Evolution of Whole Genomes Through Inversions: Duplicates, Ancestors, and Edit Scenarios

Krister M. Swenson

November 10, 2009

Abstract

Advances in sequencing technology are yielding DNA sequence data at an alarming rate – a rate reminiscent of Moore's law. Biologists' abilities to analyze this data, however, have not kept pace. On the other hand, the discrete and mechanical nature of the cell life-cycle has been tantalizing to computer scientists. Thus in the 1980s, pioneers of the field now called Computational Biology began to uncover a wealth of computer science problems, some confronting modern Biologists and some hidden in the annals of the biological literature. In particular, many interesting twists were introduced to classical string matching, sorting, and graph problems.

One such problem, first posed in 1941 but rediscovered in the early 1980s, is that of sorting by inversions (also called reversals): given two permutations, find the minimum number of inversions required to transform one into the other, where an inversion inverts the order of a subpermutation. Indeed, many genomes have evolved mostly or only through inversions. Thus it becomes possible to trace evolutionary histories by inferring sequences of such inversions that led to today's genomes from a distant common ancestor. But unlike the classic edit distance problem where string editing was relatively simple, editing permutation in this way has proved to be more complex.

In this dissertation, we extend the theory so as to make these edit distances more broadly applicable and faster to compute, and work towards more powerful tools that can accurately infer evolutionary histories. In particular, we present work that for the first time considers genomic distances between *any* pair of genomes, with no limitation on the number of occurrences of a gene. Next we show that there are conditions under which an ancestral genome (or one close to the true ancestor) can be reliably reconstructed. Finally we present new methodology that computes a minimum-length sequence of inversions to transform one permutation into another in, on average, $O(n \log n)$ steps, whereas the best worst-case algorithm to compute such a sequence uses $O(n\sqrt{n \log n})$ steps.

keywords: inversions, reversals, sorting, pairwise distance, duplications, median, genomes, evolution, phylogeny, orthology, positional homology

Résumé

Les avancées dans la technologie de séquencage sont en train de fournir une quantité de données génétiques à un rythme alarmant - un rythme rappelant la loi de Moore. Cependant, la capacité des biologistes à analyser toutes ces données ne suit pas le même rythme. la nature mécanique et discrète du cycle cellulaire a toujours attiré les informaticiens. Dans les années 1980, les pionniers du domaine maintenant appelé Biologie Computationelle ont commencé à découvrir une quantité de problèmes informatiques dont certains qui posaient déjá problème aux biologistes, d'autres étant cachés dans les annales de la littérature. En particulier, il fut introduit de nouvelles variations sur des problèmes classiques de string matching, tri et graphes.

Un de ces problèmes, posé d'abord en 1941 mais redécouvert dans le début des années 80, est celui de l'assortiment par *inversion* (aussi appelé *reversal*): étant donné deux permutations, trouver le nombre minimum d'inversion nécessaires pour transformer l'une dans l'autre, une inversion inversant l'ordre d'une sous-permutation. En effet, beaucoup de génomes ont évolué surtout ou uniquement par inversion. Il devient donc possible de retracer l'histoire évolutive en inférant de telles séquences d'inversions qui ont amené aux génomes actuels à partir d'un ancêtre commun distant. Mais à la différence du problème classique de distance edit ou le string editing était relativement simple, l'édition de permutation de cette façon s'est révélée être plus complexe que cela.

Dans cette dissertation, nous alimentons la théorie pour élargir le champ d'application des edit distances et accélérer leur temps de calcul. Nous concevons aussi des outils plus puissants permettant d'inférer les histoires évolutives de manière plus précise. En particulier, nous présentons un travail qui pour la première fois prend en considérations les distances génomiques entre *n'importe* quelle paire de génomes, sans aucune limitation sur le nombre d'occurrences d'un gène. Ensuite nous montrons qu'il y a des conditions sous lesquelles un génome ancestral (ou un génome proche de véritable ancêtre) peut être reconstruit fiablement. Enfin nous présentons une nouvelle méthodologie qui calcule une séquence d'inversion minimum pour transformer une permutation en une autre en moyenne en $O(n \log n)$ étapes, alors que l'algorithme du worst-case se calcule en $O(n\sqrt{n \log n})$ étapes.

mots clés: inversions, reversals, sorting, pairwise distance, duplications, median, genomes, evolution, phylogeny, orthology, positional homology

Acknowledgements

Bernard Moret is the best advisor a research student could hope for. He fosters the unrestrictive environment necessary to release the (very personal) enthusiasm for discovery, while possessing the technical ability and breadth of knowledge to move a student through the difficult moments. He also has a knack for attracting interesting people, without whom I would not have started, let alone completed, any of this work. Mark Marron was the first to show me how research is done, and was a good companion through my years in New Mexico. I was fortunate to have met and befriended students with whom I worked closely: Yu Lin, Mark Marron, Nick Pattengale, and Vaibhav Rajan. Among my office mates who provided a daily laugh as well as a serious conversation from time to time (you know who you are: Alexandros, Alisa, Eric, Jijun, Kremena, Monique, Oana, Sukanya, Yann, and Zak) Xiuwei Zhang and Guojing Cong made a particular effort to help me on my way; thank you. I will miss you.

My family has prepared me well for this environment. My parents raised me in a way that allowed for natural curiosity because they had no expectations. They would treat me exactly as they do even if I had quit my education at high school and were still working at the movie theatre. By living a full and happy life in Washington D.C. my sister continues to show me how to make the best of whatever situation I'm in and my brother taught me an important lesson about being a teacher: don't lecture!

Finally I would like to acknowledge the people in New Hampshire who skillfully sparked my interest for Computer Science: Thomas Cormen and Pilar De La Torre.

Contents

Chapter 1

Introduction and Background

1.1 Introduction

Inside each organism, the machine of life is constantly turning. Messenger RNAs are copied directly from the DNA strand and interact with each other in potentially complicated ways before finally producing proteins. On top of this, proteins interact to create a physical scaffold on which other proteins carry out the necessities of sustaining the machine. In an attempt to understand this cycle of life biologists rely on clues gained through observation, but unfortunately these processes are too small and too vital to an organism's survival to be directly observed. On the other hand, many recent technological advancements in chemistry and engineering have enabled new assays to probe these molecular workings. No new technology has so dramatically fueled the increase in data collection as DNA sequencing. But the influx of new data has posed more new questions than answers. Indeed, the process of life is encoded into the DNA strand in a way so convoluted that most have only attempted to describe its structure in a statistical manner.

Yet there exists work — actually older than the discovery of the double helix [90] — observing specific events that modify the genetic code. In particular, Sturtevant [74] noticed that a substrand of the fruit fly DNA can be inverted; in some strains of fruit fly the sequence of genes on the chromosome appears in reverse order. Further, he showed that these *inversions* were linked to the phenotype of those individuals that possessed it: male flies with a particular inversion had few or no male offspring [75]. So as early as 1936, evolutionary histories between species of fruit fly were being inferred based on inversion histories [76].

By 1941 genomes were being modeled by permutations so as to study properties of their evolution; all permutations of up to 5 elements were being tabulated, by hand, with minimum inversion scenarios being calculated between them [77]. It was not until 1982 that the fundamental problem was posed: given two permutations, find a shortest scenario of inversions to transform one into the other, where an inversion inverts the order of a substring of the permutation [91]. So, for two such permutations (5 3 2 4 1) and (1 2 3 4 5), a shortest scenario would have three inversions:

$$
\begin{array}{cccccc}
(& 5 & 3 & \overline{2 \quad 4} & 1 &) \\
(& 5 & \overline{3 \quad 4} & 2 & 1 &) \\
(& \overline{5 \quad 4 \quad 3 \quad 2 \quad 1} & & & &) \\
(& 1 & 2 & 3 & 4 & 5 \quad)
\end{array}
$$

Ten years later the more biologically relevant *signed* version of the problem was stated: given two *signed* permutations, find a shortest scenario of *signed* inversions to transform one into the other, where a signed inversion inverts the order and the signs of the elements in a substring of the permutation [65]. In this

setting the pair of permutations from above would require a scenario of five signed inversions:

$$
\begin{array}{cccccccc}
(& 5 & 3 & 2 & 4 & 1 &) \\
(& 5 & 3 & 2 & -1 & -4 &) \\
(& 5 & -3 & 2 & -1 & -4 &) \\
(& 1 & -2 & 3 & -5 & -4 &) \\
(& 1 & 2 & 3 & -5 & -4 &) \\
(& 1 & 2 & 3 & 4 & 5 &)
\end{array}
$$

Over the following three years a flurry of work on the subject culminated in an impressive theory describing the exact inversion distance between two genomes, based upon structure apparent in the so-called *breakpoint* graph, and provided a polynomial time algorithm to compute this distance and to extract a shortest scenario of inversions [42] (The unsigned version of the problem was later shown to be APX-Hard [14]).

On its own, an accurate evolutionary *distance* has proved useful in phylogenetic tree reconstruction [58]. For this reason, much effort has been spent trying to improve the original, somewhat difficult, algorithm that was presented in 1995. In 2001 this effort culminated in a linear time algorithm to compute the minimum inversion distance between two genomes that efficiently analyzes the structure of the breakpoint graph [8]. However, it remains an open question as to how fast a minimum inversion *scenario* can be calculated; the fastest algorithm takes (in the worst case) $O(n\sqrt{n \log n})$ time [87].

Unfortunately current methods have yet to make a large impact due to methodological as well as modeling and data limitations. Indeed, the number of whole genomes sequenced ten years ago was extremely low, the cost of producing one being prohibitive. More importantly, until the turn of the century, no consideration had been payed to the fact that many genomes cannot be represented by a permutation.

On the other hand, recent research is considering more complex evolutionary models that compare genomes with unequal gene content; the ice was broken by Sankoff's group (and, in particular, El-Mabrouk) [34, 66, 32] near the turn of the century. Along with formulating many problems for the first time, they showed that deletions of contiguous segments can be handled within the framework of the breakpoint graph. For a deeper introduction into this area (including examples) see Section 3.2.2.

Sequencing is also cheaper now. A full bacterial genome can be sequenced for only three thousand dollars. Consequently, a few thousand prokaryotic and tens of eukaryotic sequences now exist. With talk of the one-thousand-dollar genome on the horizon the number of fully sequenced organisms is sure to increase dramatically.

Of late, more difficult questions have been addressed by the community. How many minimum sorting scenarios between two permutations exist [11, 19] and which is the most likely? What are some properties of these scenarios [61]? Can we sample all minimum scenarios to extract useful information like average inversion length or breakpoint reuse statistics [1, 51, 55]? These are questions that still lack a satisfactory answer [56]. Another such question is that of ancestor reconstruction; given a phylogenetic tree and known genomes at the leaves of the tree, what are the most likely genomes for the internal nodes? In 2002, Bourque and Pevzner [17] associated this question with a better understood problem called the *median* problem: given three permutations, find a fourth that minimizes the pairwise distance between it and the other three.

In this dissertation, we make progress in three of the aforementioned areas. We improve on the work of Bourque and Pevzner and offer a new perspective for attacking the ancestral reconstruction problem in Chapter 4. In Chapter 3 we describe foundational work for computing the inversion distance in the presence of duplicate genes. In particular we offer the only known (constant factor) approximation algorithm for finding the evolutionary distance between any genome and the identity permutation along with an algorithm that, in practice, accurately predicts the distance between any two genomes. We show that this algorithm, combined with good tree reconstruction techniques can reconstruct phylogenies

better than the other known methods (Section 3.4.7). Finally, in Chapter 5 we show that we can find a sorting scenario between most pairs of permutations in $O(n \log n)$ time. Supporting much of the work presented here is the simplifying assumption that certain structures in genomes are rarely encountered. We justify the use of this assumption in Chapter 2.

All of the work presented has been accomplished by close collaboration with Bernard Moret. Most of the work presented has included collaboration with some subset of past and current lab mates including Guojing Cong, Joel Earnest-DeYoung, Yu Lin, Mark Marron, Nick Pattengale, Vaibhav Rajan, and Jijun Tang. We will only present work which we feel we were instrumental in seeing through, indicating collaborations at the beginning of each section.

1.2 Background

In our study, we represent a chromosome of n genes by a signed permutation on the elements $\{1, 2, \ldots, n\}$, that is, a permutation with positive or negative signs associated to each element. The signs reflect the fact that genes can be coded in reverse order on the strand (it would be read on the other of the two strands that compose a DNA molecule). An *inversion* $\rho(i, j)$ is a permutation that, when applied to π, reverses the order and the sign of the segment of π that begins at the ith gene and ends at the jth one. Thus

$$\rho(i, j) = \big(1, \ldots, i-1, -\mathbf{j}, -(\mathbf{j}-\mathbf{1}), \ldots, -(\mathbf{i}+\mathbf{1}), -\mathbf{i}, j+1, \ldots, n\big),$$

and we denote $\pi \cdot \rho$ as the composition of ρ with π. For example, with π = (2 4 1 −3) and $\rho(2, 3)$=(1 −3 −2 4) we get $\pi \cdot \rho(2, 3)$ = (2 −1 −4 −3).

An n gene chromosome could be linear or circular, but note that most models of evolution (ours being that of inversions) are unaffected by a change in representation from linear to circular or vice-versa. Each linear permutation corresponds to $n + 1$ circular permutations (of length $n + 1$), which are equivalent in terms of the scenario of inversions used to sort them: if we join the ends of a linear permutation to form a circular permutation then an inversion $\rho(i, j)$ can be thought of as inverting the subpermutation from i to j, or as inverting the rest of the permutation while fixing the subpermutation from i to j in place. Thus, throughout this presentation we will consider permutations to be either linear or circular as we see fit. Without loss of generality we consider that every linear permutation has an implicit far left element 0 and implicit far right element $n + 1$.

Say I represents the identity permutation (1 2 3 4...n). Then we can define the following genome comparison problems:

Problem 1.2.1. *The* sorting by inversions problem *for signed permutations π_1 and π_2 asks for a minimum length scenario of inversions $\rho_1, \rho_2, \ldots, \rho_d$ that transforms π_1 into π_2. In other words $\pi_1 \cdot \rho_1 \cdot \ldots \cdot \rho_d = \pi_2$.*

We call $\rho_1, \rho_2, \ldots, \rho_d$ an *edit scenario.*

Problem 1.2.2. *The* inversion distance problem *for signed permutations π_1 and π_2 asks for $d(\pi_1, \pi_2)$, the minimum number of inversions needed to transform π_1 into π_2. The number $d(\pi_1, \pi_2)$ is called the* inversion distance.

Note that an edit scenario takes π_1 to π_2 if and only if that scenario takes $\pi_1 \cdot \pi_2^{-1}$ to I. This motivates the following equivalent, but simpler formulations:

Problem 1.2.3 (SBI problem)**.** *The* sorting by inversions problem *for signed permutation π asks for a minimum length scenario of inversions $\rho_1, \rho_2, \ldots, \rho_d$ that transforms π_1 into the identity permutation. In other words $\pi_1 \cdot \rho_1 \cdot \ldots \cdot \rho_d = I$.*

Problem 1.2.4 (ID problem)**.** *The* inversion distance problem *for signed permutation π asks for $d(\pi)$, the minimum number of inversions needed to transform π into I. The number $d(\pi_1)$ is called the* inversion distance.

For example, with π_1 = (3 2 −1 4) and π_2 = (−1 3 −4 2) we have minimum edit scenario $\rho(2, 3) \cdot \rho(3, 4) \cdot \rho(1, 2) \cdot \rho(2, 2)$, which is also a minimum edit scenario for $\pi_1 \cdot \pi_2^{-1}$ = (2 4 1 −3).

1.2.1 The Breakpoint Graph

The breakthrough in the SBI problem came when Kececioglu and Sankoff [47] and Bafna and Pevzner [10] independently derived bounds based on a graph theoretic framework. Although the frameworks were different and neither were the one that would eventually be used in the final theory, both groups

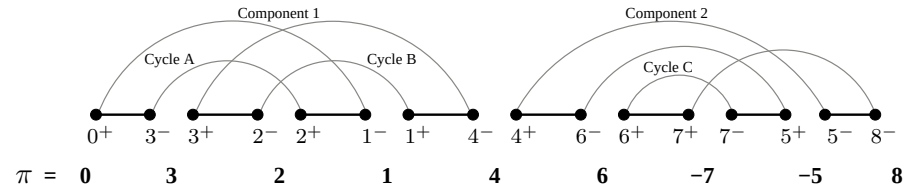


Figure 1.1: A permutation and its breakpoint graph. Desire edges are shown in gray, reality edges in black.

noticed a close correlation between the number of cycles in their graphs and the inversion distance. Kececioglu and Sankoff [48] again gave improved algorithms and bounds for sorting signed permutation while, only 2 years after the original Kececioglu paper, Hannenhalli and Pevzner solidified the theory in the landmark paper [42], presenting an exact formula for calculating the distance based on certain structures found in the *breakpoint graph*. The following exposition describes this correspondence.

Each permutation element will be represented by two vertices; one for each "side" of the element. Edges are included that represent adjacent elements in the permutations. Figure 1.1 shows the breakpoint graph for π = (3 2 1 4 6 −7 −5). The fully sorted identity permutation I has adjacencies between consecutive integers. Thus, our desired configuration is represented by the gray edges in the graph. The black edges, on the other hand, represent the reality of the adjacencies in the current permutation.

To compute $d(\pi)$ we must look further into the structure of the breakpoint graph. Denote the two vertices representing a permutation element π_i in the breakpoint graph by π_i^- and π_i^+ ($\pi^\pm$ can denote either). Embed the breakpoint graph on a line as follows: place all $2n$ vertices on the line so that:

1. π_i^+ and π_i^- are adjacent,
2. π_i^- is left of π_i^+ if and only if π_i is positive, and
3. $\pi_i^\pm$ is adjacent to $\pi_{i+1}^\pm$ if and only if π_i and π_{i+1} are contiguous in π.

Also add a vertex 0^+ as the leftmost vertex and $(n+1)^-$ as the rightmost vertex. For two vertices $v_1 = \pi_i^\pm$ and $v_2 = \pi_j^\pm$ ($i \neq j$) that are adjacent on the line, add the edge (v_1, v_2)—a reality edge; also add edges (π_i^+, π_{i+1}^-) for all i along with $(0^+, \pi_1^-)$ and $(\pi_n^+, (n+1)^-)$—the desire edges.

The breakpoint graph is just as described in [42], but its embedding clarifies the notion of orientation of edges. Note that since the degree of every vertex is exactly 2, the graph decomposes naturally into cycles. Say inversion $\rho(i,j)$ *acts upon* a reality edge if it is either the i^{th} or $j+1^{st}$ reality edge from the left. Say an inversion *acts upon* a desire edge if the edge is incident to the rightmost vertex of the i^{th} reality edge or leftmost vertex of the $j+1^{st}$ reality edge. The vertices that connect the acted upon reality and desire edges are those that are *affected* by the inversion. In our example, the inversion over substring "6 -7 -5" (also known as $\rho(5,7)$) acts upon reality edges $(4^+, 6^-)$ and $(5^-, 8^-)$. It acts upon desire edges $(6^-, 5^+)$ and $(5^-, 4^+)$ while it affects vertices 6^- and 5^-.

Two reality edges on the same cycle are *convergent* if a traversal of their cycle visits each edge in the same direction in the linear embedding; otherwise they are *divergent*. The action of an inversion $\rho(i,j)$ on π is to swap the connectivity of reality edge $(\pi_{i-1}^\pm, \pi_i^\pm)$ and reality edge $(\pi_j^\pm, \pi_{j+1}^\pm)$. Thus, any inversion that acts on a pair of divergent reality edges splits the cycle to which the edges belong, so is called a *cycle-splitting* inversion. Conversely, no inversion that acts on a pair of convergent reality edges can split their common cycle. (An inversion that acts upon a pair of reality edges in two different cycles simply merges the two cycles.) Notice that at most one cycle can be created by this action on the graph. Thus we get the inequality

$$d(\pi) \geq (n+1) - c(\pi), \tag{1.1}$$

where $c(\pi)$ is the number of cycles in the breakpoint graph.

This lower bound cannot always be realized. Consider the prefix P = (3 2 1) of the permutation from Figure 1.1 for example. An enumeration of all scenarios of 2 inversions shows that P can be sorted in no fewer than 3 inversions, whereas inequality 1.1 gives $d(P) \geq 4 - 2$. Hannenhalli and Pevzner [42] found the structures that indicate the gap between the lower and bound and the inversion distance, and coined the terms "hurdles" and "fortresses" to refer to them. We will visit a full exposition of the inversion distance in Chapter 2 and show strong evidence as to why we can safely use inequality 1.1 as an equality, an assumption that is particularly useful when dealing with duplicate elements (as in Chapter 3).

Chapter 2

Ignoring Hurdles and Fortresses

(This is joint work with Yu Lin and Vaibhav Rajan)

The result of Hannenhalli and Pevzner [42] gives us

$$d(\pi) = (n+1) - c(\pi) + h(\pi) + \{1, 0\}, \tag{2.1}$$

where n is the length of π, $c(\pi)$ and $h(\pi)$ are the number of cycles and so-called *hurdles* in the breakpoint graph of π, and $\{1, 0\}$ is a correction factor that accounts for the possible occurrence of a rarely occurring phenomenon, the *fortress*. We will see in this section that the machinery behind equation 2.1 is considerably more complicated than inequality 1.1. Further, for problems that require us to build permutations in order to minimize distance, like the duplicate assignment problem in Section 3, it is advantageous to optimize only one factor (cycles) rather than three (cycles, hurdles, and fortresses). Fortunately, Caprara [24] showed that hurdles occur in only $\theta(n^{-2})$ proportion of the random permutations of length n, effectively justifying the use of inequality 1.1 as an equality. In this section we prove the same result using markedly simpler means, a technique that also extends to the first analysis of the rarity of fortresses.

In this section we consider the permutation to be circular. That is, the last element π_n is adjacent to the first element π_1.

2.1 Hurdles and Fortresses as Framed Common Intervals

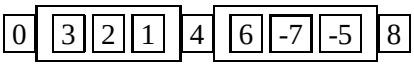
A pair of elements in a circular permutation (π_i, π_{i+1}) is called a *breakpoint* whenever we have $\pi_{i+1} - \pi_i \neq 1$ (for $1 \leq i \leq n-1$) or $\pi_1 - \pi_n \neq 1$. Since there is one-to-one mapping between π and the corresponding breakpoint graph, we identify the second with the first and so write that π contains cycles, hurdles, or fortresses if its breakpoint graph does. Let Σ_n denote the set of signed permutations over n elements and Σ_n^0 to denote the set of those permutations with $n+1$ breakpoints. Bergeron *et al* [11] proved the following result about $|\Sigma_n^0|$.

Lemma 2.1.1 ([11])**.** *For all $n > 1$, $\frac{1}{2}|\Sigma_n| < |\Sigma_n^0| < |\Sigma_n|$.*

Definition 2.1.2 (FCI)**.** *A* framed common interval *(FCI) of a permutation (made circular by considering the first and last elements as being adjacent) is a substring of the permutation, $as_1s_2 \ldots s_kb$ or -$bs_1s_2 \ldots s_k$-a such that*

- *for each i, $1 \leq i \leq k$, $|a| < |s_i| < |b|$, and*
- *for each l, $|a| < l < |b|$, there exists a j with $|s_j| = l$, and*
- *it does not contain a proper substrings satisfying the previous two properties.*

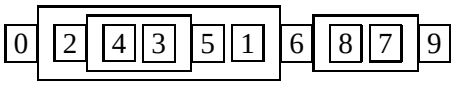
So the substring $s_1 s_2 \ldots s_k$ is a (possibly empty) signed permutation of the integers that are greater than a and less than b; a and b form the *frame*. The framed interval is said to be common, in that it also exists as an interval $\big(a(a+1)(a+2)\ldots b\big)$ in the identity permutation. Recall the permutation from Figure 1.1. The FCIs in the permutation can be illustrated as follows.



In this example there are exactly two FCIs, one framed by 0 and 4 and the other framed by 4 and 8.

The *span* of an FCI is the number of elements between a and b, plus two, or $b - a + 1$. FCI B is *nested* inside FCI A if and only if the left and right frame elements of A occur, respectively, before and after the frame elements of B. A *component* is comprised of all elements inside a framed interval that are not inside any nested subinterval, plus the frame elements. A *bad component* is a component whose elements all have the same sign, otherwise the component is *good*. For example, the permutation from Figure 1.1 has two components, the leftmost of which is bad.

Bad component A *separates* bad components B and C if and only if every substring containing an element of B and an element of C also has an element of A in it. We say that A *protects* B if A separates B from all other bad components. A *superhurdle* is a bad component that protects another bad component. The component framed by 0 and 6 is a superhurdle in the permutation



because it protects the nested component with frame elements 2 and 5. A *hurdle* is a bad component that is not a superhurdle. In the above permutation the components framed by 2 and 5, and 6 and 9 are hurdles while in the permutation from Figure 1.1, only the leftmost component is a hurdle. A *fortress* is a permutation that has an odd number (larger than 1) of hurdles, all of which are superhurdles. The permutation of Figure 2.1 is one of the shortest possible fortresses.

We will use the following useful facts about FCIs; all but fact 3 follow immediately from the definitions.

1. A bad component indicates the existence of a hurdle.
2. To every hurdle can be assigned a unique bad component.
3. Two FCIs can only overlap at their endpoints and at most both the endpoints of an FCI can overlap with other FCIs [13].
4. An interval shorter than 4 elements cannot be bad.

2.2 The Rarity of Hurdles and Fortresses

In this section, we provide asymptotic characterizations of the probability that a hurdle or fortress is found in a signed permutation selected uniformly at random. Each proof has two parts, an upper bound and a lower bound; for readability, we phrase each part as a lemma and develop it independently.

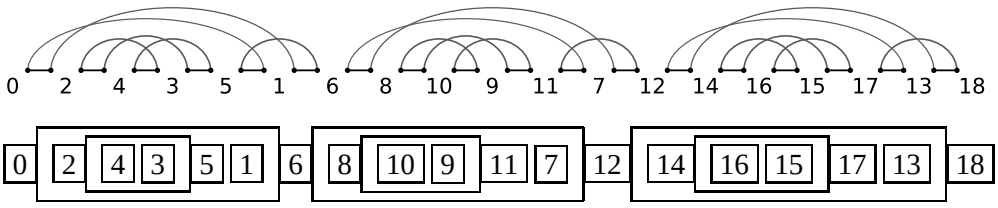


Figure 2.1: A fortress and its breakpoint graph.

2.2.1 Hurdles

We begin with hurdles; the characterization for these structures was already known, but the original proof of Caprara [24] is long and complex. The proof is based on the observation that more than one component is unlikely to be found in a random permutation because the structure of an FCI is so particular. Thus, most of the following proof evaluates the probability of seeing more than one component.

Theorem 2.2.1. *The probability that a random signed permutation on n elements contains a hurdle is $\Theta(n^{-2})$.*

Lemma 2.2.2 (Upper bound for shorter than $n-1$)**.** *The probability that a random signed permutation on n elements contains a hurdle spanning no more than $n-2$ elements is $O(n^{-2})$.*

Proof. Fact 4 tells us that we need only consider intervals of at least four elements. Call $F_{\leq n-2}$ the indicator random variable corresponding to the event that an FCI spanning no more than $n-2$ and no less than four elements exists. Call $F(i)_{\leq n-2}$ the indicator random variable corresponding to event that such an FCI exists with a left endpoint at π_i. We thus have $F_{\leq n-2} = 1$ if and only if there exists an i, $1 \leq i \leq n$, with $F(i)_{\leq n-2} = 1$. Note that $F(i)_{\leq n-2} = 1$ implies either $\pi_i = a$ or $\pi_i = {-}b$ for some FCI. Thus we can write

$$Pr\big(F(i)_{\leq n-2} = 1\big) \leq \sum_{l=4}^{n-2} \frac{1}{2(n-1)} \binom{n-2}{l-2}^{-1} \tag{2.2}$$

since $\frac{1}{2(n-1)}$ is the probability the right endpoint matches the left endpoint (π_l is -a or b if π_i is -b or a respectively) of an interval of span l and $\binom{n-2}{l-2}^{-1}$ is the probability that the appropriate elements are inside the frame. We can bound the probability from (2.2) as

$$\begin{aligned} Pr\big(F(i)_{\leq n-2} = 1\big) &\leq \frac{1}{2(n-1)} \sum_{l=2}^{n-4} \binom{n-2}{l}^{-1} \\ &\leq \frac{1}{n-1} \sum_{l=2}^{\lceil n/2 \rceil - 1} \binom{n-2}{l}^{-1} \\ &\leq \frac{1}{n-1} \Bigg(\sum_{l=2}^{\sqrt{n}} \Big(\frac{l}{n-2}\Big)^{l} + \sum_{l=\sqrt{n}+1}^{\lceil n/2 \rceil - 1} \binom{n-2}{l}^{-1} \Bigg) \end{aligned} \tag{2.3}$$

where the second term is no greater than

$$\sum_{l=\sqrt{n}+1}^{\lceil n/2 \rceil - 1} \binom{n-2}{l}^{-1} \leq \sum_{l=\sqrt{n}+1}^{\lceil n/2 \rceil - 1} \Big(\frac{1}{2}\Big)^{\sqrt{n}+1} \in O(n^{-2}) \tag{2.4}$$

and the first term can be simplified

$$\begin{aligned} \sum_{l=2}^{\sqrt{n}} \Big(\frac{l}{n-2}\Big)^{l} &= \sum_{l=2}^{4} \Big(\frac{l}{n-2}\Big)^{l} + \sum_{l=5}^{\sqrt{n}} \Big(\frac{l}{n-2}\Big)^{l} \\ &\leq \sum_{l=2}^{4} \Big(\frac{l}{n-2}\Big)^{l} + \sum_{l=5}^{\sqrt{n}} \Big(\frac{n}{n-2}\frac{\sqrt{n}}{n}\Big)^{5} \\ &\in O\Big(3 \times \frac{16}{(n-2)^2} + \sqrt{n}\, n^{-5/2}\Big) = O(n^{-2}). \end{aligned} \tag{2.5}$$

To compute $Pr(F_{\leq n-2})$ we use the union bound on $Pr(\bigcup_{i=1}^{n} F(i)_{\leq n-2})$. This removes the factor of $\frac{1}{n-1}$ from (2.3) yielding just the sum of (2.5) and (2.4) which is $O(n^{-2})$. The probability of observing a hurdle in some subsequence of a permutation can be no greater than the probability of observing an FCI (by fact 2). Thus we know the probability of observing a hurdle that spans no more than $n-2$ elements is $O(n^{-2})$. □

We now proceed to bound the probability of a hurdle that spans $n-1$ or n elements. Call intervals with such spans *n-intervals*. For a bad component spanning n elements with $a = i$, there is only a single $b = (i-1)$ that must be a's left neighbor (in the circular order), and for a hurdle spanning $n-1$ elements with $a = i$, there are only two configurations ("+(i-2) +(i-1) +i" and its counterpart "+(i-2) −(i-1) +i") that will create a framed interval. Thus the probability that we see an n-interval with a particular $a = i$ is $O(1/n)$ and the expected number of n-intervals in a permutation is $O(1)$.

We now use the fact that a bad component is comprised of elements with all the same sign. Thus the probability that an n-interval uses all the elements in its span (i.e., there exist no nested subintervals) is $O(2^{-n})$. Call a bad component that does not use all of the elements in its span (i.e., there must exist nested subintervals) a *fragmented* interval.

Lemma 2.2.3 (Upper bound for longer than $n-2$)**.** *The probability that a fragmented n-interval is a hurdle is $O(n^{-2})$.*

Proof. We divide the analysis into three cases where the fragment-causing subinterval is of span

1. $n-1$,
2. 4 through $n-2$, and
3. less than 4.

The existence of a subinterval of span $n-1$ precludes the possibility of the frame elements from the larger n-interval being in the same component, so there cannot be a hurdle using this frame. We have already established that $Pr(F_{\leq n-2})$ is $O(n^{-2})$. Thus we turn to the third case. If an interval is bad, then the frame elements of any fragmenting subinterval must have the same sign as the frame elements of the larger one. If we view each such subinterval and each element not included in such an interval as single characters, we know that there must be at least $n/3$ signed characters. Since the signs of the characters are independent, the probability that all characters have the same sign is $1/2^{O(n)}$ and is thus negligible. □

Thus the probability of a bad n-interval is $O(n^{-2})$. Using fact 4 we conclude that the probability of existence of a hurdle in a random signed permutation on n elements is $O(n^{-2})$.

Lemma 2.2.4 (Lower bound)**.** *The probability that a signed permutation on n elements has a hurdle with a span of four elements is $\Omega(n^{-2})$.*

Proof. Call h_k the hurdle with span four that starts with element $4k+1$. So the subsequence that corresponds to h_k must be $+(4k+1)+(4k+3)+(4k+2)+(4k+4)$ or $-(4k+4)-(4k+2)-(4k+3)-(4k+1)$. We can count the number of permutations with h_0, for instance. The four elements of h_0 are contiguous in $4!(n-3)!2^n$ permutations of length n. In $c = 2/(4!2^4)$ of those cases, the contiguous elements form a hurdle, so the total proportion of permutations with h_0 is

$$c\frac{4!(n-3)!2^n}{n!2^n} \in \Omega\Big(\frac{1}{n^3}\Big).$$

Similarly, the proportion of permutations that have both h_0 and h_1 is

$$F_2 = c^2 \frac{(4!)^2(n-6)!2^n}{n!2^n} \in O\Big(\frac{1}{n^6}\Big)$$

and, therefore, the proportion of permutations that have at least one of h_0 or h_1 is

$$2 \times c\frac{4!(n-3)!2^n}{n!2^n} - F_2. \tag{2.6}$$

We generalize (2.6) to count the proportion of permutations with at least one of the hurdles h_0,h_1,...,$h_{\lfloor n/4 \rfloor}$; this proportion is at least

$$\left\lfloor \frac{n}{4} \right\rfloor \times c\frac{4!(n-3)!2^n}{n!2^n} - \binom{\lfloor n/4 \rfloor}{2} F_2 \tag{2.7}$$

which is $\Omega(n^{-2})$ since the second term is $O(n^{-4})$. □

2.2.2 Fortresses

Now we turn to the much rarer fortresses. We start by using the fact that the smallest fortress that could exist requires the existence of an FCI spanning 19 elements (see figure 2.1), a very unlikely event.

Theorem 2.2.5. *The probability that a random signed permutation on n elements includes a fortress is $\Theta(n^{-15})$.*

Lemma 2.2.6 (Upper bound)**.** *The probability that a random signed permutation on n elements includes a fortress is $O(n^{-15})$.*

Proof. We bound the probability that at least three superhurdles occur in a random permutation by bounding the probability that three non-overlapping bad components of length seven exist. We divide the analysis into three cases depending on the number l of elements spanned by a bad component.

1. For one of the three FCIs we have $n - 14 \leq l \leq n - 11$.
2. For one of the three FCIs we have $17 \leq l \leq n - 15$.
3. For all FCIs we have $7 \leq l < 17$.

As we did in Lemma 2.2.2 (equation 2.2), we can bound the probability that we get an FCI of length l starting at a particular position by

$$Pr\big(F_l = 1\big) \leq \frac{1}{2(n-1)} \binom{n-2}{l-2}^{-1}. \tag{2.8}$$

In the first case the probability that the FCI is a superhurdle is $O(n^{-11} \cdot 2^{-n})$ if the FCI is not fragmented and $O(n^{-15})$ if it is (using the same technique as for the proof of Lemma 2.2.3). In the second case the probability is at most

$$n \sum_{l=17}^{n-15} F_l = n \sum_{k=15}^{n-17} \frac{1}{2(n-1)} \binom{n-2}{k}^{-1}$$

which, by the same reasoning used for equation 2.3 to derive $O(n^{-2})$, is $O(n^{-15})$. Thus the first two cases both give us an upper bound of $O(n^{-15})$.

Fact 3 tells us that any pair of FCIs can overlap only on their endpoints. Thus, if we first consider the probability of finding a smallest FCI, we know that no other FCI will have an endpoint inside it. So the

probability of having a second FCI, conditioned on having a smaller first one, is dependent only on the size of the first. The same reasoning extends to the probability of having a third conditioned on having two smaller FCIs. Since each of the three FCIs spans less than seventeen elements, the probability of each FCI appearing is at most $n \sum_{l=7}^{17} F_k = O(n^{-5})$, and the probability of there being at least three of them is $O(n^{-15})$. □

We now turn to the lower bound. Consider the probability of the existence, among random permutations, of a permutation with exactly three superhurdles spanning seven elements each. A lower bound on this probability is a lower bound on the probability of existence of a fortress in a random permutation.

Lemma 2.2.7 (Lower bound)**.** *The probability that a random signed permutation on n elements includes a fortress is $\Omega(n^{-15})$.*

Proof. Denote by $F_{3,7}(n)$ the number of permutations on n elements with exactly 3 superhurdles spanning 7 elements each. To create such a permutation, choose a permutation of length $n-18$ (with zero adjacencies and without hurdles), select three elements, and extend each of these three elements to a superhurdle, renaming the elements of the permutation as needed. That is, replace element $+i$ by the framed interval of length 7 $f = +(i)+(i+2)+(i+4)+(i+3)+(i+5)+(i+1)+(i+6)$ and rename all the elements with magnitude j to have magnitude $j+6$ (for those with $|j| > |i|$). After extending the three selected elements, we get a permutation on n elements where there are exactly 3 superhurdles each spanning 7 elements.

From Lemma 2.1.1 and the results about the rarity of hurdles from the previous section, we have

$$F_{3,7}(n) > \frac{(n-18)!2^{n-18}}{2}\Big(1 - O(n^{-2})\Big)\binom{n-18}{3}$$

where $\frac{(n-18)!2^{n-18}}{2}(1 - O(n^{-2}))$ is a lower bound for the number of permutations of length $n-18$ (with zero adjacencies and without hurdles) and $\binom{n-18}{3}$ is the number of ways to choose the elements for extension. Therefore we have

$$\begin{aligned} \frac{F_{3,7}(n)}{n!2^n} &> \frac{(n-18)!2^{n-18}}{2}\Big(1 - O(n^{-2})\Big)\binom{n-18}{3}\frac{1}{n!2^n} \\ &\in \Omega(n^{-15}) \end{aligned} \tag{2.9}$$

□

Chapter 3

Unequal Gene Content

The biggest challenge towards applying current methods to real sequences lies in modelling sequences as permutations. Most sequences, in fact, do not have single copies of each gene. Two sequences may not have the same set of genes either. Thus, the use of permutations in a model for genome rearrangements can be limited. While it has been sufficient in some cases [42, 43, 29, 28] to simply ignore genes that occur more than once by considering only the genes that are common to two sequences, much information can be lost in the process [82]. Indeed there exists sequences that have close to half of their genome duplicated. For one such case, methods we present here have led to accurate phylogeny reconstruction on real-world data [30, 15]. While related problems — those that ask for a parsimonious ancestor given a single genome — have admitted nice solutions [33, 34, 3], most of the problems discussed in this section have yet to be satisfactorily solved.

In this chapter we refer to a genome as a *gene sequence* (or *sequence*), as it may not be a permutation. In our case a *sequence* is a string over the alphabet $\mathbb{Z}$ (i.e. a sequence is any element of $\mathbb{Z}^*$). We first present known formulations of evolutionary models and corresponding problems that deal with comparing gene sequences, then the (recent) history of the problems, and finally present our results.

3.1 Insertions and Deletions

Consider the sequences A = (7 −1 −3 8 4 −6 5) and B = (1 2 3 4 5). Notice that the elements 6, 7, and 8 only occur in A while the element 2 only occurs in B. To account for the unequal content between A and B we must permit insertions and deletions of elements in our model of evolution. Denote a deletion of elements in the subsequence from i to j as $del(i, j)$ and the insertion of the string α before the element at position i as $ins(\alpha, i)$. One scenario of inversions, insertions, and deletions for the above example uses 4 inversions, 2 deletions, and 1 insertion:

$$\begin{array}{lcll}
(7{-}1{-}3\ \ 8\ \ 4{-}6\ \ 5) & \cdot & \rho(5,6) & = \\
(7{-}1{-}3\ \ 8\ \ 6{-}4\ \ 5) & \cdot & \rho(6,6)\cdot\rho(2,2) & = \\
(7\ \ 1{-}3\ \ 8\ \ 6\ \ 4\ \ 5) & \cdot & del(4,5)\cdot del(1,1) & = \\
(\ \ 1{-}3\ \ \ \ \ \ 4\ \ 5) & \cdot & \rho(2,2) & = \\
(\ \ 1\ \ 3\ \ \ \ \ \ 4\ \ 5) & \cdot & ins(\text{“2”},2) & = \\
\ \ \ \ (1\ \ 2\ \ 3\ \ 4\ \ 5) & & & = B
\end{array}$$

Yet, another scenario takes only 2 inversions, 2 deletions, and 1 insertion:

$$\begin{array}{lcll}
(7{-}1{-}3\ \ 8\ \ 4{-}6\ \ 5) & \cdot & \rho(A,B)\cdot\rho(3,4) & = \\
(1{-}7{-}8\ \ 3\ \ 4{-}6\ \ 5) & \cdot & del(2,3)\cdot del(6,6) & = \\
(1\ \ \ \ \ \ 3\ \ 4\ \ \ \ 5) & \cdot & ins(\text{“2”},2) & = \\
\ \ \ \ (1\ \ 2\ \ 3\ \ 4\ \ 5) & & & = B
\end{array}$$

Notice that we have deleted contiguous elements of some intermediate permutation in each of the edit scenarios, and that insertions never introduce elements that already exist in the permutation. For that matter, we could simply delete all of A in one move and insert all of B in the second move. To avoid scenarios like this we impose a parsimony criterion: we can either have insertions of a particular number or deletions of a particular number, but not both.

El-Mabrouk [32] showed that a minimum edit scenario of inversions with deletions of contiguous segments can be computed in polynomial time. However, it remains unknown as to whether minimum edit scenarios of inversions with inversions, deletions, and insertions is in P. We combine the result of El-Mabrouk with new insight in Section 3.3.

3.2 Duplicate Elements

Some genomes may have duplicated genes; we represent these genomes by sequences that contain more than one occurrence of a number. We call the set of all elements x from a sequence the *gene family* (or *family*) x, and the size of the family in sequence S is $occ(x, S)$. A family with occurrence greater than 1 is a *multi-element* family. For example, the sequence S = (3 7 −1 6 −3 4 −6 5 3) has multi-element family 3 and multi-element family 6 where $occ(6, S) = 2$.

3.2.1 Problem Definitions

The first work dealing with multi-element families and inversion minimization was initiated by Sankoff [66], who posed the exemplar problem:

Problem 3.2.1 ((**ERD**) Exemplar Reversal Distance)**.** *Given sequences A and B, each with at least one element from some alphabet Σ, find a (not necessary contiguous) subsequence A' and B' of each sequence with exactly one occurrence of each element of Σ, so that $d(A', B')$ is minimized.*

The elements of A' and B' are called the *exemplars*. The ERD problem was proven NP-Hard [20] and currently no good algorithms exist to solve it. The following two related problems are of particular interest in this section:

Problem 3.2.2 ((**OtMDA**) One-to-Many Duplicate Assignment problem)**.** *Given a sequences $A \in \Sigma^*$ and an integer n, rename all but a single element from each multi-element family to be unique, so as to minimize the number of inversions, insertions, and deletions necessary to turn A into the identity permutation of length n.*

Problem 3.2.3 ((**MtMDA**) Many-to-Many Duplicate Assignment problem)**.** *Given two sequences $A, B \in \Sigma^*$, find a renaming of elements from multi-element families, yielding A' and B', so that the following conditions are satisfied:*

1. *for each multi-element family x of A or B, there exist exactly $min(occ(x, A), occ(x, B))$ pairs of elements — one from A and one from B — each pair having been renamed to the same unique element,*
2. *all other occurrences of x (in one of the sequences) have been renamed to be unique elements, and*
3. *the length of the minimum scenario of inversions, insertions, and deletions from A' to B' is minimized.*

OtMDA and MtMDA remain tricky to reason about due to the combination of operations that are considered in the objective function (inversions, insertion, deletion). Thus, there exist few results beyond those presented in Sections 3.3 and 3.4 that directly apply to OtMDA and MtMDA. For instance, it remained unclear as to whether they are in P or not. For this reason researchers have chosen to focus on specific aspects of the problem. The following is a summary of the variations and simplifications found in the literature:

(OtMRD) One-to-Many Reversal Distance The same as OtMDA except the objective function only counts the number of inversions in the subsequence restricted to the remapped elements.

(MtMRD) Many-to-Many Reversal Distance The same as MtMDA except the objective function only counts the number of inversions in the subsequences restricted to the remapped elements.

(RDD) Reversal Distance with Duplicates The same as MtMRD except the input is restricted to sequences with equal size gene families (for any family x, $occ(x, a) = occ(x, B)$).

(OtMCM) One-to-Many Cycle Maximization The same as OtMRD except the objective function only counts the number of cycles in the induced breakpoint graph.

(MtMCM) Many-to-Many Cycle Maximization The same as MtMRD except the objective function only counts the number of cycles in the induced breakpoint graph.

(OtMBM) One-to-Many Breakpoint Minimization The same as OtMRD except the objective function only counts the number of breakpoints in the induced permutation.

(MtMBM) Many-to-Many Breakpoint Minimization The same as MtMRD except the objective function only counts the number of breakpoints in the induced permutation.

In the following section we attempt to put these problems along with our own work into context.

3.2.2 Background

Our approach [54] was the first to address the OtMDA problem by presenting a constant-factor approximation algorithm. We briefly present a refined (and improved, in terms of the error bound) version of this work in Section 3.3. One of the main steps in finding our solution to the OtMDA problem requires the computation of the *minimum cover*: a minimum cardinality set of non-overlapping substrings (from the input sequence A) that match (in forward or reverse direction) the maximum number of elements from the length n identity permutation. For example, $\{$(1 2),(3 4),(−6 −5)$\}$ and $\{$(1 2 3),(−6 −5 −4)$\}$ are both covers for A = (−6 −5 1 2 −6 −5 −4 3 4 1 2 3) with $n = 6$, but only the later is a minimum cover. Computing the minimum cover in this case is equivalent to OtMBM. It turns out that the greedy method of repeatedly choosing a largest common substring from an unused portion of A and the identity yields a minimum cover (see Lemma 3.3.3). The relaxed analogue we call MtMBM, applicable to MtMDA, is unfortunately APX-Hard to compute even with a guarantee that $occ(x, A) = occ(x, B)$ for all x [41]. We show in Section 3.4 that the MtMDA problem has a satisfactory solution in practice.

Chen *et al.* [26] attempted to solve RDD as a step in solving a larger problem: given two nucleotide sequences, find genes that are most likely to have been the same in the nearest ancestor (called orthologous genes). A step in their algorithm uses an analogue to the minimum cover, named more verbosely "minimum common string partition": find a minimum cardinality partition of two strings into the same collection of substrings or report that none exists (equivalent to MtMBM). A thread of work exists that addresses MtMBM under various restrictions [41, 27, 50, 49]. All of these results apply only to instances where for all x, $occ(x, A) = occ(x, B)$, so cannot be directly used within our approximation framework, but could lead to progress in the future.

While, as we stated, the use of common partition-based methods is limited (APX-Hard) for the MtMDA problem [41], these methods are also somewhat limited for OtMDA due to the existence of the following family of permutations.

Theorem 3.2.4. *There exists a family of permutations where the minimum common string partition yields a distance twice that of the optimal for OtMDA.*

Proof. Take A to be a sequence of length $2n$ that is created by concatenating the permutation A_1=(−1 −2 $\ldots$ $-n$) and a permutation A_2 where every adjacency is a breakpoint and there are only cycles of length four [1]. Because it is comprised only of length four cycles, A_2 must have an odd number of elements (n is odd) and will need exactly $(n + 1)/2$ inversions to sort. Choice of either all the elements of A_1 or all the elements of A_2 will yield the minimum size cover n. Thus, if all the elements of A_1 are chosen to match those in the identity, n inversions and 1 deletion are required whereas if all the elements of A_2 are chosen $(n + 1)/2$ inversions and 1 deletion are required. So as n grows the ratio of optimal to worst case cover choice $\frac{n+3}{2n+2}$, goes to 2. □

[1] For instance, take the permutation of length n = 2m+(m+1):
A_2 = (*-(2m+1)* **-1** (2m+2) *2m* **2** -(2m+3) *-(2m-1)* **-3** (2m+4) *(2m-2)* $\ldots$ **-(m-1)** (n-1) *(m+2)* **m** -n *-(m+1)*)

Since cover (minimum common partition) based methods simply attempt to rename the sequence so as to minimize the number of breakpoints, better solutions to OtMDA can sometimes be found by attempting to minimize cycles in the resulting breakpoint graph. We offered a first look into the power of this approach in [83], which we present in Section 3.5. The work by Chen *et al.* [26] also has a cycle maximization heuristic applied after assigning a minimum common partition. The only other known work attempts to directly solve MtMCM by formulating an integer linear program which has an exponential (in n, the length of the sequence) number of formulas and variables [78].

Bryant [20] established that the exemplar reversal distance (ERD) problem is NP-hard via a reduction (the simple version is found as an addendum to the original paper) from the unsigned reversal distance problem (proven hard by Caprara[25]). The reduction takes the unsigned permutation — the input for the unsigned reversal distance — and replaces each element e with two elements ($+e$ $-e$). This way the exemplar problem picks a sign for the elements of the unsigned permutation so as to minimize the signed inversion distance, which yields a minimum reversal scenario for the unsigned permutation. Notice that the identical reduction to OtMRD (and hence, MtMRD) holds. Chen *et al.* [26] showed that RDD is NP-Hard with a similar technique. Note, however, that this reduction cannot be applied to the cycle maximization problems (OtMCM and MtMCM) due to the fact that the assignment that maximizes cycles does not necessarily give the minimum reversal distance (as when hurdles are created)[2]. In Section 3.6 we give a more complicated reduction that applies to the cycle maximization problems. The proof is more general than existing proofs because it subsumes the aforementioned result of Bryant and Chen.

[2]With A = (2 -2 1), both choices yield a single cycle but (2 1) is a hurdle whereas (-2 1) is not. There are also instances when the assignment that minimizes the reversal distance will always yield *fewer* cycles than the maximum cycle assignment: for A = (2 1 3 -3 5 4) there is one assignment that gives 2 cycles and 2 hurdles whereas the optimal assignment gives 1 cycle and 0 hurdles.

3.3 Approximating the One to Many Duplicate Assignment (OtMDA) Problem

(This is joint work with Mark Marron)

In this section, we extend work of El-Mabrouk [32] by providing a polynomial-time approximation algorithm with constant error bound to compute edit distances under inversions, deletions, and unrestricted insertions (including duplications) from the any sequence to the identity permutation. We also show that the algorithm we implemented works well in practice. An approximation with heuristics that perform well in practice, is the best we can hope for due to the NP-Hardness result of Section 3.6.

As in the standard statement of the equal gene content problem, we assume that the desired (optimal) edit scenario is that which uses the fewest operations, with all operations counted equally. We move from a subject sequence S to a perfectly sorted target T.

Our approach is based on a canonical form for edit scenarios which we introduced in [54]: we showed that shortest edit scenarios can be transformed into equivalent sequences of equal length in which all insertions are performed first, followed by all inversions, and then by all deletions. We state the theorem here without proof.

Theorem 3.3.1 ([54])**.** *Given a minimum edit scenario $S \cdot o_1 \cdot o_2 \cdot \ldots \cdot o_m = T$ there is an equivalent edit scenario $S \cdot ins_1 \cdot \ldots \cdot ins_p \cdot inv_1 \cdot \ldots \cdot inv_q \cdots del_1 \cdot \ldots \cdot del_r = T$ where all insertions are followed by all inversions which are followed by all deletions.*

The utility of this theorem is two-fold. As we will see, it is instrumental in the application of Lemma 3.3.4 to our approximation algorithm. It also allows us, in practice, to take advantage of El-Mabrouk's exact algorithm for inversions and deletions, which we then extend by finding the best possible prefix of insertions, producing an approximate solution with bounded error.

Section 3.3.1 outlines our method for handling unrestricted insertions. Section 3.3.1 gives the algorithm matching that method. Section 3.3.2 presents the complete algorithm outline as well as an analysis of its error bounds. Finally, Section 3.3.3 gives some empirical results for method presented here.

3.3.1 Unrestricted Insertions

The presence of duplicates in the sequence makes an analysis much more difficult; in particular, it prevents a direct application of the method of Hannenhalli and Pevzner and thus also of that of El-Mabrouk. We could solve this problem by assigning distinct names to each copy, but this approach begs the question of how to assign such names. Sankoff proposed the exemplar strategy [66], which attempts to identify, for each gene family, the "original" gene (as distinct from its copies) and then discards all copies, thereby reducing a multi-set problem to the simpler set version. However, identifying exemplars is itself NP-hard [21]—and much potentially useful information is lost by discarding copies. We found a simple selection method that discards none of the elements of the sequence, based on substring pairing, while yielding a constant error bound.

Sequence Covers

Our job is to pick a group of substrings from the subject such that every element in the target appears in one of those substrings. To formalize and use this property, we need a few definitions. Call a substring $e_1 e_2 \ldots e_m$ a *block* if we have $\forall j,\ e_{j+1} = e_j + 1$. Given a block s_i, define the *normalized* version of s_i to be s_i itself if the first element in s_i is positive, and the inversion of s_i otherwise; thus the normalized version of s_i is a substring of the identity. Call a subsequence T_{nd} of the target string T, the *non-deleted* portion of T if T_{nd} (i.e. only the elements from T that also exist in S). Note that T_{nd} is not a substring, but a subsequence; that is, it may consist of several disjoint parts of T. Thus it is unique. Given a set C

of normalized blocks in S such that all the elements are also incomparable under the substring relation, define $\uplus C$ to be the string produced as follows; order the strings of C lexicographically and concatenate them in that order, removing any overlap. We will say that a set C of blocks from S is a *cover* for T if T_{nd} is $\uplus C$. Note that a cover must contain only blocks.

Set $T = (1, 2, 3, 4, 5, 6, 7)$ and $S = (3, 4, 5, -4, -3, 5, 6, 7)$. The set of normalized maximal blocks is $\{(3, 4, 5), (3, 4), (5, 6, 7)\}$; T_{nd} is $(3, 4, 5, 6, 7)$; a possible cover for T is $\{(3, 4, 5), (5, 6, 7)\}$; and $\uplus C_p$ is $(3, 4, 5, 6, 7)$.

Lemma 3.3.2. *For a subject S, d operations from the identity T, there exists a cover of size $2d + 1$.*

Proof. By induction on d. For $d = 0$, S itself forms a cover, since it is a block; hence the cover has size 1, obeying the bound. For the inductive step, note that deletions are irrelevant, since the cover only deals with the non-deleted portion; thus we need only verify that insertions and inversions obey the bound. An insertion between two blocks simply creates another block, while one inside a block splits it and adds a new block, for an increase of two blocks. Similarly, an inversion within a block cuts it into at most three blocks, for a net increase of two blocks, while an inversion across two or more blocks at worst cuts each of the two on the ends into two blocks, leaving the intervening sequence contiguous, also for a net increase of two blocks. Since we have $(2(d - 1) + 1) + 2 = 2d + 1$, the bound is obeyed in all cases. □

Building the Minimum Cover

Let $\mathcal{C}(T, S)$ be the set of all (normalized versions of) maximal contiguous substrings (blocks) shared between T and S. We will build our cover greedily from left to right with this simple idea: if, at some stage, we have a collection of strings in the current cover that, when run through the $\uplus$ operator, produces a string that is a prefix of length i of our target T, we consider all remaining strings in $\mathcal{C}(T, S)$ that begin at or to the left of position i—that can extend the current cover—and select that which extends farthest to the right of position i. Although this is a simple (and efficient) greedy construction, it actually returns a minimum cover, as we can easily show by contradiction.

Lemma 3.3.3. *The cover derived by our greedy algorithm is optimal.*

Proof. Assume there exists a cover, say C_{min}, that is smaller than the one provided by our construction, C_{const}. Order the sequences in C_{min} by increasing value of the smallest index in the sequence. Let α be the smallest element, say the kth element in this order such that α is not the same as the kth sequence of C_{const} under the same order. We have three cases:

1. During the construction of C_{const}, α was not selected for C_{const} because the previous selection of a cover element in C_{const} did not cover all the way to the start index of α. Then α is not the first differing element in the order, a contradiction.

2. During the construction of C_{const}, α was not selected for C_{const} because there was a sequence that had the same start index as α, but covered fewer elements than α. But this contradicts the selection criteria for our construction.

3. During the construction of C_{const}, α was not selected for C_{const} because there was a sequence that had the same start index as α, but covered more elements than α. Then C_{const} has at most as many elements as C_{min}, a contradiction.

□

3.3.2 Our Algorithm

Now that we have a method to construct a minimal cover, we can assign unique labels to all duplicates, which in turn enables the use of El-Mabrouk's method for computing the edit scenario.

We first present a result relating the number of blocks in the cover to the maximum number of insertions and deletions. To do this we will need to look at the target sequence T with all the elements that do not appear in S removed, we call this new sequence T_{ir} to denote that all the inserted elements have been removed.

Lemma 3.3.4. *Let α be the minimal edit scenario from S to T, using l insertions and m inversions. Let α' be the minimal edit scenario of just inversions and deletions from S to T_{ir}. The extension $\hat{\alpha}$ (extending α' with the needed insertions) has at most $l + m$ insertions.*

Proof. Clearly, our method will do at least as well as looking at each inserted string in T and taking that as an insertion for $\hat{\alpha}$. Now, looking at the possible effect of each type of operation on splitting a previous insertion, we have 3 cases. Take v as the inserted substring:

1. Inserting another substring cannot split an inserted substring—it just creates a longer string of inserted elements. (If x is inserted, $uv_1v_2w \to uv_1xv_2w$)
2. Deletion of a substring cannot split an inserted substring—it just shortens it, even perhaps to the point of eliminating it and thus potentially merging two neighboring strings. (If v_2 is deleted, $uv_1v_2v_3w \to uv_1v_3w$)
3. An inversion may split an inserted substring into two separate strings, thus increasing the number of inserted substrings by one. It cannot split a pair of inserted substrings because the inversion only rearranges the inserted substrings; it does not create new blocks. (If u_2v_1 is the substring inverted, $u_1u_2v_1v_2w \to u_1\overline{v_1u_2}v_2w$)

Thus, if we have l insertions and m inversions in α, there can be at most $l + m \leq |\alpha| = d$ inserted substrings in T. □

Starting with the subject S with cover elements $(s_1, s_2, \ldots, s_k)$ numbered by the order in which they appear in the target T. We place in order (for i from 1 to k) each s_i in its final location in T with at most two inversions; one to place it and one to orient its sign. Thus, we use at most $2k$ inversions. By Lemma 3.3.2 and Lemma 3.3.3 we have $k \leq 2d + 1$, so our inversion scenario will have at most $4d + 2$ inversions. Theorem 3.3.1 tells us that Lemma 3.3.4 applies to both insertions and deletions, thus there at most k insertions and k deletions. Thus, the edit scenario produced by the proposed method has at most $6d + 2$ operations, where d is the minimum distance.

While this error bound is large — it is a factor of 3 larger than the lower bound given in Theorem 3.2.4 — it is the lowest known bound for OtMRD. Furthermore, the bounds can be easily computed on a case-by-case basis in order to provide information on the accuracy of the results for each run. We expect the error encountered in practice to be much lower and that further refinements in the algorithm and error analysis should bring the bound closer to that of the lower bound.

3.3.3 Experimental Results

To test our algorithm and get an estimate of its performance in practice, we ran simulations. We generated pairs of sequences, one the sequence $(1, 2, 3, \ldots, n)$, for $n = 200, 400, 800$, and the other derived from the first through an edit scenario. Our edit scenarios, of various lengths, include 80% of randomly generated inversions (the two boundaries of each inversions are uniformly distributed through the array), 10% of deletions (the left end of the deleted string is selected uniformly at random, the length of the deleted string is given by a Gaussian distribution of mean 20 and deviation 7), and 10% insertions (the

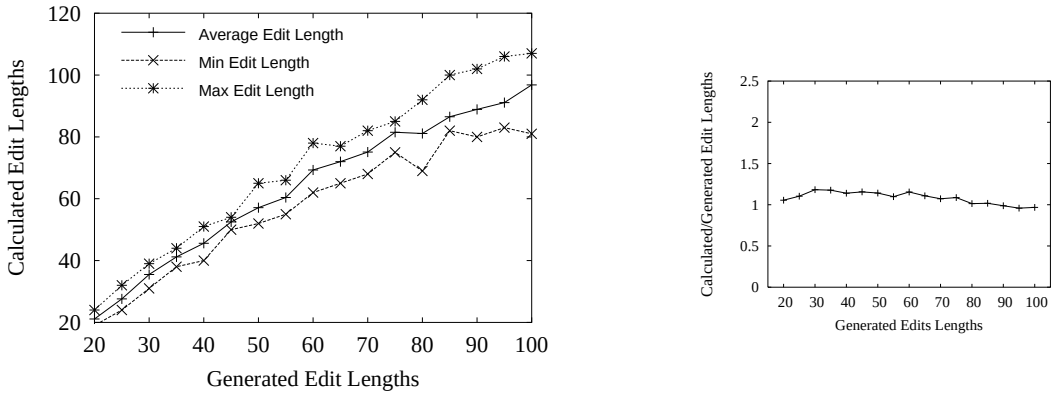


Figure 3.1: Experimental results for 200 genes. Left: generated edit length vs. reconstructed length; right: the ratio of the two.

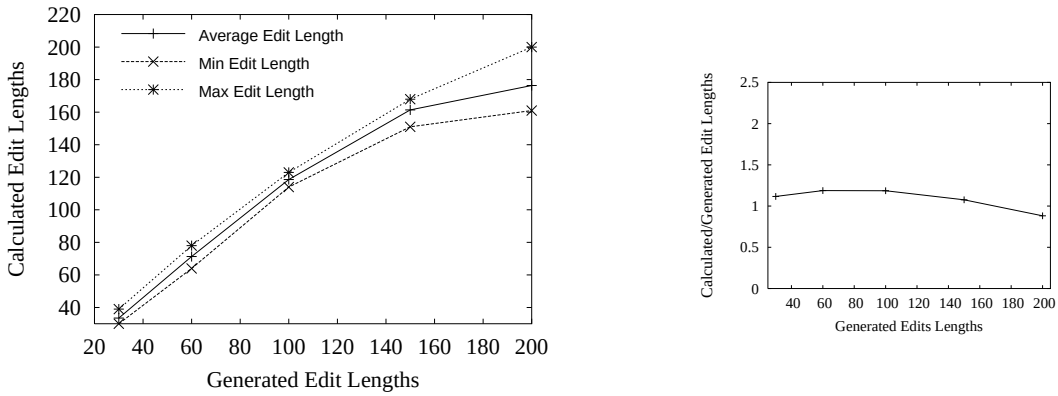


Figure 3.2: Experimental results for 400 genes. Left: generated edit length vs. reconstructed length; right: the ratio of the two.

locus of insertion is uniformly distributed at random and the length of the inserted string is as for deletion), with half of the insertions consisting of new elements and the other half repeating a substring of the current sequence (with the initial position of the substring selected uniformly at random). Thus, in particular, the expected total number of duplicates in the subject sequence equals the generated number of edit operations—up to 400 in the case of 800-gene sequences. We ran 10 instances for each combination of parameters (in the figures below, we show the average, minimum, and maximum values over the 10 instances). The results are gratifying: the error is consistently very low, with the computed edit distance staying below 3% of the length of the generated edit scenario in the linear part of the curve—that is, below saturation. (Of course, when the generated edit scenario gets long, we move into a regime of saturation where the minimum edit scenario becomes arbitrarily shorter than the generated one; our estimated length shows this phenomenon very clearly.) Figures 1, 2, and 3 show our results for sequences of 200, 400, and 800 genes, respectively.

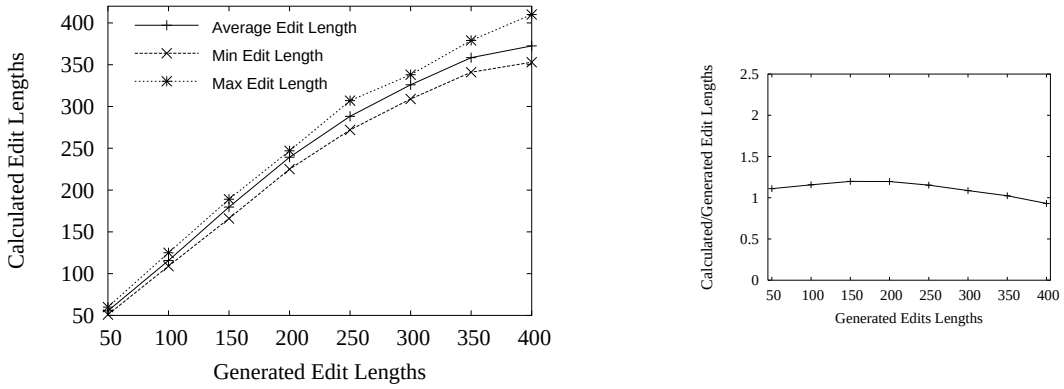


Figure 3.3: Experimental results for 800 genes. Left: generated edit length vs. reconstructed length; right: the ratio of the two.

3.4 Applying the Cover to the Many to Many Duplicate Assignment (MtMDA) Problem

(This is joint work with Mark Marron, Jijun Tang, and William Arndt)

Here, we generalize the approach from the previous section to compute the distance between two arbitrary sequences and show through extensive simulations that we reconstruct a scenario of operations that reflects the true evolutionary distance. Since this is an experimental result, it is hard to verify exactly what the minimum number of operations is; hence the following results do not apply to MtMDA directly, but give an indication of how well our distances track the true distance. Our algorithm computes distances between two sequences in the presence of insertions (including duplications), deletions, and inversions; in our simulations, the distance computed very closely approximates the true evolutionary distance up to a (high) saturation level. The approximation is in fact good enough that it can be used in conjunction with a distance-based phylogenetic reconstruction method (we used the most common one, neighbor-joining) to reconstruct trees of reasonable sizes (up to 100 sequences) and very large pairwise distances with high accuracy.

It is worthwhile to note that although we consider only inversions (aside from duplicating insertions and deletions), the properties of a minimum cover discussed in Section 3.4.1 imply that it would likely perform well with other operations such as transpositions: the cover is a model-independent method. However, due to the fact that transpositions distances are yet to be well understood we do not consider them in this exposition.

The rest of the section is organized as follows. Section 3.4.1 establishes the background. Section 3.4.2 discusses the difficulties faced when using two arbitrary sequences and how we solve them to recover a solution in the spirit of our earlier results; it outlines our method for producing a cover in quadratic time. Section 3.4.3 presents the design of our two studies while Section 3.4.4 shows how our constructed cover performs when estimating pairwise tree distances and how these distances can be used in tree reconstruction. Finally, Section 3.4.7 uses our distance method with a more sophisticated tree building method, and shows that the combined methods rebuild trees more accurately than other know methods.

3.4.1 Background

The Cover

Our solution attempts to assign each gene in the subject to a gene from the same family in the target; that is, it creates a maximum matching between the genes in corresponding gene families of the two sequences. However, some matchings are clearly preferable to others because they reduce the number of insertions, deletions, and rearrangement operations required to transform one genome into the other. We define a *minimum cover* to be a cover that maps the subject to the target with the fewest common substrings. The effect of renaming according to a minimal cover is to yield a breakpoint graph [42] with maximum number of cycles of length 2, minimizing the number of breakpoints between the renamed sequences. Thus a minimum cover is a solution to MtMBM.

Difficulties With an Arbitrary Target

The difference between our work from Section 3.3 and that of this section is the presence of duplicate genes in the target. When building the cover with the identity permutation as the target, all candidate cover elements from the subject are immediately apparent because of the unique correlation between their identity and their index in the target genome. In the case of an arbitrary target, however, this correlation no longer exists. Moreover, a cover may no longer cover all genes from one or the other genome: clearly, if genome A has more duplicates of gene x than genome B, and genome B has more duplicates of gene y than genome A, then any matching between these two sequences must leave some duplicates of gene x unassigned in A and some duplicates of gene y unassigned in B. For example, with subject (1 2 3 −5 −2) and the identity permutation (1 2 3 4 5) as target, we have a cover, using indices in the target, for indices 1 through 3, one for index 5, and one for index 2; but for the same subject and for target (−7 1 2 3 5 −3), we obtain partial covers for indices 2 through 4 or for indices 5 through 6. We settle for fast heuristics to build our cover due to the results of Goldstein *et al.* [41], who show that computing a minimum cover is APX-Hard even with a guarantee that $occ(x, A) = occ(x, B)$ for all x.

3.4.2 Constructing a Small Cover

The algorithm used in Section 3.3 looks for the longest matching substring. As long as such a longest match is unique, there is no difficulty beyond identifying such matches as quickly as possible. (A naïve cubic-time algorithm will do, although, as we shall see, the same job can be done in quadratic time.) When the longest match is not unique, however, finding a minimum cover may require an exploration of the alternatives and thus exponential time. Instead, we use a greedy heuristic to break ties.

We have tried several tie-breaking heuristics (and compared them to breaking ties at random). One heuristic is based on identifying a possible extension of the match (to one or the other side). If the substring to one side of the match is the inverse of the substring to the same side of the match in the other genome, for instance, if we had substrings (1 2 −4 −2) in the target and (1 2 2 4) in the subject, we may prefer to match these substrings to each other (even though there may be another (1 2) elsewhere in both sequences) because they are only a single inversion from each other. Another heuristic is to minimize the interaction between matches. The longer the match we make at each iteration, the fewer potential matches may be needed overall, so we may want to choose the match with a range of indices that crosses the smallest number of other match ranges. Section 3.4.5 contains some conclusions about the effectiveness of these heuristics.

To find the longest match, we begin by finding all possible maximal matching substrings and then repeatedly pick the next largest substring, doing necessary bookkeeping to reflect our successive choices. Let M be the set of all maximal matching substrings between the subject and the target that have not yet been picked. For instance, if we start with target genome (1 2 1 3 4 5 6 7 8) and subject genome (6 7 3 4 5 6 1 2 3 6 7 8), we initially have $M = \{(67), (3456), (12), (3), (678)\}$. We say that two matches

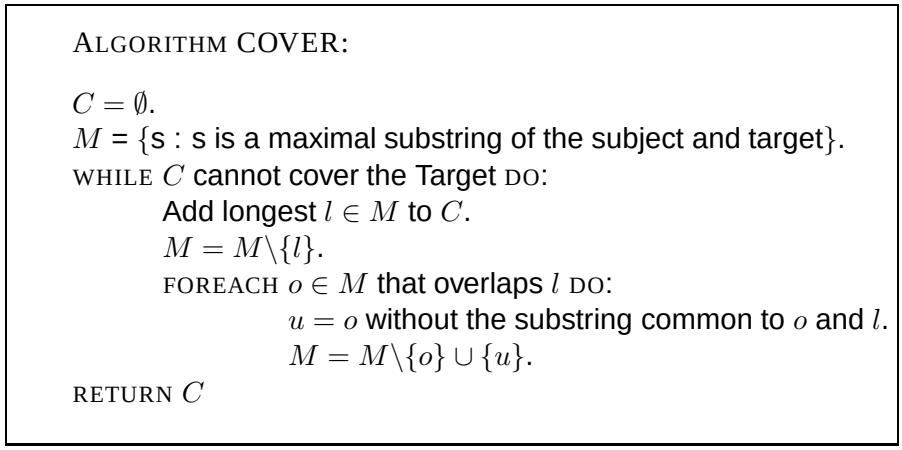


Figure 3.4: Choosing a nearly minimal cover.

overlap if their indices in the target intersect. By picking the longest match l, we cover a part of the target that may overlap with some number s of other matches, call them $o_1, o_2, \ldots, o_s \in M$. In our example, match (3 4 5 6) would be chosen first, covering the 6 from matches (6 7) and (6 7 8) and the 3 from match (3). The overlapping portion of each match $o_i, 1 \leq i \leq s$ is then removed, resulting in shorter matches. Thus, three of those matches in our example will be shortened yielding (7), (7 8), and (). The resulting algorithm is described in Figure 3.4.

We proceed to show that COVER can be implemented to run efficiently, first stating the theorem and then providing the necessary background to prove it.

Theorem 3.4.1. *Algorithm* COVER *can be implemented to run in quadratic time.*

We represent M by a list arranged by match length. We keep an auxiliary data structure, the *index reference*, to maintain the set M through each iteration. This index reference is an array (0 indexed) of lists, one for each index of the target; each such list, an *index list*, contains the matches that have an endpoint on that target index. For instance, in our example three such matches would be (3 4 5 6), (6 7), and (6 7 8). These matches are associated with indices 3 through 6, 6 through 7, and 6 through 8 of the target. Thus index 6 of the target would have three members to its index list, because the matches (3 4 5 6), (6 7), and (6 7 8) all have the 6 (from position 6 in the target) as an endpoint. Index 7, however, would have a single match (6 7), because (6 7 8) does not have 7 as an endpoint. A simple way to find all possible maximal matches in quadratic time is to slide the subject over the target, comparing all possible combinations of indices between the two. Each match found is placed in M and the index lists for its endpoints. The key to this implementation is the efficient update of overlapping matches. With the index lists we can find all $o \in M$ that overlap a given $m \in M$ by examining each list that corresponds to an index that m spans. When the match m that spans indices i through k is chosen, we can shorten each o_i that overlaps from the left by relocating it from the index list for $j, i \geq j \geq k$, to the index list for $i - 1$. Similarly, each o_k that overlaps m from the right can be relocated to the index list for $k + 1$.

Lemma 3.4.2. *The maximum number of matches that can have an endpoint at a given index of the target is bounded by $4n$, where n is the length of the longer genome.*

Proof. Each index in subject or target can be of two types: a left or right endpoint of a match. All four combinations of endpoint types can occur for a given pair of indices. If there were more than one match per pairing of endpoint types then one of them could not be maximal, therefore there can be at most four distinct maximal matches associated with every pair of indices. Since there are n indices in the subject, there can be at most $4n$ matches associated with a single index of the target. □

It follows immediately that the number of maximal matches between two sequences, the larger of which has size n, is $O(n^2)$.

Lemma 3.4.3. *Initialization of M and of the index reference takes quadratic time.*

Proof. We know that the number of maximal matches is $O(n^2)$ and that the length of a match is bounded by the size of the sequences. We can add a match to a list organized by length in constant time through direct indexing. Likewise, addition to the end of a given index list can be done in constant time. Since there are $O(n^2)$ matches and placement into the index reference is $O(1)$, we can build these lists in quadratic time. □

Lemma 3.4.4. *A match can be relocated between index lists at most twice before being removed from consideration.*

Proof. It is sufficient to show that a match e will not be encroached upon from the same side twice. Assume that e is shortened from one direction by match m and later from the same direction by match m' without being covered. Because m was picked by the algorithm first, m' must not stretch past the opposite end of m. Therefore, either m' covers less than e or e must be completely covered —a contradiction in either case. □

We are finally ready to prove Theorem 3.4.1.

Proof. (of Theorem 3.4.1) Initialization takes quadratic time (Lemma 3.4.3). Each match in each index list is visited a constant number of times (Lemma 3.4.4). When visited, each match is shortened, removed from consideration or relocated to the index list at the edge of the most recently chosen match, and then relocated in the length list. Since each of these operations runs in constant time, the running time is bounded by a constant times the total number of matches visited. Since each index list is visited at most once and the length of that list is at most linear (Lemmata 3.4.2 and 3.4.4), the running time is $O(n^2)$. □

Theorem 3.4.5. *The distance function can be computed in $O(n^2)$ time.*

Proof. The cover can be generated and applied in $O(n^2)$ time. Then the algorithm presented in [54] or [32] can be applied. Both methods run in $O(n^2)$ time. □

3.4.3 Experimental Design

We used two types of tests to assess the accuracy and utility of our tree distance algorithm. The first set of tests were designed to determine if our distance function accurately modeled the true pairwise tree (true evolutionary) distances. The second set of tests were used to evaluate the effectiveness of our distance function within the most simple distance-based phylogenetic reconstruction algorithm.

Pairwise Error

For this experiment, we generated evolutionary trees with known edge lengths and compared the pairwise distances between the leaves with those computed by our algorithm. Variance in tree shape does not matter here; in fact, since we want a large range of pairwise tree distances, a perfectly balanced tree is best.

In the following tests we used the simplest version of the method described earlier. The algorithm picks the largest match to make and in the case of ties picks one of the tied matches at random. Clearly other information is present in the sequences that could provide a better choice of match and thus lead to a more accurate distance score. However, all of the heuristic methods that we used failed to have a noticeable impact on the accuracy of the distance value returned. Furthermore, in experiments with

a large number of random restarts, we found that most of the values clustered around the true value with a small number of outliers; we also found that averaging over a smaller number of random restarts and discarding any substantially outlying points provided a distance estimate that was nearly indistinguishable from the distance estimate computed with the use of our best heuristics (see Section 3.4.5). While the use of biological information to select the best match could prove effective in generating more biologically plausible evolutionary paths, the current method seems to perform quite well in terms of distance computations.

Not enough is known about inversions, deletions, insertions, and duplications to enable one to set good parameters (such as lengths of inversion, for instance) *a priori*, so we chose values so as to ensure that a single operation would not completely alter the genome. Most of our tests were conducted with a root genome of 800 genes on a tree of depth 4; such a tree has 16 leaves and thus 120 pairs of sequences with paths from 2 to a maximum of 8 edges between sequences.

Tree Reconstruction

We tested the performance of our distance functions using neighbor-joining, the standard distance-based tree reconstruction method. Due to the dearth of real-world trees reconstructed using biological techniques, we had to generate model trees that would exercise our algorithm over a wide range of plausible models of gene-order evolution. (We conducted one study using real data with very large numbers of insertions and deletions; partial results to date show promise [30].) We generated one thousand trees using a variation of the birth-death model that produces a larger variation in tree topologies, especially imbalanced ones that are known to be insufficiently represented in a pure birth-death model [44]. The only constraint that was placed on the operations was that the expected number of inserted elements was equal to the expected number of deleted elements, in order to keep all genome sizes within a reasonable range. (Cases where certain sequences are much smaller than others, due, e.g., to symbiosis, certainly exist, but the variation generated by our mechanism nearly encompasses that case already.) Three random restarts of our distance algorithm were used for each pair of nodes to produce the pairwise distance matrix.

Within the thousand trees the percentage of inversions varied from 50% to 90%. The remaining percentages were split evenly between insertions (duplicating and non-duplicating) and deletions. Non-duplicating insertion and duplication percentages were varied over three different tests, in which each received a quarter, a half, and three quarters of the percentage. The expected Gaussian distributed length of each operation filled a range of combinations from 5 to 30 operations per operation type. Finally, the expected number of event per edge was 20 with a Gaussian distributed variance of 10 operations.

To generate a tree we began with the identity genome on 800 genes and performed 200 evolutionary operations on it using the same parameters that are specified for generating the tree. This genome was then used as the root of the tree. For each node we checked if it should become a leaf, based on the maximum depth allowed and a random choice, if not we stopped. Otherwise we created each of the two children by performing the randomly selected operations (as specified in the previous paragraph) on the parent. Each type of operation (inversion, non-duplicating insertion, duplication, and deletion) was selected at random according to a fixed distribution. The interval over which an operation acts is produced with one endpoint selected at random and a length drawn from a Gaussian distribution. For duplications, the interval to be duplicated is selected and then inserted at an index chosen uniformly at random in the genome.

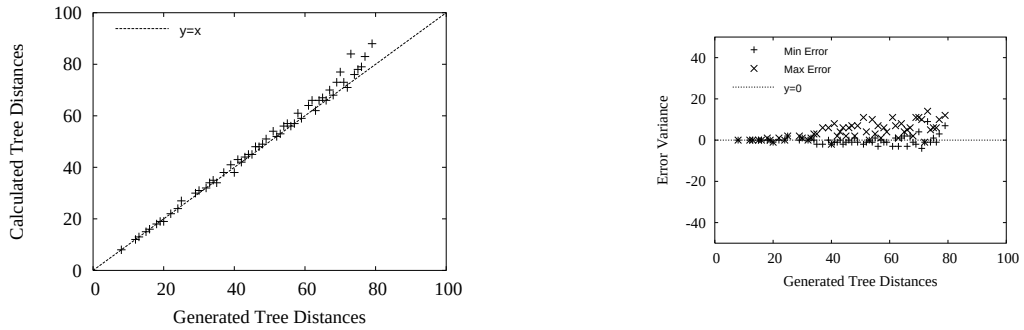


Figure 3.5: Experimental results for 800 genes with expected edge length 10. Left: generated distance vs. reconstructed distance; right: the variance of computed distances per generated distance.

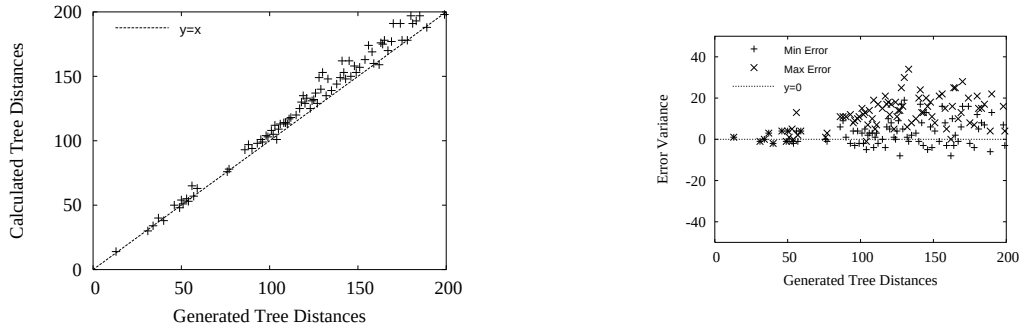


Figure 3.6: Experimental results for 800 genes with expected edge length 20. Left: generated distance vs. reconstructed distance; right: the variance of computed distances per generated distance.

3.4.4 Experimental Results

Pairwise Error

We present results for one of the many mixes of operations used in our simulations; other mixes gave very similar results. This particular data set used a mix of 70% inversions, 16% deletions, 7% insertions, and 7% duplications. The inversions had a mean length of 20 and a standard deviation of 10. The deletions, insertions, and duplications all had a mean length of 10 with a standard deviation of 5. We used four trees of 16 leaves as described earlier, with 10, 20, 40, and 60 expected operations per tree edge; these choices can result in very large pairwise distances—up to an expected 480 operations (on just 800 genes) for the most distant pairs. For these four trees, our algorithm was run with 10 random restarts and simple randomization for the selection of the matchings.

Figures 3.5 through 3.8 show the results (as a scatter plot of the 120 data points for each experiment) for these four datasets. In each figure, the left-hand plot shows the estimated tree distance on the ordinate against the true evolutionary distance (from the simulation) on the abscissa. A perfect result would simply trace the 1:1 diagonal, which is lightly marked on each plot to aid in evaluating the results. The right-hand plot displays the deviation from the 1:1 ideal as a function of the true evolutionary distance, plotting largest and smallest differences between computed values and the true value, for each true value.

These plots show that our distance estimator tracks the true evolutionary distance very closely up to a saturation threshold, where it starts lagging seriously behind the true value. Such saturation is of course expected; what is surprising is how high that saturation threshold is. On sequences of roughly 800 genes, saturation appears to occur only around 250 evolutionary events and our estimator tracks very accurately to at least 200 events. Moreover, the smaller plots indicate that the variance is very small up

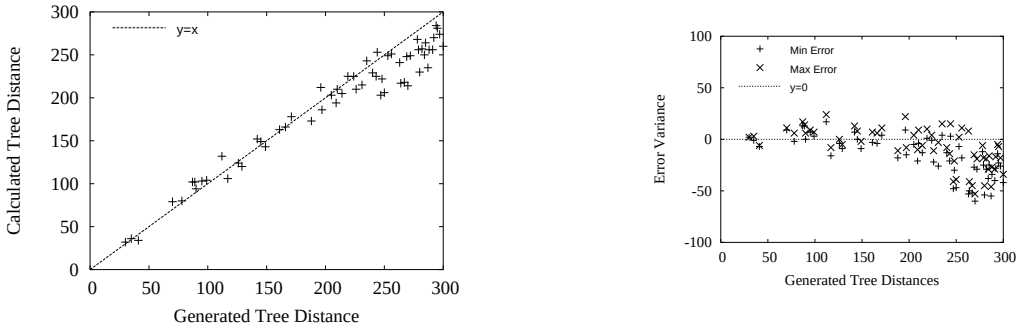


Figure 3.7: Experimental results for 800 genes with expected edge length 40. Left: generated distance vs. reconstructed distance; right: the variance of computed distances per generated distance.

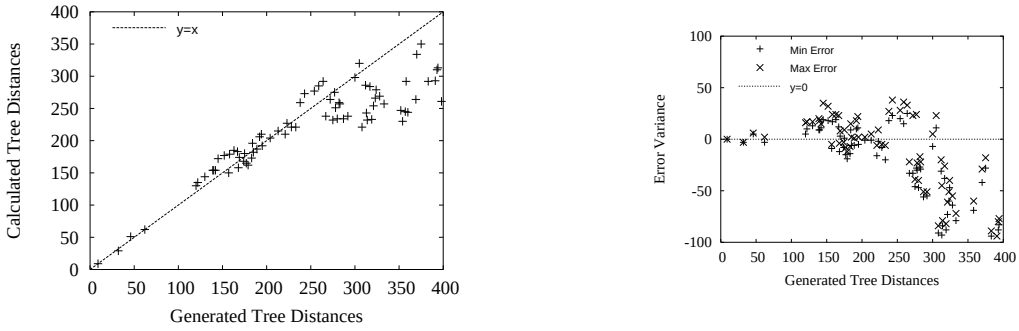


Figure 3.8: Experimental results for 800 genes with expected edge length 60. Left: generated distance vs. reconstructed distance; right: the variance of computed distances per generated distance.

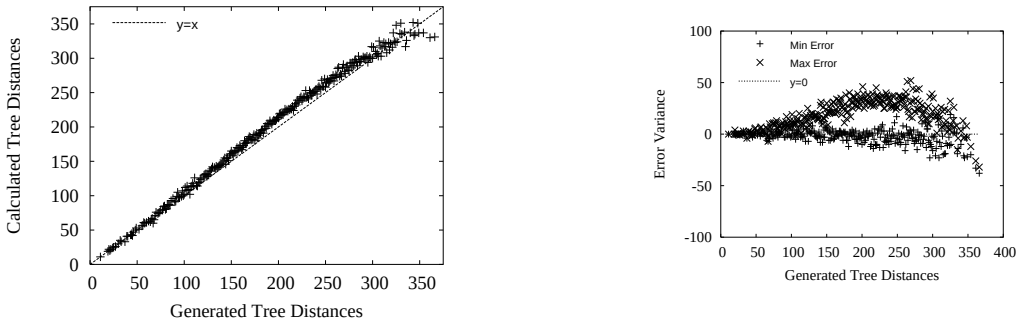


Figure 3.9: Experimental results for 1,200 genes with expected edge length 20. Left: generated distance vs. reconstructed distance; right: the variance of computed distances per generated distance.

to 200 events and remains reasonable up to 250 events.

These results are not limited to small trees. We ran another series of tests involving trees of 50 leaves; while the main purpose of these tests was to assess the quality of tree reconstruction using our distance computations, we checked the computed distances against the true distances for these trees as well. Figure 3.9 shows the same two scatter plots (this time on roughly 1,250 data points) for one such tree. For these larger trees, we used a root genome of 1,200 genes in order to prevent early saturation; the example reported in the figure used an expected edge length of 20 evolutionary events. With the larger number of genes, saturation now does not occur until we reach at least 350 evolutionary events. The error plot shows that the error remains sharply bounded throughout the range of values tested.

Tree Reconstruction

Since our distance computation tracks tree distances so accurately and since distance-based methods are guaranteed to do well when given distances that are close to the true evolutionary distances, we also ran a series of tests designed to ascertain the quality of tree reconstruction obtained with the most commonly used distance-based reconstruction method, neighbor-joining (NJ). The NJ method runs in low cubic time and thus is applicable to large datasets, but, like all distance-based methods, it is known to produce poor results when the range of tree distances gets large (see, e.g., [60].

Recall that we generated a very large number of diverse tree topologies, producing a population of trees that more closely matches the observed balance statistics [44] than would be the case with a pure birth-death process. We evaluated results using the standard *Robinson-Foulds (RF) distance* [64], which is simply (in the case of binary trees, as in our series of experiments) the number of edges (or bipartitions) present in one tree, but not in the other. In several cases, we present the *RF error rate*, which is the ratio of the RF distance to the number of taxa in the tree. In terms of the latter measure, most systematists will consider rates above 10% to be unacceptable and rates below 5% to be very good.

The tree reconstruction performed very well on the generated trees, as shown in Figure 3.10. Approximately 65% of the reconstructed trees had a Robinson-Foulds error rate of less than 5% and only 15% of the trees had an error above 10%. This reconstruction was done without any use of error correction, variances, or knowledge of the underlying model that generated the trees; it also used the simplest form of neighbor-joining. Thus, it would be easy to improve these results by refining the reconstruction method.

As an additional check, we also compared how well our method performs with respect to simply removing duplicate content and applying El-Mabrouk's exact method [32]. This comparison gives us an indication of how important it is to handle duplication in estimating true tree distances. We computed a distance matrix for each tree where a single entry of a matrix was obtained by pairwise removal of all duplicate content and subsequent computation using El-Mabrouk's exact method. The NJ method was

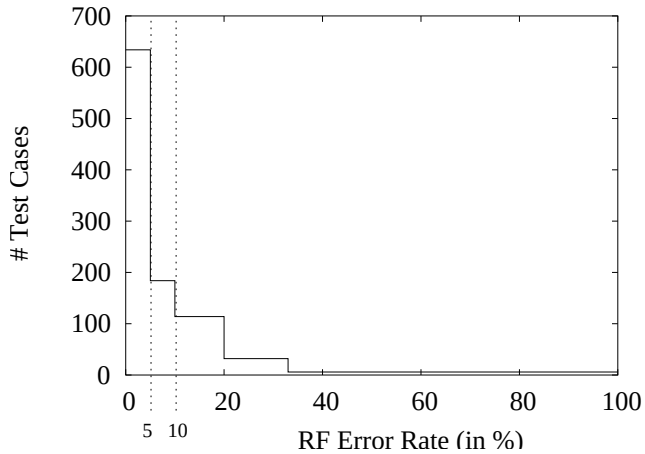


Figure 3.10: The histogram of RF error rates for reconstructions based on our distance computation.

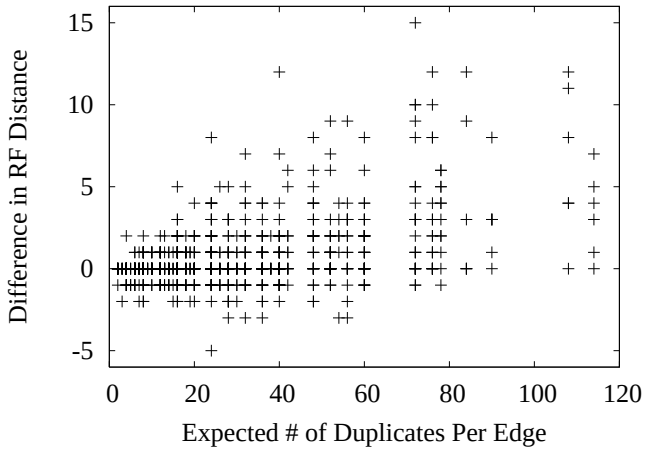
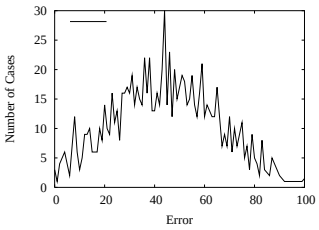


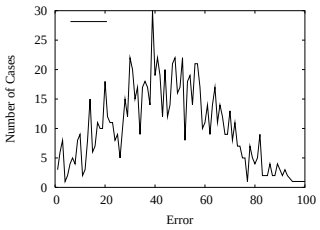
Figure 3.11: The difference in RF distance between the method without duplicates and our method as a function of the number of duplicates on an edge.

applied to each matrix to obtain a tree. Over all thousand trees the reconstruction without duplicates had a lower RF error rate than ours on only 14% of the trees; furthermore, in three quarters of those cases, the overall RF error rate for both methods was lower than 10%—that is, these were relatively easy cases. Thus, our method does better on the harder cases; the average difference in RF error rate on the trees where our method did worse on was 1.2, while the average difference in RF error rate on the trees our method did better on was 3.5. This is strong evidence that our method makes significant improvements on the state of the art. Furthermore, because of this low error rate in the 14% of cases where our method was not the best, there is good reason to believe that a slightly better tie breaker (see Section 3.4.5) will yield even more cases where the method presented here wins.

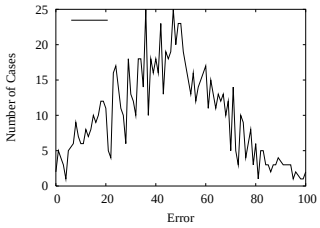
To examine how well our technique handled copies, we compared (for every test run) the RF distances of our reconstruction with those of the reconstruction without duplicates as a function of the total number of duplications. Figure 3.11, a scatter plot of the differences in RF distance, indicates that, as the number of duplicates increases, our method does correspondingly better at reconstructing the tree.



(a) random tie-breaker



(b) overlaps tie-breaker



(c) overlaps with context tie-breaker (reduced scale)

Figure 3.12: Number of cases with error for each tie-breaker.

3.4.5 Improved Heuristics

For distances used in tree reconstruction, the relative ordering of the values is more important than their absolute magnitude; it is most important to see computed distances increase as the simulated distances do. Our major goal with the introduction of more sophisticated heuristics is to reduce the variance of the scores so that the distance ordering will be more consistent and potentially result in more accurate trees.

The results presented earlier in the section used a very simple heuristic; we selected the longest match for a cover element and then chose a match at random in the cases of ties. We investigate two more promising tie-breaking heuristics (introduced in Section 3.4.2): picking a match that has the smallest overlap with the other cover elements or picking a match by looking at the immediate context of the cover elements in the source genome. By choosing the match that has minimal overlap with all other matches, we maximize the number of longest-match candidates for the next round. To understand the motivation for the context driven heuristic suppose we are trying to find a cover element for a subsequence (of genes) s in the target. Also suppose that in the target, the subsequence to the left of s is s_l and to the right of s is s_r. Then we would like to pick a match in the source genome that has the context subsequences s'_l and s'_r that are as similar to s_l and s_r as possible.

To assess the improvements when using these heuristics we ran two sets of pairwise distance comparisons. One set used sequences of length 800 with 200 operations from the identity to the first taxa and 200 operations between the taxa. The second set used sequences of length 1200 and took 400 operations between the identity genome and the first and between the first and the second taxa. In both data sets the probability of an operation being an inversion was 80%, of being a deletion 10%, of being a duplicating insertion 5%, and of being a non-duplicating insertion 5%. The distance between each pair was then computed using three heuristics, first the random selection was run, then the score was computed using overlap minimization, and finally the score was computed using the overlap minimization with context. Figure 3.12 indicates that there is little difference in the error values for the various methods. More importantly, the more sophisticated heuristics have very little impact on the variance. All methods resulted in a sample variance of about 22.6 for the sequences constructed with 400 operations.

3.4.6 Saturation

Unsurprisingly, the high-error trees have arisen from saturation in the pairwise distance data. To this point, we have referred to saturation as being the point where the variance grows too large to make the

(a) Worst RF Saturation Distribution *(b) Best RF Saturation Distribution*

Figure 3.13: Saturation occurs early for those cases where the RF error is bad.

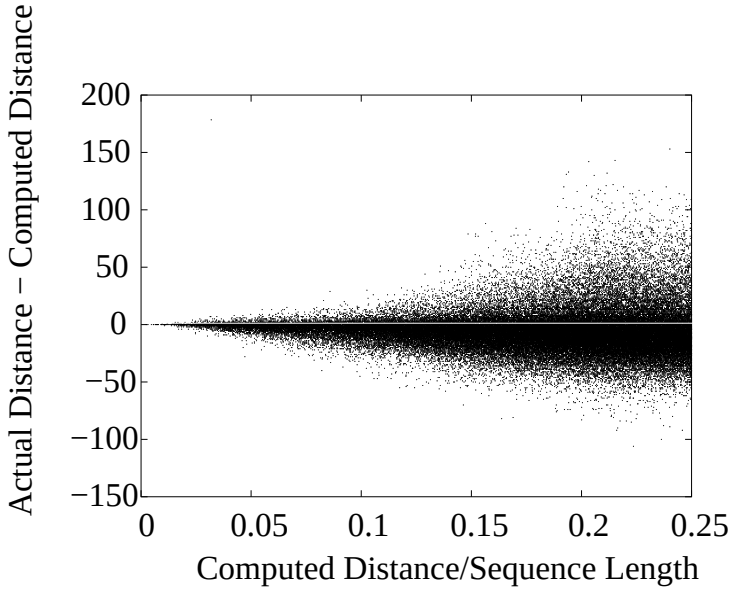


Figure 3.14: Histogram of the error (actual distance − computed distance) as the ratio of the computed distance to the genome size increases.

calculated distances useful. We now use a numerical definition: saturation occurs whenever the true evolutionary distance exceeds the distance computed under our method (which, it should be recalled, is not necessarily a minimum edit distance).

We compared reconstructed trees with an RF error greater than 10% to trees with RF errors of less than 5%. In the high RF error category over 91% of the distance matrices show saturation, whereas in the low RF error category 75.5% of the matrices are devoid of any saturation. The distribution of the number of operations where saturation occurs for the high and low RF error groups is shown in Figure 3.13. Further investigation into the properties of the trees in the high and low RF categories revealed little correlation between factors such as tree size, genome size (in genes), or distribution of operations. The major limiting factor in the accurate reconstruction of trees using this distance score is thus definitely the onset of saturation. Since the average genome size in our experiments was approximately 1000 elements, reconstruction is highly accurate when the computed edit distance does not exceed 10% of the genome size and in general performs well until the number of operations exceeds 25% of the genome size. Even in these cases the distance computation performs quite well up to the saturation point, as illustrated in Figure 3.14. The vertical axis is the difference between the actual and computed distances while the horizontal axis is the ratio of the computed distance to the genome size. Note that in a regime of saturation the computed distance stays the same while the actual distance is rising, so only the positive points should be considered when looking for saturation.

3.4.7 Sophisticated Tree Reconstruction

We still lack a good approach for inference of ancestral gene orders under the *insertion, duplication, loss, and rearrangement* (iDLR) model, both from the point of view of computational effort and from that of accuracy. Indeed, Theorem 3.3.1 and the study of Earnest-DeYoung *et al.*[30], indicate that internal gene orders are seriously underconstrained and so may not be reliably inferred—we need a more detailed and sensitive model of the evolutionary operations on a gene ordering.

Thus, we tested our method with a reconstruction algorithm that searches for a parsimonious tree from all possible topologies, using linear programming [84]. Tang and Moret[84] proposed a linear programming (LP) formulation that obviated the need to score over 99.99% of candidate topologies in their experiments. It turns out that the LP score was close enough to the actual score that Tang and Moret proposed using this score in lieu of scoring the tree, avoiding any median computation. The resulting reconstruction lacks ancestral orderings, but gives a topology, an estimated score, and estimated edge lengths (the values of the LP variables), much as a maximum-likelihood reconstruction does for sequence data. Specifics about the full tree reconstruction algorithm can be found in our paper [79].

Experimental Design

Our objective is to verify that computing under the full iDLR mode, i.e., handling both rearrangements and changes in gene content, allows for better reconstruction than handling only rearrangements on genomes reduced to signed permutations. Relative accuracy is thus our main evaluation criterion. However, absolute accuracy is needed in order to put the comparison in perspective. Since, in phylogenetic reconstruction, error rates larger than 10% are considered unacceptable, there is obviously little use in improving the error rate by a factor of two if the result is just bringing it from 60% down to 30%. We also need to test a wide range of parameters in the iDLR model, as well as to test the sensitivity of the methods to the rate of evolution. These considerations argue for testing on simulated data, where we can conduct both absolute and relative evaluations of accuracy, before we move to applying the tools to biological data, where only relative assessments of scores can be made. The range of dataset sizes need not be large, however, as we know that applying DCM methods [85] scales up results from datasets of fewer than 15 taxa to datasets of over one thousand taxa with little loss in accuracy and very little distortion over the range of parameters. As we can run many more tests on small datasets and as our primary interest is the effect of model parameters on accuracy, we generated datasets in the range of 10 to 13 taxa.

Simulated trees are often generated under the Yule-Harding model—they are birth-death trees. Many researchers observed that these trees are better balanced than most published ones. Other simulations have used trees chosen uniformly at random from the set of all tree topologies, so-called "random" trees; these, in contrast, are more imbalanced than most published trees. Aldous[2] proposed the β-split model to generate trees with a tailored level of balance; depending on the choice of β, this model can produce random trees ($\beta = -1.5$), birth-death trees ($\beta = 0$), and even perfectly balanced trees. We use all three types of trees in our experiments; for β-split trees, Aldous recommended using $\beta = -1$ to match the balance of most published trees; instead, we chose the parameter to match the computational effort on the datasets from which those trees were computed, which led us to using $\beta = -0.8$. On random and β-split trees, expected edge lengths are set after the tree generation by sampling from a uniform distribution on values in the set $\{1, 2, \ldots, r\}$, where r is a parameter that determines the overall rate of evolution. In the case of birth-death trees, we used both the same process and the edge lengths naturally generated by the birth-death process, deviated from ultrametricity and then scaled to fit the desired diameter.

We generate the true tree by turning each edge length into a corresponding number of iDLR evolutionary events on that edge. The events we consider under the iDLR model are insertions, duplications, losses, and inversions of genes or contiguous segments made of several genes—in particular, inserting, duplicating, or deleting a block of k consecutive genes has the same cost regardless of the value of k. We

forced the expected number of inserted and duplicated elements to equal the expected number of deleted elements, in order to keep genome sizes within a general range. We varied the percentage of inversions as a function of the total number of operations from 20% to 90%. The remaining percentages were split evenly between insertions/duplications and losses, with the balance of insertions and duplications tested at one quarter, one half, and three quarters. The expected Gaussian-distributed length of each operation filled a range of combinations from 5 to 30 genes. These are conditions similar to, but broader in scope than, those used in the experiments reported in Swenson *et al.*[81]

In all our simulations, we used initial (root) genomes of 1,000 genes. The resulting leaf genomes are large enough to retain phylogenetic information while exhibiting large-scale changes in structure. These sizes correspond to the smaller bacterial genomes and allow us to conclude that our results will extend naturally to all unichromosomal bacterial genomes.

The collections of gene orders produced by these simulations are then fed to our various competing algorithms. These are of two types: (i) algorithms running on the full gene orders, namely NJ and our new LP-based algorithm; and (ii) algorithms running on equalized gene contents, which include NJ again (running on the inversion distance matrix produced by GRAPPA), GRAPPA [59], and MGR [88]. Gene contents are equalized by removing genes from families with more than one gene, then keeping only singleton genes common to all genomes. On some of these datasets, the equalized gene content is minuscule—with high rates of evolution, the number of genes shared by all 12 taxa is occasionally in the single digits, obviously leading to serious inaccuracies on the part of reconstruction algorithms. We collect the data (including running times, the actual trees, internal inferred gene orders, inferred edge lengths, etc.) and compute basic measures, particularly the Robinson-Foulds[64] distance from the true tree—the most common error measure in phylogenetic reconstruction.

Results and Discussion

We ran collections of 100 datasets of 10 to 13 genomes, each of 1,000 genes, under various models of tree generation and various parameters of the iDLR model. We used birth-death, random, and β-split (with $\beta = -0.8$) models, with evolutionary diameters (the length of the longest path, as measured in terms of evolutionary operations, in the true tree) of 200, 400, 500, and 800 operations. (We ran tests with diameters of 800, but noted that most resulting instances exhibited strong saturation—that is, that many of the true edge lengths were significantly larger than the edit distances between the genomes at the ends of the edge; since no reconstruction method can do well in the presence of strong saturation, we did not pursue diameters larger than 800.) For each tree returned, we measured its RF error rate (the percent of edges in error with respect to the true tree) and then averaged the ratios over the set of test instances for each fixed parameter. We computed the ratio of the RF rate for our approach with that for NJ on full genomic distances and with those for the three approaches with equalized gene contents, binning the results into one "losing" bin (the other method did better), one bin of ties, and 5 bins of winners, according to the amount of improvement. Not all 100 instances are included in these averages, because some instances had equalized gene contents of just 2 or 3 genes and could not be run with GRAPPA.

We present below a few snapshots of our results. Table 3.1 shows the results of using full genomic distances for β-split trees on datasets of diameters 200, 400, and 500, using 80% inversions. In this case, no difference was found between the results returned by our LP-based method and those returned by NJ using full genomic distances. The average RF error rate for MGR was 23% for diameter 200, 32% for diameter 400, and 42% for diameter 500. As simple a method as NJ handily beats existing methods that must rely on equalized gene contents, often by large factors (e.g., factors of 4 or more in 26% of the cases with diameter 200 with respect to MGR). The reduction in error rate was sufficient in many cases to turn unacceptable results (with error rates well in excess of 10%) into acceptable ones.

Experience with sequence data leads us to expect that an MP method, should do better than NJ when the diameter and deviation from ultrametricity get large. Our LP-based approach is a hybrid: unlike an

Dataset	NJ			GRAPPA			MGR		
200	16-4-25-1-0	50	4	14-0-11-4-0	1	3	26-6-21-4-1	36	6
400	4-0-5-4-0	23	0	3-0-6-1-0	0	0	5-1-7-6-12	1	4
500	5-5-5-8-0	69	8	11-2-14-17-15	18	23	17-7-14-17-14	24	7
	w	**t**	**l**	**w**	**t**	**l**	**w**	**t**	**l**

Table 3.1: The accuracy for NJ on full genomic distances and for three evolutionary diameters compared to three methods on equalized gene contents. Column triples show wins, ties, and losses, in percent. Quintiles in the winning columns denote error reductions by factors larger than 4, 3, 2, 1.5, and 1.

MP method, it does not reconstruct ancestral labels, but like an MP method, it attempts to minimize the total length of the tree; thus it should at least occasionally outperform NJ. We tested this hypothesis on random trees and birth-death trees where, in both cases, we generated edge lengths by uniform sampling from the set $\{1, 2, \ldots, r\}$, for values of r ranging from 20 to 100, still using 80% inversions. Tables 3.2 and 3.3 present the results, this time limited to the reference MGR and to the two methods using full genomic data.

	20	40	60	80	100
LP	0.9	8.0	7.8	6.0	26.0
NJ	0.5	8.5	8.7	9.5	25.5
MGR	11.3	31.8	34.0	35.0	49.0

Table 3.2: Error rates, in percent, on random trees for the two approaches using full genomic data and for MGR on equalized gene contents.

	20	40	60	80	100
LP	0.2	8.5	7.6	5.7	19.4
NJ	1.4	9.0	8.5	8.0	18.0
MGR	9.7	31.7	31.8	33.7	51.4

Table 3.3: Error rates, in percent, on birth-death trees for the two approaches using full genomic data and for MGR on equalized gene contents.

Both tables show gains for the LP-based method over simple NJ as evolutionary rates increase, until both methods start failing at $r = 100$. Note that the accuracy gains over MGR are consistently very high.

Keeping the proportions of inversions to 80%, however, is neither very realistic, as gene duplications and losses are presumably more frequent in nature than rearrangements, nor very challenging, as, given a bounded set of possible gene choices, duplications and losses will saturate sooner than inversions. The experiments of Swenson *et al.*[81] did not test low percentages of inversions, so we ran sets of tests with 20% inversions only, keeping all other relative percentages of events identical. Table 3.4 shows these results. We were pleased, and somewhat surprised, to observe actual improvements in the quality of trees for rates up to $r = 40$; the threshold effect to $r = 60$ corresponds to a type of saturation caused by too many insertions and deletions. (Approaches with equalized gene contents are not reported, since they failed completely, as expected.)

Finally, we reproduced the results of Earnest-DeYoung[31] on the dataset of 13 bacteria, with genome sizes ranging from 1,000 to over 5,000 genes and gene families of up to 70 members, this time without any special preprocessing, and using our LP-based approach rather than NJ. Once again the resulting phylogeny is one SPR (subtree) move away from that of Lerat et al. The large disparity in

	20	40	60	80
LP	3.8	3.0	21.0	37.8
NJ	3.1	4.9	18.9	33.7

Table 3.4: Error rates, in percent, on birth-death trees with only 20% inversions.

gene content between species in this dataset was handled automatically, for the first time for this dataset (or, indeed, for any other set of cellular genomes).

3.4.8 Conclusion and Future Directions

We have outlined a method that accurately computes tree distances (true evolutionary distances) under the full range of evolutionary operations between two arbitrary sequences. Our experimental results indicate that the accuracy is excellent up to saturation, which is reached remarkably late—for instance, with sequences of roughly 800 genes, our distance computation remains highly accurate up to 200–250 evolutionary events. Indeed, these distances are accurate enough that the simple neighbor-joining method applied to distance matrices computed with our algorithm reconstructs trees with high accuracy. These findings open up the possibility of reconstructing phylogenies from whole-genome nuclear data, as opposed to the organellar data that have been used so far. We've shown that the more sophisticated LP method can utilize our distances better than the simple neighbor-joining procedure.

While our experiments show that our distance computation is accurate, the accompanying scenario of evolutionary events is only one of many possible sequences (it uses a "canonical form" [54]); hence our level of confidence in the correctness of reconstructed ancestral sequences is low. In order to reconstruct good ancestral sequences, we will need additional biological information, such as boundary constraints (centromere, origin of replication, etc.), length distributions, and sequence data around each gene. Unfortunately, direct comparisons between the method of Chen *et al.* [26] and those of this section are hard due to the fact that the Chen algorithm takes nucleotide sequences as input. The second half of the algorithm works on two gene sequences where the number of occurrences of a particular gene in each genome is equal, but other than that their method is actually quite similar to ours. No study has been done to discern whether the minute differences in our methods make any difference in distance estimation or duplicate assignment.

3.5 Towards a Practical Solution to the One to Many Duplicate Assignment (OtMDA) Problem

(Work in this section was joint work with Nick Pattengale)

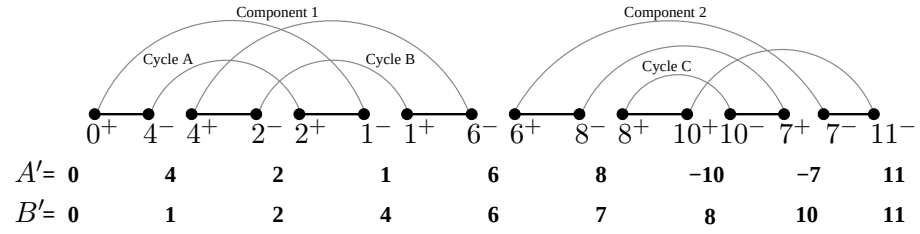
Whereas many of the results earlier in this chapter showed that we can get close to the true pairwise distance in the presence of duplicate and missing genes, in this section we show that some instances of OtMCM can be solved optimally. If the particular optimal solution to OtMCM has no hurdles in it — a fact that we can check in linear time [9] — then we know we also have a solution for OtMDA. Fortunately, the results of Section 2 suggest that this would likely be the case, making a minimal solution for OtMCM a minimal solution for OtMDA. We conclude the section by giving a framework for approximating OtMCM.

3.5.1 The Generalized Breakpoint Graph

We have seen in Section 1.2.1 that the basic structure describing a pair of sequences with no duplicates and equal gene content is the *breakpoint graph* (actually a multigraph). For this section, however, gene families need not be singletons, so we generalize the construction to include *only* singleton gene families as follows. Let $BG_{A,B}$ denote the breakpoint graph for sequences A and B. As with the normal breakpoint graph, each singleton gene g in A becomes a pair of vertices, g^- and g^+ (the "negative" and "positive" terminals); however, we leave out the gene families with multiple members, since only the singletons have a readily usable structure. We need to accommodate gaps left in the sequence where duplicate genes exist in A. Call the versions of A and B without multi-gene families A' and B' respectively. We add an edge (a *desire* edge, in the charming terminology of [69]) (x^-, y^+) for each singleton x and y, whenever x occurs immediately to the left of y in B'. We add a *reality* edge (also known elsewhere as a black edge), (x^p, y^q) if x is the element to the left of y in A' and we have either $p = q$ if x and y have different parities (in A', naturally) or $p \neq q$ if x and y have the same parity. Thus desire edges trace the (re-)ordering of A that we need to achieve to match B, while reality edges trace the given ordering of A. Figure 3.15 illustrates the construction.

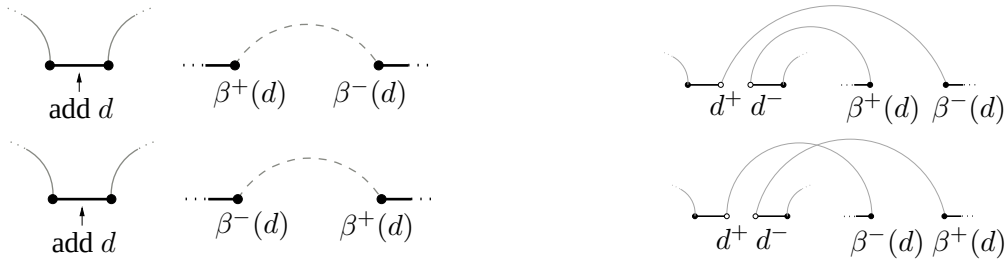
$A =$ (4 -3 2 3 1 6 9 3 8 -10 -7 9)

(a) the genome A



(b) the breakpoint graph $BG_{A,B}$

Figure 3.15: A genome A and its associated breakpoint graph $BG_{A,B}$ (with respect to the identity permutation B) after genes from families with duplicates (3 and 9) are removed; desire edges are shown in gray, reality edges in black.



(a) a subgraph of $BG_{A,B}$ before adding d; the dashed line is the desire edge that will be split

(b) the two possibilities after adding d

Figure 3.16: Adding an element d to a breakpoint graph.

3.5.2 The Consequences of An Assignment

Our job of assigning duplicates may be compared to that of reshelving books in a library with unlabeled shelves. Each book has a proper location on a shelf and multiple copies of a book must be shelved together. A librarian can proceed by first removing misshelved books and then identifying the appropriate location of each book based on the context of the books that remain in their correct spot.

In our problem each multi-gene family has been removed from the ordering, leaving a structure of cycles defined by singleton genes. We call each gene in a multi-gene family of B a *candidate*, since it is one of the choices for a duplicate assignment to a corresponding gene in A. Like each book in the library, each candidate has a location between two remaining elements in B'; each family, like each group of book copies, contains candidates that all share the same destination (when sorted) between elements of A. For each candidate d, denote by $\beta^+(d)$ the positive terminal of the next smaller (in value) element in $BG_{A,B}$ and by $\beta^-(d)$ the negative terminal of the next larger element. We call these vertices the *bookends* of d and the cycle on which they reside the *shelf* of d. For instance, in Figure 3.15, the bookends for the family of gene 3 (a family of 3 members) are 2^+ and 4^- and therefore the shelf for the family of 3s is cycle A. Although the definition of bookends applies equally well to singletons, we are only interested in bookends for candidates: bookends are part of the breakpoint graph, but candidates are not, since multi-gene families do not appear in the breakpoint graph.

Once we have chosen a candidate, the candidate and its matching gene in A effectively form a singleton gene family, so we can add the candidate to the breakpoint graph. The consequences of that choice are summarized in the following easy lemma, which underlies many of our results.

Lemma 3.5.1. *When a candidate d is chosen, exactly two edges are affected: the reality edge that spans the location where d is added and the edge between its bookends.*

Proof. Refer to Figure 3.16. Adding d to $BG_{A,B}$ splits the reality edge that spans the location where d is added, creating two new endpoints d^+ and d^-, as well as splitting the desire edge that links $\beta^+(d)$ and $\beta^-(d)$ to meet each of d^+ and d^-. □

We say that a candidate d is added *on-cycle* if, once added, it lies on its own shelf; otherwise it is added *off-cycle*. The following is an immediate consequence of Lemma 3.5.1.

Lemma 3.5.2. *When a candidate is added off-cycle, two cycles get joined.*

3.5.3 The Cycle Maximization Problem

We have formulated duplicate assignment as two optimization problems (OtMDA or MtMDA): choose an assignment of duplicates that maximizes the number of cycles in the resulting breakpoint graph (that

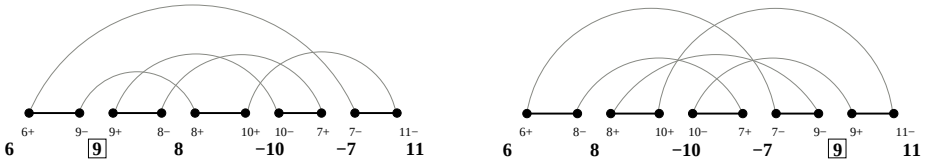


Figure 3.17: Breakpoint graphs corresponding to choice of one of two candidates for gene 9 on cycle C.

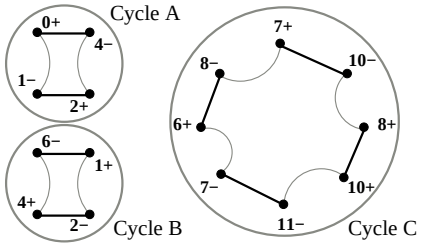


Figure 3.18: The breakpoint graph of Figure 3.15 inscribed in three circles (cycle D is not shown).

is, $BG_{A,B}$ to which the chosen candidates have been added). Note that the order in which the chosen candidates are added does not affect the structure of the resulting breakpoint graph.

Consider cycle C in Figure 3.15. This cycle is associated with the subsequence $(6, 9, 8, -10, -7, 9, 11)$, which contains two occurrences of gene 9; thus we must choose which of these two occurrences to call the match of gene 9 in B. Figure 3.17 shows the augmented breakpoint graphs resulting from each choice of candidate. The graph on the left, where we chose the candidate between 6 and 8, has one more cycle than the graph on the right, where we chose the candidate between -7 and 11, and is thus the better choice.

As we've seen, the choice of a candidate is advantageously viewed on a breakpoint graph inscribed in a series of circles, one for each cycle in the graph. We embed each cycle of $BG_{A,B}$ in a circle by choosing any start vertex and then following the cycle. Figure 3.18 shows three of the four cycles of Figure 3.15 inscribed in three circles. Returning to the two possible duplicate assignments shown in Figure 3.17, we can look at the inscribed versions of these graphs, as illustrated in Figure 3.19(a). Choosing candidates adds edges across the circle, edges that may cross each other, depending on the parity of the candidates and the locations of their bookends. The effects on the graph can be represented in just one *circle-drawing*, as shown in Figure 3.19(b). In this representation, we denote the two choices by drawing two curved line segments, both originating on the perimeter between the bookends 10^- and 8^+ and each ending between the two terminals of the corresponding candidate. Choosing the candidate between 6^+ and 8^- gives rise to desire edges that do not cross in the inscribed representation; we represent such choices with solid lines. The other candidate, between 7^- and 11^-, does give rise to crossing desire edges; we represent such choices with dashed lines.

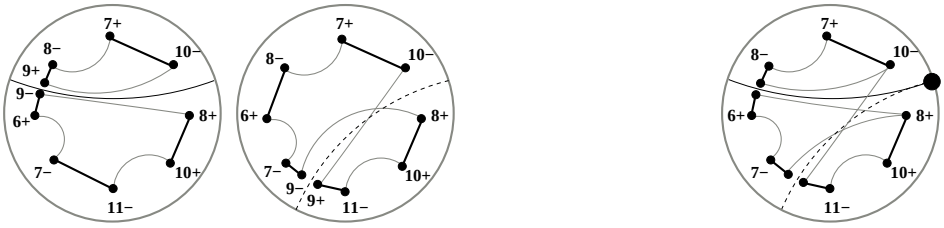
These curved lines represent assignment *operations*; we will call an operation represented by a solid line a *straight* operation (because it does not introduce crossings) and one represented by a dashed line a *cross* operation. The collection of all operations that share an endpoint represents all members of a gene family from A, so we also call it a *family* and call its common endpoint (between the bookends and represented by a solid disk on the periphery of the circle in the figures) the *family home*. We can now state the three constraints for our optimization problem:

1: Each family home is a distinct point on the circle.

2: The family home is not the endpoint of any operation not in that family.

3: The other endpoint of each operation is unique to that operation.

The objective to be maximized is the number of cycles. Figure 3.20 shows the operations for each of the



(a) the graphs of Figure 3.17 inscribed in circles

(b) the two choices of part (a) superimposed

Figure 3.19: How the cycle splitting problem can be inscribed in a single circle.

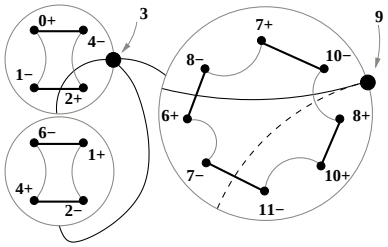


Figure 3.20: The operations that represent the gene families for our running example.

gene families from our running example. Operations that cross cycles are off-cycle and therefore will join cycles.

Figure 3.21 shows a single cycle and its operations for the simplified ("one-to-many") case where B has only singletons and for the general ("many-to-many") case where both A and B have multi-gene families. (The case where two multi-gene families have the same bookends can be handled because the relative location of the bookends does not change.) In the general case we have multiple homes per family, with one additional constraint:

4: Each home in the same family must connect to all of the same endpoints.

The problem thus becomes picking as many operations as there are homes per family such that the cycle count is maximized. The only additional complication is that applying an operation removes that operation from consideration in all other homes for its family (as required by the fourth constraint).

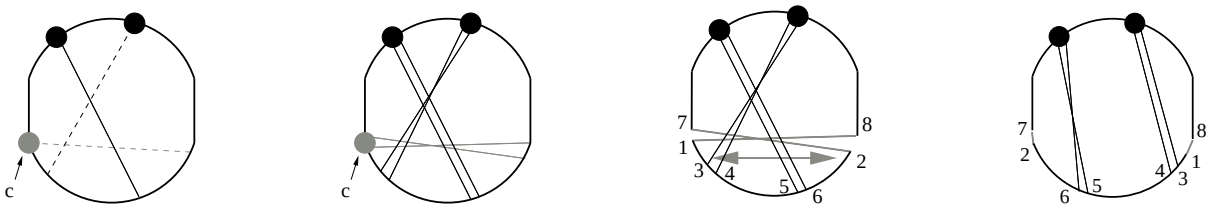
Straight and cross operations display a form of duality that suggests we can focus on straight operations alone.

Theorem 3.5.3. *Applying a cross operation c converts all operations that intersect c (call the set of such operations I) to their complement—crosses are replaced by straights and straights by crosses.*

(a) a one-to-many instance

(b) a many-to-many instance

Figure 3.21: Examples of circle-drawings for the simplified case (left) and the general one (right).



(a) drawn on the circle as usual

(b) zooming in on each operation

(c) after applying cross operation c

(d) result redrawn on the circle

Figure 3.22: Illustration for Theorem 3.5.3. Labels for the points along the circle are numbered.

(a) before application

(b) after application

Figure 3.23: Applying cross operation c.

Furthermore, for any two operations in I, if they intersected before applying c, then they no longer do after applying c, and vice versa.

Proof. We sketch the proof graphically, using Figure 3.22, a typical situation where three operations, two of which are crosses, one a straight, overlap each other. The cross operation shown in parts (a) and (b) twists, but does not break the cycle, as shown in part (c). If we redraw the cycle inscribed neatly in a circle, we find we must reverse the indices on half of the cycle; Figure 3.22(c) shows the result after reversing indices on the bottom half of the cycle. Intersecting operations no longer intersect and the identities of the operations have been inverted. □

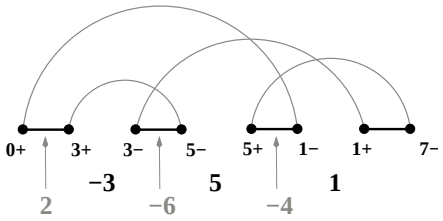
Figure 3.23 shows the implications of Theorem 3.5.3 in a more complicated setting.

3.5.4 Buried Operations

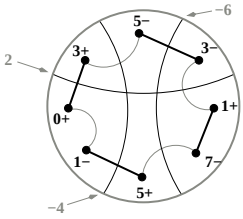
An operation makes no contribution to the cycle count of a complete assignment if the two new desire edges it creates lie on the same cycle. In Figure 3.24, the choices of candidates for the gene families are indicated in the breakpoint graph on the left and shown as operations in the inscribed representation on the right.

In Figure 3.25, we show again the three operations depicted in Figure 3.24(b), but this time only the three operations and the resulting two cycles are shown. Note the operation corresponding to gene family 2 (shown as a heavy curve): the curved edge is bounded on each side by the same cycle; we say that such an operation is *buried*. Since the two desire edges created by this operation lie on the same cycle, the operation does not increase the number of cycles (in fact it actually reduces the number of cycles, which stood, in this particular example, at 3 after operations -6 and -4).

Theorem 3.5.4. *If a duplicate assignment creates a total of b buried edges, then the number of cycles is bounded by $a - b + 1$, where a is the number of cycles present in the breakpoint graph induced by the shared singleton genes plus the total number of duplicate assignments to be made.*



(a) the breakpoint graph



(b) the inscribed version of $BG_{A,B}$

Figure 3.24: An example with A = (2 -3 4 -6 5 -4 -2 6 1). Chosen duplicates are shown in grey.

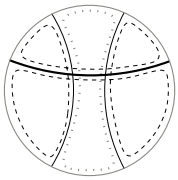


Figure 3.25: The cycle and the operations; operation "2" (the heavy curve) is buried.

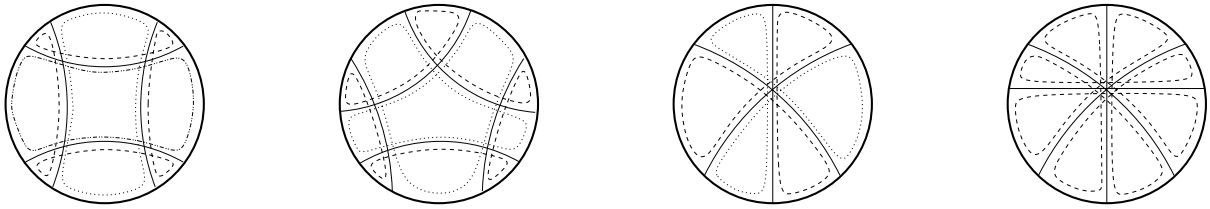
Proof. The number of cycles cannot exceed $n + 1$, since each duplicate assignment can give rise to at most one new cycle. Consider the effect on the breakpoint graph of choosing an operation: a single desire edge d is replaced with two desire edges d'_1 and d'_2, and a single reality edge r is replaced with two reality edges r'_1 and r'_2. By construction, d'_1 and d'_2 each inherit one of the original endpoints of d; similarly, r'_1 and r'_2 each inherit one of the original endpoints of r. By assumption, the chosen edge is buried, so that d'_1 and d'_2 lie on the same cycle; therefore so do all of the original endpoints of d and r. Thus all of the newly created edges must lie on a cycle that already existed. Since this is true of any buried operation, every one of the buried operations decreases by one the maximum number of attainable cycles. □

3.5.5 Chains and Stars

There exist two patters of straights that, while need not contain buried operations, nevertheless impose sharp bounds on the number of cycles. A k-*chain* (for $k \geq 3$) is an assignment in which k operations form a chain, that is, each chosen operation overlaps two of the other k, its predecessor and successor around the circle. Figure 3.26(a,b) illustrates k-chains. A k-*star* (for $k \geq 1$) is an assignment in which k operations form a clique (each overlaps every other). Figure 3.26(c,d) illustrates k-stars.

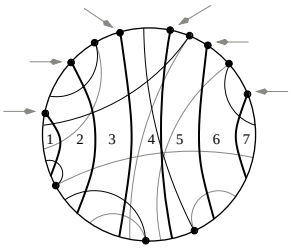
Remark 3.5.5. *For any integer $k \geq 1$ (but recall that k-chains are only defined for $k \geq 3$), we have:*

1. *a k-chain has no buried operations;*

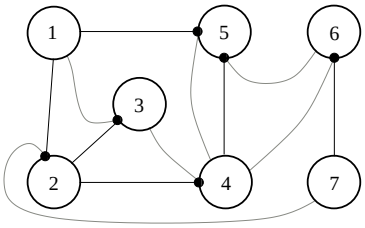


(a) a 4-chain *(b) a 5-chain* *(c) a 3-star* *(d) a 4-star*

Figure 3.26: Some examples of stars and chains.



(a) operations indicated by heavy lines (and arrows) are those chosen to produce the reduced form of part (b)



(b) the resulting reduced instance; heavy edges will produce an optimal solution to the reduced instance

Figure 3.27: Creating a reduced instance and solving it.

2. *in a k-chain with k odd, the cycle count is 2;*
3. *in a k-chain with k even, the cycle count is 3;*
4. *in a k-star with k even, every operation is buried and the cycle count is 1;*
5. *in a k-star with k odd, no operation is buried and the cycle count is 2.*

We conjecture that these two patterns, along with buried operations, describe all operations that may reduce the number of straights that do not create a new cycle.

3.5.6 Reduced Forms

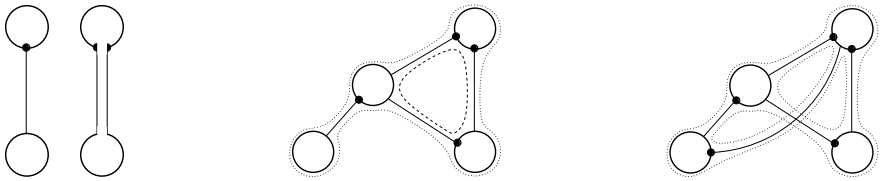
A serial assignment procedure could reach a state in which no operation remains that could split a cycle. We call such a state a *reduced form* of the instance. In a reduced form, an instance is composed of multiple cycles linked by the operations from the remaining families. This structure lends itself naturally to a graph representation; an analysis of this graph reveals conditions under which optimality can be verified.

Theorem 3.5.6. *After applying a maximal nonoverlapping set of straight operations M, remaining operations can only (by themselves) join two cycles.*

Proof. The application of a set of k nonoverlapping straights always yields k new cycles, each separated from the others by two adjacent operations or, in the case of an outermost cycle, by one operation that separates it from all others. Since M is maximal, every remaining operation from every family overlaps an element of M. Application of any $m \in M$, therefore, must span two of the new cycles, joining them into one. □

Figure 3.27(b) shows the reduced instance induced by applying each of the (straight) operations chosen in Figure 3.27(a). We are left with a reduced form that can be viewed as a graph where the vertices are the cycles created so far; but because that graph is embedded in the plane, the edges incident on a vertex are strictly ordered, in distinction to a normal graph.

We can now take advantage of graph properties such as planarity, cycles, and connected components. Because of the ordered nature of the edges incident upon a given vertex, planarity is somewhat specialized in our case: nonplanar edges can occur in simpler situations than in general graphs, as shown in Figure 3.28(c). Cycles again play a vital role in these new graphs. If we restrict our attention to planar graphs, we can look at the elementary cycles (those that delimit an inside face of the planar embedding) and obtain directly the value of an optimal solution. As shown in Figure 3.28, each connected component produces a cycle around its outer hull (one of the cycles for the outer face of the planar graph).



(a) the effect of applying an operation between two circles

(b) a reduced form: lines trace the cycles created by the operations

(c) adding a "nonplanar" operation to the reduced form from (b) joins the cycles

Figure 3.28: The effect of choosing operations on a reduced form.

(a) the solution embedded through a reduced form

(b) the circle-drawing of the solution

Figure 3.29: An optimal solution to the reduced instance in Figure 3.27

Each elementary cycle yields another cycle to its inside. Figure 3.28(c) shows how nonplanar edges can join these two cycles.

Theorem 3.5.7. *The number of cycles in a solution S to a planar reduced instance with m elementary cycles and cc connected components is $R(S) = m + cc$ (m is the cycle rank of S).*

Proof. This certainly holds for a reduced instance with no operations. Assume $R(S) = m + cc$ for a particular instance and solution, then look at the effect of adding another edge. If that edge links two previously disconnected components, then the cycles around the hulls of these components will get merged, removing a cycle and a connected component. If that edge links two connected components, then an elementary cycle will be created. Since the edge added is planar, we know that the same cycle runs past both endpoints of the operations and thus the operation will split it. □

It remains to relate results on reduced forms back to the original inscribed breakpoint graph formulation; we illustrate the process in Figure 3.29, where the left part shows the solution obtained on a reduced form and the right part shows the corresponding solution inscribed in the circle.

3.5.7 An Approximation Framework

We evaluate a solution S against an optimal solution O for a particular instance. Call $hpc(S)$ the number of breakpoint graph cycles created for a solution S so that the approximation ratio for the algorithm that creates solution S is $\frac{hpc(S)}{hpc(O)}$. Call the maximum set of non-overlapping operations (for the circle-drawing of the instance) in the solution $S^* \subseteq S$ and the same in the optimal $O^* \subseteq O$. We continue by comparing the score given for the reduced form induced by S^* and O^*.

Properties of Solutions

In this section we show that a solution with a maximum number of connected components (in a reduced form) will be no worse than half the optimal. Theorem 3.5.7 is helpful if a solution S happens to be

planar (in our peculiar way, where the edges incident to a particular vertex have a fixed ordering). We can turn the formula of Theorem 3.5.7 into an inequality where we ignore the effect (on the score) introduced by non-planar edges:

Theorem 3.5.8. *$cc \leq hpc(S) \leq m + cc$ where cc is the number of connected components in a solution S and m is the cycle rank of S.*

Proof. The lower bound comes from the fact that, as noted earlier, each component of the solution represents at least one cycle. Now we prove the upper bound:

Take the maximum cardinality planar subgraph P of S and the score of that subgraph $hpc(P) = m_P + cc_P$. Trivially, we know that $cc = cc_P$. Call the set of non-planar edges $N = S \setminus P$. Each edge in N will add one to the cycle rank so $m = m_P + |N|$. Each edge of N must span two elementary cycles in P. Take a maximum subset of those edges $N^* \subseteq N$ such that no two edges in N^* span the same two cycles. Each edge in N^* will join the two elementary cycles into one so we have $hpc(S) \leq cc+m_P-|N^*|+|N\setminus N^*|$. Since $m_P-|N^*|+|N\setminus N^*| \leq m_P+|N| = m$ we have $hpc(S) \leq cc+m$. □

The inequality in the above formula arises from the fact that only a subset of the edges in $S \setminus P$ can positively contribute to $hpc(P)$. This formula provides us a way to compare solutions without worrying if a solution is planar or not; at first glance it says that if we maximize the number of connected components in S we are within some m of the optimal. That m is at most $n-|O^*|$ where n is the number of families so any solution that maximizes the number of connected components would be no worse than $m = O(n)$ from the optimal. This does not appear to help since the distance between two permutations is $O(n)$.

However, it is well known that a graph with cycle rank m, v vertices, and e edges has $cc = v-e+m$ connected components. Thus we can find a version of Theorem 3.5.8 that suites us.

Corollary 3.5.9. $v - e + m \leq hpc(S) \leq v - e + 2m$

This is interesting because it relates m and $2m$. Indeed, if we were to find a solution with maximum $v-e+m$ we would be within half of the optimal solution. Apparently, the non-planar factor can detract at most m from a solution. We proceed to show that when cc is maximized to obtain a solution S, $m \leq cc$ and thus the optimal solution is at most double $hpc(S)\ldots$

Theorem 3.5.10. *For a solution S where $v_S-e_S+m_S$ (in the reduced form induced by S^*) is maximum and an optimal solution O (with v_O,e_O,m_O) it must be that $2hpc(S) \geq hpc(O)$, provided $v_O - e_O \geq 0$ or $v_S - e_S \geq 0, m_O \geq m_S$.*

Proof. By Corollary 3.5.9, we know that $hpc(S)$ is at least $v_S - e_S + m_S$, so we have $2(v_S - e_S + m_S) \leq 2hpc(S)$. Also by Corollary 3.5.9, we have $hpc(O) \leq v_O - e_O + 2m_O$, which (because when $v_S-e_S+m_S$ is maximum so is $2(v_S-e_S+m_S)$) is no greater than $2(v_S-e_S+m_S)$ provided we have $v_O-e_O \geq 0$ or $v_S-e_S \geq 0, m_O \geq m_S$. We conclude $hpc(O) \leq v_O-e_O+2m_O \leq 2(v_S-e_S+m_S) \leq 2hpc(S)$. □

3.5.8 Conclusion

We have described a graph-theoretical framework in which to represent and reason about duplicate assignments and their effect on the number of cycles present in the resulting breakpoint graph. We have given some foundational results about this framework, including several that point us directly to to algorithmic strategies for optimizing this assignment. We believe that this framework will lead to a characterization of the duplicate assignment problem as well as to the development of practical algorithmic solutions. We showed that this framework gives an avenue that could lead to an approximation algorithm for certain classes of instances of OtMCM and MtMCM. We will see in the next section that the work here also leads to NP-Hardness results.

3.6 NP-Hardness Proof for OtMCM, MtMCM, RDD, OtMRD, and MtMRD

We have seen that a choice of a duplicate has the effect of splitting and joining the cycles of the breakpoint graph; in order to minimize the distance, we choose duplicates so as to maximize the number of cycles. We show that One-to-Many Cycle Maximization (OtMCM) is NP-Hard by a reduction from a restricted version of 3-Dimensional Matching (called Triangle Matching). We conclude the section by showing how this powerful reduction extends to MtMCM, RDD, OtMRD, and MtMRD.

3.6.1 Triangle Matching

We pose a restricted version of the 3-Dimensional Matching (3DM) problem (called "Triangle Matching" or TriM) as a graph problem on colored vertices. Note that the input to TriM is restricted in our presentation to only chordal (triangulated) graphs; this restriction is imposed only for ease explanation since the two reductions that follow carry through even when the input is a general graph on colored vertices.

Input: A chordal (triangulated) graph $G = (V, E)$ such that $V = X \cup Y \cup Z$ (X,Y, and Z are the colored sets) and $X \cap Y = X \cap Z = Y \cap Z = \emptyset$.
Output: A set of triangles $\{(x, y, z) \mid (x, y), (y, z), (x, z) \in E \text{ for } x \in X, y \in Y, z \in Z\}$ such that every vertex exists in exactly one triangle.

The difference between TriM and 3DM is subtle; TriM is the version of 3DM suitable for drawing on a page. Figure 3.30 shows an instance of 3DM where an unintended triple is produced from three other triples when drawn on a page. Since TriM is a graph problem this case is built into its structure so there is no such thing as an "unintended triple".

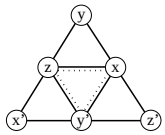


Figure 3.30: An instance of 3DM with triples $\{(y, z, x), (z, x', y'), (x, y', z')\}$ that can't be represented by a graph. The dotted triple (z, x, y') is an unintended byproduct of the other three.

Theorem 3.6.1. *TriM is NP-Hard.*

Proof. The standard reduction from 1in3SAT to 3DM (see [39]) can be directly applied to TriM so as to show it NP-Hard. □

3.6.2 Preliminaries

Section 3.5.3 describes the terminology and concepts that we use to reason about the cycle maximization problems. A consequence of Lemma 3.5.1 is that a cycle can be split into two if the orientation (sign) of the chosen candidate is correct. Without loss of generality – as we later see – we can consider only those instances that are a single cycle. Thus, the choice of a candidate is advantageously viewed on a graph inscribed on a circle; a problem instance of OtMCM can be given by a circle-drawing Figure 3.21(a).

Our reduction will rely heavily on the fact that we can create a permutation that yields a desired configuration of separate cycles. As depicted in Figure 3.31, we can include, in a constructed instance to OtMCM, single operation families (no choice is allowed) that start us off in a desired configuration. So each single operation family would be its own cycle-vertex. Note that operations that exist between cycle-vertices now link the cycle-vertices creating a situation where the two breakpoint graph cycles are

(a) the gray operations induce a particular instance

(b) the induced instance with three cycle-vertices

Figure 3.31: Creating a three cycle instance of OtMCM.

(a) length 3 cycles from TriM are (1,3,5), (1,4,5), (2,3,5), (2,3,6)

(b) a solution to k-OtMCM (dark edges are chosen candidates)

Figure 3.32: An instance of TriM converted to k-OtMCM.

joined into one. Figure 3.33 and Figure 3.28 give the reader a feel for the effect of linking cycle-vertices by edges.

3.6.3 TriM to OtMCM

We reduce TriM to OtMCM through the decision version of OtMCM called k-OtMCM. k-OtMCM asks the question: "can we find a solution for OtMCM that yields k cycles". k-OtMCM reduces to OtMCM because the number of cycles in a solution to OtMCM, of course, can be compared to k to obtain an answer to k-OtMCM. We assume that the number of vertices in a TriM instance is divisible by three because we can immediately return "no" if it is not.

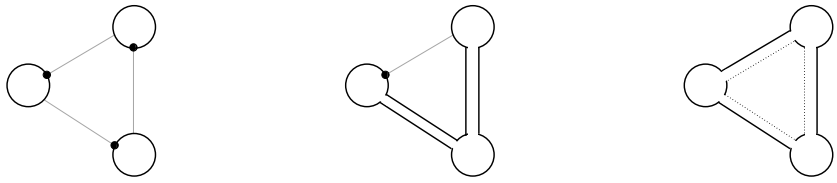
Setup

We say that a cycle-vertex A *links to* another cycle-vertex B if there exists a candidate from a family on A which connects to B. We convert an instance of TriM to an instance of k-OtMCM. For each $v \in V$ we create a separate cycle-vertex $c(v) \in C$ using the method described in Section 3.6.2.

Cycles are linked based on the edges in E. Each $c \in C$ has a single family associated with it where candidates from it will connect. For each edge $(a, b) \in E$ we create a candidate linking $c(a)$ to $c(b)$ where ($a \in X$ and $b \in Y$) or ($a \in Y$ and $b \in Z$) or ($a \in Z$ and $b \in X$). This construction results in a setup where all cycle-vertices representing elements of X link only to elements of Y, elements of Y link only to elements of Z, and elements of Z link only to elements of X. The possible configurations of solutions that satisfy k-OtMCM, as we will see, is thus very limited. Figure 3.32 shows an instance of TriM that has been converted to k-OtMCM and one choice of candidates that k-OtMCM might take.

A key notion is summarized in the following remark (illustrated in Figure 3.33)...

Remark 3.6.2. *Two cycle-vertices bridged by a candidate cannot create a new cycle. Three cycle-vertices linked in a triangle, however, combine to form a new cycle.*



(a) Three cycles to be joined by straight operations.

(b) One cycle (strong black) exists when two are joined.

(c) A new cycle (dotted) is created once all three are joined.

Figure 3.33: The effect of choosing candidates across cycles.

Limitations on Structure

As stated, the structure of feasible solutions that satisfy k-OtMCM is greatly limited by our construction. We explore these limitations here by viewing our instance of k-OtMCM as a graph where the cycle-vertices in C are the vertices and the edges are dictated by the candidates the obvious way. This is done so that we can use the terms *path*, *cycle*, and *connected component* in the expected manner on this meta graph (the word cycle no longer refers to the cycles in the HP graph unless explicitly noted).

The most important corollary of our construction deals with its inherent "directionality". When inspecting a solution to our k-OtMCM instance we can start at a cycle-vertex in C and follow the candidate from its associated family; from the cycle-vertex that we next reach we can follow its candidate and so forth. Without loss of generality we call this movement clock-wise (and movement in the opposite direction counter clock-wise); this matches the way we have drawn the example in Figure 3.32.

We restate Theorem 3.5.7...

Theorem 3.6.3. *The number of breakpoint graph cycles in a planar solution S with m elementary cycles and cc connected components is $hpc(S) = m + cc$ (m is the cycle rank of S).*

The following lemmata refer to properties of k-OtMCM instances that have been reduced from TriM....

Lemma 3.6.4. *Any clock-wise path on a connected component in a solution must terminate at a cycle.*

Proof. If this is not true then there exists a cycle-vertex that terminates a clock-wise path. This is a contradiction because every cycle-vertex has at least one candidate coming from it and there are a finite number of cycle-vertices ($|V|$ is finite). □

Lemma 3.6.5. *Every connected component in a solution must have exactly one cycle.*

Proof. A consequence of Lemma 3.6.4 is that every connected component must have at least one cycle. Assume that there is more than one cycle in a connected component: starting from a cycle-vertex on one cycle we must be able to follow a clock-wise path from it to another cycle. This is a contradiction, however, because the clock-wise degree of any node is 1. □

Lemma 3.6.6. *The length of all cycles in a solution must be a multiple of 3.*

Proof. By our construction a cycle-vertex for X only links to a cycle-vertex for Y, a cycle-vertex for Y only links to a cycle-vertex for Z, and a cycle-vertex for Z only links to a cycle-vertex for X. So every cycle is a multiple of 3. □

Lemma 3.6.7. *All feasible solutions to k-OtMCM must be planar.*

Proof. This is a direct consequence of Lemma 3.6.5. □

Mapping the Solution

We know from Lemma 3.6.5 and Lemma 3.6.6 that the candidate assignment given by k-OtMCM will be comprised of connected components with a single cycle that posses length that is a multiple of 3. We set $k = \frac{|V|}{3}2\ldots$

Lemma 3.6.8. *With $k = \frac{|V|}{3}2$, k-OtMCM can be satisfied if the candidate assignment is comprised of only length 3 cycles.*

Proof. From Lemmata 3.6.5 and 3.6.6 we know that there exists no component of size less than 3. Since every feasible solution to k-OtMCM is planar (Lemma 3.6.7), we know the number of breakpoint graph cycles can be calculated from the number of connected components and cycle rank by Theorem 3.6.3. By Lemma 3.6.5 that formula can be simplified, in our case, to be $hpc(S) = 2cc$. Since the number of splits created is related only to the number of connected components and the number of connected components will be maximized when they are all of minimum size, we know that the maximum number of splits occurs when a solution is comprised of only length 3 cycles. The Lemma follows immediately from the fact that the greatest number of length 3 cycles possible is $\frac{|V|}{3}$ (yielding $\frac{|V|}{3}2$ breakpoint graph cycles). □

Theorem 3.6.9. *OtMCM is NP-Hard.*

Proof. It is straightforward to see that by construction, and Lemma 3.6.8, k-OtMCM will give a solution with disjoint triangles of cycle-vertices if and only if there exists a partition of the vertices for TriM into disjoint triangles. As previously noted, k-OtMCM is NP-Hard implies OtMCM is NP-Hard. □

3.6.4 TriM to MtMCM, ERD, OtMRD, and MtMRD

It follows immediately from Theorem 3.6.9 that MtMCM is NP-Hard. The same reduction can be used to show OtMRD — and hence, RDD and MtMRD — to be NP-Hard as long as no hurdles are created in the breakpoint graph implied by a feasible solution to OtMCM. We show that any instance with a feasible solution containing cc cycles and some hurdles can be converted into an instance with cc cycles and no hurdle (hurdles can be detected in linear time).

Notice that our reduction used only straight operations. A straight associated to an element with a particular sign will be a cross when the element has the opposite sign (see Section 3.5.3). For a particular feasible solution to OtMCM that has bad components, we can simply change two straight operations on the cycle (we know there is one by Lemma 3.6.5) of each bad component to a cross. Since this entails flipping the sign of two elements in the permutation we know that each such component will now have two elements of opposite sign to the others and hence, will be good (see the definition of bad component in Section 2.1). The number of cycles will remain the same (imagine the analogue to Figure 3.33 that possesses two cross operations).

Chapter 4

Reconstructing Ancestors

Suppose that a set of extant species have evolved so that the phylogenetic relationship between these species can be represented by a binary tree. A toy example of one such phylogenetic tree is in Figure 4.1. The problem of ancestral reconstruction calls for us to label the internal nodes of this tree with the states of the genome just prior to each speciation event. In the traditional approach of Tesler and Pevzner [88], one of many so-called median permutations is taken to be the ancestor permutation. To this end, in Section 4.1 we introduce a fast heuristic to speed up and improve the median score of the most commonly used median algorithm. In Section 4.2 we introduce a new and powerful means to accurately compute ancestral permutations.

4.1 Noninterfering Inversions

(This is joint work with Jijun Tang and William Arndt)

Phylogeneticists have sought to exploit the advantages of gene-order data (no need for reconciliation of gene trees, very little saturation, existence of rare events that uniquely characterize some very old divergences, etc.), but have had to contend with the high computational complexity of working with such data. Of particular interest in a phylogenetic context is the problem of finding the median of three genomes, that is, finding a fourth genome that minimizes the sum of the pairwise distances between it and the three given genomes [67]. This problem, while being fairly easy for aligned sequence data, is NP-hard for gene-order data [23, 63]. Since phylogenetic reconstruction based on reconstructing ancestral states may need to compute such medians repeatedly, fast approximations or heuristics are usually needed, although exact methods have done well for small genomes (from organelles, for instance) [57, 70]. One such heuristic, implemented in the popular software MGR [88], attempts to find a longest sequence of inversions from one of the three given genomes that, at each step in the sequence, moves closer to the other two genomes. However, nothing is know about the theoretical behavior of this heuristic and no systematic experimental investigation of its usefulness has been conducted. Recently,

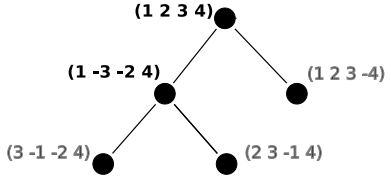


Figure 4.1: A phylogenetic tree. The ancestral (internal) node labels are bold.

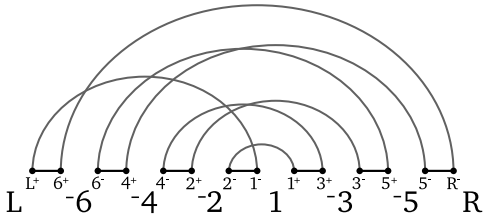


Figure 4.2: The breakpoint graph $G(\pi$ = (-6 -4 -2 1 -3 -5)).

Arndt and Tang [7] provided significant improvement on this heuristic by considering sets of *commuting* inversions, that is, inversions that can be arbitrarily reordered among themselves without affecting the end result.

In this section, we show that finding maximum cardinality sets of commuting inversions is equivalent to finding maximum independent sets on circle graphs and so can be done in low polynomial time—we give a simple algorithm for this purpose. We also shed light on the relationship between maximal sets of noninterfering inversions and independent sets on circle graphs. We further classify sets of commuting inversions into *interfering* and noninterfering inversions, where *noninterfering inversions* are commuting inversions that also make maximal progress (e.g., towards a median). Finally, we characterize the relationship of sets of noninterfering inversions to signatures and that of signatures to inversion medians.

For most of the section, we show how to analyze single permutations in terms of commuting and noninterfering inversions; in Section 4.1.4, we show how to extend the analysis to multiple permutations.

4.1.1 Definitions

In this section we use extensively the fundamental definitions from Section 1.2.

Commuting and Noninterfering Inversions

Depicted in Figure 4.2 is the breakpoint graph that we will use for our running example in this section. Cycle-splitting inversions on this graph are of particular interest to us because, in the absence of hurdles, they are the inversions that move π one inversion closer to the identity. A set of cycle-splitting inversions on a permutation π are *commuting* if and only if the application of them in any order yield the same permutation τ.

Definition 4.1.1. *A set of m inversions on π (with respect to τ) is* noninterfering *if and only if*

1. *the set is commuting; and*
2. *applying these inversions in any order moves π closer to τ by m inversions.*

Example 4.1.2. *For $\pi = (-6{-}4{-}21{-}3{-}5)$ a maximum cardinality set of commuting inversions is $\{\rho(1,1), \rho(1,4), \rho(1,5), \rho(1,6), \rho(2,3), \rho(3,3), \rho(4,4)\}$ while a maximum cardinality set of noninterfering inversions is $\{\rho(1,1), \rho(1,2), \rho(1,4), \rho(4,4)\}$.*

Edit Partial Orders and Inversion Signatures

We informally introduce some notions here that are used heavily in the Section 4.2. Recall that an *edit scenario* is a minimum-length sequence of inversions that turn π into, say, τ. The *edit partial order* (EPO), then, is the graph of all edit scenarios between π and τ; the permutations are vertices and edges link those permutations one inversion away from each other. The intersection of all EPOs from a set of permutations P to permutation τ is the *signature graph* and any vertex (permutation) in this graph is an *inversion signature*. See Definition 4.2.1 for a more formal treatment.

So a set of noninterfering inversions of size m constitutes a subgraph of the signature graph of size $\sum_{i=0}^{m} \binom{m}{i} = 2^m$. This motivates our use of noninterfering inversions for fast computation of inversion signatures; experiments in Section 4.2 confirm that this method is often faster that other known methods.

Circle Graphs and Permutation Graphs

When a set of chords is drawn so that each endpoint of the chord lies on a circle we have a *chord model* of a circle graph. The *circle graph* represents the intersection of these chords where each vertex corresponds to a chord and each edge corresponds to intersecting chords [36]. For a permutation we can define a *permutation graph* as follows. Each vertex is an element of the permutation and an edge (u, v) exists if and only if $v > u$ and v appears to the left of u in the permutation [37]. It is simple to see that a permutation graph is a circle graph.

4.1.2 Maximum Sets of Commuting Inversions

We now show how to find a maximum cardinality set of commuting inversions efficiently, omitting proofs due to space limitations. We can interpret the indices of an inversion to be indices of an interval on a line. Two intervals *overlap* if and only if they are disjoint or if one is contained inside the other, and two intervals that share the same endpoint do not overlap. In this way each oriented inversion of π could be mapped to an interval yielding a set of intervals, some of which overlap and some of which may meet at an endpoint.

Lemma 4.1.3. *A set C of inversions commute if and only if no two inversions from C overlap.*

Thus, we have a set of intervals that when projected onto a circle yield a chord model of a circle graph [40]. Call this circle graph G_C. See Fig. 4.4(a) for an example of such a graph. It is clear that a maximum independent set of G_C corresponds exactly to a maximum independent set of commuting inversions. With the use of the $O(n)$ algorithms by Bader et al. [8] to build the HP-graph and the $O(n^2)$ algorithm of Valiente [89] for maximum independent set on a circle graph, we get the following theorem.

Theorem 4.1.4. *A maximum cardinality set of commuting inversions can be found in $O(n^2)$ steps.*

4.1.3 Maximum Sets of Noninterfering Inversions

In this section we show how to relate the problem of finding a set of noninterfering inversions to finding an independent set on the union of two circle graphs.

Since a set of noninterfering inversions is also a set of commuting inversions, the constraints of G_C (from Section 4.1.2) will have to be satisfied. Additional constraints must be introduced to ensure that the set of commuting inversions that are picked also sort the permutation, call the graph representing these constraints G_S. We will see that these intricate interactions can also be represented by a circle graph; first this is shown for a component that can be represented by a single cycle in the breakpoint graph and then is generalized to any permutation.

Single Cycle Components

One important property of commuting inversions is that the application of one inversion can not disturb the orientation of an inversion it commutes with.

Lemma 4.1.5. *Given mutually commutative oriented inversions $\rho(i, j)$ and $\sigma(k, l)$, the application of (without loss of generality) ρ will either*

1. *make σ span two different cycles or*

2. leave σ oriented.

Proof. Call r and s the reality edges being acted upon by σ. At least one of r or s will remain intact after the application of ρ, say it is r. At least one of the vertices incident to s must remain intact, call it v. There is a path P from v to a u incident to r that does not include r. Note that the adjacencies of v and u are not affected by ρ and that, because σ is oriented, if v is on some side of s then u is on the same side of r. But ρ can only remove a subpath of the cycle when creating another cycle. Because ρ and σ commute, whether the removed subpath is also a subpath of P or not, u and v will remain on the same sides of their respective reality edges, thus leaving the inversion σ oriented. □

Each oriented inversion will split the cycle into two by swapping the affected vertices of the desire edges being acted upon. Thus, when we embed the cycle on a circle we can represent the action of an inversion as a cord with its endpoints on those desire edges. For two inversions that intersect and act upon a disjoint set of desire edges we know that applying one of them will put the reality edges acted upon by the other on different cycles; so in this case intersecting chords represent inversions that interfere.

Finding the interactions between inversions that share a reality edge takes more care however. More specifically, consider the set of inversions that all share a reality edge as an endpoint and share the same desire edge. For example the set of inversions that share reality edge $(2^-, 1^-)$ is $\{\rho(2,3), \rho(3,3), \rho(4,4), \rho(4,5), \rho(4,6)\}$, which can be partitioned into inversions that share edge $(2^-, 1^+)$ $\{\rho(2,3), \rho(3,3)\}$ and those that share $(1^-, \mathrm{L}^+)$ $\{\rho(4,4), \rho(4,5), \rho(4,6)\}$.

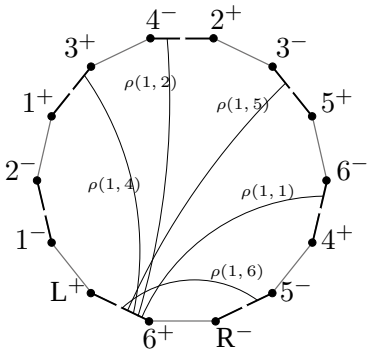
The following lemma describes the structure of the interference between those that share a desire edge. First, let us order such a set I in two ways. Call the ordering $\alpha : I \mapsto \mathbb{N}$ that which numbers inversions from shortest to longest. As stated, the action of an inversion on $G(\cdot)$ is to swap endpoints of the two desire edges being acted upon. Because they share an acted upon reality and desire edge we can look at the shared vertex v that will be affected by all inversions in I. The ordering $\beta : I \mapsto \mathbb{N}$ is that which numbers inversions by the order in which we visit its non-v endpoint, starting at the common reality edge and proceeding through v.

Lemma 4.1.6. *Take inversions $i, j \in I$. i interferes with j if and only if $\alpha(i) > \alpha(j)$ and $\beta(i) < \beta(j)$.*
(In other words, an inversion interferes with all shorter inversions that appear after it on the cycle.)

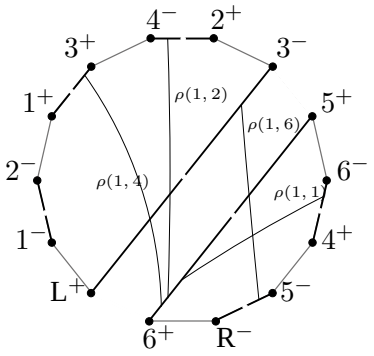
Proof. Recall that v is the shared vertex that will be affected by all inversions in I. For an inversion $i \in I$ and any $j \in \{k \mid k \in I \setminus \{i\} \text{ and } \alpha(i) > \alpha(k)\}$ with endpoints v and u respectively, we know that i interferes with j if and only if u ends up on a different cycle than v after applying i. If we follow the cycle in the same order used to build β, the reality edges we visit before encountering u are those that will be remain on the cycle with v when it is attached by the new reality edge. So those inversions that act upon such reality edges will remain oriented, and they are exactly those j that have $\beta(j) < \beta(i)$. The others will respect $\beta(i) < \beta(j)$. □

Example 4.1.7. *Fig. 4.3(a) shows the graph from Fig. 4.2 embedded on a circle. α imposes the ordering on all inversions that share desire edge $(6^+, R^-)$ so that $\alpha(\rho(1,1)) < \alpha(\rho(1,2)) < \alpha(\rho(1,4)) < \alpha(\rho(1,5)) < \alpha(\rho(1,6))$. We also have $\beta(\rho(1,6)) < \beta(\rho(1,1)) < \beta(\rho(1,5)) < \beta(\rho(1,2)) < \beta(\rho(1,4))$. So for $\rho(1,5)$ we have $\alpha(\rho(1,5)) > \alpha(\rho(1,4)) > \alpha(\rho(1,2))$, as well as $\beta(\rho(1,5)) < \beta(\rho(1,2)) < \beta(\rho(1,4))$, which tells us that $\rho(1,5)$ interferes with $\rho(1,2)$ and $\rho(1,4)$. Further, $\alpha(\rho(1,5)) < \alpha(\rho(1,6))$ and $\beta(\rho(1,5)) > \beta(\rho(1,6))$ shows that $\rho(1,5)$ interferes with $\rho(1,6)$. Fig. 4.3(b) shows the result of applying inversion $\rho(1,5)$ on the graph.*

Corollary 4.1.8. *The interference relationship between all inversions that act on the same desire edge can be represented by a permutation graph.*



(a) Chords represent inversions that will effect the desire edge $(6^+, \mathrm{R}^-)$.



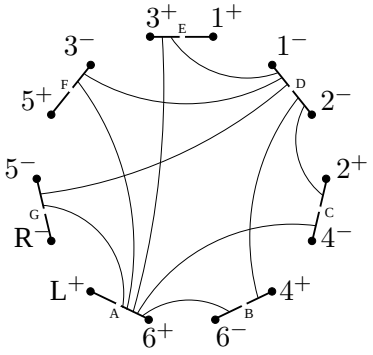
(b) After applying inversion $\rho(1, 5)$.

Figure 4.3: $G(\pi$ = (-6 -4 -2 1 -3 -5)) embedded on a circle. We see the affect that inversion $\rho(1, 5)$ has on those inversions acting upon the same desire edge; $\rho(1, 5)$ interferes with $\rho(1, 2)$, $\rho(1, 4)$, and $\rho(1, 6)$ but not $\rho(1, 1)$.

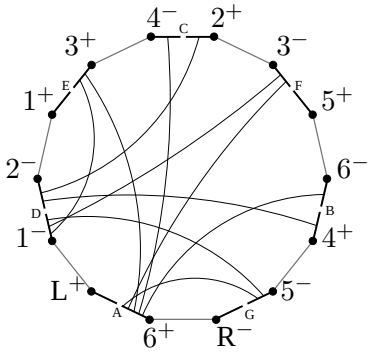
Theorem 4.1.9. *G_S can be represented by a circle graph.*

Proof. If two inversions both act an a reality edge then apply Corollary 4.1.8. Otherwise, embed the cycle on a circle and notice that the effect of an inversion is to split the circle (see Fig. 4.3). Therefore, a chord model representing the interference between two inversions that don't share a reality edge is obtained by drawing a chord for each inversion between the reality edges it acts upon. □

Fig. 4.4 shows the two circle graphs that represent the constraints of the HP-graph from Fig. 4.2. In this case, G_C is a subgraph of G_S so $G_C \cup G_S$ is a circle graph. A maximum cardinality set of noninterfering inversion would be represented by the set of chords $\{AB, AC, AE, DE\}$ (matching that from Example 4.1.2).



(a) The chord model for G_C.



(b) The chord model for G_S.

Figure 4.4: The chord models for circle graphs representing the constraints on $G(\pi$ = (-6 -4 -2 1 -3 -5)).

The union of two circle graphs, however, does not necessarily yield a circle graph. We handle this by decomposing the problem into computationally easy-to-handle and hard-to-handle subproblems by using the first of two phases from the polynomial time circle graph recognition algorithm of Bouchet [16, 73]. This first phase repeatedly decomposes the graph by the *join decomposition*. This is done by finding a partition on the vertices V_1 and V_2 ($|V_1| \geq 2$ and $|V_2| \geq 2$) so that the set of all edges between V_1 and V_2 form a complete bipartite graph. Call the sets of vertices that compose this complete bipartite graph $V_{1c} \subseteq V_1$ and $V_{2c} \subseteq V_2$. This subgraph is then replaced by the two graphs induced by taking only vertices in V_1 and V_2, and adding a marker vertex to each graph connected to only V_{1c} and V_{2c} respectively.

Once no such decomposition exists (i.e. a subgraph is prime) a chord model is found for each subgraph in the second phase. If every prime subgraph yields a chord model, then we can apply the quadratic algorithm of Valiente[89] to find the maximum independent set of the circle graph. If only some subgraphs yield a chord model, we can handle those independently with the same algorithm. The computationally hard-to-handle subgraphs are those that do not yield a chord model. It is on these subgraphs that we are forced to run a general algorithm for maximum independent set. Call this algorithm $MIS(\cdot)$. Fig. 4.5 shows how a set of vertices is partitioned into connected components $V_1 = V_{1a} \cup V_{1b} \cup V_{1c}$ and $V_2 = V_{2a} \cup V_{2b} \cup V_{2c}$ where V_{1a}, V_{2a}, V_{1b}, and V_{2b} are possibly empty sets. In our setting, the

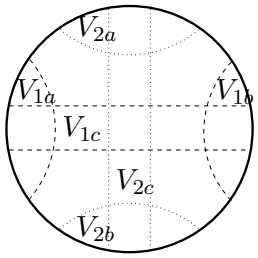


Figure 4.5: What the chord model of a join decomposition would look like if such a chord model exists.

sets V_{1a}, V_{1b}, and V_{1c} (resp. V_{2a}, V_{2b}, and V_{2c}) may not actually yield chord models, but the representation of Fig. 4.5 is instructive in seeing how the independent sets of such a decomposition interact with each other. Now when composing solutions of independent sets on hard-to-handle subgraphs we must consider two possibilities: either 1) vertices from V_{1c} and V_{2c} are used for $MIS(V_1)$ and $MIS(V_2)$ respectively, or 2) vertices from none or only one of the two are used. For the later case all vertices from both independent sets will be in the independent set for $G_S \cap G_C$. In the former case we can use the vertices from V_{1c} or from V_{2c} but not both, so we recursively test $MIS(V_{1a} \cup V_{1b}) + MIS(V_2)$ and $MIS(V_{2a} \cup V_{2b}) + MIS(V_1)$ and use the larger of the two as the score for the subproblem.

Multiple Cycle Instances

In Section 4.1.3 we show how to represent the constraints of a component that is comprised of a single HP-cycle. In this section we show how to transform a multiple cycle component into a single cycle while appropriately ignoring inversions that are created by the process.

In [42], Hannenhalli and Pevzner introduce the notion of a (g, b)-*split* where a cycle of length six or larger is split into two (by adding two vertices) in such a way that preserves at least one minimum edit scenario in the process. Such a change in the graph can be represented in the corresponding permutation by a remapping of some vertex labels, this process is called a (g, b)-*padding*. Here we introduce the inverse operation to the split, the (d, r)-*join*, which takes two cycles and joins them in such a way that preserves all edit scenarios. Similarly, the analogue to the padding is the (d, r)-*shrink*. A (d, r)-join removes the vertices x^- and x^+ (from two different cycles) for some permutation element x along with reality edges (x^-, r_1) and (x^+, r_2), and desire edges (x^-, d_1) and (x^+, d_2). After removal, the edges $r = (r_1, r_2)$ and $d = (d_1, d_2)$ are created to form a valid HP-graph $\check{G}(\pi)$. It is easy to verify that a (d, r)-join operation is equivalent to a (d, r)-shrink which acts by removing the element x and renaming all other elements with magnitude $i > x$ to have magnitude $i - 1$ with the same sign. So $G(\check{\pi}) = \check{G}(\pi)$.

Lemma 4.1.10. *Apply to permutation π a (d, r)-shrink by removing an element x (corresponding to vertices x^- and x^+ from two different cycles) to obtain $\check{\pi}$. The EPO for π will be a subgraph of the inversion EPO for $\check{\pi}$ and $d(\pi) = d(\check{\pi})$.*

Proof. The length of the permutation decreases by one but so does the number of cycles, therefore $d(\pi) = d(\check{\pi})$. We now show that the (d, r)-join of cycles C_1 and C_2 turning $G(\pi)$ to $\check{G}(\pi)$ will preserve

the relative direction between edges. Fix a direction on the cycle with reality edge (x^-, r_1) by visiting r_1 before x^- followed by d_1. Conversely, fix a direction on the cycle with edge (x^+, r_2) by visiting d_2 before x^+ followed by r_2. Thus, after the application of the (d, r)-join the remaining reality edge r can be visited from r_1 to r_2 in a tour continuing to d_2 and d_1 from desire edge d. Since the direction for the new edges is consistent with the direction of the removed edges, the direction of r to reality edges in C_1 and C_2 is also consistent. So any inversion that acts on edges (x^-, r_1) and (x^+, r_2) for a edit scenario on π will now act on r for a edit scenario on $\breve{\pi}$. Since (x^-, r_1) and (x^+, r_2) are on different cycles of $G(\pi)$, there can be no oriented inversions done that act on the two at the same time. □

An important corollary to Lemma 4.1.10 is that all oriented inversions on π will be preserved. Thus, we can shrink a multiple cycle component to an "equivalent" cycle and then run the algorithm ignoring oriented inversions introduced by the shrinking process.

4.1.4 Handling Multiple Permutations

When improving the MGR heuristic for medians or implementing a greedy heuristic for maximum signature computation, one needs to consider sets of inversions that occur in multiple permutations. This is done by simply ignoring intervals that don't occur as oriented inversions in all permutations, while merging the constraints on the remainder of the permutations. That is, to find the maximum independent set of commuting/noninterfering inversions on many permutations, take the intersection of the sets of oriented inversions over all permutations and run the above algorithm on the union of the remaining constraints.

4.1.5 Two Notes on Hurdles

There are two places that hurdles complicate our analysis. The first concerns the existence of *unsafe* oriented inversions, those that make an oriented component unoriented. Of course, inversion that are unsafe on their own are easily identified (one way is to just apply the inversion and check the component), thus we simply remove all unsafe inversions from consideration before running our algorithm. It is possible, however, that a set of noninterfering inversions can collude to create a hurdle.

Given a permutation that already contains hurdles, we could be left with another tough situation. Suppose every component is a hurdle and that there are $O(n)$ of them (there exist unoriented components of size three). There are an exponential number of ways to merge these hurdles. It is clear that each combination of merges yields a new set of oriented inversions in the end, it is not clear as to whether an exponential search of these combinations in necessary. Suffice it to say that hurdles are very rare in practice, as confirmed by a theorem of Caprara [24, 80].

4.1.6 Experimental Results

We improved the MGR heuristic using maximum independent set of commuting/noninterfering inversions. Given three genomes G_1, G_2 and G_3, we define the median score of a genome G to be $d(G, G_1) + d(G, G_2) + d(G, G_3)$, where $d(G, G_i)$ is the distance between genome G and G_i. To find the genome that minimizes the median score, the new median solver chooses the maximum independent set of inversions which brings G_1 closer to both G_2 and G_3. The algorithm will then iteratively carry on maximum independent set of inversions in the three genomes until the maximum sets are empty. At the end of this procedure, the three given genomes are transformed to potentially three new genomes, and we report the one with the lowest median score as the resulted median.

To assess the speed and accuracy of this new solver, we tested it using the the same datasets of Arndt and Tang [7]. The datasets were generated by assigning the identity permutation on the internal node, and then the three leaves were created by applying rearrangement events along each edge respectively.

There are two factors governing the number of events on each edge: the number of total evolutionary events and the tree shape. The total number of events was in the range of 80 to 140, and three tree shapes were used: trees with almost equal length edges, i.e., the ratio of three edges are $(1:1:1)$; trees with one edge a bit longer than the other two, i.e., of ratio $(2:1:1)$; trees with one edge much longer than the other two, i.e., of ratio $(3:1:1)$. We compared the new method to Caprara's median solver (exact but slow), MGR and Arndt's solver. For each combination of parameters, ten trees were generated and the average results were reported.

Tables 4.1 and 4.2 show the median score found by each method, and Tables 4.3 and 4.4 show the time used by each method. We found from these tables that the new method not only runs faster than MGR, but also returns more accurate medians. When the datasets are difficult ($r \geq 120$), the new method is about $20 \sim 30$ times faster than MGR. Compared to Arndt's method, the new solver is about $3 \sim 100$ times faster with a $1 \sim 2\%$ sacrifice of accuracy.

We believe with some small amount of extra computation, the accuracy of our new solver can be further improved. The three new genomes obtained when the search stops actually form a new instance of median problem. We applied Caprara's solver to these new (smaller) median problems for all the testings and found that the scores were improved for most cases when $r \leq 100$–the median scores are almost the same as applying Caprara's solver to the original median problems. However, for $r \geq 120$, the new median instances were still very difficult and none was able to finish. These results suggest that a better method should be developed to handle the new median instances.

	(1:1:1)		(2:1:1)		(3:1:1)	
	r=80	r=100	r=80	r=100	r=80	r=100
Score lower bound	86.2	104.2	89.4	105.8	85.7	101.3
Caprara's median score	87.9	107.6	91.4	109.8	88.0	105.2
Arndt's median score	88.2	109.5	91.8	111.4	89.1	106.7
MGR median score	90.3	113.7	94.3	116.8	89.8	110.0
New method's median score	89.1	111.8	92.6	114.1	90.0	108.1

Table 4.1: Comparison of median scores for $r \leq 100$.

	(1:1:1)		(2:1:1)		(3:1:1)	
	r=120	r=140	r=120	r=140	r=120	r=140
Score lower bound	116.1	123.5	116.1	122.7	110.3	117.6
Caprara's median score	N/A	N/A	N/A	N/A	N/A	N/A
Arndt's median score	125.8	135.3	124.5	134.7	117.9	127.0
MGR median score	132.9	143.6	131.4	142.8	123.6	135.1
New method's median score	127.9	139.5	126.9	138.5	120.6	130.1

Table 4.2: Comparison of median scores for $r \geq 120$. N/A indicates a method cannot finish.

4.1.7 Conclusions

There were two algorithms introduced, one that computes a maximum set of commuting inversions and one that computes a maximum set of noninterfering inversions. The former has a worst case running time of $O(n^2)$ while the latter runs, under certain detectable conditions, in $O(n^2)$ time when the circle graph recognition algorithm of Spinrad [72] is used. When those conditions aren't met, we show that the problem can be decomposed so that only certain subproblems require exponential work (in the size of

	(1:1:1)		(2:1:1)		(3:1:1)	
	r=80	r=100	r=80	r=100	r=80	r=100
Caprara's time	3.6	12876	57.2	31387	4.3	6908
Arndt's time	324	551	123	409	1.6	9.3
MGR time	11.2	51.9	11.6	78.2	10.3	35
New method's time	3.3	5.3	4.1	8.4	4.6	9.1

Table 4.3: Comparison of running time for $r \leq 100$ (in seconds).

	(1:1:1)		(2:1:1)		(3:1:1)	
	r=120	r=140	r=120	r=140	r=120	r=140
Caprara's time	> 172880	> 172880	> 172880	> 172880	> 172880	> 172880
Arndt's time	1485	1187	673	453	30	226
MGR time	271.6	560.1	237.8	626.9	135.3	385.4
New method's time	13.8	19.7	11.1	21.3	9.2	12.4

Table 4.4: Comparison of running time for $r \geq 120$ (in seconds).

the subproblem). Let us comment that due to the intersection step described in Section 4.1.4, the more sequences we are comparing, the sparser the intersection is likely to be. We expect this to contribute to lower running times in practice.

The work of Arndt et al. [7] has shown that the MGR-style search for medians can be improved by the use of a more deliberate choice of inversions during a search. We expect the algorithms presented to continue those improvements to provide fast and accurate method for large genomes. The MGR-style objective function has also been formalized as a search for a maximum signature. While we have shown some relationships of sets of noninterfering inversions to signatures, and signatures to medians, we show in the next section a direct application of our noninterfering inversion algorithm to signature computation.

4.2 Inversion Signatures

The study of evolution is a study of patterns of change, but also of conservation, the latter being typically easier to detect and characterize. Moreover, elements conserved across many species were probably present in their last common ancestor and preserved through selection pressures, so that these conserved elements probably play a major role in the fitness of the organisms. Biologists have long studied patterns of conservation in DNA sequences: first pairwise sequence similarity in large databases (as in the widely used FASTA [62] and BLAST [5]), then multiple sequence alignments and phylogenetic reconstruction, and finally the reconstruction of ancestral sequences, an avenue of enquiry that has seen much activity of late (see, e.g., [52]). Recently, researchers have also started to look for characteristic patterns of change across a collection of species—an example being the *discriminating subsequences* of Angelov *et al.* [6]. All of these efforts aim at recovering what one could term *genomic signatures*—subsequences that best characterize the evolutionary history of the given group of organisms.

As more genomes are fully sequenced, interest in reconstructing complete ancestral genomes has grown; Pevzner's group, for instance, has published extensively on the topic in the context of vertebrate genomes (see, e.g., [18, 17]), as has a group headed by Haussler and Miller [53]. However, while rearrangements such as inversions, transpositions, translocations, and others are complex and powerful operations, our models for them remain poorly parameterized, often reduced to the simplest case of uniform distributions. Under such models, reconstruction of ancestral genomes for organisms that exhibit significant divergence (in contrast to mammals or even vertebrates) remains poor, mostly due to the enormous number of equally "good" evolutionary scenarios [30, 35]. It is therefore natural to turn once again to genomic signatures, this time formulated in terms of a rearrangement (rather than a sequence evolution) model.

In this section we introduce a measure of similarity defined between two genomes *with respect to a third*. The key idea is the introduction of the third genome, which allows us to take into consideration the evolutionary paths from the two genomes under study to the third, thus basing our measure of similarity on the evolutionary history of the two genomes rather than just on their current configuration. Naturally, these evolutionary paths are not unique under current models and thus a number of ancestral states can be reached on the way from the two genomes under study to the third genome. We call these states *rearrangement signatures* and further distinguish those that are farthest from the third genome (the most recent, as viewed from the perspective of our two genomes under study) as *maximum rearrangement signatures*. Although the concepts introduced here apply to any rearrangement operation, we study these signatures under the operation of inversion, the most commonly used rearrangement operation in work to date [58]. We show that maximum signatures carry much information about ancestral genomes and that they can often be computed within a reasonable amount of time in spite of the very large search space. We use simulations under a wide variety of conditions to show that the maximum signatures pinpoint the true ancestral genome, either recovering it outright or producing one very close to it, and to show that these signatures can be used to reconstruct reliable phylogenies, all using a polynomial-time heuristic that runs much faster than a full exhaustive search.

4.2.1 Notation and Definitions

Remember that $\pi_0, \pi_1, \ldots, \pi_d$ forms an *edit scenario* from π_0 to π_d if for all π_i, $0 \leq i < d$, we have $d(\pi_i, \pi_{i+1}) = 1$; each inversion applied along this path is then deemed an *edit inversion*. Take each π_i to be a vertex and link two vertices with an edge whenever the corresponding permutations occur consecutively on an edit scenario. This graph represents a partial order with relation "is part of an edit scenario from". We call this the *edit partial order*, or *EPO*. We denote the EPO between π_0 and π_d as $EPO_{\pi_0}(\pi_d)$ or $EPO_{\pi_d}(\pi_0)$. So if we have π_3 = (2 −1 −3) and π_0 = (1 2 3), then an edit scenario between them might visit permutations π_2 = (−2 −1 −3) and π_1 = (−2 −1 3) before reaching π_0. Figure 4.6 shows the EPOs for (2 −1 −3) and (−2 3 1).

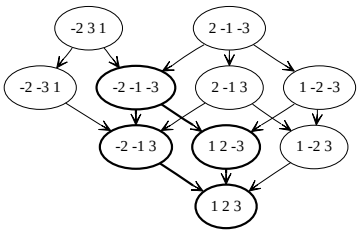


Figure 4.6: The union of the edit partial orders for P = {(-2 3 1), (2 -1 -3)} and τ = (1 2 3). The signature graph for P is highlighted in bold.

We are interested in the intersection of EPOs, which will yield the desired inversion signatures. For a set of $k+1$ permutations, one of which is the reference permutation called the *locus*, an *inversion signature* is the permutation corresponding to a vertex in the intersection of the k EPOs from each of the other k permutations to the locus.

Definition 4.2.1. *The set of all* inversion signatures *for permutations $\pi_1, \ldots, \pi_k$ with locus π_L is $S_{\pi_L}(\pi_1, \ldots, \pi_k) = V\Big(EPO_{\pi_L}(\pi_1) \cap EPO_{\pi_L}(\pi_2) \cap \cdots \cap EPO_{\pi_L}(\pi_k)\Big)$, where $V(G)$ denotes the set of vertices of graph G.*

Whenever the context is unambiguous, we shall simply write S_{π_L} for $S_{\pi_L}(\pi_1, \ldots, \pi_k)$. Similarly, the *signature graph* on $\pi_1, \ldots, \pi_k$ with respect to π_L is the graph $EPO_{\pi_L}(\pi_1) \cap EPO_{\pi_L}(\pi_2) \cap \cdots \cap EPO_{\pi_L}(\pi_k)$. An inversion signature $\pi_s \in S_{\pi_L}$ is thus a permutation that embodies some of the commonality between the k other permutations with respect to π_L, in the sense that they all possess an edit scenarios to π_L that passes through π_s. A *maximum signature* is a signature in S_{π_L} that is as far away from π_L (and thus as close to the k other permutations) as possible.

Definition 4.2.2. *The set of all* maximum signatures *is $S^*_{\pi_L} = \{\pi_s \in S_{\pi_L} |$ for all $\pi'_s \in S_{\pi_L}, d(\pi_L, \pi_s) \geq d(\pi_L, \pi'_s)\}$.*

A maximum inversion signature is thus a permutation that represents the "maximum commonality" between the k permutations: it is as close to these k permutations as possible while still being part of all edit scenarios to π_L. From a biological perspective, this edit scenarios from π_L to the signature can be thought of as the evolution that happened before speciation, or the pattern of change that the k sequences have in common.

As with the special case for Steiner points called the *median* [68], we find it helpful to name the case with $k+1=3$. For this case we have two permutations π_A and π_B and an ancestor locus π_L and we call $S^*_{\pi_L}(\pi_A, \pi_B)$ the *pairwise maximum signature*.

In Figure 4.6 we have π_A = (2 −1 −3), π_B = (−2 3 1), and π_L = (1 2 3) (the *identity* permutation of length 3). The signature graph is outlined in bold. The signatures in this case are (−2 −1 −3), (−2 −1 3), (1 2 −3), and the trivial signature π_L = (1 2 3). The only maximum signature is also the only maximal signature (−2 −1 −3).

4.2.2 Methods

We begin with an investigation of rearrangement-based genomic signatures as defined above, then give procedures for signature-based phylogenetic and ancestral reconstruction.

Computing Signatures

Definition 4.2.1 can be restated inductively in terms of edit scenarios that move from the locus π_L towards the other permutations $\pi_1, \ldots, \pi_k$. We say that some permutation π has a *common edit inversion* r with respect to $\pi_1, \ldots, \pi_k$ if we observe $d(\pi_L, \pi_i) - d(\pi_L, r\pi_i) = 1$ for $1 \leq i \leq k$.

Definition 4.2.3. *The locus π_L is an inversion signature for permutations $\pi_1, \ldots, \pi_k$. If permutation π is an inversion signature and r is a common edit inversion with respect to $\pi_1, \ldots, \pi_k$, then $r\pi$ is also an inversion signature.*

Thus, starting at the locus (which is the smallest possible signature), one can enumerate all signatures by repeatedly applying every possible common edit inversion to the current collection of signatures; maximal signatures are those signatures for which no common edit inversion exists and maximum signatures are the largest of these maximal signatures (i.e., the farthest away from the locus). Common edit inversions form the basis for the MGR algorithm of Bourque and Pevzner [17], who used a greedy algorithm that picks a single path by always choosing the common edit inversion that provides the largest number of common edit inversions at the next step.

The signature space is of course very large. In particular, if the two permutations of interest are just one inversion apart, then the space of all signatures can be roughly the same size as the inversion EPO between one of the permutations and the locus—and that is, in expectation, exponentially large in the pairwise distance. (However, the complexity of finding a maximal signature is unknown at this time.) We use the greedy heuristic of MGR to construct maximal signatures and show that it often returns the maximum signature. It is not optimal, however: consider the permutations $\pi_A = (-4\ 1\ -5\ 2\ -6\ 3)$, $\pi_B = (-4\ 1\ 6\ 2\ -5\ 3)$, and $\pi_L = (1\ 2\ 3\ 4\ 5\ 6)$. In the signature graph of Figure 4.2.1, vertices that can be produced by the greedy heuristic are highlighted, none of which are a maximum signature.

Noninterfering Independent Sets

We introduced noninterfering sets of inversion in Section 4.1. The concept of noninterfering inversions extends naturally to our framework with a defined ancestor.

Definition 4.2.4. *A set of inversions R is* mutually noninterfering *for π_A and π_B with locus π_L if it is noninterfering for π_L with respect to π_A and also for π_L with respect to π_B.*

Such mutually noninterfering sets form the basis for another greedy algorithm: we repeatedly find and apply to π_L sets of mutually noninterfering inversions until there are none left. Mutually noninterfering sets can be found very quickly, so a greedy algorithm based on this approach runs very fast. We use this particular greedy heuristic in our experiments.

Signature-Based Tree Reconstruction

Since signatures are just nodes along evolutionary paths, they can be used as internal nodes in a process of phylogenetic reconstruction. We begin with a naïve algorithm to illustrate the basic approach.

The idea is to overlay the EPOs from each of the leaves $\pi_1, \ldots, \pi_k$ to the locus π_L and construct a tree representative of the resulting structure. Consider the set of these EPOs, $O = \{EPO_{\pi_L}(\pi_i) | 1 \leq i \leq k\}$;

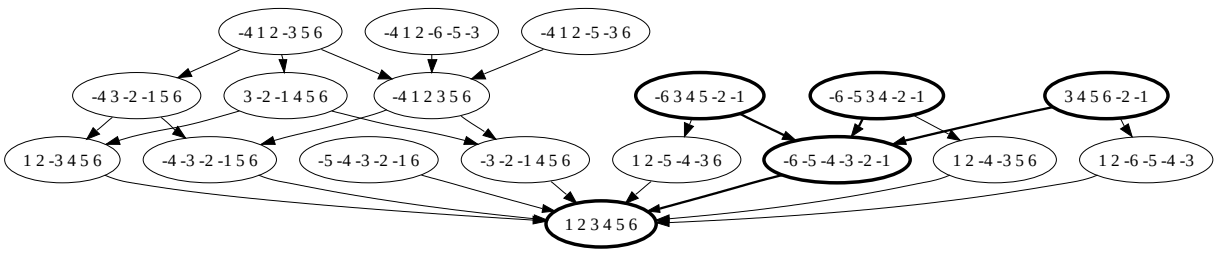


Figure 4.7: The signature graph for π_A=(-4 1 -5 2 -6 3), π_B=(-4 1 6 2 -5 3), and π_L=(1 2 3 4 5 6).

our algorithm constructs a tree from the current version of O, iteratively choosing a node from pairwise intersections of graphs in O and updating O to reflect this choice. Specifically, at iteration i,

1. select from O a vertex π_s that maximizes $d(\pi_L, \pi_s)$;
2. if the vertex selected in the previous step belongs to the intersections of $P_A, P_B \in O$, then create a node in the tree to be the parent of the subtrees represented by P_A and P_B;
3. in O replace $EPO_{\pi_L}(\pi_A)$ and $EPO_{\pi_L}(\pi_B)$ with their intersection.

This algorithm yields a tree without internal node labels, because EPOs are not closed under intersection, so that a node in the tree may represent two graphs from O that no longer have a least upper bound.

Our second algorithm overcomes this problem; in addition, it yields implicit edit scenarios from the leaves to the root that join at the internal nodes. In this improved version, we maintain the invariant that elements of O are always EPOs. Thus only the third step of the iteration is affected, and replaced by the following:

- in O replace $EPO_{\pi_L}(\pi_A)$ and $EPO_{\pi_L}(\pi_B)$ with $EPO_{\pi_L}(\pi_s)$.

Step 1 in each iteration is obviously the computationally intensive one; our implementation for this step uses the MGR heuristic.

Distance-Based Bound on Signature Size

We develop an upper bound based on pairwise distances to help us evaluate our greedy signature methods in the experimental phase. Denote by A, resp. B, the inversion distance between the locus and π_A, resp. π_B, and by D the inversion distance between π_A and π_B. (Inversions distances can be computed in linear time [9].) Now consider some arbitrary signature π_S for this triple and denote its size, or distance from the locus, by c; Figure 4.8 depicts the situation. As all distances are edit distances, we can write $A - a = B - b$ and, by the triangle inequality, $a + b \geq D$; combining the two, we get

$$a \geq \frac{D + A - B}{2},$$

with the symmetric version for b. Without loss of generality, assume $A \geq B$; then we get

$$d(\pi_L, \pi_S) = c \leq A - \Big(\frac{D + A - B}{2}\Big),$$

the desired upper bound.

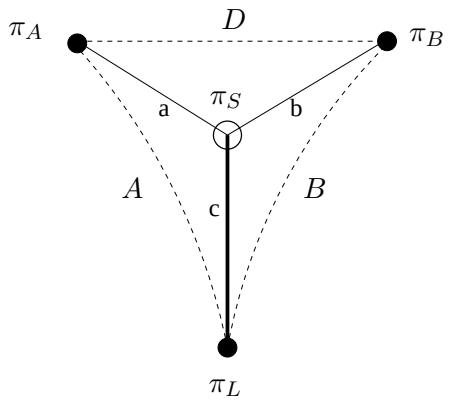


Figure 4.8: The distances around a signature π_S.

4.2.3 Results and Discussion

We demonstrate the use of pairwise inversion signatures for ancestral reconstruction and for phylogenetic reconstruction through extensive simulations. We first show that, under certain reasonable conditions, maximum signatures coincide with ancestral genomes most of the time, then proceed to show that, under more stringent conditions, maximum signatures always coincide with ancestral genomes. Since no polynomial-time algorithm for computing maximum signatures is known at present, we show that our heuristics perform well, both in terms of accuracy and running time, even when applied to larger genomes (to the size of small prokaryotic genome). Finally, we show that the signature method use for phylogenetic reconstruction produces trees comparable in quality to neighbor-joining while providing ancestral reconstructions along the way.

Maximum Signatures as Ancestral Genomes

Our experiments for ancestral reconstruction simply use triplets of genomes generated from an ancestral genome by generating three evolutionary paths, using randomly chosen inversions. The locations of these inversions is distributed uniformly at random, but their lengths are distributed according to one of two possible distributions: uniform and normal. The lengths of the edges from the ancestor to the three leaves are chosen in both a balanced manner and several skewed manners. All of our experiments used 1,000 repetitions unless stated otherwise and the results presented show averages over these 1,000 tests.

We present most of our results in the form of tables. Tables 1 through 6 group columns by the percentage of the length of the longest simulated path P in the triplet. For instance, column two of Table 4.5 shows the percentage of true ancestors that are within $0.15 \times |P|$ inversions away from a maximum signature (in this case, no more than one inversion away because $|P|$ is no greater than 8 for any row of column two). The rows in these cases are labeled by the edge length as a percentage of the genome size.

The first set of tables apply to triplets where all edges have the same length (that is, the same number of random inversions). Table 4.5, for normally distributed inversion lengths, shows that the simulated ancestor is a maximum signature most of the time, even when the evolutionary rates are extremely high. When the rates are already high 10% of the genome size, 97% of the true ancestral genomes are maximum signatures. The table also shows that (the last two rows aside) the true ancestor is within 2 inversions from a maximum signature more than 90% of the time. Table 4.6 shows similar, but slightly weaker results for uniformly distributed inversion lengths.

The next set of tables examines the influence of the size of the genome. Table 4.7 shows that the accuracy scales well. In addition, we tested genomes of size 100; the results are shown in Table 4.8.

Table 4.5: Percentage of the time that the true ancestor is a maximum signature, under normally distributed inversion lengths on genomes of size $n = 30$.

# of ops as	% of $\|P\|$			
% of n	0	$\leq 15\%$	$\leq 20\%$	$\leq 50\%$
10	97	97	97	100
15	93	93	93	100
20	84	84	93	100
25	78	88	88	100
29	68	83	93	100

Table 4.6: Percentage of the time that the true ancestor is a maximum signature, under uniformly distributed inversion lengths on genomes of size $n = 30$.

# of ops as	% of $\|P\|$			
% of n	0	$\leq 15\%$	$\leq 20\%$	$\leq 50\%$
10	94	94	94	99
15	87	87	87	100
20	69	69	84	100
25	53	73	73	100
29	36	58	77	100

Table 4.7: Percentage of the time that the true ancestor is a maximum signature as a function of the genome size n for simulated edge lengths of $n \times 0.1$.

	% of $\|P\|$			
n	0	$\leq 15\%$	$\leq 20\%$	$\leq 50\%$
30	97	97	97	100
35	96	96	96	100
40	95	95	95	100
45	95	95	95	100
50	94	94	98	100
55	95	95	98	100
60	91	91	97	100
65	93	93	98	100
70	91	96	96	100
75	86	92	92	100

Table 4.8: Percentage of the time the true ancestor is a maximum signature, under normally distributed inversion lengths on genomes of size $n = 100$.

# of ops as	% of $\|P\|$					
% of n	0	$\leq 5\%$	$\leq 10\%$	$\leq 15\%$	$\leq 20\%$	$\leq 50\%$
5	95	95	95	95	99	100
8	91	91	91	97	99	100
10	90	90	100	100	100	100

Table 4.9: Percentage of the time that the true ancestor is a maximum (method 1) or maximal (methods 2 and 3) signature, under normally distributed inversion lengths on genomes of size $n = 30$. Method 1 finds a maximum signature by exhaustive search; method 2 uses the greedy Bourque-like approach; and method 3 uses the approach based on maximum sets of noninterfering inversions.

# of ops as % of n	Method	% of $\lvert P \rvert$ 0	$\leq 15\%$	$\leq 20\%$	$\leq 50\%$
10	1	97	97	97	100
	2	97	97	97	100
	3	96	96	96	99
15	1	93	93	93	100
	2	93	93	93	100
	3	89	89	89	100
20	1	84	84	93	100
	2	83	83	92	100
	3	76	76	85	100
25	1	78	88	88	100
	2	76	86	86	100
	3	67	77	77	100
29	1	68	83	93	100
	2	66	81	89	100
	3	57	69	76	100

Computing Maximal Signatures

The exhaustive algorithm rapidly reaches its limits: for genomes of size 100 with edge lengths of 10, computations already take on the order of hours instead of minutes. Table 4.8 shows favorable results for exhaustive computation of maximum signatures on such genomes. We now proceed to compare these results with those of our new maximal signature algorithms. Under most circumstances, the true ancestor is found by such maximal signature computations.

Table 4.9 shows that the Bourque-like approach and the approach based on noninterfering inversions fare well with respect to the exhaustive search, the latter dropping off first. Table 4.10 shows results for the two greedy methods on genomes of size 100. For reasonable rates of evolution (10% or less per edge), we again see that the true ancestor is found most of the time.

Finally, we tested on genomes of more realistic sizes, but of a size usually considered forbidding for ancestral inference—up to 1,000 genes. With 50 random events per edge the Bourque-like computations take just under 30 minutes, while for 80 random events they take under 2 hours. The accuracy remains very high: in 99% of the 380 trials with 50 random events per edge, the signature returned is within 5 inversions of the true ancestor, while in 66% of these trials, the signature returned is in fact the true ancestor. The approach based on noninterfering inversions is by far the fastest, taking under a half a minute for each of these trials, even with 80 random events per edge. Using 50 random inversions per edge, we found that 97% of the 1000 trials gave an ancestor within 5 inversions of the true ancestor, while 57% gave the true ancestor. With 80 events per edge, 91% gave an ancestor within 8 inversions of the true ancestor, while 15% gave the true ancestor.

The largest genomes we tested had size 2000 (corresponding to small bacterial genomes, for instance) and 100 operations per edge, and 5000 (corresponding to the genomes of free-living bacteria such as E. coli) with 250 operations per edge. All trials gave a signature within 10 inversions of the true ancestor, while 90% gave one within 4 inversions, all running in under 2 minutes per trial for size 2000 and 4 minutes per trial for size 5000. These speeds are enormously higher than methods such as

Table 4.10: Percentage of the time that the true ancestor is a maximal signature, under normally distributed inversion lengths on genomes of size $n = 100$. Method 2 uses the greedy Bourque-like approach while method 3 uses the approach based on maximum sets of noninterfering inversions.

# of ops as		% of $\|P\|$					
% of n	Method	0	$\leq 5\%$	$\leq 10\%$	$\leq 15\%$	$\leq 20\%$	$\leq 50\%$
5	2	95	95	95	95	99	100
	3	94	94	94	94	98	100
8	2	90	90	90	97	99	100
	3	86	86	86	92	94	100
10	2	85	85	94	97	100	100
	3	77	77	85	87	98	100
15	2	68	68	92	98	100	100
	3	54	54	73	90	98	100
20	2	43	63	89	98	100	100
	3	28	41	74	90	98	100

Table 4.11: Percentage of the time that the true ancestor is a maximal signature, under normally distributed inversion lengths on genomes of size $n = 50$. Edge lengths b (to a child) and c (to an ancestor) vary from 5 to $2a$ while $a = 5$ (number of inversions to the other child). Each entry shows the exhaustive method to the left of the Bourque-like method.

	c							
b	5		7		10		12	
5	94	94	92	91	87	87	–	82
7	90	90	88	88	82	82	–	79
10	88	88	84	83	80	80	–	73
12	86	86	83	83	–	76	–	66

MGR or median-based reconstructions, yet the accuracy is also much higher. Thus, by focusing on the characteristic (shared) patterns of inversions, we are able to win on two fronts at once, mostly because we avoid the confusion and long explorations associated with multiple reconverging paths.

Skewed Trees

The true ancestor will not always be equidistant from the leaves and the locus. While large amounts of skew can sometimes move an ancestor farther from a maximum signature, the true ancestor usually remains very close to a maximum signature.

We call the number of random inversions from the locus to the true ancestor c and the number of random inversions from the true ancestor to each of the leaves a and b. We fix a to be 10% of the total length and vary c and b from values equal to a up to 2.5 times a. Table 4.11 shows that for genomes of size 50, the true ancestor is a maximum signature in most cases and that almost as often it is a maximal signature found by the Bourque-like greedy method. Our maximum signature method appears slightly more robust to skew on one of the child branches as opposed to skew on the branch to the locus.

Tree Reconstruction

We simulated evolution over 300 trees to test our signature-based tree reconstruction method. We found that our method (using the Bourque-like signatures for efficiency) reconstructs the true topology most of the time and that any error remains very small. The trees were constructed using the birth-death model

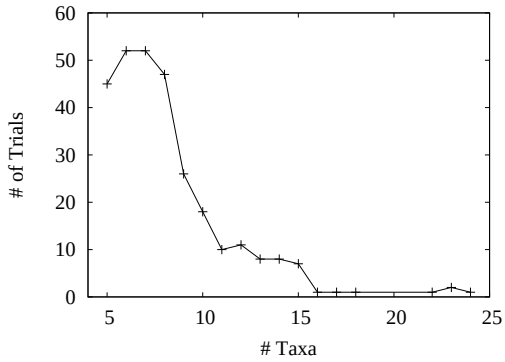


Figure 4.9: The size of the generated trees.

and the mean of the normally distributed edge lengths was varied from 5 to 9 operations with a standard deviation varying from 2 to 3. The mean of the normally distributed inversion lengths was varied from 8 to 30 with a standard deviations varying from 5 to 10. The generated trees have from 5 to 24 taxa and are distributed as shown in Figure 4.9.

Two methods were used for choosing a locus. The first method used the true root of the tree given by the simulation (an ideal method not available in practice, of course), while the second method used a random leaf as the locus. With the true root as the locus, we found that 94% of the trees were reconstructed perfectly, while 16 of the 17 remaining trees had a Robinson-Foulds error of 2, giving an average RF error of 0.15. With a random leaf as the locus, we found that 85% of the trees were reconstructed perfectly, while 28 of the 45 remaining trees had an RF error of 2 and 11 of the last 27 had an RF error of 4, giving an average RF error of 0.5.

Using the true root as the locus demonstrates that the pairwise signature contain a great deal of information about the phylogeny. Using a random leaf as the locus demonstrates that such information remains recoverable even when the choice of locus is arbitrary (and usually far from ideal), justifying our initial claim that comparing two genomes with respect to a third tremendously enriches what can be had from a direct pairwise comparison. (As an example, trees that were not properly reconstructed by the neighbor-joining method, which uses strictly pairwise comparisons, were commonly reconstructed correctly by our signature-based method.) Our tests for phylogenetic reconstruction are obviously of limited scope, meant to exemplify the usefulness of the method rather than provide a full evaluation; and the method itself is subject to many obvious improvements (better ways to choose a locus, using k-way signatures rather than pairwise ones to support a top-down reconstruction method, etc.)

Tightness of the Upper Bound

Finally, we present experimental results suggesting that our upper bound is on average very tight and then use the bound to show that the greedy signatures, used for ancestral reconstruction of genomes too large for the exhaustive computation, are indeed close to a maximum signature. Since the computed ancestor is bracketed within this bound, our results imply that the maximum signature is very close to the true ancestor with high probability.

The upper bound was computed for each trial in Table 4.5. For each of the sets of 1000 trials, the average difference between the upper bound and the maximum signature was 0.029, 0.073, 0.176, 0.27, and 0.327 for trials with 10, 15, 20, 25, and 29 percent respectively. For the length-dependent data from Table 4.7, the average difference stays between 0.021 and 0.082. Table 4.12 indicates similar performance for experiments run on skewed triplets. The tests from Table 4.10 give average differences from 0.024 up to 1.375 for the Bourque-like method and differences from 0.048 up to 2.228 for the

Table 4.12: The average difference between the upper bound and the computed signatures with normally distributed inversion lengths and genomes of size $n = 50$. Edge lengths b (to a child) and c (to an ancestor) vary from 5 to $2a$ while $a = 5$ (number of inversions to the other child). Each entry shows the exhaustive method to the left of the Bourque-like method.

	c							
b	5		7		10		12	
5	0.053	0.053	0.080	0.081	0.138	0.143	–	0.176
7	0.106	0.106	0.114	0.114	0.173	0.173	–	0.224
10	0.097	0.098	0.165	0.167	0.203	0.203	–	0.290
12	0.131	0.132	0.158	0.158	–	0.279	–	0.359

noninterfering inversions method. Only 1 of the tests from genomes of size 1000 did not match the upper bound for the greedy method.

4.2.4 Conclusions

In any study of evolutionary changes, the challenge is to distinguish global patterns from a background of many local changes—or, to put it another way, to find commonalities among many equally plausible evolutionary paths that lead to the same modern organism. We have proposed an approach to this problem that focuses on intermediate states along such paths in the setting of a speciation event and seeks to return the last (most recent) states from which both species of organisms could still have been derived. This approach offers multiple benefits: the focus on intermediate states translates readily into one on ancestral reconstruction; the study of paths going through a fork (the speciation event) stresses the role of evolutionary history rather than just final states; and the search for the most recent states that are part of the fork naturally separates common evolutionary changes (prior to the fork) from individual variations (subsequent to the fork). Although finding such signatures appears hard, we gave an efficient heuristic that does very well through an extensive range of simulations. Our signatures are based on inversions, since inversions are the best studied of the various genomic rearrangements to date, but the concept readily extends to any other rearrangement operation or family of such operations.

Chapter 5

Sorting By Inversions in $O(n \log n)$ Time

(This is joint work with Vaibhav Rajan and Yu Lin)

In 1992 Sankoff posed two fundamental questions about inversions: given two signed permutations, what is the smallest number of inversions required to transform one permutation into the other and what is a scenario of inversions implementing this transformation [65]. The first problem is thus to compute an edit distance, where the edit operation is the inversion; the second is to return an edit scenario—a problem usually known as "sorting," since a simple re-indexing can turn one of the permutations into the identity. Many years of work were needed to ascertain the complexity of each of these problems. The breakthrough came in 1995, when Hannenhalli and Pevzner provided a polynomial-time algorithm to solve both problems. (In contrast, in 1997, Caprara [22] showed that both problems were NP-hard if phrased in terms of unsigned permutations.) The running time for both problems has been steadily reduced over the years. In 2001, Bader et. al. gave an optimal linear-time algorithm to compute the edit distance [8]; and in 2004, Tannier and Sagot, building on the work of Kaplan and Verbin [46], gave an $O(n\sqrt{n \log n})$ algorithm to produce a sorting scenario. Remaining open was the question of whether signed permutations can be sorted by inversions in $O(n \log n)$ time.

In this chapter, we give a qualified positive answer to this question by describing two new algorithms for sorting signed permutations by inversions. The first is a randomized algorithm that runs in guaranteed $O(n \log n)$ time, but may fail; successive restarts reduce the probability of failure, but we cannot guarantee that every permutation will be sorted with high probability with a finite number of restarts, so that it is not a true Las Vegas algorithm. (Indeed, we give a family of permutations that cannot be sorted by this algorithm regardless of the number of restarts.) The other is a deterministic algorithm that always sorts the permutation and runs in $O(n \log n + kn)$ time, where k is the number of successive "corrections" (detailed in Section 5.4) that must be applied—a value, incidentally, that appears not to be related to the edit distance, although it is bounded by it. We give a family of permutations for which k is $\Theta(n)$ (the worst case value for k) and thus for which our sorting algorithm will run in quadratic time. However, we present the results of very extensive experimentation showing that the expected value and the standard deviation of k are small constants (less than 1), independent of n, so that the running time of the algorithm is, with high probability, $O(n \log n)$. Thus we conclude (but do not prove) that almost all permutations can be sorted in optimal $O(n \log n)$ time.

5.1 Preliminaries

In this chapter we assume that every permutation of n elements is framed by elements 0 and $n + 1$. In this way we consider each permutation to be linear, noting that each linear permutation corresponds to $n+1$ circular permutations (of length $n+1$), which are equivalent in terms of the scenarios of inversions

used to sort them. The *span* of an inversion $\rho(i, j)$ is the closed interval on the natural numbers $[i, j]$ and two spans $[i, j]$ and $[k, l]$ *overlap* if and only if either $i < k$ and $k < j$ or $k < i$ and $j < l$.

We remind you that two adjacent elements, π_i and π_{i+1} for $0 \leq i \leq n + 1$, form an *adjacency*. An adjacency is a *non-breakpoint* if and only if we have $\pi_{i+1} - \pi_i = 1$, otherwise it is a *breakpoint*. An *oriented pair*, (π_i, π_j), in a permutation is a pair of integers with opposite signs such that $\pi_i + \pi_j = \pm 1$. The inversion induced by an oriented pair (π_i, π_j), called an *oriented inversion*, is $\rho(i, j - 1)$ for $\pi_i + \pi_j = +1$, and $\rho(i+1, j)$ for $\pi_i + \pi_j = -1$. An oriented inversion always creates a non-breakpoint; we say that it *heals* the breakpoint (or breakpoints—there could be two) to which the elements of the oriented pair belonged before the inversion.

We refer you to Definition 2.1.2 of a *framed common interval* (FCI), and the paragraph following that for definitions of *good* and *bad components*. An inversion is said to be *unsafe* if it creates a bad component, otherwise it is *safe*.

A permutation is *positive* if it is not the identity permutation and every element is positive. A positive permutation indicates the existence of at least one bad component. Any permutation containing bad components can be transformed to another permutation that does not contain any bad component in linear time [8]. Thus, in the algorithms we describe, we assume that the input permutation does not contain any bad components.

5.2 Background: Data Structures for Permutations

To implement an algorithm for sorting by inversions, we need a data structure for handling permutations that supports two basic operations: (i) choose an oriented inversion, and (ii) perform an inversion.

We now describe the data structure of Kaplan and Verbin [46] that stores a permutation in linear space and allows us to perform an inversion in logarithmic time. The structure is a splay tree, in which the nodes are ordered by the indices of the permutation, with one additional flag maintained at each node.

To perform an inversion $\rho(i, j)$ between (and including) indices i and j, index $i - 1$ is splayed and the right subtree of the root is split from the root yielding subtrees $T_{<i}$ and $T_{\geq i}$ where $T_{<i}$ ($T_{\geq i}$) contains all elements with indices less than (greater than or equal to) i. Next, index j is splayed in $T_{\geq i}$ and again the right subtree is split from its root yielding subtrees T_{rev} and $T_{>j}$ where $T_{>j}$ contains all elements with indices greater than j and T_{rev} contains the elements of the permutation that have to be reversed. Finally, there are three subtrees: $T_{<i}$, T_{rev} and $T_{>j}$. Now, actually reversing the elements in T_{rev} can take $\Theta(n)$ time since $\Theta(n)$ elements could be reversed in a single inversion. To achieve logarithmic time complexity a lazy approach is taken: a *reversed* flag is maintained in each node, which if turned on indicates that the subtree rooted at the node is reversed. Now instead of immediately reversing a subtree, we just set its reversed flag. During an inversion the reversed flag of the root of T_{rev} is flipped and $T_{<i}$ is joined to T_{rev} to get $T_{\leq j}$. This is achieved by making T_{rev} the right child of the root of $T_{<i}$, which still contains the element at index $i-1$, yielding the tree $T_{\leq j}$. $T_{\leq j}$ is then joined to $T_{>j}$ by splaying j in $T_{\leq j}$, after which $T_{>j}$ is made the right child of the root of $T_{\leq j}$, yielding the final tree which represents the permutation after the inversion. Since the only operation that takes more than constant time is the splay and since splaying takes amortized logarithmic time [71], each inversion takes amortized logarithmic time.

A tree could have several reversed flags, but the invariant maintained is that an in-order traversal modified by the reversed flags yields the permutation. So to read the permutation one would traverse a reversed subtree in reverse order while flipping signs of elements read. Nested reversed flags cancel in the sense that a reversed flag on a node within a reversed subtree, implies that the inner subtree (rooted at that node) is not reversed. Thus, a subtree rooted at a node is reversed if and only if there is an odd number of reversed flags in the path from the root to the node (including the node).

When a scenario of inversions is performed, reversed flags can get nested to arbitrarily deep levels.

We can push the flag down a traversed path in the tree, by flipping the sign of the element in the node, exchanging the left and right subtrees, and flipping the reversed flags in both children. The reversed flag of a leaf is cleared by just flipping its sign. Pushing down a flag takes constant time per node so the logarithmic time complexity of splaying is maintained. By pushing down the flags in the splay path we ensure that the three subtrees created ($T_{<i}$, T_{rev} and $T_{>j}$) reflect the changes made in all the previous inversions.

This is exactly the data structure described in [46]; it can handle a scenario of d inversions in $O(d \log n)$ time. The data structure maintains only the state of the permutation at each step (in a lazy way). However it does not maintain information about oriented pairs, nor could it do so efficiently, as a single inversion could change the orientation of $\Theta(n)$ pairs. Indeed, using this data structure to maintain the information necessary to choose an oriented inversion at each step would increase the running time by a factor of n.

To overcome this problem both Kaplan and Verbin [46], and later Tannier and Sagot [87], used a two-level version of the data structure in which a permutation is stored in linear blocks of size $O(\sqrt{n \log n})$ each. Corresponding to each block is a splay tree that maintains information about all oriented pairs (π_i, π_j) such that either π_i or π_j is in the block. Performing an inversion while maintaining information about all oriented pairs takes $O(\sqrt{n \log n})$ time and choosing an inversion at each sorting step takes $O(\log n)$ time, so that the total time complexity of their algorithms is $O(n\sqrt{n \log n})$.

In order to run in $O(n \log n)$ time, these algorithms need to be able to choose an oriented inversion in logarithmic time and thus information to identify such inversions must also be maintained in logarithmic time through an inversion.

5.3 Our Algorithm

Instead of addressing the data structure (by designing a new data structure that can somehow process $O(n)$ new pair orientations in logarithmic time), we address the root question of identifying an oriented inversion. Our key contribution is that we need not maintain information about *all* oriented inversions for every permutation at each sorting step—a couple suffice in most cases.

5.3.1 MAX inversions

Definition 5.3.1. *Let (π_i, π_j) be an oriented pair in a permutation and let π_j be the negative element in the pair. The oriented inversion corresponding to (π_i, π_j) is a* MAX inversion *if π_j has the maximum value of all negative elements in the permutation. The pair (π_i, π_j) is called the* MAX pair *of the permutation.*

For example the MAX inversion in the permutation $(4\ 5\ {-3}\ 1\ {-6}\ 2\ {-7})$ is $\rho(4, 6)$, corresponding to the oriented pair $(2, -3)$, and the MAX inversion in the permutation $(2\ 3\ {-1}\ {-4})$ is $\rho(1, 3)$, corresponding to the oriented pair $(0, -1)$. We maintain information about only the MAX inversions in the data structure and correspondingly perform a MAX inversion in each sorting step. The result is algorithm MAX.

Algorithm 1 MAX

1: **while** there exists a negative element in the permutation **do**
2: Find index of maximum negative element π_j.
3: Find index of $\pi_i = |\pi_j| - 1$.
4: Perform inversion corresponding to oriented pair (π_i, π_j).
5: **end while**

Because any permutation that contains a negative element contains a MAX inversion and because any scenario of oriented safe inversions is optimal [42], we can conclude as follows.

Lemma 5.3.2. *In the absence of unsafe MAX inversions at any sorting step, algorithm MAX produces an optimal sorting scenario.*

Algorithm MAX fails to sort only when it is "stuck" at an all-positive permutation that is not the identity, which happens when a MAX inversion was unsafe. (We deal with unsafe inversions in the next section.) The same arguments hold *mutatis mutandis* if we choose an oriented pair with the minimum negative element, yielding another algorithm, algorithm MIN. Combining the two strategies and picking one at random at each step gives us a randomized algorithm: algorithm RAND.

Algorithm 2 RAND

while there exists a negative element in the permutation **do**
  randomly select either MAX or MIN
  if MAX **then**
    Find index of maximum negative element π_j.
    Find index of $\pi_i = |\pi_j| - 1$.
    Perform inversion corresponding to oriented pair (π_i, π_j).
  else if MIN **then**
    Find index of minimum negative element π_k.
    Find index of $\pi_l = |\pi_k| + 1$.
    Perform inversion corresponding to oriented pair (π_k, π_l).
  end if
end while

5.3.2 Maintaining information through an inversion

We now show how to maintain information about the maximum negative element of a permutation through an inversion using the splay tree data structure. We describe the process for MAX, but the obvious analog works for MIN.

Let the maximum negative element of a subtree, MAX_{neg}, be the element in the subtree that has the maximum value among all negative elements in the subtree. The minimum positive element, MIN_{pos}, of a subtree is defined similarly. These values are stored in each node of the splay tree. Note that the MAX_{neg} of the root node is the maximum negative element of the permutation, that is, the negative element of the MAX pair of the permutation. The MAX_{neg} of a node is the maximum of the following three: the MAX_{neg} of the left subtree, the MAX_{neg} of the right subtree, and the element in the node if the element is negative. Also notice that whenever the reversed flag of a node is turned on, MAX_{neg} and MIN_{pos} are swapped. Therefore pushing down a reversed flag applies this swap to the children, unless there is a cancellation of flags.

A splay operation performs a series of rotations based on the structure of the tree and the index being queried. Each rotation changes at most three edges of a connected subtree while maintaining the binary search tree property. MAX_{neg} can be recalculated for only the subtree that is affected, Recall that to perform an inversion $\rho(i, j)$ the splay tree is split into three subtrees which are rejoined after the reversed flag has been set for one of the trees. The value of MAX_{neg} can be kept for each of the subtrees in the process by simply checking the children of the root after each operation.

By maintaining the MAX_{neg} values in this fashion, one can maintain the invariant that the MAX_{neg} of the root node is the maximum negative element of the permutation through any scenario of inversions. Since calculating MAX_{neg} takes $O(1)$ time per node, these modifications do not alter the time complexity of the data structure.

Lemma 5.3.3. *For any (signed) permutation of size n, there exists a data structure that handles an inversion in $O(\log n)$ time while maintaining information about the maximum negative element of the permutation.*

5.3.3 Finding the MAX pair

We now describe how to obtain the elements of the MAX pair in a permutation using the modified data structure described above.

First the maximum negative element of the permutation is located. If the element in a node is not equal to the MAX_{neg} of the node then MAX_{neg} of the node lies in either the left subtree or the right subtree of the node. Therefore starting at the root one can go down the tree looking for the maximum negative element. Reversed flags must be pushed down along the path to ensure that MAX_{neg} values are updated and the correct path is followed.

To find the second element of the MAX pair, a lookup vector of pointers (of n elements) maps each element to the node that contains the element. These pointers do not change throughout the computation and enable constant-time lookup of the node containing the second element of the MAX pair.

5.3.4 Finding the indices of the MAX inversion

In absence of reversed flags, the indices of the MAX inversion can be obtained directly from the current location of the nodes corresponding to the MAX pair. However, the presence of a reversed flag indicates nodes that have outdated indices, forcing additional work to retrieve the correct indices.

The index of a node (with respect to the current state of the permutation) can be calculated using the index of the parent node and the sizes of the left and right subtrees. Thus the current index of a node can be calculated whenever the reversed flag is pushed down from it. The size of the subtree rooted at a node is easily maintained. If the node is a right child, then its index is one more than the sum of its parent's index and the size of the left subtree. If the node is a left child, then its index is one less than the difference of its parent's index and the size of the right subtree. The index of the root is just the size of its left subtree. Thus starting at the root, as the reversed flags are pushed down along any path in the tree, the current indices can be calculated.

As one traverses the tree from the root searching for the maximum negative element, the indices are recalculated. After the node corresponding to the second element in the MAX pair is found using the lookup vector, its updated index can be retrieved by traversing up to the root (using parent pointers) and returning down the same path, pushing down the reversed flags and recalculating indices at each node.

5.3.5 Putting it all together

The previous subsections detail all the steps for performing a MAX inversion. The time complexity of each of these steps is easy to analyze. Pushing down the reversed flag takes $O(1)$ time per node. Thus, finding the maximum negative element and its updated index takes $O(\log n)$ time. Finding the other element of the MAX pair takes $O(1)$ time and obtaining its updated index takes $O(\log n)$ time. Therefore the complexity of finding the two indices (steps 2 and 3 in algorithm MAX) is $O(\log n)$. For each inversion, maintaining MAX_{neg}, MIN_{pos}, MIN_{neg}, and MAX_{pos} in the nodes takes $O(1)$ time during split and join operations, and $O(1)$ time for each rotation in the two splays. Therefore performing the inversion in step 4 of algorithm MAX takes $O(\log n)$ time. So we have proved:

Theorem 5.3.4. *For any signed permutation of size n, a data structure exists that*

- *allows checking whether there exists an oriented inversion in $O(1)$ time,*
- *allows performing a MAX (or MIN) inversion, while maintaining the permutation, in $O(\log n)$ time,*
- *and is of size $O(n)$.*

Theorem 5.3.5. *In the absence of unsafe inversions at any sorting step, algorithm MAX produces an optimal sorting scenario in $O(n \log n)$ time.*

5.4 Bypassing Bad Components

We saw that algorithms MAX and RAND can get stuck at a positive permutation by choosing an unsafe inversion. We offer two strategies for recovery.

5.4.1 Randomized restarts

For algorithm RAND we can simply restart the computation hoping that a better outcome is met in the next run. Indeed, the experiments from Section 5.5 show that, for most permutations, this simple approach suffices. However, this approach cannot always sort a permutation as there exists a family of permutations that it cannot handle. For instance, take the permutation (3 1−4−2): both MAX and MIN inversions are unsafe because they yield the same positive permutation (3 1 2 4); this small example can be extended to any length by appending the requisite number of positive elements.

5.4.2 Recovering from an unsafe inversion: Tannier and Sagot's approach

Tannier and Sagot [87] introduced a powerful approach for finding unsafe inversions and augmenting the sorting scenario till it is optimal. They noticed that it is computationally difficult to detect an unsafe inversion as it is applied; but it is of course trivial to find out that one has occurred when the process is stuck at a positive permutation. Their approach is thus *postmortem*: their algorithm traces the sorting process back to the most recent unsafe inversion and inserts two or more oriented inversions before the unsafe one without invalidating the already computed inversions (this ensures that the sorting scenario grows in every trace-back phase.) After the trace-back, the sorting process continues from the state of the permutation just before the unsafe inversion. The new inversions that are inserted are chosen such that the bad component created by the previous unsafe inversion is no longer created and so, the (previously) unsafe inversion and all the inversions that followed it can be applied again.

They use the *overlap graph* [45] to keep track of the remaining breakpoints (and whether or not they are oriented). Using the overlap graph they can find the most recent unsafe inversion, find and insert more inversions before the unsafe one, and continue sorting without invalidating the inversions that have been applied after the most recent unsafe inversion [87]. However, the process may have to be repeated, as, even after augmenting the sorting scenario, their algorithm may again get stuck at a positive permutation.

5.4.3 Recovering from an unsafe inversion: Our approach

We use a similar idea, but do not maintain the full overlap graph, as it is too expensive to maintain. Denote by p_1 the first positive permutation at which the algorithm gets stuck and by p_i the i^{th} such positive permutation. Recovering from a positive permutation p_i involves three steps: finding the most recent unsafe inversion μ_i, finding and inserting two new oriented inversions before μ_i, and appending inversions without invalidating those oriented inversions that had been applied after μ_i. We describe each of these steps in turn.

Finding the most recent unsafe inversion:

In the trace-back phase, starting at p_i we undo the inversions that have been done so far in order to find the most recent unsafe inversion μ_i. Thus, each inversion undone joins two cycles and an unsafe inversion is an inversion that, when undone, creates a good component from bad component(s). Denote by $\pi \cdot S$ and $\pi \cdot \rho$ the result of applying the inversions from the scenario of inversions S and the single inversion ρ to the permutation π, respectively. *Undoing* the inversion ρ in $\pi \cdot \rho$ refers to performing ρ on $\pi \cdot \rho$ which yields π, and undoing the inversions $S = \rho_1, \rho_2, \ldots, \rho_n$ in $\pi \cdot S$ refers to performing the

inversions of S in the reverse order which yields $\pi \cdot S \cdot S^{-1} = \pi$. The scenario of inversions on input permutation π^0 that results in the positive permutation p_i is denoted by S_i, so $p_i = \pi^0 \cdot S_i$. Let $B(\pi)$ be the set of bad components in permutation π.

Remark 5.4.1. *When undoing inversions from S_i, the most recent unsafe inversion μ_i is the first inversion met that turns an element of $B(p_i)$ from bad to good.*

Finding μ_i is not trivial because framed intervals can be nested. For example the positive permutation (2 3 6 7 4 5 8 9 10 1) has two components: the one framed by the implicit frame elements 0 and 11, and the nested component framed by the elements 3 and 8. Undoing the inversion $\rho(2, 7)$ will leave both bad components intact despite the fact that it occurs within the frame elements of the larger component. Thus, in the trace-back phase, $\rho(2, 7)$ cannot be an unsafe inversion. However, undoing the inversions $\mu(5, 7)$ and $\mu(4, 5)$ will make the inner component good and so these two inversions, had they been the most recent inversion performed, would have also been unsafe. The following remark characterizes undoing an unsafe inversion in terms of the components in $B(p_i)$.

Remark 5.4.2. *An inversion is the most recent unsafe inversion μ_i if and only if it is the most recent inversion to change the indices of a proper nonempty subset of the elements from some component in $B(p_i)$.*

The trace-back algorithm is thus as follows: start undoing the inversion scenario S_i, checking after each inversion whether there exist components in $B(p_i)$ with both changed and unchanged indices and stop when an unsafe inversion is found. We describe how to do this by keeping an ancillary splay tree where nodes represent adjacencies in the permutation rather than permutation elements.

The heart of the problem deals with how non-breakpoints interact with the undoing of unsafe inversions. We present a labeling of the ancillary tree so that the safety check can be carried out by a constant-time comparison on the two adjacencies broken by an inversion. Each adjacency has a label indicating the innermost overlying component along with a second label that is non-null only for non-breakpoints. For a given component, each group of consecutive non-breakpoints (ignoring nested components) gets a unique second label. Thus an inversion displaces only a fraction of the elements of a component if and only if both broken adjacencies are labeled as non-breakpoints with the same component and non-breakpoint labels.

In the example, the permutation (2 3 6 7 4 5 8 9 10 1) has component label X for adjacencies (0,2), (2,3), (8,9), (9,10), (10,1), and (1,11), and component label Y for the others. The non-breakpoint labels are the same for (2,3), (8,9), and (9,10), but different between (6,7) and (4,5). Inversion $\rho(2, 7)$ acts upon non-breakpoints with the same pair of labels while inversion $\mu(5, 7)$ acts upon non-breakpoints with different component labels and $\mu(4, 5)$ acts upon non-breakpoints with different non-breakpoint labels.

We can list the endpoints of the components of a permutation in linear time [8, 12]. A simple traversal of the permutation, keeping one stack for each label, can perform the node labeling described above. Thus the setup of the ancillary tree can be done in $O(n)$ time. Let $S1_i$ be the scenario of inversions applied before μ_i in S_i and $S2_i$ be the scenario of inversions applied after $S1_i$ (including μ_i) in S_i. Each safe inversion in $S2_i$ that is undone will cost $O(\log n)$ time so the total cost for finding a most recent unsafe inversion is $O(n + |S2_i| \log n)$.

Inserting oriented inversions before the unsafe inversion:

Recall that μ_i is the most recent unsafe inversion in the scenario S_i. Theorem 3 in Tannier, Bergeron, and Sagot [86] shows that there always exists two oriented inversions $\nu1_i$ and $\nu2_i$ that can be applied before the inversion μ_i in S_i. According to [86], inversions $\nu1_i$ and $\nu2_i$ must have the following properties:

- the span of $\nu1_i$ overlaps the span of μ_i, and

- either the span of $\nu 2_i$ overlaps the span of $\nu 1_i$ and does not overlap the span of μ_i, or the span of $\nu 2_i$ overlaps the span of μ_i and does not overlap the span of $\nu 1_i$.

In the following we show how to find a $\nu 1_i$ and $\nu 2_i$ with these properties in time proportional to the size of the bad component that we created.

Lemma 5.4.3. *Given an unsafe oriented inversion μ_i and the bad component b of size m created by μ_i, one can always find, in $O(m)$ time, inversions $\nu 1_i$ and $\nu 2_i$ such that $S1_i \cdot \nu 1_i \cdot \nu 2_i \cdot \mu_i$ is a valid scenario of oriented inversions.*

Proof. We proceed by finding oriented $\nu 1_i$ and $\nu 2_i$ with the properties listed above. A bad component could have been created in one of three ways when μ_i was applied. Without loss of generality we ignore the symmetric counterpart to the first case below (both cannot happen at once). We also ignore the inverted versions of each case where the hurdle created has only negative elements. This leaves us with three cases to consider.

- $(\pm\pi_0 \ldots +\mathbf{l}+\mathbf{x_1} \ldots +\mathbf{x_s} \underbrace{\pm\pi_x \ldots -\mathbf{r}-\mathbf{x_{k-1}} \ldots -\mathbf{x_{s+1}}} \pm\pi_{x+1} \ldots \pm\pi_n)$
  where the braced inversion creates the bad component
  $b = +l+x_1 \ldots +x_s+x_{s+1} \ldots +x_{k-1}+r$.
- $(\pm\pi_0 \ldots +\mathbf{l}+\mathbf{x_1} \ldots +\mathbf{x_l} \underbrace{-\mathbf{x_{r-1}} \ldots -\mathbf{x_{l+1}}} +\mathbf{x_r} \ldots \mathbf{x_{k-1}}+\mathbf{r} \ldots \pm\pi_n)$
  where the braced inversion creates the bad component
  $b = +l+x_1 \ldots +x_l+x_{l+1} \ldots +x_{r-1}+x_r \ldots +x_{k-1}+r$.
- The third case is the same as the first, except that one or more bad components are created which span the internal component
  $+l\pm x_1 \ldots \pm x_s \pm x_{s+1} \ldots \pm x_{k-1}+r$.

For the first case, write $L = +l+x_1 \ldots +x_s$ and $R = -r-x_{k-1} \ldots -x_{s+1}$ and examine the substrings L and R. Since the component $(l, \ldots, r)$ is a bad component, there must exist an element t in L such that either $t+1$ or $t-1$ is negative and not in L. Assume w is the first such element we encounter by scanning from $+l$ to $+x_s$. We locate the rightmost $-(w-1)$ or $-(w+1)$ in R by scanning from $-x_{s+1}$ to $-r$. Now, there are two possibilities.

1. The rightmost element is $-(w-1)$. We have $w > l+1$ and thus $(w, -(w-1))$ is an oriented pair; consequently, there exists an oriented inversion, $\nu 1_i$, which is different from μ_i. Now consider the position of those elements with absolute values between (and including) l and $w-1$. Let y be the element with the smallest value that does not appear to the left of w in L (such an element must exist because l is to the left of w but $w-1$ is in R). Thus $y-1$ must appear to the left of w in L. Not that y cannot be in R, as this would contradict the fact that w is the leftmost element in L with $-(w+1)$ or $-(w-1)$ in R. Thus y must be in L and to the right of w. After applying $\nu 1_i$, we will have the oriented pair $(y-1, -y)$, and consequently, another oriented inversion $\nu 2_i$. Notice that the span of $\nu 1_i$ overlaps the span of μ_i and the span of $\nu 2_i$ overlaps the span of $\nu 1_i$ but not that of μ_i.
2. The rightmost element is $-(w+1)$. Note that $(w, -(w+1))$ is an oriented pair, so that there exists an oriented inversion $\nu 1_1$. This inversion must be different from μ_i as otherwise L would a bad component in itself. Now we examine the substring to the right of w in L. Let z be the element with the largest absolute value in that substring. Consider the following two cases:
   (a) The absolute value of z is less than w: we consider the elements with absolute values in the interval $[l, z]$. Let y be the element with the largest absolute value in $[l, z]$ that appears to the left of w (such an element must exist because l is to the left of w but z is to the right

of w in L). $y+1$ cannot be in R, as this would contradict the fact that w is the leftmost element in L with $-(w+1)$ or $-(w-1)$ in R. Thus $y+1$ must be in L and to the right of w. After applying $\nu 1_i$, we will have the oriented pair $(y, -(y+1))$, and consequently, another oriented inversion $\nu 2_i$. Notice that the span of $\nu 1_i$ overlaps the span of μ_i and the span of $\nu 2_i$ overlaps the span of $\nu 1_i$ but not that of μ_i.

(b) The absolute value of z is larger than $w+1$: We consider the elements with absolute values in $[z, r]$. Let y be the element with the largest absolute value in $[z, r]$ that appears to the left of $-(w+1)$ in R (such an element must exist because r is to the left of $-(w+1)$ in R but z is in L). $y-1$ cannot be to the left of w in L, as this would contradict the fact that w is the leftmost element in L with $-(w+1)$ or $-(w-1)$ in R. Thus $y-1$ must be either to the right of w in L or to the right of $-(w+1)$ in R. If $y-1$ is to the right of w in L, the oriented pair $(-(y-1), y)$ defines the oriented inversion $\nu 2_i$. Notice that the spans of $\nu 1_i$ and $\nu 2_i$ overlap the span of μ_i but $\nu 1_i$ and $\nu 2_i$ do not overlap. If $y-1$ is to the right of $-(w+1)$ in R, after applying $\nu 1_i$, we will have the oriented pair $(y, -(y-1))$, and consequently, another oriented inversion $\nu 2_i$. In this case the span of $\nu 1_i$ overlaps the span of μ_i and the span of $\nu 2_i$ overlaps the span of $\nu 1_i$ but not that of μ_i.

For the second case (where the span of the unsafe inversion is a proper subset of the span of the bad component), write $L = {}^+l {}^+x_1 \ldots {}^+x_l$, $M = {}^-x_{r-1} \ldots {}^-x_{l+1}$ and $R = {}^-r {}^-x_{k-1} \ldots {}^-x_{s+1}$. In substrings L and R, there must exist one element t such that $-(t+1)$ or $-(t-1)$ is in M and the inversion induced by this pair is not μ_i. Thus, the oriented pair $(t, -(t-1))$ or $(t, -(t+1))$ defines the oriented inversion $\nu 1_i$. Since $\nu 1_i$ is different from μ_i, there will be some negative elements after applying $\nu 1_i$; assume that the maximum negative element among them is $-y$. Thus, $y-1$ must be positive and the oriented pair $(-y, {}^+(y-1))$ defines the other oriented inversion $\nu 2_i$. It is easy to verify that these inversions have the required properties.

For the third case, if the innermost component is also bad then we can find the two new inversions using the first case. If it is good, then we find the inversions using the logic of the second case.

The linear-time complexity can be achieved by using a lookup vector that maps each element to its index in the permutation. (This is created in the beginning and maintained throughout the sorting process.) Thus, for the first case, with a single scan of L, we can find w and $-(w-1)$ and with another scan of elements between l and $w-1$ in the lookup vector, the pair $((y-1), {}^-y)$. The other cases can be analyzed similarly. Note that in no case do we need to scan any element that is not a part of b. Thus the inversions $\nu 1_i$ and $\nu 2_i$ can be found in $O(m)$ time. □

Appending inversions to the sorting scenario:

To reiterate, after we get the permutation $q_i = \pi \cdot S1_i$ we apply the scenario $\nu 1_i \cdot \nu 2_i \cdot \mu_i$ on q_i. Now we would like to ensure that some scenario of inversions S'_i we append after μ_i does not invalidate the scenario $S2_i$ (i.e. $S1_i \cdot \nu 1_i \cdot \nu 2_i \cdot \mu_i \cdot S'_i \cdot S2_i$ is a valid scenario of oriented inversions). Call $G(\pi)$ the set of good components for a permutation π. A slight extension to the proof of Theorem 3 from [86] shows the following:

Lemma 5.4.4. *The set of good components $G(q_i \cdot \mu_i)$ is identical to $G(q_i \cdot \nu 1_i \cdot \nu 2_i \cdot \mu_i)$.*

This tells us that inversions associated with $S2_i$ will be part of distinct components in $q_i \cdot \nu 1_i \cdot \nu 2_i \cdot \mu_i$ and that these components will be exactly as they are in $q_i \cdot \mu_i$. So any scenario of inversions S'_i will only heal breakpoints on components other than those of $S2_i$.

We continue by showing how to compute an S'_i, ensuring that we only work on the components of $G(q_i \cdot \mu_i)$. We achieve this by renaming the permutation q_i in the following way. By definition, $q_i \cdot \mu_i$ has at least one bad component created by μ_i along with a possibly nonempty set $G(q_i \cdot \mu_i)$ The inversions

that sort the components of $G(q_i \cdot \mu_i)$ correspond exactly to the scenario $S2_i$. Thus, our desired scenario S'_i of inversions should only displace (if at all) such components without affecting their structure.

Say there is a component c of length m with left frame element l. The *canonical form* $\hat{c}$ of c is a permutation of length m with $\hat{c}[i] = c[i] - l + 1$, $1 \leq i \leq m$, where $p[i]$ denotes the ith element of a permutation p. Components c and d are said to be *structurally equivalent* if and only if we have $\hat{c} = \hat{d}$.

Lemma 5.4.5. *Let q_i be a permutation without a bad component and μ_i be an inversion such that $q_i \cdot \mu_i$ has at least one bad component and a set of good components $G(q_i \cdot \mu_i)$. There exists a q'_i where any scenario S'_i that sorts q'_i to the identity, when applied to q_i, will result in a permutation whose only components are those in $G(q_i \cdot \mu_i)$.*

Proof. Rename the permutation $q_i \cdot \mu_i$ such that all breakpoints from components in $G(q_i \cdot \mu_i)$ become non-breakpoints and then undo μ_i to get q'_i. Note that this renaming leaves one structurally equivalent bad component in place of each bad component, so that the renaming is unique. An inversion scenario that sorts q'_i to the identity heals all breakpoints from the bad components in $q_i \cdot \mu_i$; moreover, it does not act upon any adjacency or heal any breakpoint from components in $G(q_i \cdot \mu_i)$ due to the nesting property of FCIs. □

For example, take q_i = (2 3 6 7 4 −8 −5 −9 10 −1) and $\mu_i = \mu(6, 7)$. Now $q_i \cdot \mu_i$ is (2 3 6 7 4 5 8 −9 10 −1), so that $G(q_i \cdot \mu_i)$ is comprised of the components framed by the pair (of frame elements) (0,11) and the pair (8,10). $q_i \cdot \mu_i$ is renamed to $q'_i \cdot \mu_i$ = (1 2 5 6 3 4 7 8 9 10), yielding q'_i = (1 2 5 6 3 −7 −4 8 9 10). The sorting scenario $S'_i = (\rho(3, 6), \rho(3, 4), \rho(4, 7))$ for q'_i can be applied to q_i to get (2 3 4 5 6 7 8 −9 10 −1).

Lemma 5.4.6. *Given a permutation p with a set of bad components $B(p)$, permutation p' that has one structurally equivalent bad component in place of each $b \in B(p)$ and only non-breakpoints everywhere else, can be constructed in linear time.*

Proof. If an adjacency is not part of a bad component then label it with a null value; otherwise label it by the bad component of which it is part of. Also label adjacencies with the left and right endpoints of each component, which can be done in linear time [8, 12]. We use a stack R, the top of which we denote by $top(R)$. Perform the following steps until the end of the permutation is reached, i.e., until we have $i = n$.

1. $p[0] = 0$, $i = 1$.
2. Label each element $p'[i]$ with the value $p'[i-1] + 1$, incrementing i until the adjacency ($p[i-1]$ $p[i]$) corresponds to a bad component.
3. If the adjacency ($p[i-1]$ $p[i]$) is a left endpoint, then push onto R the value $p[i-1] - p'[i-1]$. Go to step 4.
4. Do this, incrementing i, until an adjacency with a different component label is reached: Label each element $p'[i] = p[i] - top(R)$ and if it is a right endpoint, then pop the $top(R)$. Go to step 2.

□

Overall running time analysis:

We call this algorithm, with the recovery phase included, MAX-RECOVER or RAND-RECOVER, depending on whether algorithm MAX or algorithm RAND is used in the forward-sorting phase. If algorithm MAX or RAND gets stuck at a positive permutation p_i, we proceed by undoing inversions until a permutation q_i is found such that $q_i \cdot \mu_i$ has fewer bad components than q_i. Finding such a q_i

Table 5.1: *The failure rates for MAX, RAND and RAND+RESTART*

Length	100	200	500	1,000	2,000	5,000	10,000	20,000
MAX	39.5%	38.9%	39.0%	39.1%	39.3%	39.3%	39.3%	39.2%
RAND	39.0%	39.2%	39.5%	39.5%	39.6%	39.5%	39.6%	39.5%
RAND-RESTART	17.2 %	17.1 %	16.8 %	16.4 %	16.3 %	16.2 %	16.0 %	16.0 %

Table 5.2: *Number of recovery steps (k) for MAX-RECOVER: Average and Standard Deviation*

Length	100	200	500	1,000	2,000	5,000	10,000	20,000
AVE(k)	0.513	0.518	0.522	0.524	0.524	0.525	0.524	0.525
SD(k)	0.765	0.770	0.772	0.774	0.773	0.775	0.774	0.777

and μ_i alone takes $O(n + |S2_i| \log n)$ time. The inversions undone in this step are not discarded as they can be applied after inserting at least two more inversions. Notice that each inversion undone in the trace-back must be done or undone on a splay tree at most three times and that $S2_i$ and $S2_j$ for any two p_i and p_j, $i \neq j$, will be disjoint. Thus the $O(n \log n)$ term describes the amount of time spent for undoing inversions over the entire course of the algorithm and just a linear amount of work beyond that must be done in each recovery phase.

Theorem 5.4.7. *The running time of MAX-RECOVER or RAND-RECOVER is $O(n \log n + kn)$ where k is the total number of unsafe inversions performed in the algorithm.*

In Section 5.5 we show strong empirical evidence that, on random permutations of length n, the average value and standard deviation of k remain constant (about $\frac{1}{2}$) even as n grows very large, leading us to conjecture that these algorithms sort almost all permutations in $O(n \log n)$ time. In the worst case, however, RAND-RECOVER and MAX-RECOVER can use $\Theta(n^2)$ time, as in the following family of permutations: build a permutation of length n by starting with the identity permutation of length n mod 5 as the first block, followed by $n/5$ copies of the block $i(i+3)(i+1)^-(i+4)^-(i+2)(i+5)$, each of which shares its first element with the last element of the preceding block.

5.5 Experimental Results

We present experimental results for algorithms MAX, RAND, MAX-RECOVER and RAND-RECOVER. All of the experiments are on random permutations of length 100, 200, 500, 1000, 2000, 5000, $10,000$ and $20,000$. For each length, we tested our algorithms on $1,000,000$ permutations.

Table 5.1 lists the failure rates for algorithm MAX and algorithm RAND. Algorithm MAX and algorithm RAND produce a full sorting scenario with frequency 61%. We also include the failure rates for RAND-RESTART: the simple heuristic that runs RAND on the input permutation a second time if it fails to sort at the first attempt. The failure rate for RAND-RESTART reduces to 16% ($\approx 0.39 \times 0.39$), which suggests that the two runs are independent with respect to the failure rate.

Tables 5.2 and 5.3 summarize the details of the number of recovery steps, k, that we observe in algorithms MAX-RECOVER and RAND-RECOVER. The average value and the standard deviation of k remain constant as n grows. Figure 5.1 shows the distribution of k for MAX-RECOVER on random permutations of length $10,000$. This figure is representative of the observed distribution for the other lengths as well. The similarity to the inverse exponential function suggests that the upper bound for the average value of k is a constant.

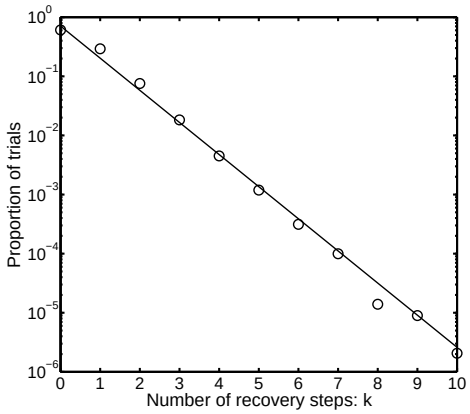


Figure 5.1: The distribution of k for MAX-RECOVER on random permutations of length $10,000$.

Table 5.3: *Number of recovery steps (k) for RAND-RECOVER: Average and Standard Deviation*

Length	100	200	500	1,000	2,000	5,000	10,000	20,000
AVE(k)	0.485	0.489	0.492	0.493	0.495	0.495	0.495	0.499
SD(k)	0.690	0.694	0.697	0.697	0.698	0.698	0.698	0.699

5.6 Conclusions

We have given two new algorithms for sorting signed permutations by inversions, one a fast heuristic that works on most permutations, the other a deterministic algorithm that sorts all permutations and takes $O(n \log n)$ time on almost all of them. We have given the results of very extensive experimentation to confirm these claims. We have thus taken a major step towards a final resolution of the sorting problem: we believe tools presented here will eventually lead to a proof that we can sort most permutations in $O(n \log n)$ time. Future work may also include design of an algorithm to deal with the few remaining permutations that require more time.

Chapter 6

Conclusion

We have described improvements in three main areas of *comparative genomics*: genomic distances in the presence of duplicate genes, ancestral genome inference, and the classical problem of sorting signed permutations by inversions. Behind all of the work in the first two areas is the simplifying assumption that certain hard-to-compare pairs of genomes are rarely encountered. In Chapter 2 we confirmed the validity of this assumption by showing that hurdles and fortresses occur in permutations with probability $\Theta(n^{-2})$ and $\Theta(n^{-15})$ respectively. This finished the fundamental work that was started by Alberto Caprara [24] ten years ago.

Foundational work for computing evolutionary distances between genomes with unequal and duplicated gene content was presented in Chapter 3. In a restricted setting where only one of the two genomes being compared contains duplicated genes we gave an approximation algorithm with constant error bound. We expect that this bound may be improved by leveraging the dichotomy between the number of cycles and the number of deletions[1] but do not address this here. We also found nontrivial but detectable conditions under which we can compute the minimum evolutionary distance between two genomes. In the process we built a machinery that facilitates a reduction showing almost every known problem[2] related to distance minimization with duplicate genes (those define in Section 3.2.1) to be NP-Hard.

The methodology of the aforementioned approximation algorithm was extended to be used in the general setting — where inversions, deletions, and unrestricted (duplicating) insertions — were considered. We found through simulation studies that this extension tracked the true evolutionary distance quite well, and that simulated trees that evolved through the supposed model could be reconstructed very accurately. Further, on the one real dataset we tried — the dataset of 13 bacteria from Earnest-DeYoung [31] with genome sizes ranging from 1,000 to over 5,000 genes and gene families of up to 70 members — we reconstructed the true (accepted by the biologists) tree almost exactly.

In the process of computing these distances, we always create a mapping from the genes of some family in one genome to those from the same family in another. This mapping is of particular interest because it can give insight into the evolutionary relationship of two genes; the most parsimonious assignment of duplicate genes could indicate which genes originated during some duplication event. Sometimes such genes are called *positional homologs*. Information of this sort may be instrumental in identifying *orthology* between genes [38, 4]. Future work must draw this connection between positional homologs and orthology, as well as reconcile the relationship between positional homology and gene function.

[1] For example, take the permutations A = (1 2 3 4) and B = (5 3 4 3 -1 -2). If the first 3 of B is chosen then there are 2 deletions and 2 inversions necessary but choosing the second 3, while increasing the number of inversions by 2, reduces the number of deletions necessary. We conjecture that under most circumstances any assignment would be within three halves of the optimal.

[2] The result does not apply to problems that concern genome halving/doubling [34].

In Chapter 4 we showed that there are certain conditions under which ancestral gene sequences can be reliably reconstructed. But the questions surrounding ancestral reconstruction seem to be numerous, the most striking of which are: if ancestral sequences can be inferred, what will biologists find most interesting about these sequences; can knowing the ancestral sequence give insight into the regulatory interdependence of a group of genes; and if some full sequences cannot be reconstructed, what other approximations can we settle for? Computational methods have only started to scratch the surface in this field.

Finally, we gave important steps towards a resolution of the sorting by inversions problem. In particular, we showed how to find an oriented inversion in constant time while maintaining a data structure first applied in this context by Kaplan and Verbin. We also showed that we can recover from an unsafe inversion in linear time without disturbing already computed inversions, all without the knowledge of the overlap graph. Both pave the way for many avenues that we are currently exploring to finalize an algorithm that provably runs in $O(n \log n)$ time for almost all permutations.

Bibliography

[1] Y. Ajana, J.-F. Lefebvre, E.R.M. Tillier, and N. El-Mabrouk. Exploring the set of all minimal sequences of reversals - an application to test the replication-directed reversal hypothesis. In *WABI '02: Proceedings of the Second International Workshop on Algorithms in Bioinformatics*, pages 300–315, London, UK, 2002. Springer-Verlag.

[2] D.J. Aldous. Stochastic models and descriptive statistics for phylogenetic trees, from yule to today. *Statistical Science*, 16:23–34, 2001.

[3] M.A. Alekseyev and P.A. Pevzner. Whole genome duplications, multi-break rearrangements, and genome halving problem. In *SODA '07: Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 665–679, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[4] A.M. Altenhoff and C. Dessimoz. Phylogenetic and functional assessment of orthologs inference projects and methods. *PLoS Comput Biol*, 5(1):e1000262, 2009.

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

[6] S. Angelov, B. Harb, S. Kannan, S. Khanna, and J. Kim. Efficient enumeration of phylogenetically informative substrings. In *Proc. 10th Ann. Int'l Conf. Comput. Mol. Biol. (RECOMB'06)*, pages 248–264, 2006.

[7] W. Arndt and J. Tang. Improving inversion median computation using commuting reversals and-cycle information. In *Proc. 5th Workshop Comp. Genomics (RECOMB-CG'07)*, volume 4751 of *Lecture Notes in Computer Science*, pages 30–44. Springer Verlag, Berlin, 2007.

[8] D.A. Bader, B.M.E. Moret, and M. Yan. A fast linear-time algorithm for inversion distance with an experimental comparison. *J. Comput. Biol.*, 8(5):483–491, 2001.

[9] D.A. Bader, B.M.E. Moret, and M. Yan. A linear-time algorithm for computing inversion distance between signed permutations with an experimental study. *J. Comput. Biol.*, 8(5):483–491, 2001. A preliminary version appeared in WADS'01, pp. 365–376.

[10] V. Bafna and P.A. Pevzner. Genome rearrangements and sorting by reversals. In *SFCS '93: Proceedings of the Proceedings of 1993 IEEE 34th Annual Foundations of Computer Science*, pages 148–157, Washington, DC, USA, 1993. IEEE Computer Society.

[11] A. Bergeron, C. Chauve, T. Hartman, and K. Saint-Onge. On the properties of sequences of reversals that sort a signed permutation. *JOBIM*, pages 99–108, June 2002.

[12] A. Bergeron, S. Heber, and J. Stoye. Common intervals and sorting by reversals: a marriage of necessity. In *Proc. 2nd European Conf. Comput. Biol. ECCB'02*, pages 54–63, 2002.

[13] A. Bergeron and J. Stoye. On the similarity of sets of permutations and its applications to genome comparison. In *Proc. 9th Int'l Conf. Computing and Combinatorics (COCOON'03)*, volume 2697 of *Lecture Notes in Computer Science*, pages 68–79. Springer Verlag, Berlin, 2003.

[14] P. Berman and M. Karpinski. On some tighter inapproximability results. Technical Report 29, University of Trier, Trier, Germany, 1998. http://www.eccc.uni-trier.de/.

[15] G. Blin, C. Chauve, and G. Fertin. Genes order and phylogenetic reconstruction: Application to γ-proteobacteria. In *Proc. 2nd Workshop Comp. Genomics (RECOMB-CG'05)*, volume 3388 of *Lecture Notes in Computer Science*, pages 11–20. Springer Verlag, Berlin, 2004.

[16] A. Bouchet. Reducing prime graphs and recognizing circle graphs. *Combinatorica*, 7(3):243–254, 1987.

[17] G. Bourque and P.A. Pevzner. Genome-scale evolution: reconstructing gene orders in the ancestral species. *Genome Research*, 12:26–36, 2002.

[18] G. Bourque, G. Tesler, and P.A. Pevzner. The convergence of cytogenetics and rearrangement-based models for ancestral genome reconstruction. *Genome Research*, 16:311–313, 2006.

[19] M.D.V. Braga, M. Sagot, C. Scornavacca, and E. Tannier. The Solution Space of Sorting by Reversals. In *Bioinformatics Research and Applications: Proceedings from ISBRA 2007*. Springer, 2007.

[20] D. Bryant. The complexity of calculating exemplar distances. In D. Sankoff and J. Nadeau, editors, *Comparative Genomics: Empirical and Analytical Approaches to Gene Order Dynamics, Map Alignment, and the Evolution of Gene Families*, pages 207–212. Kluwer Academic Publishers, Dordrecht, NL, 2000.

[21] D. Bryant, V. Berry, T. Jiang, P. Kearney, M. Li, T. Wareham, and H. Zhang. A practical algorithm for recovering the best supported edges of an evolutionary tree. In *Proc. 11th Ann. ACM/SIAM Symp. Discrete Algs. (SODA'00)*, pages 287–296. ACM Press, New York, 2000.

[22] A. Caprara. Sorting by reversals is difficult. In *Proc. 1st Ann. Int'l Conf. Comput. Mol. Biol. (RECOMB'97)*, pages 75–83. ACM Press, New York, 1997.

[23] A. Caprara. Formulations and hardness of multiple sorting by reversals. In *Proc. 3rd Ann. Int'l Conf. Comput. Mol. Biol. (RECOMB'99)*, pages 84–93. ACM Press, New York, 1999.

[24] A. Caprara. On the tightness of the alternating-cycle lower bound for sorting by reversals. *J. Combin. Optimization*, 3:149–182, 1999.

[25] A. Caprara. Sorting permutations by reversals and eulerian cycle decompositions. *SIAM J. Discrete Math.*, 12(1):91–110, 1999.

[26] X. Chen, J. Zheng, Z. Fu, P. Nan, Y. Zhong, S. Lonardi, and T. Jiang. Computing the assignment of orthologous genes via genome rearrangement. In *Proc. 3rd Asia Pacific Bioinformatics Conf. (APBC'05)*, pages 363–378. Imperial College Press, London, 2005.

[27] M. Chrobak, P. Kolman, and J. Sgall. The greedy algorithm for the minimum common string partition problem. *ACM Transactions on Algorithms*, 1(2):350–366, 2005.

[28] M.E. Cosner, R.K. Jansen, B.M.E. Moret, L.A. Raubeson, L. Wang, T. Warnow, and S.K. Wyman. A new fast heuristic for computing the breakpoint phylogeny and experimental phylogenetic analyses of real and synthetic data. In *Proc. 8th Int'l Conf. on Intelligent Systems for Mol. Biol. (ISMB'00)*, pages 104–115, 2000.

[29] M.E. Cosner, R.K. Jansen, B.M.E. Moret, L.A. Raubeson, L.-S. Wang, T. Warnow, and S.K. Wyman. A new fast heuristic for computing the breakpoint phylogeny and a phylogenetic analysis of a group of highly rearranged chloroplast genomes. In *Proc. 8th Int'l Conf. on Intelligent Systems for Mol. Biol. (ISMB'00)*, pages 104–115, 2000.

[30] J. Earnest-DeYoung, E. Lerat, and B.M.E. Moret. Reversing gene erosion: reconstructing ancestral bacterial genomes from gene-content and gene-order data. In *Proc. 4th Int'l Workshop Algs. in Bioinformatics (WABI'04)*, volume 3240 of *Lecture Notes in Computer Science*, pages 1–13. Springer Verlag, Berlin, 2004.

[31] J.V. Earnest-DeYoung. Reversing gene erosion: Reconstructing ancestral bacterial gene orders. Master's thesis, University of New Mexico, 2004.

[32] N. El-Mabrouk. Genome rearrangement by reversals and insertions/deletions of contiguous segments. In *Proc. 11th Ann. Symp. Combin. Pattern Matching (CPM'00)*, volume 1848 of *Lecture Notes in Computer Science*, pages 222–234. Springer Verlag, Berlin, 2000.

[33] N. El-Mabrouk. Reconstructing an ancestral genome using minimum segments duplications and reversals. *J. Comput. Syst. Sci.*, 65:442–464, 2002.

[34] N. El-Mabrouk, J.H. Nadeau, and D. Sankoff. Genome halving. In *CPM*, pages 235–250, 1998.

[35] N. Eriksen. Reversal and transposition medians. *Theor. Comput. Sci.*, 374(1-3):111–126, 2007.

[36] S. Even and A. Itai. Queues, stacks, and graphs. In *Theory of Machines and Computations*. Academic Press, 1971.

[37] S. Even, A. Pnueli, and A. Lempel. Permutation graphs and transitive graphs. *JACM*, 19(3):400–410, 1972.

[38] W.M. Fitch. Distinguishing homologous from analogous proteins. *Syst Zool*, 19:99–113, 1970.

[39] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman & Co., New York, NY, USA, 1979.

[40] F. Gavril. Algorithms for a maximum clique and a maximum independent set of a circle graph. *Networks*, 3:261–273, 1973.

[41] A. Goldstein, P. Kolman, and J. Zheng. Minimum common string partition problem: Hardness and approximations. *Electr. J. Comb.*, 12, 2005.

[42] S. Hannenhalli and P.A. Pevzner. Transforming cabbage into turnip (polynomial algorithm for sorting signed permutations by reversals). In *Proc. 27th Ann. ACM Symp. Theory of Comput. (STOC'95)*, pages 178–189. ACM Press, New York, 1995.

[43] S. Hannenhalli and P.A. Pevzner. Transforming mice into men (polynomial algorithm for genomic distance problems). In *Proc. 36th Ann. IEEE Symp. Foundations of Comput. Sci. (FOCS'95)*, pages 581–592. IEEE Press, Piscataway, NJ, 1995.

[44] S.B. Heard. Patterns in phylogenetic tree balance with variable and evolving speciation rates. *Evol.*, 50:2141–2148, 1996.

[45] H. Kaplan, R. Shamir, and R.E. Tarjan. Faster and simpler algorithm for sorting signed permutations by reversals. *SIAM J. Computing*, 29(3):880–892, 1999.

[46] H. Kaplan and E. Verbin. Efficient data structures and a new randomized approach for sorting signed permutations by reversals. In *Proc. 14th Ann. Symp. Combin. Pattern Matching (CPM'03)*, volume 2676 of *Lecture Notes in Computer Science*, pages 170–185. Springer Verlag, Berlin, 2003.

[47] J.D. Kececioglu and D Sankoff. Exact and approximation algorithms for the inversion distance between two chromosomes. In *CPM*, pages 87–105, 1993.

[48] J.D. Kececioglu and D Sankoff. Efficient bounds for oriented chromosome inversion distance. In *CPM*, pages 307–325, 1994.

[49] P. Kolman and T. Walen. Approximating reversal distance for strings with bounded number of duplicates. *Discrete Applied Mathematics*, 155(3):327–336, 2007.

[50] P. Kolman and T. Walen. Reversal distance for strings with duplicates: Linear time approximation using hitting set. *Electr. J. Comb.*, 14(1), 2007.

[51] B. Larget, D.L. Simon, J.B. Kadane, and D. Sweet. A bayesian analysis of metazoan mitochondrial genome arrangements. *Mol. Biol. Evol.*, 22(3):486–495, 2005.

[52] D.A. Liberles, editor. *Ancestral Sequence Reconstruction*. Oxford University Press, UK, 2007.

[53] J. Ma, L. Zhang, B. Suh, B. Raney, R. Burhans, W. Kent W, M. Blanchette, D. Haussler, and W. Miller. Reconstructing contiguous regions of an ancestral genome. *Genome Research*, 16:1557–1565, 2006.

[54] M. Marron, K.M. Swenson, and B.M.E. Moret. Genomic distances under deletions and insertions. *Theor. Computer Science*, 325(3):347–360, 2004.

[55] I. Miklós and J. Hein. Genome rearrangement in mitochondria and its computational biology. In *Proc. 2nd Workshop Comp. Genomics (RECOMB-CG'05)*, volume 3388 of *Lecture Notes in Computer Science*, pages 85–96. Springer Verlag, Berlin, 2004.

[56] I. Miklós, B. Mélykúti, and K.M. Swenson. The metropolized partial importance sampling mcmc mixes slowly on minimal reversal rearrangement paths. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 0(0):0–0, 2009.

[57] B.M.E. Moret, A.C. Siepel, J. Tang, and T. Liu. Inversion medians outperform breakpoint medians in phylogeny reconstruction from gene-order data. In *Proc. 2nd Int'l Workshop Algs. in Bioinformatics (WABI'02)*, volume 2452 of *Lecture Notes in Computer Science*, pages 521–536. Springer Verlag, Berlin, 2002.

[58] B.M.E. Moret and T. Warnow. Advances in phylogeny reconstruction from gene order and content data. In E.A. Zimmer and E.H. Roalson, editors, *Molecular Evolution: Producing the Biochemical Data, Part B*, volume 395 of *Methods in Enzymology*, pages 673–700. Elsevier, 2005.

[59] B.M.E. Moret, S.K. Wyman, D.A. Bader, T. Warnow, and M. Yan. A new implementation and detailed study of breakpoint analysis. In *Proc. 6th Pacific Symp. on Biocomputing (PSB'01)*, pages 583–594. World Scientific Pub., 2001.

[60] L. Nakhleh, B.M.E. Moret, U. Roshan, K. St. John, and T. Warnow. The accuracy of fast phylogenetic methods for large datasets. In *Proc. 7th Pacific Symp. on Biocomputing (PSB'02)*, pages 211–222. World Scientific Pub., 2002.

[61] A. Ouangraoua and A. Bergeron. Parking functions, labeled trees and dcj sorting scenarios. *CoRR*, abs/0903.2499, 2009.

[62] W.R. Pearson and D.J. Lipman. Improved tools for biological sequence comparison. *Proc. Nat'l Acad. Sci., USA*, 85:2444–2448, 1988.

[63] I. Pe'er and R. Shamir. The median problems for breakpoints are NP-complete. *Elec. Colloq. on Comput. Complexity*, 71, 1998.

[64] D.R. Robinson and L.R. Foulds. Comparison of phylogenetic trees. *Mathematical Biosciences*, 53:131–147, 1981.

[65] D. Sankoff. Edit distance for genome comparison based on non-local operations. In *Proc. 3rd Ann. Symp. Combin. Pattern Matching (CPM'92)*, volume 644 of *Lecture Notes in Computer Science*, pages 121–135. Springer Verlag, Berlin, 1992.

[66] D. Sankoff. Genome rearrangement with gene families. *Bioinformatics*, 15(11):990–917, 1999.

[67] D. Sankoff and M. Blanchette. The median problem for breakpoints in comparative genomics. In *Proc. 3rd Int'l Conf. Computing and Combinatorics (COCOON'97)*, volume 1276 of *Lecture Notes in Computer Science*, pages 251–264. Springer Verlag, Berlin, 1997.

[68] D. Sankoff, G. Sundaram, and J. Kececioglu. Steiner points in the space of genome rearrangements. *Int'l J. Foundations of Computer Science*, 7:1–9, 1996.

[69] J.C. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology*. PWS Publishers, Boston, MA, 1997.

[70] A.C. Siepel and B.M.E. Moret. Finding an optimal inversion median: Experimental results. In *Proc. 1st Int'l Workshop Algs. in Bioinformatics (WABI'01)*, volume 2149 of *Lecture Notes in Computer Science*, pages 189–203. Springer Verlag, Berlin, 2001.

[71] D.D. Sleator and R.E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.

[72] J.P. Spinrad. Recognition of circle graphs. *Journal of Algorithms*, 16(2):264–282, 1994.

[73] J.P. Spinrad. *Efficient Graph Representations*. American Mathematical Society, 2003.

[74] A.H. Sturtevant and G.W. Beadle. The relation of inversions in the x-chromosome of drosophila melanogaster to crossing over and disjunction. *Genetics*, 21:554–604, 1936.

[75] A.H. Sturtevant and Th. Dobzhansky. Geographical distribution and cytology of "sex ratio" in drosophila pseudoobscura and related species. *Genetics*, 21:473–490, 1936.

[76] A.H. Sturtevant and Th. Dobzhansky. Inversions in the third chromosome of wild races of drosophila pseudoobscura and their use in the study of the history of the species. *Proc. Nat'l Acad. Sci., USA*, 22:448–450, 1936.

[77] A.H. Sturtevant and E. Novitski. The homologies of chromosome elements in the genus drosophila. *Genetics*, 26:517–541, 1941.

[78] J. Suksawatchon, C. Lursinsap, and M. Bodén. Heuristic algorithm for computing reversal distance with multigene families via binary integer programming. In *CIBCB*, pages 187–193, 2005.

[79] K.M. Swenson, W. Arndt, J. Tang, and B.M.E. Moret. Phylogenetic reconstruction from complete gene orders of whole genomes. In *Proc. 6rd Asia Pacific Bioinformatics Conf. (APBC'08)*, pages 241–250, 2008.

[80] K.M. Swenson, Y. Lin, V. Rajan, and B.M.E. Moret. Hurdles hardly have to be heeded. In *Proc. 6th Workshop Comp. Genomics (RECOMB-CG'08)*, volume 5267 of *Lecture Notes in Computer Science*, pages 239–249. Springer Verlag, Berlin, 2008.

[81] K.M. Swenson, M. Marron, J.V. Earnest-DeYoung, and B.M.E. Moret. Approximating the true evolutionary distance between two genomes. In *Proc. 7th SIAM Workshop on Algorithm Engineering & Experiments (ALENEX'05)*. SIAM Press, Philadelphia, 2005.

[82] K.M. Swenson, M. Marron, J.V. Earnest-Deyoung, and B.M.E. Moret. Approximating the true evolutionary distance between two genomes. *J. Exp. Algorithmics*, 12:1–17, 2008.

[83] K.M. Swenson, N.D. Pattengale, and B.M.E. Moret. A framework for orthology assignment from gene rearrangement data. In *Comparative Genomics, RECOMB 2005 International Workshop, RCG 2005*, pages 153–166, 2005.

[84] J. Tang and B.M.E. Moret. Linear programming for phylogenetic reconstruction based on gene rearrangements. In *Proc. 15th Ann. Symp. Combin. Pattern Matching (CPM'04)*, Lecture Notes in Computer Science, 1995. to appear.

[85] J. Tang and B.M.E. Moret. Scaling up accurate phylogenetic reconstruction from gene-order data. In *Proc. 11th Int'l Conf. on Intelligent Systems for Mol. Biol. (ISMB'03)*, volume 19 of *Bioinformatics*, pages i305–i312. Oxford U. Press, 2003.

[86] E. Tannier, A. Bergeron, and M.-F. Sagot. Advances on sorting by reversals. *Disc. Appl. Math.*, 155(6–7):881–888, 2007.

[87] E. Tannier and M. Sagot. Sorting by reversals in subquadratic time. In *Proc. 15th Ann. Symp. Combin. Pattern Matching (CPM'04)*, volume 3109 of *Lecture Notes in Computer Science*, pages 1–13. Springer Verlag, Berlin, 2004.

[88] G. Tesler. Efficient algorithms for multichromosomal genome rearrangements. *J. Comput. Syst. Sci.*, 65(3):587–609, 2002.

[89] G. Valiente. A new simple algorithm for the maximum-weight independent set problem on circle graphs. In *Proc. 14th Int'l. Symp. Alg. and Comp. (ISAAC'03)*, volume 2906 of *Lecture Notes in Computer Science*, pages 129–137. Springer Verlag, Berlin, 2003.

[90] J.D. Watson and F.H.C. Crick. Molecular structure of nucleic acids: A structure for deoxyribose nucleic acid. *Nature*, 171:737–738, 1953.

[91] G.A. Watterson, W.J. Ewens, T.E. Hall, and A. Morgan. The chromosome inversion problem. *J. Theoretical Biology*, 99:1–7, 1982.