset3 genome

let rec second_cut inv_merge invert_and_merge is_circular first_half
  second_half first_cut position2 invert genome =
    let first_element = Stack.pop genome in
    let genome = Stack.rest genome in
    if Church.not_zero position2 then
        if is_chromosome_limit first_element then
```

```
            Stack.push (marker is_circular )
            (inv_merge first_half
            (second_cut inv_merge invert_and_merge (first_element second)
            Stack.empty Stack.empty first_cut
            (Church.predecessor position2) invert genome))
        else
            second_cut inv_merge invert_and_merge is_circular
            (Stack.push first_element first_half )
            second_half first_cut (Church.predecessor position2) invert
            genome
else
    (* Wow we have reached the breaking point! At last! *)
    if first_cut first then
        (* The first one was circular, this is easy *)
        invert_and_merge first_half
        (invert_and_merge ( first_cut second)
        (Stack.push first_element genome))
    else
        (* The first cut was linear, then handling depends on the kind
        * of this one *)
        if is_circular then
            if is_chromosome_limit first_element then
                Stack.push linear_mark
                (inv_merge
                ( first_cut second first )
                (apply1
                    (if invert then invert_and_merge first_half else
                        inv_merge second_half)
                (apply1
                    (if invert then invert_and_merge second_half else
                        inv_merge first_half )
                    (inv_merge ( first_cut second second)
                    (Stack.push first_element genome)))))
            else
                second_cut inv_merge invert_and_merge is_circular
                    first_half (Stack.push first_element second_half)
                        first_cut position2 invert genome
        else
            Stack.push linear_mark
            (inv_merge first_half
            (apply1 (if invert then inv_merge (first_cut second second)
                else invert_and_merge ( first_cut second first ))
```

```
                (Stack.push linear_mark
                (apply1 (if invert then inv_merge (first_cut second first )
                    else invert_and_merge ( first_cut second second))
                (Stack.push first_element genome)))))

let rec first_cut invert_and_merge is_circular first_half second_half
  position1 position2 invert genome =
    let position2 = Church.predecessor position2 in
    let first_element = Stack.pop genome in
    let genome = Stack.rest genome in
    if Church.not_zero position1 then
        if is_chromosome_limit first_element then
            Stack.push (marker is_circular )
            (invert_and_merge first_half
                (let is_circular = first_element second in
                 first_cut invert_and_merge is_circular Stack.empty Stack.empty
                (Church.predecessor position1) position2 invert genome))
        else
             first_cut invert_and_merge is_circular
            (Stack.push first_element first_half ) second_half
            (Church.predecessor position1) position2 invert genome
    else
        if is_chromosome_limit first_element then
            second_cut Stack.inv_merge invert_and_merge (first_element second)
            Stack.empty Stack.empty
            (pair is_circular
                (if is_circular then
                    Stack.inv_merge second_half
                        (invert_and_merge first_half Stack.empty)
                else pair first_half second_half)) position2 invert genome
        else
             first_cut invert_and_merge is_circular first_half
            (Stack.push first_element second_half)
            position1 position2 invert genome

let selector continuation genome position1 position2 invert =
    if are_in_same_chromosome position1 position2 genome then
        continuation
            (apply1
            (in_one_chromosome
            (if invert then invert_in_place else make_circular_in_place)
            Stack.empty)
```

```
                position1 position2 genome)
        else
            let genome = Stack.rest genome in
            let is_circular = head second in
            continuation
                ( first_cut invert_and_merge is_circular Stack.empty Stack.empty
                position1 position2 invert genome)

    let rec do_dcj uniform_max continuation length genome do_nothing =
        if do_nothing then
            continuation genome
        else
            let decode_invert continuation position1 position2 invert =
                ( selector (do_dcj uniform_max continuation length)
                position1 position2 (Stream.to_bool invert) )
            in
            let decode_second_position position1 =
                uniform_max (decode_invert position1) 0 length
            in
            let decode_first_position =
                uniform_max decode_second_position 0 length
            in
             decode_first_position

    let dcj continuation length genome do_nothing =
        do_dcj (IntegerDecoder._uniform_max Church.add)
        continuation length genome do_nothing

end
```

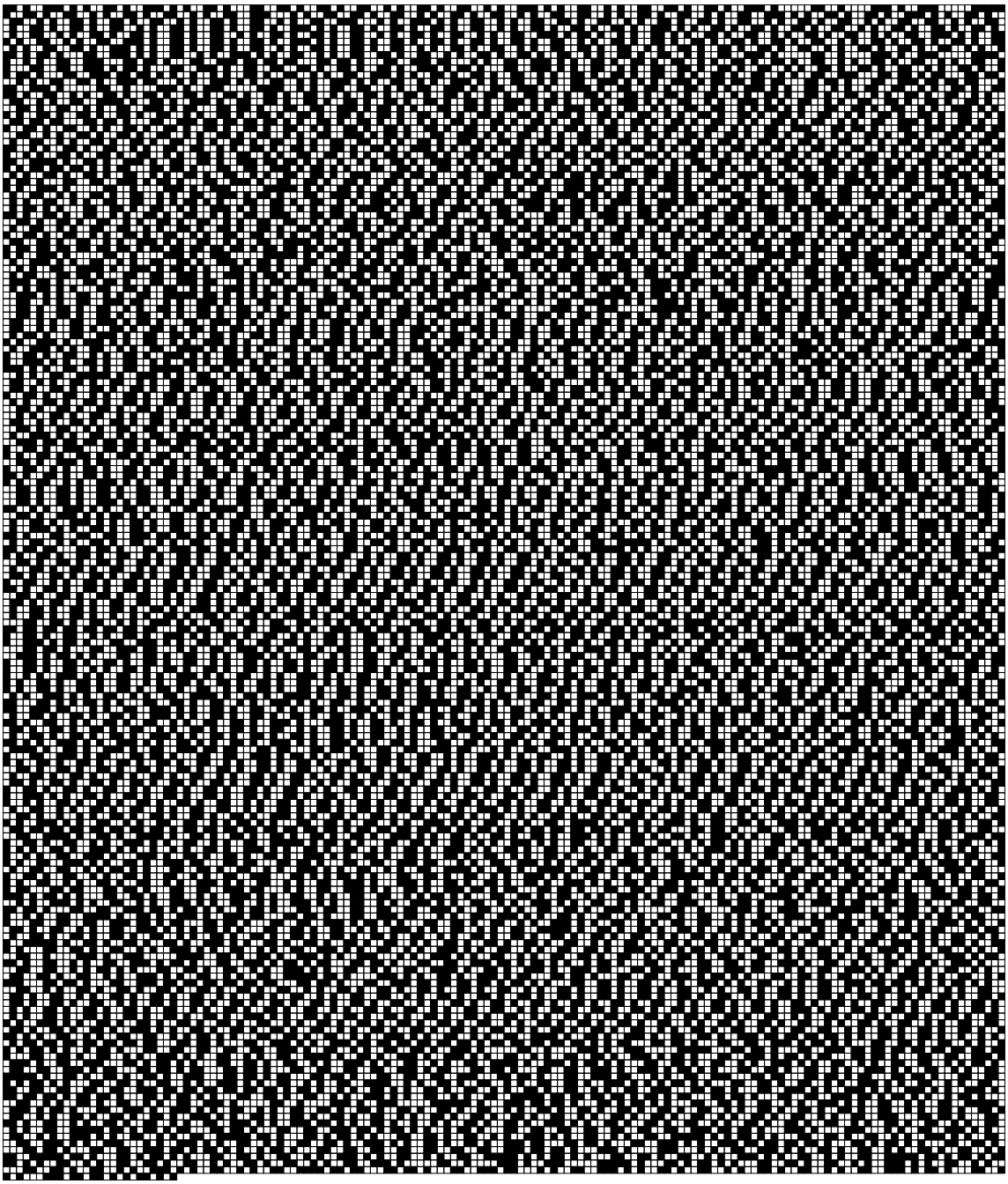


Figure C.16: SK Machine `dcj` for the DCJ model.

C.1.10 Inversion, Fusion, Fission, and Translocation

```
module Grimm = struct
    let is_genome_limit base = base first

    let rec invert_and_merge stack1 stack2 =
        if Stack.is_empty stack1 then stack2
        else
            let a_head = Stack.pop stack1 in
            let a_sign = m_not (a_head second first) in
            let a_base = a_sign second second in
            let new_a_head = pair m_false (pair a_sign a_base) in
            Stack.push new_a_head (invert_and_merge (Stack.rest stack1) stack2)


    let rec invert genome stack offset1 offset2 =
        if Church.not_zero offset1 then
            Stack.push (Stack.pop genome)
            (invert (Stack.rest genome) stack (Church.predecessor offset1)
            offset2)
        else
            if Church.not_zero offset2 then
                invert
                (Stack.rest genome)
                (Stack.push (Stack.pop genome) stack)
                offset1
                (Church.predecessor offset2)
            else
                invert_and_merge stack genome

    let rec beginning_of_offset genome beginning position offset1 offset2 =
        if Church.not_zero offset1 then
            beginning_of_offset (Stack.rest genome)
            (if is_genome_limit (Stack.pop genome) then position
                else beginning)
            (Church.successor position) (Church.predecessor offset1) offset2
        else if Church.not_zero offset2 then
            beginning_of_offset (Stack.rest genome)
            (if is_genome_limit (Stack.pop genome) then position
                else beginning)
            (Church.successor position) offset1 (Church.predecessor offset2)
        else beginning
```

```
let rec end_of_offset genome position offset1 =
    let reached_end_of_chromosome=
        m_and (Church.is_zero offset1) (is_genome_limit (Stack.pop genome))
    in
    if reached_end_of_chromosome then position
    else
        end_of_offset (Stack.rest genome)
        (Church.successor position) (Church.predecessor offset1)

let do_ifft invert genome offset1 offset2 =
    let chromosome_of_1 = beginning_of_offset genome 0 0 offset1 0 in
    let chromosome_of_2 = beginning_of_offset genome 0 0 offset1 offset2 in
    if Church.equal chromosome_of_1 chromosome_of_2 then
        invert genome Stack.empty offset1 offset2
    else
        let end_of_chromosome_1 = end_of_offset genome 0 offset1 in
        let choose_translocation is_simple_translocation =
            let genome =
                invert genome Stack.empty end_of_chromosome_1 chromosome_of_2
            in
            if Stream.to_bool is_simple_translocation then
                invert genome Stack.empty offset1 offset2
            else
                let genome =
                    invert genome Stack.empty chromosome_of_1
                    end_of_chromosome_1
                in
                let genome = invert genome Stack.empty
                    (Church.add
                        chromosome_of_1
                        (Church.substract end_of_chromosome_1 offset1))
                    offset2
                in
                invert genome Stack.empty
                chromosome_of_1
                    (Church.add
                        (Church.substract end_of_chromosome_1 offset1)
                        (Church.substract offset2 chromosome_of_2))
        in
        choose_translocation
```

```
    let rec my_process _uniform_max continuation genome length do_nothing =
        if do_nothing then continuation genome length
        else
            let decode_operation offset1 offset2 =
                my_process _uniform_max continuation
                ( do_ifft invert genome Stack.empty offset1 offset2) length
            in
            let decode_offset2 offset1 =
                _uniform_max (decode_operation offset1) 0
                length
            in
            let decode_offset1 =
                _uniform_max decode_offset2 0 length
            in
            decode_offset1

    let ifft continuation genome length do_nothing =
        my_process IntegerDecoder._uniform_max continuation genome length
        do_nothing
end
```

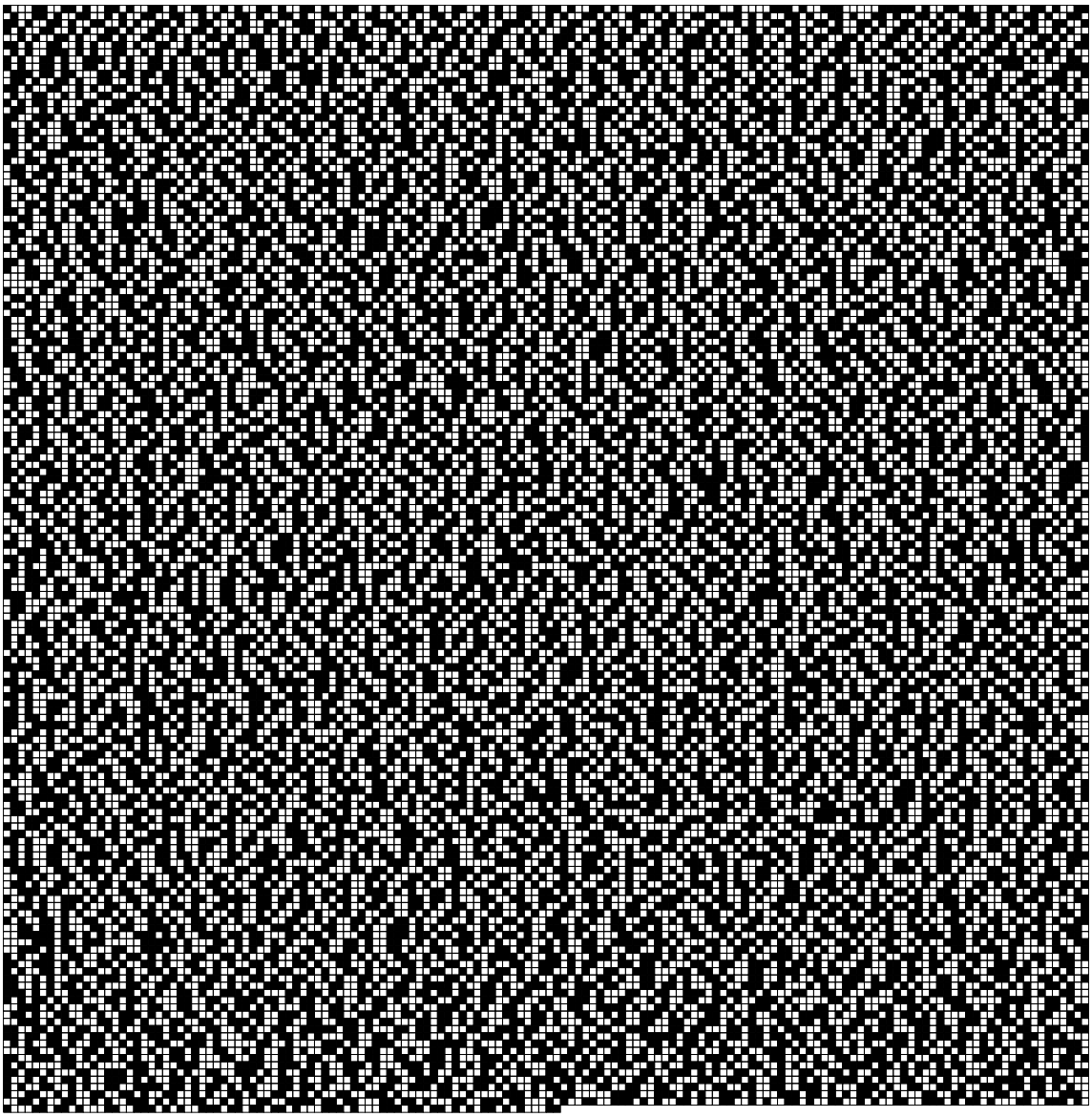
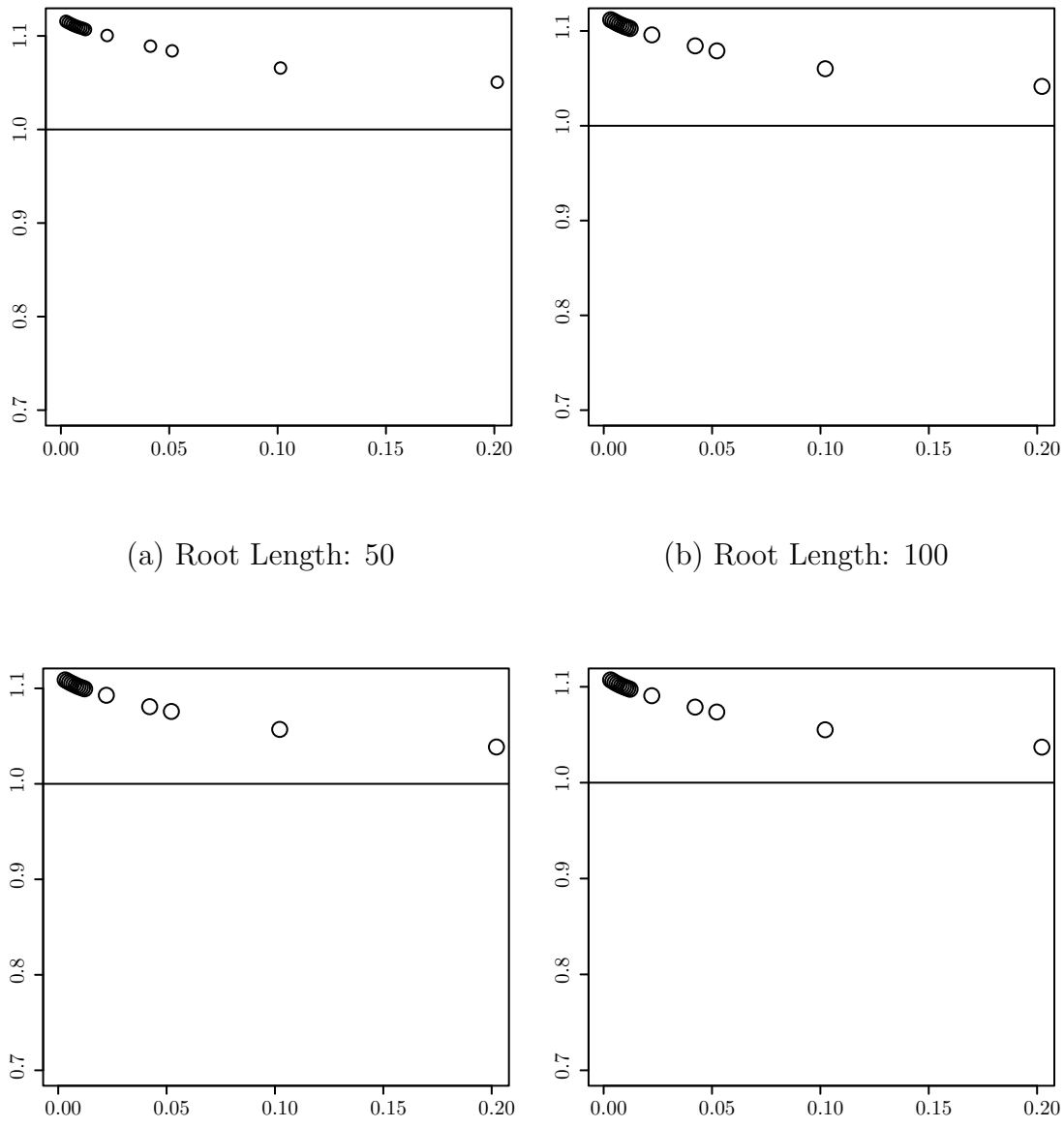


Figure C.17: SK machine `ifft` for the IFFT model.

C.2 Experimental Evaluation



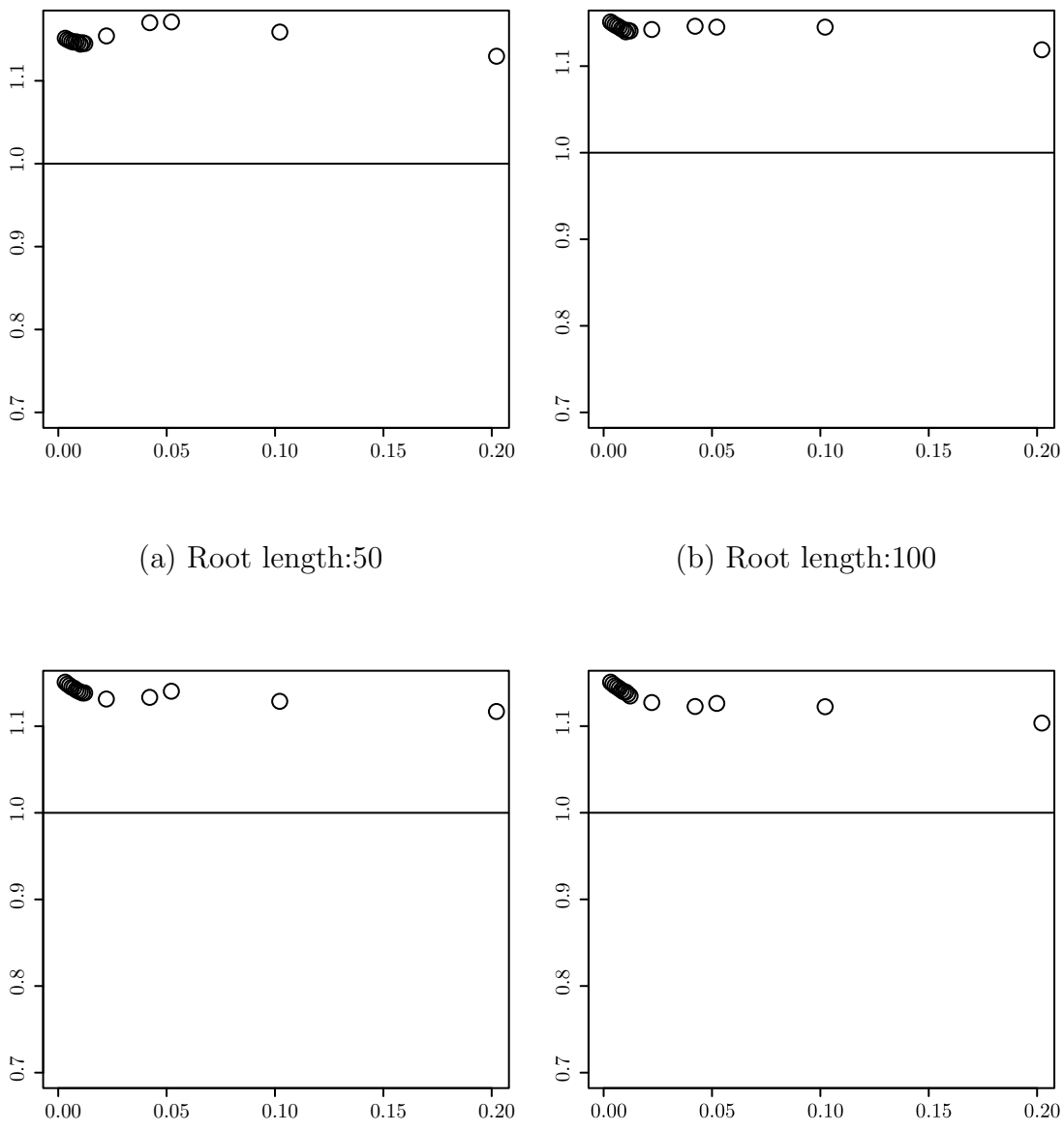
(a) Root Length: 50

(b) Root Length: 100

(c) Root Length: 150

(d) Root Length: 200

Figure C.18: DCJ compared to the GRIMM model under the simplified DCJ mechanism simulation. Points above the line are hypotheses that would prefer GRIMM, points below the line are hypotehses that would prefer DCJ.



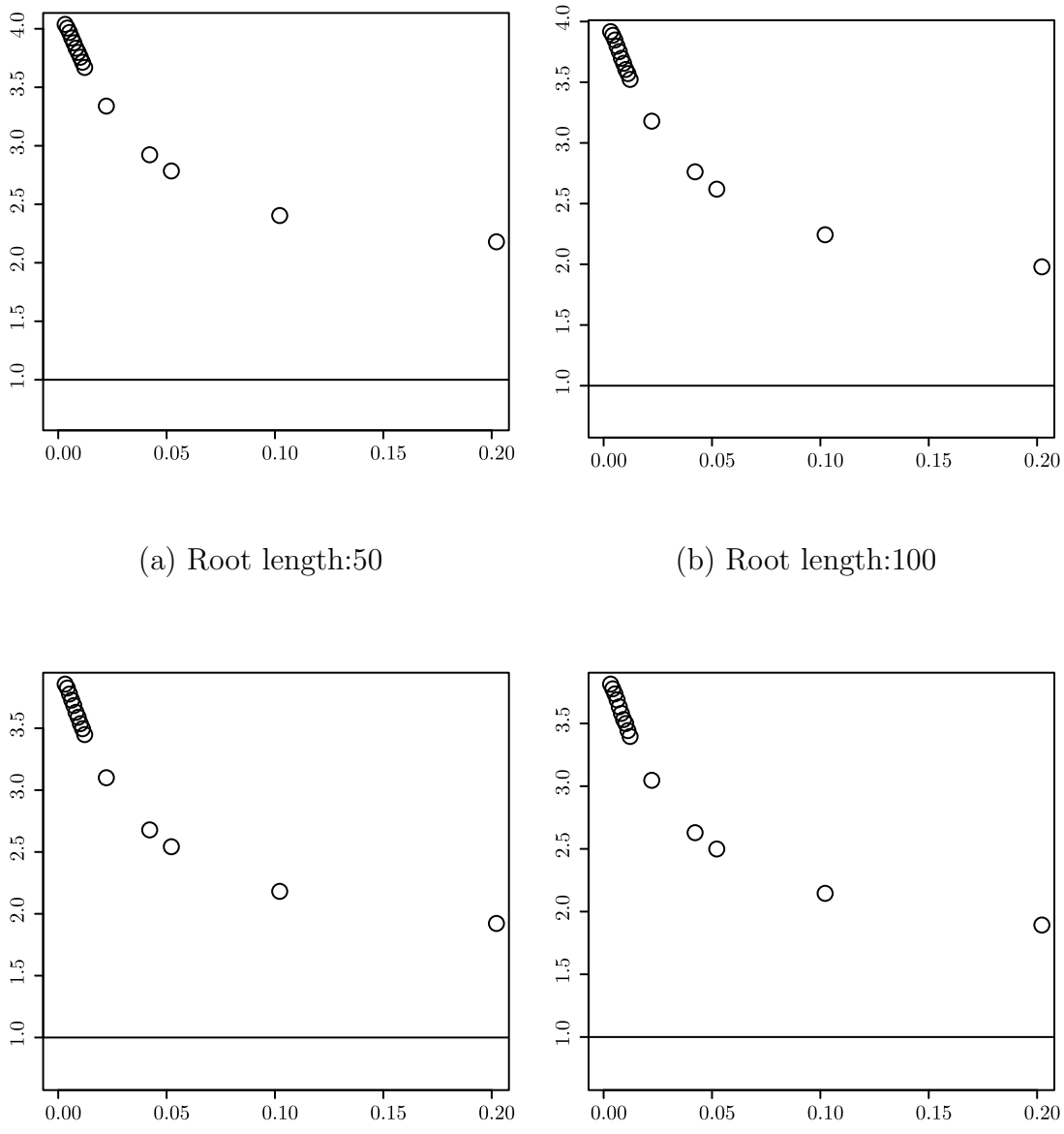
(a) Root length:50

(b) Root length:100

(c) Root length:150

(d) Root length:200

Figure C.19: DCJ compared to the GRIMM model under the GRIMM mechanism simulation. Points above the line are hypotheses that would prefer GRIMM, points below the line are hypotehses that would prefer DCJ.



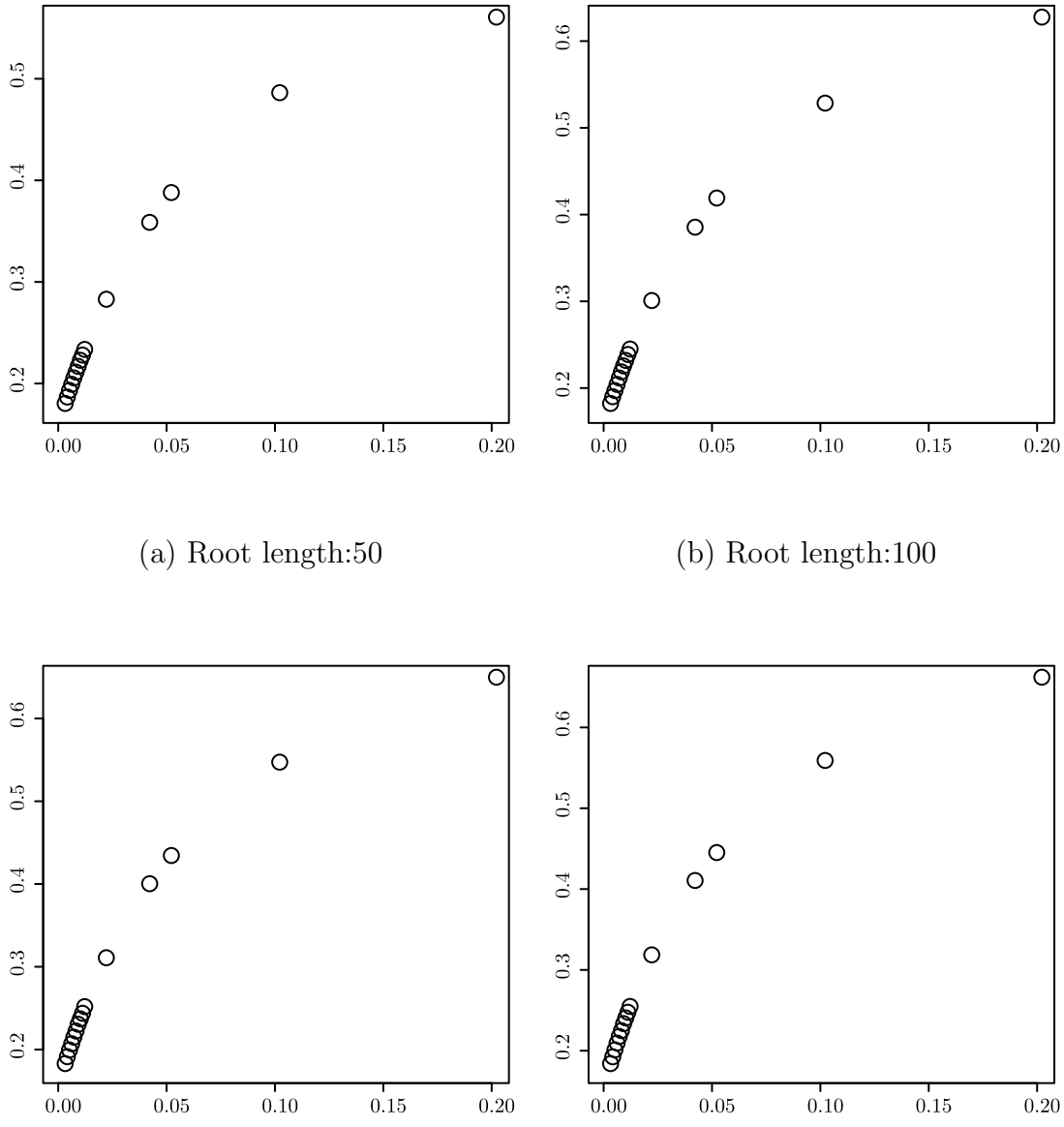
(a) Root length:50

(b) Root length:100

(c) Root length:150

(d) Root length:200

Figure C.20: DCJ compared to the Transposition model under the Transposition mechanism simulation. Points above the line are hypotheses that would prefer Transposition, points below the line are hypotehses that would prefer DCJ.



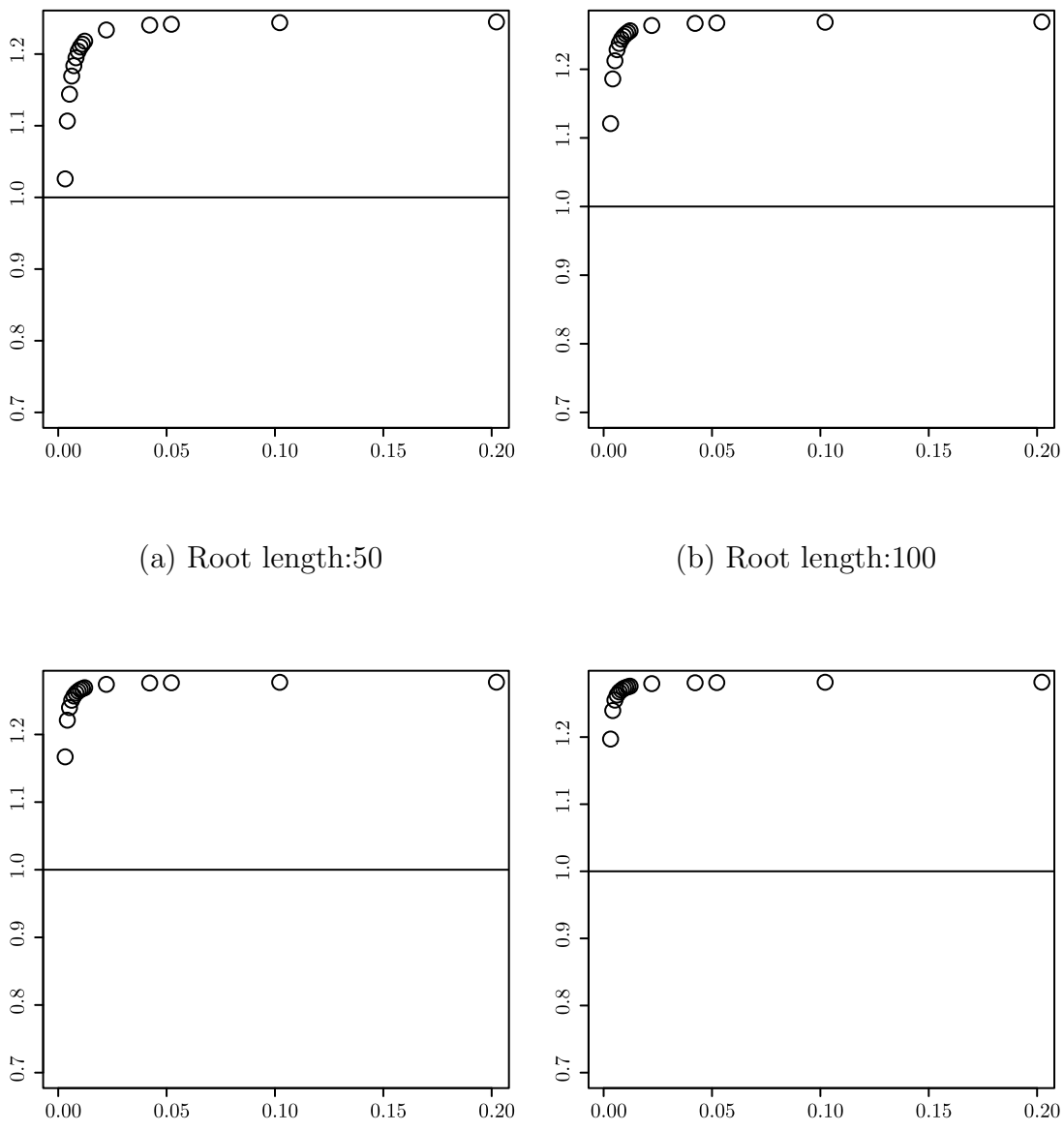
(a) Root length:50

(b) Root length:100

(c) Root length:150

(d) Root length:200

Figure C.21: Inversion compared to the DCJ model under the Inversion mechanism simulation. Points above the line are hypotheses that would prefer DCJ, points below the line are hypotehses that would prefer Inversion.



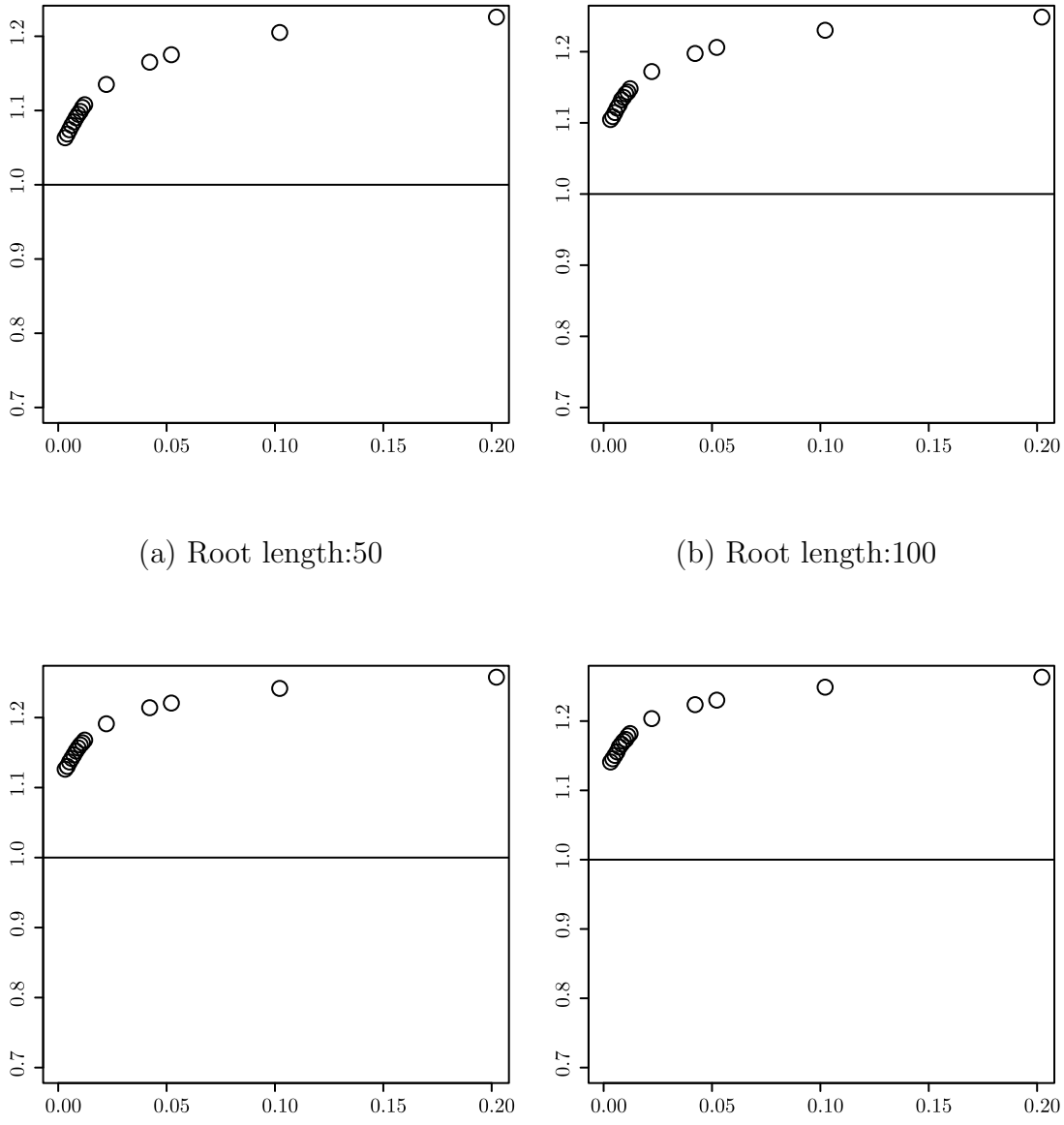
(a) Root length:50

(b) Root length:100

(c) Root length:150

(d) Root length:200

Figure C.22: Inversion compared to the Transposition model under the TDRL mechanism simulation. Points above the line are hypotheses that would prefer Transposition, points below the line are hypotehses that would prefer Inversion.



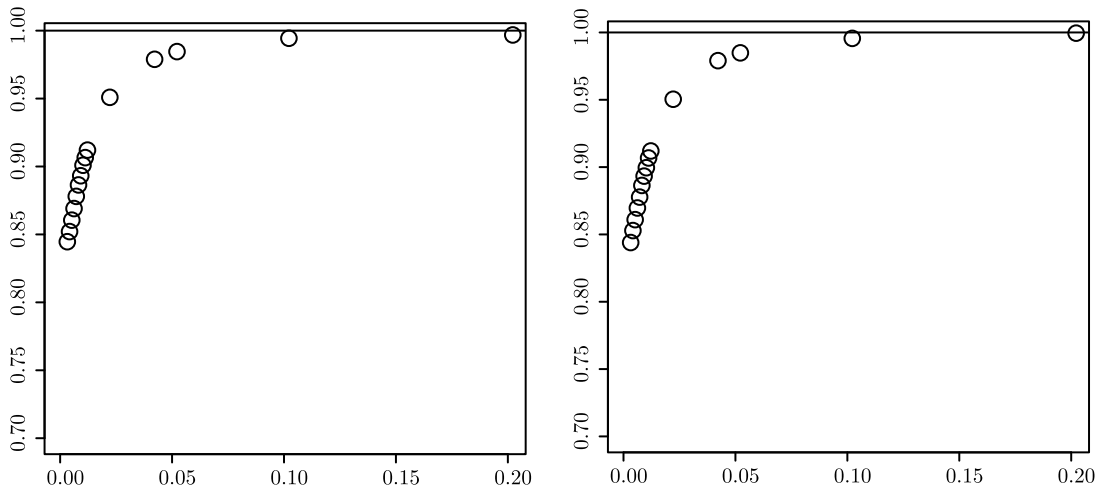
(a) Root length:50

(b) Root length:100

(c) Root length:150

(d) Root length:200

Figure C.23: Inversion compared to the Transposition model under the Transposition mechanism simulation. Points above the line are hypotheses that would prefer Transposition, points below the line are hypotehses that would prefer Inversion.



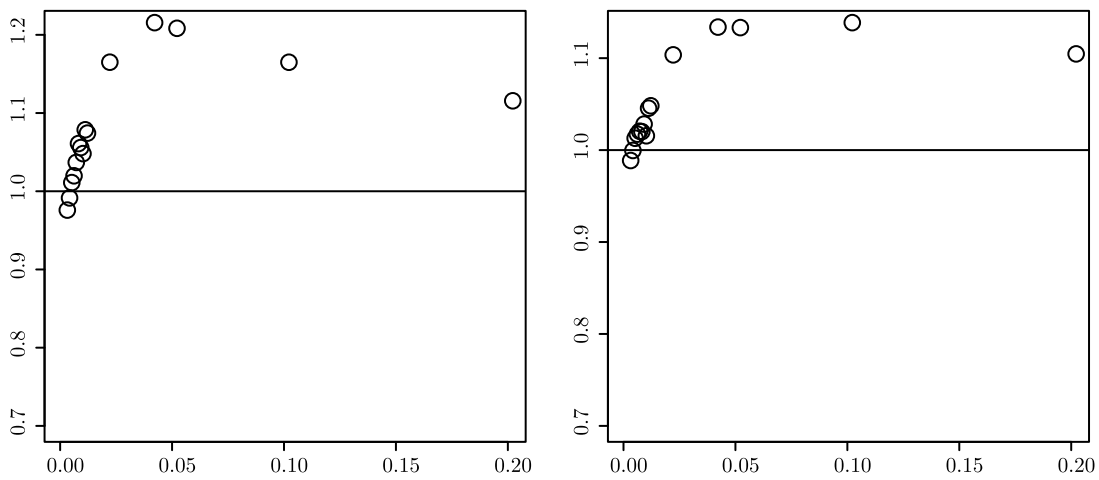
(a) Root length:50

(b) Root length:100

(c) Root length:150

(d) Root length:200

Figure C.24: Parsimony DCJ compared to the parsimony GRIMM model under the simplified DCJ mechanism simulation. Points above the line are hypotheses that would prefer GRIMM, points below the line are hypotehses that would prefer DCJ.



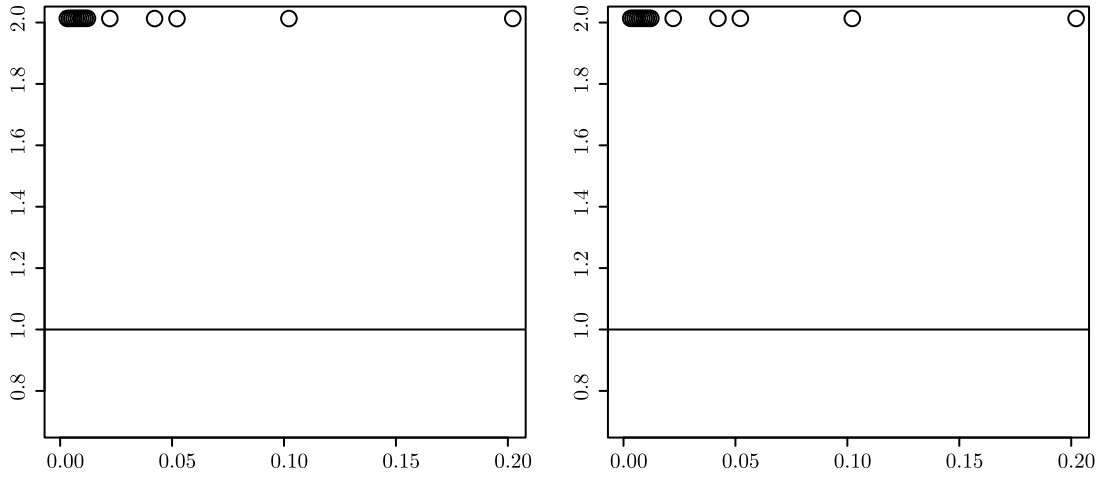
(a) Root length:50

(b) Root length:100

(c) Root length:150

(d) Root length:200

Figure C.25: Parsimony DCJ compared to the parsimony GRIMM model under the GRIMM mechanism simulation. Points above the line are hypotheses that would prefer GRIMM, points below the line are hypotehses that would prefer DCJ.



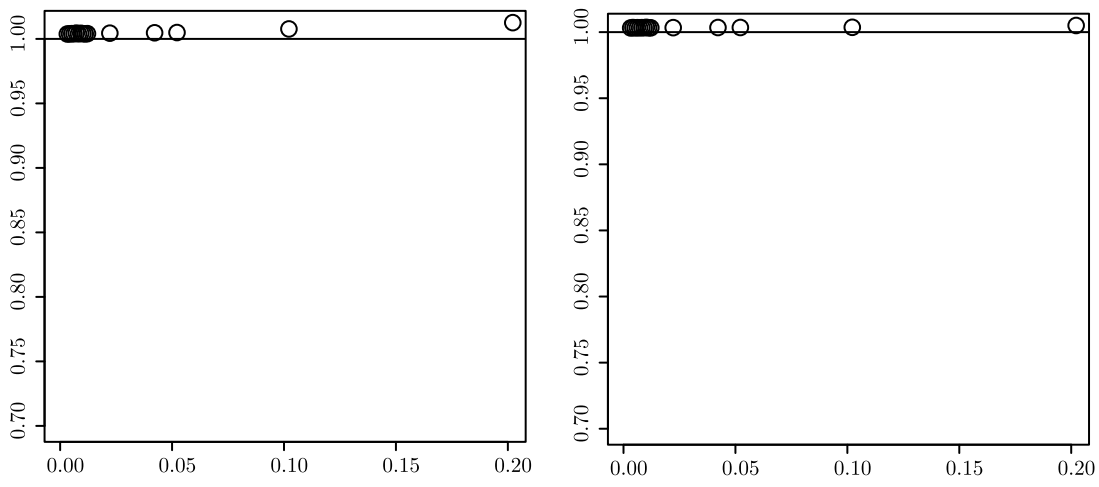
(a) Root length:50

(b) Root length:100

(c) Root length:150

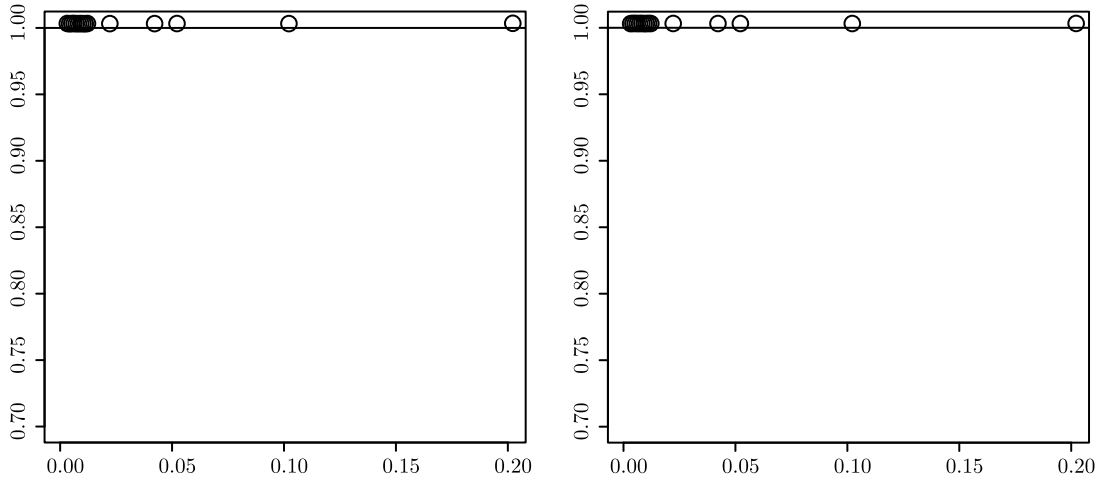
(d) Root length:200

Figure C.26: Parsimony DCJ compared to the parsimony Transposition model under the Transposition mechanism simulation. Points above the line are hypotheses that would prefer Transposition, points below the line are hypotehses that would prefer DCJ.



(a) Root length:50

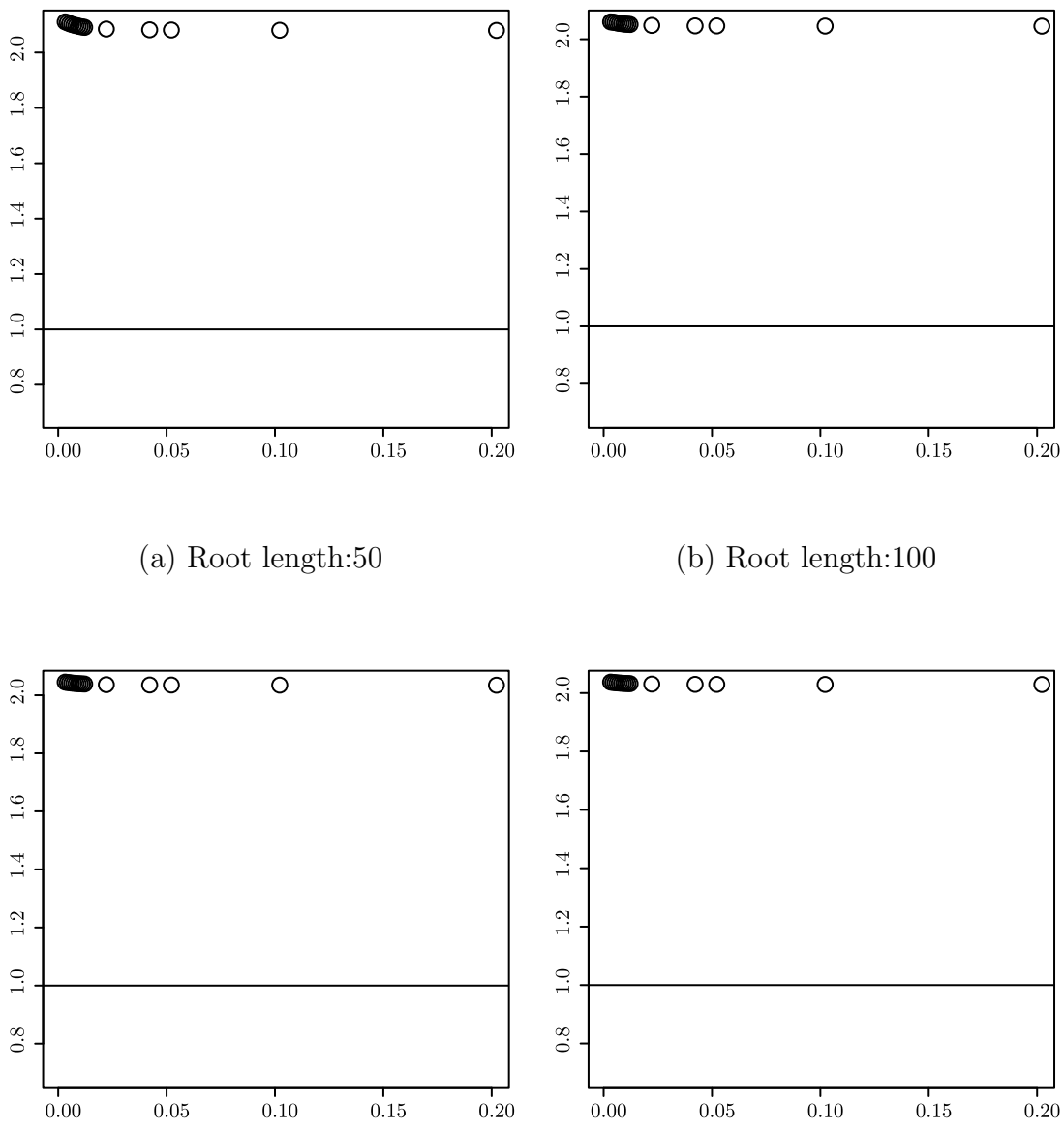
(b) Root length:100



(c) Root length:150

(d) Root length:200

Figure C.27: Parsimony Inversion compared to the parsimony DCJ model under the Inversion mechanism simulation. Points above the line are hypotheses that would prefer DCJ, points below the line are hypotehses that would prefer Inversion.



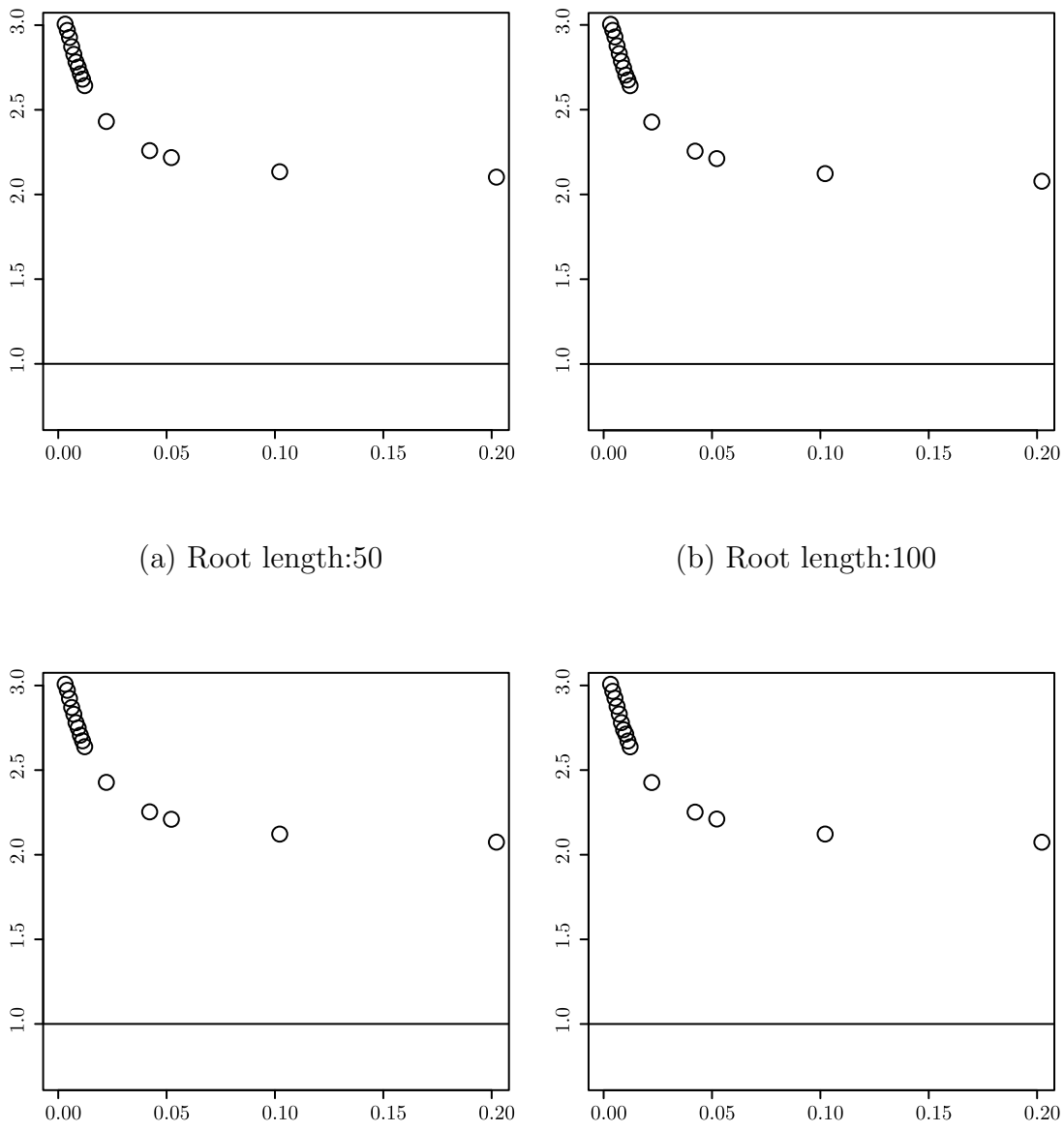
(a) Root length:50

(b) Root length:100

(c) Root length:150

(d) Root length:200

Figure C.28: Parsimony Inversion compared to the parsimony Transposition model under the TDRL mechanism simulation. Points above the line are hypotheses that would prefer Transposition, points below the line are hypotehses that would prefer Inversion.



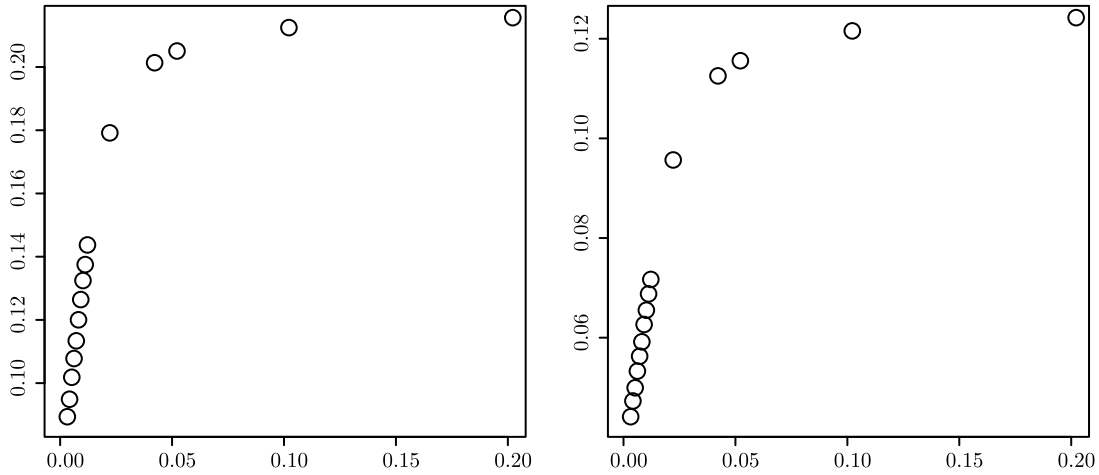
(a) Root length:50

(b) Root length:100

(c) Root length:150

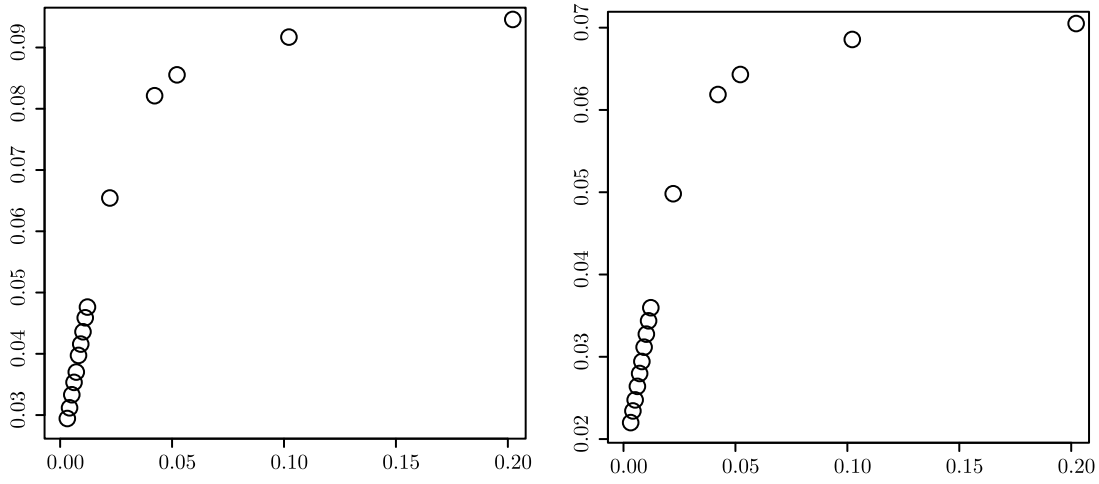
(d) Root length:200

Figure C.29: Parsimony Inversion compared to the Transposition model under the Transposition mechanism simulation. Points above the line are hypotheses that would prefer Transposition, points below the line are hypotehses that would prefer Inversion.



(a) Root length:50

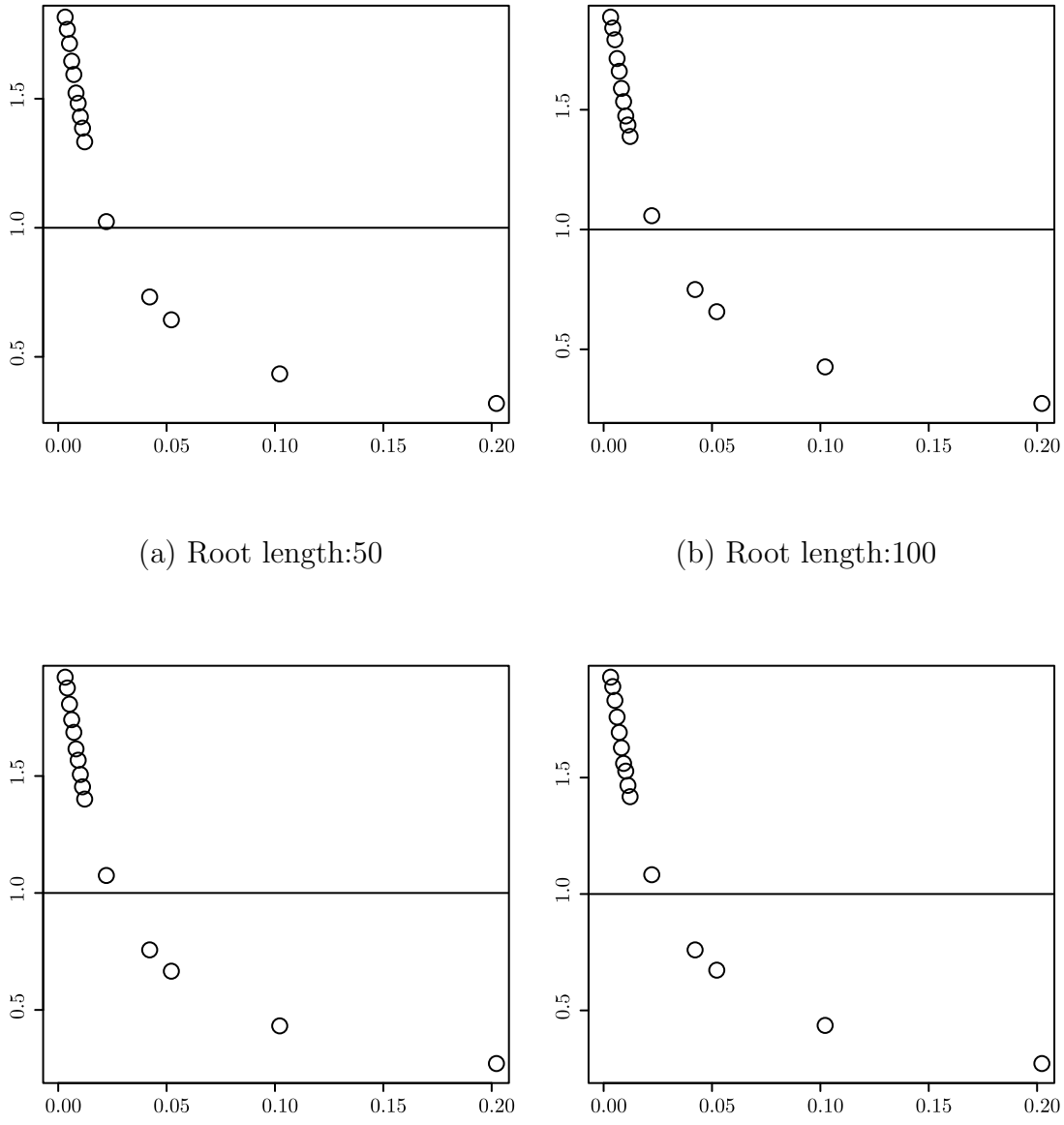
(b) Root length:100



(c) Root length:150

(d) Root length:200

Figure C.30: Parsimony TDRL compared to the parsimony Transposition model under the TDRL mechanism simulation. Points above the line are hypotheses that would prefer Transposition, points below the line are hypotehses that would prefer TDRL.



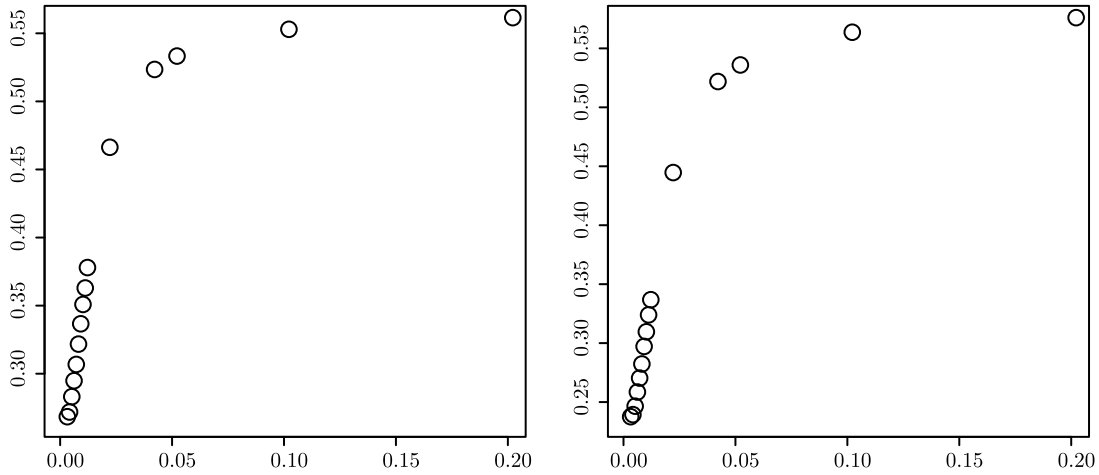
(a) Root length:50

(b) Root length:100

(c) Root length:150

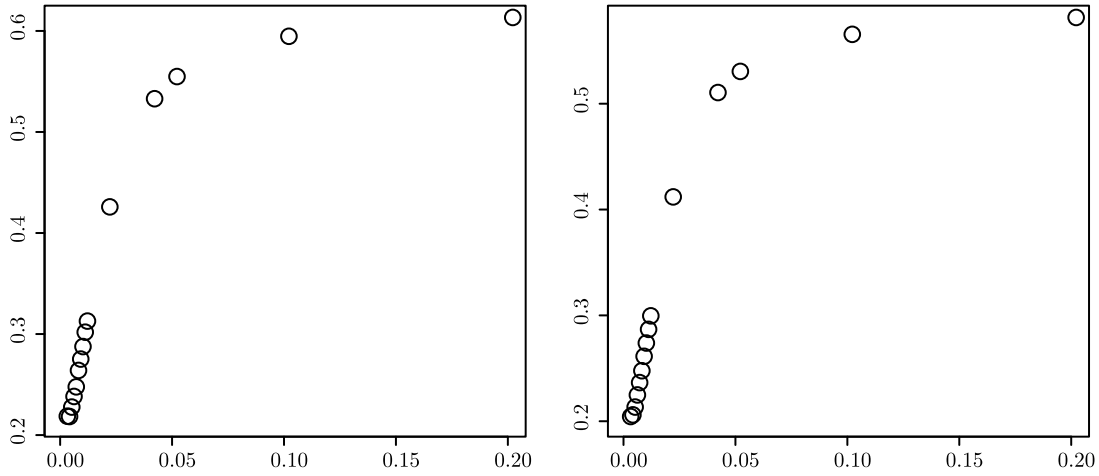
(d) Root length:200

Figure C.31: Parsimony TDRL compared to the parsimony Transposition model under the Transposition mechanism simulation. Points above the line are hypotheses that would prefer Transposition, points below the line are hypotehses that would prefer TDRL.



(a) Root length:50

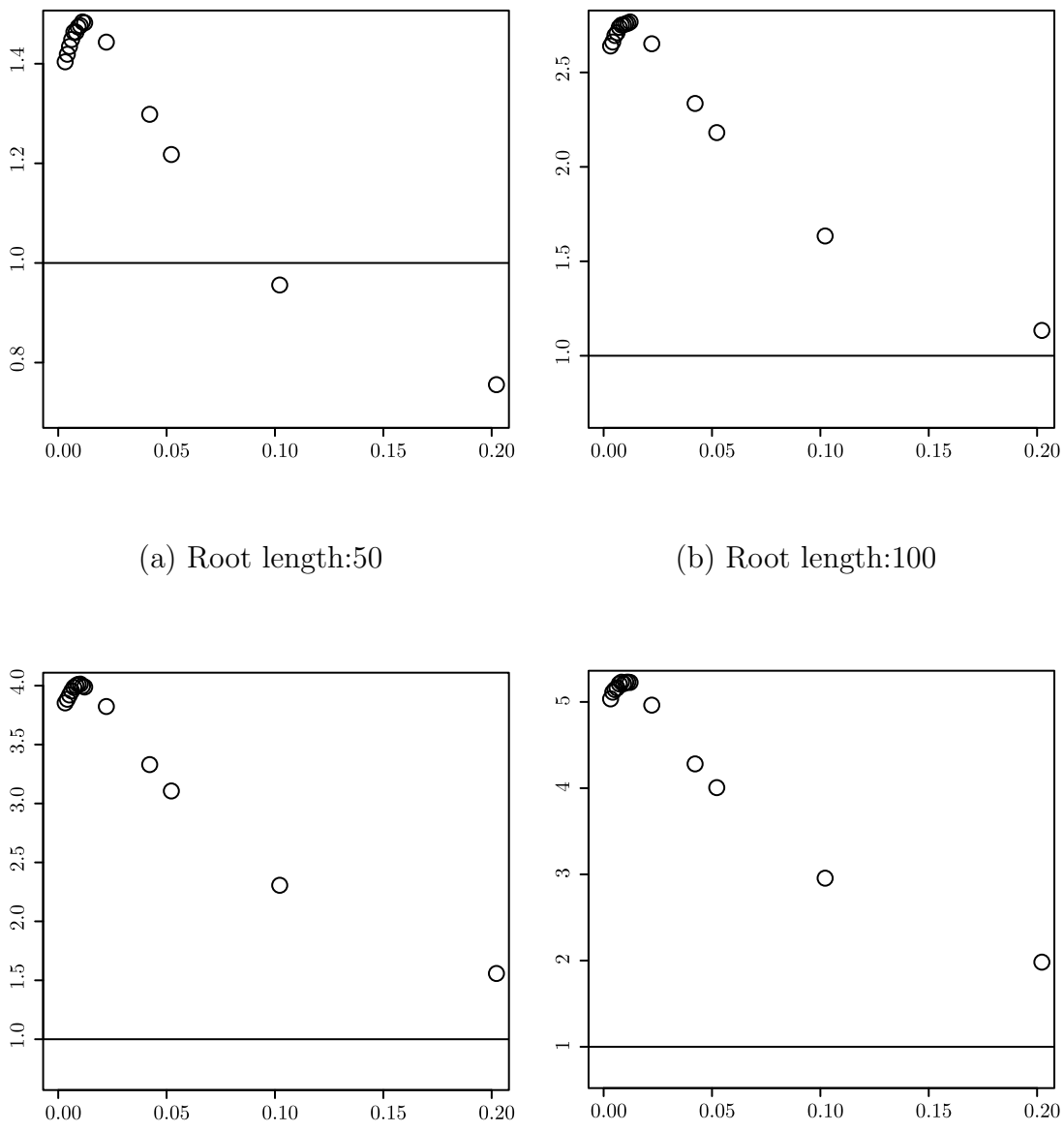
(b) Root length:100



(c) Root length:150

(d) Root length:200

Figure C.32: TDRL compared to the Transposition model under the TDRL mechanism simulation. Points above the line are hypotheses that would prefer Transposition, points below the line are hypotehses that would prefer TDRL.



(a) Root length:50

(b) Root length:100

(c) Root length:150

(d) Root length:200

Figure C.33: TDRL compared to the Transposition model under the Transposition mechanism simulation. Points above the line are hypotheses that would prefer Transposition, points below the line are hypotehses that would prefer TDRL.

Bibliography

[1] H. Akaike. Information theory and an extension of the maximum likelihood principle. In *Proceedings of the 2nd International Symposium on Information Theory (ISIT 1971)*, pages 267–281, Budapest, Hungary, 1973. Akademiai Kiado.

[2] L. Allison, D. Powell, and T. Dix. Compression and approximate matching. *Comp. J.*, 42:1–10, 1999.

[3] L. Allison, L. Stern, T. Edgoose, and T. I. Dix. Sequence complexity for biological sequence analysis. *Computers and Chemistry*, 24:43–55, 2000.

[4] L. Allison and C. N. Yee. Minimum message length encoding and the comparison of macromolecules. *Bulletin of Mathematical Biology*, 52:431–453, 1990.

[5] C. Ane and M. J. Sanderson. Missing the forest for the trees: Phylogenetic compression and its implications for inferring complex evolutionary histories. *Systematic Biology*, 54(1):146–157, 2 2005/2/1.

[6] K. Atteson. The performance of the neighbor-joining method of phylogeny reconstruction. In B. Mirkin, F. R. McMorris, F. S. Roberts, and A. Rzhetsky, editors, *Mathematical hierarchies in biology*, volume 37 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 133–147. American Mathematical Society, Providence, RI, 1997.

[7] H. P. Barendregt. *The Lambda Calculus Its Syntax and Semantics*, volume 103 of *Studies in Logic and The Foundations of Mathematics*. Elsevier, 1984.

[8] D. Barker. LVB: parsimony and simulated annealing in the search for phylogenetic trees. *Bioinformatics*, 20:274–275, 2004.

[9] S. A. Benner and M. A. Cohen. Empirical and structural models for insertions and deletions in the divergent evolution of proteins. *Journal of Molecular Evolution*, 229:1065–1082, 1993.

[10] M. Blanchette, G. Bourque, and D. Sankoff. Breakpoint phylogenies. In *Proceedings of the Genome Informatics Workshop VIII*, pages 25–34. Universal Academy Press, December 1997.

[11] J. L. Boore. *Comparative Genomics*, chapter The duplicationrandom loss model for gene rearrangement exemplified by mitochondrial genomes of deuterostome animals, pages 133–147. Kluwer Academic, 2000.

[12] G. Bourque and P. A. Pevzner. Genome-scale evolution: reconstructing gene orders in the ancestral species. *Genome Res.*, 12:26–36, 2002.

[13] G. Bourque, P. A. Pevzner, and G. Tesler. Reconstructing the genomic architecture of ancestral mammals: Lessons from human, mouse, and rat genomes. *Genome Res.*, 14:507–516, 2004.

[14] K. Bremer. Branch support and tree stability. *Cladistics*, 10:294–304, 1994.

[15] A. Caprara. Sorting by reversals is difficult. In *RECOMB '97: Proceedings of the first annual international conference on Computational molecular biology*, pages 75–83, New York, NY, USA, 1997. ACM.

[16] A. Caprara. The Reversal Median Problem. *INFORMS JOURNAL ON COMPUTING*, 15(1):93–113, 2003.

[17] R. A. Cartwright. DNA assembly with gaps (Dawg): simulating sequence evolution. *Bioinformatics*, 21(Suppl. 3):iii31–iii38, 2005.

[18] R. A. Cartwright. Logarithmic gap costs decrease alignment accuracy. *BMC Bioinformatics*, 7:527–539, 2006.

[19] G. J. Chaitin. On the length of programs for computing finite binary sequences. *Journal of the ACM*, 13:547–569, 1966.

[20] M. S. S. Chang and S. A. Benner. Empirical analysis of protein insertions and deletions determining parameters for the correct placement of gaps in protein sequence alignments. *Journal of Molecular Biology*, 341(2):617–631, 2004.

[21] K. Chaudhuri, K. Chen, R. Mihaescu, and S. Rao. On the tandem duplication-random loss model of genome rearrangement. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 564–570, New York, NY, USA, 2006. ACM.

[22] P. Cheeseman and B. Kanefsky. Evolutionary tree reconstruction. Technical Report 90.27, Research Institute for Advanced Computer Science, NASA Ames Research Center, Nasa Ames Research Center - MS: 230-5, Moffett Field, CA 94035, March 1990.

[23] D. Cieslik. The steiner ratio of several discrete metric spaces. *Discrete Mathematics*, 260:189–196, 2003.

[24] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, United States, 2 edition, 2001.

[25] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 2 edition, 2001.

[26] M. E. Cosner, L. A. Raubeson, and R. K. Jansen. Chloroplast DNA rearrangements in campanulaceae: phylogenetic utility of highly rearranged genomes. *BMC Evol. Biol.*, 4:27, 2004.

[27] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. John Wiley and Sons, 1991.

[28] M. D. Davis, R. Sigal, and E. J. Weyuker. *Computability, Complexity, and Languages. Fundamentals of Theoretical Computer Science*. Morgan Kaufmann, San Francisco, California, second edition, 1993.

[29] J. E. De Laet. *Parsimony Phylogeny and Genomics*, chapter Parsimony and the problem of inapplicables in sequence data, pages 81–116. Oxford University Press, 2004.

[30] M. C. C. de Pinna. Concepts and tests of homology in the cladistic paradigm. *Cladistics*, 7:367–394, 1991.

[31] C. B. Do, M. S. P. Mahabhashyam, M. Brudno, and S. Batzoglou. ProbCons: Probabilistic consistency-based multiple sequence alignment. *Genome Res.*, 15:330–340, 2005.

[32] R. C. Edgar. MUSCLE: a multiple sequence alignment method with reduced time and space complexity. *BMC Bioinformatics*, 5:113, 2004.

[33] O. Elemento, O. Gascuel, and M.-P. Lefranc. Reconstructing the duplication history of tandemly repeated genes. *Mol. Biol. Evol.*, 19:278–288, 2002.

[34] J. Faivovich, C. F. B. Haddad, P. C. A. Garcia, D. R. Frost, J. A. Campbell, and W. C. Wheeler. Systematic review of the frog family hylidae, with special reference to hylinae: phylogenetic analysis and taxonomic revision. *Bulletin of the American Museum of Natural History*, 294:240, June 2005.

[35] J. S. Farris. Methods for computing wagner trees. *Systematic Zoology*, 19(1):86–92, Mar. 1970.

[36] J. S. Farris. *The logical basis of phylogenetic analysis*, pages 7–36. Columbia University Press, 1983.

[37] J. S. Farris, A. G. Kluge, and M. J. Eckhardt. A numerical approach to phylogenetic systematics. *Systematic Zoology*, 19:172–189, 1970.

[38] J. Felsenstein. *Infering Phylogenies*. Sinauer Associates, Sunderland, Massachusetts, 2004.

[39] W. M. Fitch. Toward defining the course of evolution: Minimum change for a specific tree topology. *Systematic Zoology*, 20(4):406–416, 1971.

[40] R. Fleissner, D. Metzler, and A. von Haeseler. Simultaneous statistical multiple alignment and phylogeny reconstruction. *Systematic Biology*, 54(4):548–561, 2005.

[41] L. R. Foulds and R. L. Graham. The Steiner problem in phylogeny is NP-complete. *Adv. Appl. Math.*, 3:43–49, 1982.

[42] F. Frati, C. Simon, J. Sullivan, and D. L. Swofford. Evolution of the mitochondrial cytochrome oxidase ii gene in Collembola. *Journal of Molecular Evolution*, 44:145–158, 1997.

[43] G. Giribet, G. D. Edgecombe, and W. C. Wheeler. Arthropod phylogeny based on eight molecular loci and morphology. *Nature*, 413:157–161, 2001.

[44] D. S. Gladstein. Efficient incremental character optimization. *Cladistics*, 13:21–26, 1997.

[45] P. Goloboff. Nona (no name). http://www.cladistics.com, 1999.

[46] P. A. Goloboff. Character optimization and calculation of tree lenghts. *Cladistics*, 9(4):433–436, Dec. 1993.

[47] P. A. Goloboff. Tree searches under sankoff parsimony. *Cladistics*, 14:229–237, 1998.

[48] P. A. Goloboff. Analyzing large data sets in reasonable times: Solutions for comosite optima. *Cladistics*, 15(4):415–428, 1999.

[49] P. A. Goloboff, J. S. Farris, and K. C. Nixon. Tnt, a free program for phylogenetic analysis. *Cladistics*, 24(5):774–786, 2008.

[50] O. Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162:705 – 708, 1982.

[51] P. D. Grünwald. *Advances in Minimum Description Length: Theory and Applications*, chapter A Tutorial Introduction to the Minimum Description Length Principle, page 80. MIT Press, 2005.

[52] P. D. Grünwald. *The minimum description length principle*. The MIT Press, Cambridge, Massachusetts, 2007.

[53] X. Gu and W.-H. Li. The size distribution of insertions and deletions in human and rodent pseudogenes suggests the logarithmic gap penalty for sequence alignment. *Journal of Molecular Evolution*, 40(4):464–473, 1995.

[54] D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, New York, NY, USA, 1997.

[55] R. W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 26(2):147–160, 1950.

[56] S. Hannenhalli and P. A. Pevzner. Transforming cabbage into turnip (polynomial algorithm for sorting signed permutations by reversals). In *Proceedings of the 27th Annual Symposium on Theory of Computing*, volume (STOC 1995), pages 178–189, Las Vegas, Nevada, 1995.

[57] J. Hein. A new method that simultaneously aligns and reconstructs ancestral sequences for any number of homologous sequences, when the phylogeny is given. *Molecular Biology and Evolution*, 6(6):649–668, Nov. 1989.

[58] J. Hein. Unified approach to alignment and phylogenies. *Methods in Enzymology*, 183, 1990.

[59] M. D. Hendy and D. Penny. Branch and bound algorithms to determine minimal evolutionary trees. *Mathematical Biosciences*, 60:133–142, 1982.

[60] F. K. Hwang, D. S. Richards, and P. Winter. *The Steiner Tree Problem*. North-Holland, Amsterdam, 1992.

[61] D. A. Janies and W. C. Wheeler. Efficiency of parallel direct optimization. *Cladistics*, 17:S71–S82, 2001.

[62] T. H. Jukes and C. R. Cantor. Evolution of protein molecules. In H. N. Munro, editor, *Mammalian Protein Metabolism*, volume 3, pages 21–123. Academic Press, New York, 1969.

[63] K. Katoh, K. Misawa, K. Kuma, and T. Miyata. MAFFT: a novel method for rapid multiple sequence alignment based on fast Fourier transform. *Nucl. Acids Res.*, 30:3059–3066, 2002.

[64] J. Kececioglu and D. Sankoff. Efficient bounds for oriented chromosome inversion distance. In *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching*, volume 807 of *Lecture Notes in Computer Science*, pages 307–325. Springer Verlag, 1994.

[65] M. Kimura. A simple method for estimating evolutionary rates of base substitutions through comparative studies of nucleotide sequences. *J. Mol. Evol.*, 16:111–120, 1980.

[66] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[67] A. N. Kolmogorov. Some theorems on algorithmic entropy and the algorithmic quantity of information. *Uspekhi Mat. Nauk.*, 23(3):201, 1968.

[68] C. Lanave, G. Preparata, C. Saccone, and G. Serio. A new method for calculating evolutionary substitution rates. *J. Mol. Evol.*, 20:86–93, 1984.

[69] G. Lancia and R. Ravi. SALSA: Sequence alignment via steiner ancestors.

[70] G. Lancia and R. Ravi. GESTALT: Genomic steiner alignments. *Lecture Notes in Computer Science*, 1645:101, 1999.

[71] S. Lehtonen. Phylogeny estimation and alignment via POY versus clustal + PAUP*: A response to Ogden and Rosenberg (2007). *Systematic Biology*, 57(4):653–657, 2008.

[72] X. Leroy. The ocaml programming language.

[73] M. Li, J. H. Badger, X. Chen, S. Kwong, P. Kearney, and H. Zhang. An information-based sequence distance and its application to whole mitochondrial genome phylogeny. *Bioinformatics*, 17(2):149–154, 2001.

[74] M. Li and P. Vitányi. *An introduction to Kolmogorov complexity and its applications*. Springer, second edition, 1997.

[75] C. Liébecq, editor. *Biochemical Nomenclature and Related Documents*. Portland Press, London, 2 edition, 1992.

[76] K. Liu, S. Nelesen, S. Raghavan, C. R. Linder, and T. Warnow. Barking up the wrong treelength: the impact of gap penalty on alignment and tree accuracy. *IEEE Transactions on Computational Biology and Bioinformatics*, 2008.

[77] D. R. Maddison, D. L. Swofford, and W. P. Maddison. NEXUS: An extensible file format for systematic information. *Systematic Biology*, 46:590–621, 1997.

[78] B. M. E. Moret, D. A. Bader, and T. Warnow. High-performance algorithm engineering for computational phylogenetics. *The Journal of Supercomputing*, 22:99–111, 2002.

[79] B. Morgenstern. DIALIGN 2: improvement of the segment-to-segment approach to multiple sequence alignment. *Bioinformatics*, 15:211–218, 1999.

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

[81] S. Nelesen, K. Liu, D. Zhao, C. R. Linder, and T. Warnow. The effect of the guide tree on multiple sequence alignments and subsequenct phylogenetic analyses. *Pacific Symposium on Biocomputing*, 13:25–36, 2008.

[82] K. C. Nixon. The parsimony ratchet, a new method for rapid parsimony analysis. *Cladistics*, 15:407–414, 1999.

[83] T. H. Ogden and M. S. Rosenberg. Alignment and topological accuracy of the direct optimization approach via poy and traditional phylogenetics via clustalw + PAUP*. *Systematic Biology*, 56(2):182–193, 2007.

[84] C. Okasaki. *Purely functional data structures*. Cambridge University Press, Cambridge, UK, 1999.

[85] H. H. Otu and K. Sayood. A new sequence distance measure for phylogenetic tree construction. *Bioinformatics*, 19:2122–2130, 2003.

[86] N. Platnick. An empirical comparison of parsimony programs. *Cladistics*, 3:121–144, 1987.

[87] D. R. Powell, L. Allison, and T. I. Dix. Fast, optimal alignment of three sequences using linear gap costs. *Journal of Theoretical Biology*, 207:325–336, 2000.

[88] R. Ravi and J. D. Kececioglu. Approximation algorithms for multiple sequence alignment under a fixed evolutionary tree. *Discret. Appl. Math.*, 88:355–366, 1998.

[89] B. D. Redelings and M. A. Suchard. Joint Bayesian estimation of alignment and phylogeny. *Syst. Biol.*, 54:401–418, 2005.

[90] F. Ren, H. Tanaka, N. Fukuda, and T. Gojobori. Molecular evolutionary phylogenetic trees based on minimum description length principle. In *Proceedings of the 28th Annual Hawaii International Conference on System Sciences (HICSS'95)*, pages 165–173. IEEE, 1995.

[91] J. Ripplinger and J. Sullivan. Does choice in model selection affect maximum likelihood analysis? *Systematic Biology*, 57(1):76–85, 2008.

[92] S. Roch. A short proof that phylogenetic tree reconstruction by maximum likelihood is hard. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 3(1):92, April 2006.

[93] J. S. Rogers. Maximum likelihood estimation of phylogenetic trees is consistent when substitution rates vary according to the invariable sites plus gamma distribution. *Syst. Biol.*, 50:713–722, 2001.

[94] N. Saitou and M. Nei. The neighbor–joining method: A new method for reconstructing phylogenetic trees. *Mol. Biol. Evol.*, 4:406–425, 1987.

[95] D. Sankoff. Minimal mutation trees of sequences. *SIAM Journal on Applied Mathematics*, 28(1):35–42, January 1975.

[96] D. Sankoff and R. J. Cedergren. *Simultaneous Comparison of Three or more Sequences Related by a Tree*, pages 253–263. Addison-Wesley, Reading, MA, 1983.

[97] D. Sankoff, R. J. Cedergren, and G. Lapalme. Frequency of insertion-deletion, transversion, and transition in the evolution of 5S ribosomal RNA. *Journal of Molecular Evolution*, 7:133–149, 1976.

[98] D. Sankoff, G. Leduc, N. Antoine, B. Paquin, B. F. Lang, and R. Cedergren. Gene order comparisons for phylogenetic inference: evolution of the mitochondrial genome. *Proceedings of the National Academy of Sciences of the United States of America*, 89:6575–6579, July 1992.

[99] D. Sankoff and P. Rousseau. Locating the vertices of a steiner tree in an arbitrary space. *Mathematical Programming*, 9:240–246, 1975.

[100] G. Schwarz. Estimating the dimensions of a model. *Ann. Stat.*, 6:461–464, 1978.

[101] B. Schwikowski and M. Vingron. The deferred path heuristic for the generalized tree alignment problem. In *RECOMB '97: Proceedings of the first annual international conference on Computational molecular biology*, pages 257–266, New York, NY, USA, 1997. ACM Press.

[102] B. Schwikowski and M. Vingron. Weighted sequence graphs: boosting iterated dynamic programming using locally suboptimal solutions. *Discrete Appl. Math.*, 127(1):95–117, 2003.

[103] C. Semple and M. Steel. *Phylogenetics*. Oxfort University Press, Great Britain, first edition, 2003.

[104] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423, 1948.

[105] R. J. Solomonoff. A preliminary report on a general theory of inductive inference. Technical report, Zator Company, Cambridge, Massachusetts, November 1960.

[106] D. L. Swofford. *PAUP: Phylogenetic analysis using parsimony, V3.1.1.* Smithsonian Institution, Washington, D.C., 1993.

[107] D. L. Swofford, G. J. Olsen, P. J. Waddell, and D. M. Hillis. Phylogeny reconstruction. In D. M. Hillis, C. Moritz, and B. K. Mable, editors, *Molecular Systematics*, pages 407–514. Sinauer Associates, Sunderland, Massachusetts, 2 edition, 1996.

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

[109] C. Tuffley and M. Steel. Links between maximum likelihood and maximum parsimony under a simple model of site substitution. *Bulletin of Mathematical Biology*, 59(3):581–607, 1997.

[110] E. Ukkonen. Algorithms for approximate string matching. *Inf. Control*, 64(1-3):100–118, 1985.

[111] A. Varón, L. S. Vinh, I. Bomash, and W. C. Wheeler. Poy 4.0.2900. http://research.amnh.org/scicomp/, 2008.

[112] A. Varón, L. S. Vinh, and W. C. Wheeler. POY version 4: phylogenetic analysis using dynamic homologies. *Cladistics*, DOI: 10.1111/j.1096-0031.2009.00282.x 2009.

[113] A. Varón, W. Wheeler, and A. Bar-Noy. An efficient heuristic for the tree alignment problem. submitted, 2009.

[114] A. Varón and W. C. Wheeler. Application note: on extension gap in POY version 3. *Cladistics*, 24(6):1070–1070, 2008.

[115] A. Varón, W. C. Wheeler, and A. Bar-Noy. A heuristic for the tree alignment problem with affine indels. *Transactions on Computational Biology and Bioinformatics*, page *submitted*, 2009.

[116] L. S. Vinh, A. Varón, and W. C. Wheeler. Pairwise alignment with rearrangements. *Genome Informatics*, 17(2):141–151, 2006.

[117] P. M. B. Vitányi and M. Li. Minimum description length induction, bayesianism, and kolmogorov complexity. *IEEE Transactions on Information Theory*, 46(2):446–464, March 2000.

[118] L. Wang and D. Gusfield. Impoved approximation algorithms for tree alignment. *Journal of Algorithms*, 25(2):255–273, Nov. 1997.

[119] L. Wang and T. Jiang. On the complexity of multiple sequence alignment. *Journal of Computational Biology*, 1:337–348, 1994.

[120] L. Wang, T. Jiang, and D. Gusfield. A more efficient approximation scheme for tree alignment. *SIAM Journal on Computing*, 30(1):283–299, 2000.

[121] L. Wang, T. Jiang, and E. L. Lawler. Approximation algorithms for tree alignment with a given phylogeny. *Algorithmica*, 16:302–315, 1996.

[122] M. S. Waterman, T. F. Smith, and W. A. Beyer. Some biological sequence metrics. *Advances in Mathematics*, 20(3):367–387, 1976.

[123] W. C. Wheeler. Sequence alignment, parameter sensitivity, and the phylogenetic analysis of molecular data. *Systematic Biology*, 44(3):321–331, September 1995.

[124] W. C. Wheeler. Optimization alignment: The end of multiple sequence alignment in phylogenetics? *Cladistics*, 12:1–9, 1996.

[125] W. C. Wheeler. Fixed character states and the optimization of molecular sequence data. *Cladistics*, 15:379 – 385, 1999.

[126] W. C. Wheeler. Homology and the optimization of dna sequence data. *Cladistics*, 17:S3–S11, 2001.

[127] W. C. Wheeler. Iterative pass optimization of sequence data. *Cladistics*, 19:254–260, 2003.

[128] W. C. Wheeler. Search-based optimization. *Cladistics*, 19(4):348–355, Aug. 2003.

[129] W. C. Wheeler. Dynamic homology and the likelihood criterion. *Cladistics*, 22:157–170, 2006.

[130] W. C. Wheeler. *Sequence Alignment, edited by M. S. Rosenberg.*, chapter Simulation Approaches to Evaluating Alignment Error and Methods for Comparing Alternate Alignments, pages 179–208. University of California Press, Berkeley, CA, USA, 2009.

[131] W. C. Wheeler, L. Aagesen, C. P. Arango, J. Faivovich, T. Grant, C. D'Haese, D. Janies, W. L. Smith, A. Varón, and G. Giribet. *Dynamic Homology and Phylogenetic Systematics: A Unified Approach using POY.* American Museum of Natural History, 2006.

[132] W. C. Wheeler, D. Gladstein, and J. De Laet. *POY, Phylogeny Reconstruction via Optimization of DNA and other Data version 3.0.11 (May 6 of 2003).* American Museum of Natural History, May 2003.

[133] P. Winter and M. Zachariasen. Euclidean steiner minimum trees: An improved exact algorithm. *Networks*, 30:149–166, 1997.

[134] S. Yancopoulos, O. Attie, and R. Friedberg. Efficient sorting of genomic permutations by translocation, inversion and block interchange. *Bioinformatics*, 21:3340–3346, 2005.

[135] Z. Yang and B. Rannala. Bayesian phylogenetic inference using dna sequences: a markov chain monte carlo method. *Mol. Biol. Evol.*, 14:717–724, 1997.

[136] F. Yue, J. Shi, and J. Tang. Simultaneous phylogeny reconstruction and multiple sequence alignment. *BMC Bioinformatics*, 10(Suppl 1):S11, 2009.

[137] F. Yue and J. Tang. A divide-and-conquer implementation of three sequence alignment and ancestor inference. In *Proc. of the 1st IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 143–150, 2007.

[138] M. Zachariasen. Rectilinear full steiner tree generation. *Networks*, 33:125–143, 1999.

[139] Z. Zhang and M. Gerstein. Patterns of nucleotide substitution, insertion and deletion in the human genome inferred from pseudogenes. *Nucl. Acids Res.*, 31(18):5338–5348, 2003.

[140] J. Zola, D. Tryastram, A. Tchernykh, and C. Brizuela. Parallel multiple sequence alignment with local phylogeny search by simulated annealing. In *IPDPS, 20th International Parallel and Distributed Processing Symposium*. IEEE, 2006.