Computational Phylogenetics
An introduction to designing methods for phylogeny estimation

Tandy Warnow
Contents

2.2.4 Compatible sets of clades 37
2.2.5 Hasse Diagram algorithm 38
2.2.6 Difficulties in rooting trees 39

2.3 Unrooted trees 40
2.3.1 Newick notation for unrooted trees 40
2.3.2 The bipartitions of an unrooted tree 41
2.3.3 Representing non-binary trees 42
2.3.4 Comparing trees using their bipartitions 43
2.3.5 Constructing \( T \) from \( C(T) \) 43
2.3.6 Testing compatibility of a set of bipartitions 44

2.4 Constructing the Strict Consensus Tree 46
2.5 Quantifying error in estimated trees 46
2.6 The number of binary trees on \( n \) leaves 47
2.7 Rogue taxa 48
2.8 Induced subtrees 48
2.9 Some special trees 49
2.10 Further reading 50
2.11 Review questions 50
2.12 Homework problems 50

3 Constructing trees from true subtrees 54
3.1 Introduction 54
3.2 Tree compatibility 54
3.2.1 Unrooted tree compatibility 54
3.2.2 Rooted tree compatibility 55
3.3 The ASSU algorithm: constructing rooted trees from rooted triples 55
3.4 Constructing unrooted binary trees from quartet subtrees 56
3.4.1 Notation 56
3.4.2 The All Quartets Method 56
3.4.3 Inferring quartet trees from other quartet trees 57
3.4.4 Constructing a tree from a subset of its quartet trees 58
3.5 Testing compatibility of a set of trees 59
3.6 Further reading 60
3.7 Review questions 60
3.8 Homework problems 61

4 Constructing trees from qualitative characters 65
4.1 Introduction 65
4.2 Terminology 66
4.3 Tree construction based on Maximum Parsimony 67
4.3.1 The Fitch-Hartigan algorithm for fixed-tree maximum parsimony 68
4.3.2 The Sankoff algorithm for fixed tree maximum parsimony 70
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.9</td>
<td>The No Common Mechanism Model</td>
<td>153</td>
</tr>
<tr>
<td>8.10</td>
<td>Beyond statistical consistency: performance on data</td>
<td>154</td>
</tr>
<tr>
<td>8.11</td>
<td>Estimating branch support</td>
<td>155</td>
</tr>
<tr>
<td>8.12</td>
<td>Other gene tree estimation problems</td>
<td>156</td>
</tr>
<tr>
<td>8.13</td>
<td>Further reading</td>
<td>158</td>
</tr>
<tr>
<td>8.14</td>
<td>Review questions</td>
<td>158</td>
</tr>
<tr>
<td>8.15</td>
<td>Homework problems</td>
<td>159</td>
</tr>
<tr>
<td>9</td>
<td>Multiple sequence alignment</td>
<td>163</td>
</tr>
<tr>
<td>9.1</td>
<td>Introduction</td>
<td>163</td>
</tr>
<tr>
<td>9.2</td>
<td>Evolutionary history and sequence alignment</td>
<td>163</td>
</tr>
<tr>
<td>9.3</td>
<td>Evaluating multiple sequence alignments</td>
<td>165</td>
</tr>
<tr>
<td>9.4</td>
<td>Edit distances and how to compute them</td>
<td>167</td>
</tr>
<tr>
<td>9.4.1</td>
<td>Needleman-Wunsch</td>
<td>168</td>
</tr>
<tr>
<td>9.4.2</td>
<td>Smith-Waterman</td>
<td>172</td>
</tr>
<tr>
<td>9.5</td>
<td>Optimization problems for multiple sequence alignment</td>
<td>172</td>
</tr>
<tr>
<td>9.5.1</td>
<td>Sum-of-Pairs Alignment</td>
<td>172</td>
</tr>
<tr>
<td>9.5.2</td>
<td>Tree Alignment</td>
<td>173</td>
</tr>
<tr>
<td>9.5.3</td>
<td>Generalized Tree Alignment</td>
<td>175</td>
</tr>
<tr>
<td>9.6</td>
<td>Profile Hidden Markov Models</td>
<td>176</td>
</tr>
<tr>
<td>9.6.1</td>
<td>Profiles</td>
<td>176</td>
</tr>
<tr>
<td>9.6.2</td>
<td>Gap-free Profile Hidden Markov Models</td>
<td>177</td>
</tr>
<tr>
<td>9.6.3</td>
<td>Profile Hidden Markov Models with insertion and deletion states</td>
<td>177</td>
</tr>
<tr>
<td>9.6.4</td>
<td>Probability calculations on profile HMMs</td>
<td>181</td>
</tr>
<tr>
<td>9.7</td>
<td>Algorithmic techniques</td>
<td>182</td>
</tr>
<tr>
<td>9.7.1</td>
<td>Progressive alignment</td>
<td>182</td>
</tr>
<tr>
<td>9.7.2</td>
<td>Template-based methods</td>
<td>184</td>
</tr>
<tr>
<td>9.8</td>
<td>Co-estimation of alignments and trees</td>
<td>185</td>
</tr>
<tr>
<td>9.9</td>
<td>Current challenges</td>
<td>185</td>
</tr>
<tr>
<td>9.10</td>
<td>Further reading</td>
<td>185</td>
</tr>
<tr>
<td>9.11</td>
<td>Review questions</td>
<td>189</td>
</tr>
<tr>
<td>9.12</td>
<td>Homework problems</td>
<td>189</td>
</tr>
<tr>
<td>10</td>
<td>Constructing species trees under the multi-species coalescent model</td>
<td>192</td>
</tr>
<tr>
<td>10.1</td>
<td>Introduction</td>
<td>192</td>
</tr>
<tr>
<td>10.2</td>
<td>Theoretical foundations</td>
<td>193</td>
</tr>
<tr>
<td>10.2.1</td>
<td>The multi-species coalescent model</td>
<td>193</td>
</tr>
<tr>
<td>10.2.2</td>
<td>Anomalous gene trees</td>
<td>194</td>
</tr>
<tr>
<td>10.2.3</td>
<td>Concatenation under the multi-species coalescent model</td>
<td>195</td>
</tr>
<tr>
<td>10.3</td>
<td>Summary methods</td>
<td>196</td>
</tr>
<tr>
<td>10.3.1</td>
<td>Summary methods that require rooted gene trees</td>
<td>196</td>
</tr>
<tr>
<td>10.3.2</td>
<td>Summary methods that use unrooted gene trees</td>
<td>198</td>
</tr>
</tbody>
</table>
## Contents

10.4 Site-based methods 202
10.5 Co-estimation of gene trees and species trees 202
10.6 Improving scalability of coalescent-based methods 203
10.7 Further reading 204
10.8 Review questions 204
10.9 Homework problems 204

11 Designing methods for large-scale phylogeny estimation 206
11.1 Introduction 206
11.2 Standard approaches 206
11.3 Introduction to Disk-Covering Methods (DCMs) 210
  11.3.1 Objectives of a DCM: boosting a base method 210
  11.3.2 The three phases of a DCM 210
11.4 Triangulated graphs 211
  11.4.1 Definitions and basic properties of triangulated graphs 211
  11.4.2 Decompositions of triangulated graphs 212
  11.4.3 Threshold graphs 212
  11.4.4 Short subtree graphs 214
11.5 Designing DCMs 215
  11.5.1 Introduction 215
  11.5.2 DACTAL: a DCM using iteration and recursion 216
  11.5.3 More elaborate DCM designs 217
  11.5.4 Observations about DCM design strategies 218
11.6 Further reading 222
11.7 Review questions 222
11.8 Homework problems 222

12 Designing methods for large-scale multiple sequence alignment estimation 223

13 Advanced topics in biological phylogenetics 224
13.1 “Fast-converging methods” 224
  13.1.1 Sequence length requirements 224
  13.1.2 Short Quartets Methods 224
  13.1.3 Other fast-converging methods 226
13.2 Phylogenetic networks 226
13.3 Genome rearrangements 227
13.4 Phylogenetic forests 227

PART THREE LINGUISTIC PHYLOGENETICS 229

14 Constructing phylogenetic trees for languages 231
14.1 Introduction 231
14.2 Linguistic character data 231
## Contents

14.3 Models of linguistic character evolution 233
14.4 Inferring properties about proto-languages 233
14.5 Computing linguistic phylogenies 233
  14.5.1 Glottochronology 234
  14.5.2 Maximum parsimony and maximum compatibility 234
  14.5.3 Gray and Atkinson’s approach 235
  14.5.4 Geoff Nichols’ approach: Bags of words 235
  14.5.5 Performance on data 236
14.6 Controversies 236
14.7 Further reading 236
14.8 Review questions 236
14.9 Homework problems 236

15 Constructing phylogenetic networks for languages 237
  15.1 Introduction 237
  15.2 Perfect phylogenetic networks 237
  15.3 Further reading 237
  15.4 Review questions 237
  15.5 Homework problems 237

16 The future of linguistic phylogenetics 238
  16.1 Introduction 238
  16.2 Further reading 238

Appendix A Primer on biological data and evolution 239
Appendix B Algorithm design and analysis 240
Appendix C Guidelines for writing papers about computational methods 261
Appendix D Projects 265
Appendix E Glossary 273
Appendix F Notation 274

References 275
Index 291
Preface

Overview

Phylogenies are graphical models of the evolutionary history of a set of objects. Much of the time, these objects are species and the phylogeny is a tree that represents how the set of species evolved from a common ancestor. Other times, the objects are nucleotide or amino acid sequences for a particular gene, and the phylogeny is a representation of how these molecular sequences evolved from their common ancestor. Sometimes the objects are languages, manuscripts, or other types of objects that have evolved from a common ancestor. Finally, sometimes the phylogenies are networks rather than trees, in order to faithfully represent events such as horizontal gene transfer and hybridization that make the tree model an inadequate representation of the true evolutionary history.

Because phylogenies represent what has happened in the past, phylogenies cannot be directly observed but rather must be estimated. In practice, phylogenies are estimated based on observable properties of the objects, such as genomic DNA sequence data, and are nearly always based on a mathematical model of how these properties evolve over time. The evolutionary history of a set of genes, species, or individuals, provides a context in which biological questions can be addressed. For this reason, phylogeny estimation is a fundamental step in many biological studies, with many applications throughout biology such as protein structure and function prediction, analyses of microbiomes, inference of human migrations, etc. Indeed, there is a famous saying by Dobzhansky that “Nothing in biology makes sense except in the light of evolution” (Dobzhansky, 1973).

Phylogeny estimation typically depends on estimated multiple sequence alignments, another important bioinformatics problem with applications throughout biology. Yet both phylogeny estimation and multiple sequence alignment are computationally difficult problems where existing methods fail to provide adequate accuracy on large, complex datasets. This impacts biological discovery, since error in these estimations can lead to errors in the downstream inferences. Thus, new methods with improved accuracy and scalability to large, complex datasets are needed in order to improve the accuracy of phylogenetic trees and alignments.

Historical linguistics is another discipline where methods are used to address the understanding of how languages evolved, and which has multiple applications (e.g., under-
Preface

standing the properties of ancestral languages, human migrations, etc.). Because languages
evolve under different processes than those that operate on biomolecular sequences, meth-
ods for biological phylogenetics cannot be directly adapted to linguistic phylogenetics.
Unfortunately, the methods used for phylogeny estimation of languages have largely been
based on minor modifications of statistical models for molecular sequence evolution. Thus,
new methods specifically designed for linguistic data that are based on appropriate models
of linguistic evolution are also needed.

The main goal of this text is to enable researchers (typically graduate students in com-
puter science, applied mathematics, or statistics) to be able to contribute new methods for
phylogeny estimation, and in particular for large heterogeneous datasets that are charac-
teristic of the types of inputs that are increasingly of interest in practice. Thus, this text
emphasizes high-level algorithmic design strategies, such as divide-and-conquer, that en-
able statistical methods to scale to large datasets.

The majority of this textbook is focused on estimating phylogenetic trees for biological
data, focusing primarily on the analysis of molecular sequence data. However, this text-
book also includes a discussion of the models and methods used to infer phylogenies for
languages. The secondary goal is to enable biologists and linguists to understand the meth-
ods and their statistical guarantees under these models of evolution, so that they can select
appropriate methods for their datasets, and select appropriate datasets given the available
methods.

Because these goals are really focused on methods - and concretely on software that
is used to analyze data - this text also discusses how to evaluate methods with respect to
accuracy in reconstructing the true evolutionary tree. Because the true evolutionary history
cannot be known (i.e., we cannot go back in time), the data we use to explore methods will
by necessity include simulated datasets as well as real datasets. Learning how to do perfor-
mance studies using simulated and real datasets is a very important aspect of the research
program for method developers for phylogenetic estimation. Furthermore, as many have
observed, there is a substantial gap between performance as suggested by mathematical
theory (which is used to establish guarantees about methods under statistical models of
evolution) and how well the methods actually perform on data – even on data generated
under the same statistical models! For example, methods can have nearly identical the-
etrical properties, according to current mathematical knowledge, but have dramatically
different performance on data. Indeed, this gap is one of the most interesting things about
doing research in computational phylogenetics, because it means that the most impactful
research in the area must draw on mathematical theory (especially probability theory and
graph theory) as well as on observations from data.

Thus, this textbook is aimed at both the method developer as well as at users of methods
(e.g., evolutionary biologists and historical linguists). Some of the material in the textbook
is fairly mathematical, and presumes undergraduate coursework in probability theory, dis-
crete mathematics, and algorithm design and analysis. However, no background in biology
or linguistics is assumed.

Students without the assumed background (e.g., most students from biology and lin-
guistics) may find it difficult at times to understand the proofs. However, understanding the meaning of the theoretical guarantees for phylogeny estimation methods and the statistical models on which they are based empowers these students to critique the scientific literature, and choose methods and datasets that are best able to address the scientific questions they wish to answer. Indeed, students from biology or linguistics are regular attendees in my courses.

**Phylogeny estimation pipeline**

Phylogeny estimation is a multi-step process, and understanding the process is helpful to developing and evaluating methods.

- The biologist identifies a question they wish to answer, and based on the question they select the species and genes they will analyze.
- Sequence data are collected for the species and genes. This may be done by accessing public databases, or by going into the field and gathering specimens. Typically, at the end of this step, for each selected gene and each species, at most one sequence is obtained. Often this is a DNA sequence, but in some cases it may be an RNA or amino acid sequence.
- For each gene, a *multiple sequence alignment* of the different sequences is obtained. This process puts the sequences into a matrix so that the rows correspond to the different species, and the columns represent “homologies” (nucleotides having a common evolutionary history).
- For each gene, a phylogenetic tree is estimated based on the multiple sequence alignment computed in the previous step. This analysis is almost always based on a statistical model of sequence evolution, and most methods combine graph-theoretic methods with statistical estimation techniques. Furthermore, many methods attempt to solve NP-hard optimization problems, so heuristic techniques are often used.
- Statistical support for the individual branches of each gene tree is computed, typically using methods such as non-parametric bootstrapping. These support values let the biologist know which aspects of the evolutionary history are considered highly reliable, and which ones are not as reliable.
- Now we have a collection of trees and multiple sequence alignments, one for each gene. When the gene trees are either identical or very similar to each other, then the species tree is often estimated by concatenating the sequence alignments together, and then using standard techniques on the large “super-alignment” to compute a species tree. However, when the gene trees are very different from each other, estimating the species tree can require different techniques, although the choice of technique depends on the cause for the discordance. When the discordance is due to horizontal gene transfer or hybrid speciation, then a phylogenetic network is needed. However, sometimes the species evolution is treelike but still produces genes whose trees can be different from the species
tree. Thus, estimating species phylogenies (whether phylogenetic trees or phylogenetic networks) requires “phylogenomic” methods.

- After the species tree or phylogenetic network is computed, other aspects of the history (e.g., dates at internal nodes, whether selection has occurred, and how some specific trait evolved within the species phylogeny) are estimated. These are called “post-tree” analyses, and are often the main goal of the study.

The pipeline described above is often varied somewhat. For example, instead of computing a single multiple sequence alignment, sometimes several alignments are computed, and a tree constructed for each alignment. Furthermore, instead of computing a single tree for a single alignment, sometimes multiple trees are computed (perhaps based on different tree estimation methods, or sometimes even just one tree estimation methods). The sets of alignments or sets of trees are then explored to determine the features that are consistent across the different techniques, and “consensus trees” or “consensus alignments” are computed. In addition, sometimes the alignment and tree are co-estimated together, rather than having the alignment estimated first and then the tree based on that alignment.

Note that the final species tree or phylogenetic network depends on the individual gene trees and multiple sequence alignments, and that the gene trees themselves depend on the multiple sequence alignments. This dependency suggests that errors in these initial analyses could result in errors in the downstream analyses, and hence lead to errors in the conclusions of the scientific study. Indeed, research has shown that errors in estimated multiple sequence alignment results in errors in estimated gene trees (Liu et al., 2009a, 2011; Ogden and Rosenberg, 2006), and that errors in estimated gene trees results in errors in estimated species trees (Bayzid and Warnow, 2013; Bayzid et al., 2015; Gatesy and Springer, 2014; Mirarab et al., 2014b). Furthermore, research has also shown that scientific questions, such as the detection of positive selection, can be misled by errors in alignments (Fletcher and Yang, 2010).

Therefore, improving the estimation of multiple sequence alignments and phylogenetic trees can lead to improvements in downstream analyses. Statistical estimation techniques are used to produce multiple sequence alignments, gene trees, and species trees; this emphasis on statistical estimation also means that there is a corresponding effort to develop statistical models of sequence evolution that are biologically realistic. Understanding methods in terms of their performance under statistical models requires some understanding of probability theory (primarily discrete probability). However, phylogeny estimation also depends very much on combinatorial and graph-theoretic algorithms, and so graph theory and discrete mathematics are necessary tools. Finally, nearly all methods are based on optimization problems, many of which are NP-hard; hence, heuristics are needed to design effective software that can analyze large datasets.

Thus, method development in this area depends on a combination of techniques, including statistical inference, probability theory, discrete mathematics, and the design of heuristics. This text presents basic theory for these problems and methods, and also directs the interested reader to some new developments in each area.
This textbook emphasizes statistical guarantees under stochastic models of sequence evolution, algorithm design for large-scale and complex evolutionary scenarios, and performance on data. To achieve this, I have focused on the basic statistical models of sequence evolution and how to design methods for estimating phylogenies under these models so that they have statistical guarantees (most notably, so that they are "statistically consistent"). I have also emphasized the proof techniques for establishing that a method is or isn’t statistically consistent under a statistical model of evolution, so that the reader can apply the same (or similar) techniques to establish that some other method that she may have developed is or is not statistically consistent.

At the same time, performance on sequence data often is quite different from what the theory might suggest: methods that are statistically consistent may have poor accuracy on sequence data, even when the data are generated under the correct model, while statistically inconsistent methods may have good accuracy. Thus there is also an emphasis on evaluating methods on data, looking at performance studies, and using the insights that we can gain into methods through these studies to design more accurate methods.

Outline of textbook

Chapter 1 presents a quick, and relatively lightweight, introduction to the major themes involved in computational phylogenetics, addressing both theory (e.g., statistical consistency under a statistical model of evolution) and performance on data. We use the Cavender-Farris-Neyman model of binary sequence evolution since understand issues in analyzing data generated by this very simple model is helpful to understanding statistical estimation under the commonly used models of molecular sequence evolution. We also present a discussion about performance studies, using both simulated and biological data, that have explored the accuracy of phylogeny estimation methods on varying conditions, since these studies have helped to clarify the meaning of these theoretical guarantees. Thus, Chapter 1 introduces the reader to some of the fundamental challenges, in terms of both theory and practice, in algorithm design for phylogenetic estimation.

Chapters 2 through 7 comprise the “Discrete Mathematics for Phylogenetics” part of the text; the concepts and mathematics introduced in this part are the building blocks for algorithm design in phylogenetics, especially for developing methods that can scale to large datasets, and understanding these concepts makes it possible to understand theoretical guarantees of methods under statistical models of evolution. Chapter 2 introduces trees as graph-theoretic objects, and presents different representations of trees that will be useful for method development. Chapters 3, 4, and 5 present different types of methods for phylogenetic tree estimation (based on combining subtrees, using character data, or using distances, respectively). Chapter 6 presents methods for analyzing sets of trees, each on the same set of taxa, and computing consensus trees and agreement subtrees, and discusses how these methods are used to estimate support for different phylogenetic hypotheses. Chapter 7 examines the topic of supertree estimation, which is where the input is a set of trees, each on a different set of taxa, and the objective is a tree on the full set of taxa.
Preface

Chapters 8 through 16 introduce the statistical foundations of phylogenetic estimation, and its application to both biological and linguistic data. Chapter 8 presents commonly used statistical models of molecular sequence evolution, and statistical methods for phylogeny estimation under these models. However, standard sequence evolution models do not include events such as insertions, deletions, and duplications, which can change the sequence length. These are very common processes, so that biological sequences are usually of different lengths and must first be put into a multiple sequence alignment before they can be analyzed using phylogeny estimation methods. Multiple sequence alignment estimation is covered in Chapter 9.

Strictly speaking, what we have been describing here is how to model the evolution of a single genomic region, rather than how to model the evolution of the entire genome. Since different chromosomal regions can have different evolutionary histories for a number of reasons – including incomplete lineage sorting – computing the evolution of the species as a whole is quite complicated. Constructing a species tree from different gene trees in the presence of incomplete lineage sorting (ILS) is a fascinating research problem that we present in Chapter 10.

Since nearly all good approaches to phylogeny and multiple sequence alignment estimation are heuristics, and some are computationally intensive, we also discuss the design of methods for large datasets. Chapter 11 addresses tree estimation for large datasets, and the design of methods for large-scale multiple sequence alignment is discussed in Chapter 12.

Chapter 13 introduces some advanced research topics, including phylogenetic placement, fast-converging methods, phylogenetic networks, phylogenetic forests, and genome rearrangements, with a focus on providing entries to the scientific literature on these topics.

Linguistic phylogenetics – the inference of phylogenetic trees and networks for languages – is closely related to biological phylogenetics, but accurate analyses of linguistic data require different models of character evolution, and consequently different methods; we discuss these issues in Chapters 14-16.

Most chapters end with a set of review questions and homework problems. The review questions are easy to answer and do not require any significant problem solving or calculation. The homework problems are largely computational pen and paper problems that reinforce the mathematical content of the text.

The textbook comes with four appendices. The first appendix provides an introduction to biological evolution and data; the textbook can be read without it, but the reader who wishes to analyze biological data will benefit from this material. The second appendix provides an introduction to algorithm design and analysis; this material is covered in undergraduate computer science courses, and is intended as a quick review of the concepts. Students without this background may need additional materials. The third appendix provides guidelines about how to write papers that introduce new methods or evaluate existing methods. The fourth appendix provides computational projects ranging from short term (i.e., a few days) to research projects that could lead to publications. In fact, several of the final projects for my Computational Phylogenetics courses have grown into journal publications (e.g., Zimmermann et al. (2014); Davidson et al. (2015); Chou et al. (2015);
Bayzid et al. (2014)) that have influenced the computational phylogenetics research community and provided new methods for phylogenetic analysis.
PART ONE
BASIC TECHNIQUES
1

Brief introduction to phylogenetic estimation

At the heart of most modern phylogenetic estimation is the challenge of taking a set of sequences (typically DNA sequences, which can be considered strings over the alphabet \{A, C, T, G\}), and computing a tree from the set. While there are many ways to compute trees from a set of sequences, understanding whether the methods can be accurate requires having some kind of model for how the sequences relate to each other, and more specifically how they evolved from a common ancestor. We introduce the key concepts and issues in phylogeny estimation in the context of a very simple model of sequence evolution - the Cavender-Farris-Neyman model of binary sequence evolution.

1.1 The Cavender-Farris-Neyman model

The Cavender-Farris-Neyman (CFN) model describes how a trait (which can either be present or absent) evolves down a tree (Cavender, 1978; Farris, 1973; Neyman, 1971). Hence, a CFN model has a rooted binary tree \(T\) with \(n\) leaves and numerical parameters on the tree that describe the evolutionary process of a trait. Under the CFN model, the probability of absent (0) or present (1) is the same at the root, but the state can change on the edges of the tree. Thus, we associate a parameter \(p(e)\) to every edge \(e\) in the tree, where \(p(e)\) denotes the probability of changing state (from 1 to 0, or vice-versa). For reasons that we will explain later, we require that \(0 < p(e) < 0.5\).

Note that under this model, a trait (which is also called a “character”) evolves down the tree under this random process, and hence attains values at every node in the tree, and in particular at the \(n\) leaves of the tree. Now suppose you were to be able to look at the values of the trait at the \(n\) different leaves. Would you be able to reconstruct the tree \(T\)?

After some thought, it should be obvious that it isn’t possible to figure out the tree from the states of this single trait – there just isn’t enough information. However, we could run this process again, starting at the root with either 0 or 1 (picked randomly), and end up with a new trait taking values at every node. By repeating the process \(k\) times, we obtain \(k\) different traits that have evolved down the tree, each of them evolving identically and independently (\(i.i.d\.). Can we prove that we can construct the tree correctly with high probability, when \(k\) is large enough? (Equivalently, can we construct the tree correctly with high probability, given a large enough number of \(i.i.d\). trials?)
1.2 An analogy: determining whether a coin is biased towards heads or tails

Even this might seem unlikely to you, but an analogy might be helpful. Suppose you have a coin that is biased either towards heads or towards tails, but you don’t know which. Can you run an experiment to figure out which type of coin you have?

After a little thought, the answer may seem obvious – toss the coin many times, and see whether heads comes up more often than tails. If it does, say the coin is biased towards heads; otherwise, say it is biased towards tails. The probability that you guess correctly will approach 1 as you increase the number of coin tosses. However, the probability of being correct will clearly depend on the number of coin tosses, so you may need to toss it often.

Now suppose you don’t get to toss the coin yourself, but are instead shown a sequence of coin tosses of some length that is chosen by someone else. Now, you can still guess whether the coin is biased towards heads or tails, but the probability of being correct may be small if the coin is not tossed enough times. Note that for this problem - of deciding whether the coin is biased towards heads or tails - you will either be 100% correct or 100% wrong, but the probability of being 100% correct can be high, and will depend on how many coin tosses you have. The reason you can be 100% correct is that there are only a finite number of choices.

Now, suppose I were to ask you to estimate the actual probability of a head for that coin. You could do the same experiment, and report the fraction of the coin tosses that come up heads. Note that in this problem your estimations of the probability of a head will generally have error. For example, if the probability of a head is irrational, then you can never be completely correct. However, your estimate will converge to the true probability of a head for the coin as the number of coin tosses increased. In other words, if you are asked to estimate the probability of a head, then from a large enough number of coin tosses, your answer will have low error with high probability.

The problem of constructing a CFN tree is very similar to the problem of determining whether a coin is biased towards heads or tails. There are only a finite number of different trees on \( n \) distinctly labelled leaves, and you are asked to select from among these. Then, if you have a sequence of samples of a random process, you are trying to use the samples to select the tree from that finite set; this is very much like deciding between the two types of biased coins. As we will show, it is possible to correctly construct the CFN tree with high probability, given sufficiently long sequences generated on the tree. The problem of constructing the substitution probabilities on the edges of the Cavender-Farris-Neyman tree is similar to the problem of determining the actual probability of a head, in that these are real-valued parameters, and so some error will always be expected. However, if good methods are used, then as the sequence lengths increase the error in the estimated substitution probabilities will decrease, and the estimates will converge to the true values.

While estimating the numeric parameters is important for many tasks, we’ll focus here on the challenge of estimating the tree \( T \), rather than the numeric parameters. We describe
some techniques for estimating this tree from binary sequences, and discuss whether they can estimate the tree correctly with high probability given sufficiently long sequences.

1.3 Estimating the Cavender-Farris-Neyman tree

Recall that the CFN model consists of a tree $T$ that is rooted and binary (so all nodes other than the leaves have exactly two children), and also the numeric parameters on the edges that indicate the probability that the site will change state on the edge. Thus, a CFN model tree is a pair $(T, \theta)$ where $T$ is the rooted binary tree with leaves labelled $s_1, s_2, \ldots, s_n$ and $\theta$ provides the values of $p(e)$ for every edge $e \in E(T)$.

However, this stochastic process can also be described differently, and in a way that is helpful for understanding why some methods can have good statistical properties for estimating CFN model trees. Under the CFN model, the number of substitutions on an edge is modelled by a Poisson random variable with expected value $\lambda(e)$. Thus, instead of using the substitution probability $p(e)$ on each edge, we use $\lambda(e)$, with the constraint that $0 < \lambda(e)$ for all $e$. Note then that if the endpoints of an edge have different states (so that one endpoint is 0 and the other is 1), then an odd number of changes must have occurred. Using the properties of Poisson random variables, it can be shown that

$$\lambda(e) = -\frac{1}{2} \ln(1 - 2p(e)).$$

1.3.1 Estimating the CFN tree when evolution is clock-like

An assumption that is sometimes made is that sequence evolution is clock-like (also referred to as the “strict molecular clock”), which means that the expected number of changes is proportional to time. If we assume that the leaves represent extant (i.e., living) species, then under the assumption of a strict molecular clock, the total expected number of changes from the root to any leaf is the same. Thus, under the assumption of a strict molecular clock, the matrix of expected distances between the leaves in the tree is called an “ultrametric matrix”.

**Definition 1.1** An ultrametric matrix is an $n \times n$ matrix $M$ corresponding to distances between the leaves in a rooted edge-weighted tree $T$ (with non-negative edge weights) where the sum of the edge weights in the path from the root to any leaf of $T$ does not depend on the selected leaf.

Constructing trees from ultrametric matrices is much easier than the general problem of constructing trees from distance matrices that are not ultrametric. However, the assumption of clock-like evolution may not hold on a given dataset, and is generally not considered realistic. Furthermore, the ability to reconstruct the tree using a particular technique may depend on whether the evolutionary process is in fact clock-like.

That said, let’s assume we have a clock-like evolutionary process operating on a CFN tree $(T, \theta)$, and so the total number of expected changes from the root to any leaf is the
Inferring Clocklike Evolution

While |S|>2:
  find pair x,y of closest taxa;
  delete x
  Recurse on S-{x}
  Insert y as sibling to x
  Return tree

Figure 1.1 Constructing trees when evolution is clocklike. We show a cartoon of a model tree, with branch lengths drawn proportional to the expected number of changes. When evolution is clocklike (as it is for this cartoon model), simple techniques (such as the one described in the figure) will reconstruct the model tree with probability that converges to 1 as the number of sites increases.

same. Let us take a very simple case, where the tree $T$ has three leaves, $A,B$ and $C$. To reconstruct the tree $T$ we need to be able to infer which pair of leaves are siblings, from the sequences we observe at $A,B$ and $C$. Can we do this?

One very natural approach to estimating the tree would be to select as siblings the two sequences that are the most similar to each other from the three possible pairs. Because the sequence evolution model is clock-like, this technique will correctly construct rooted three-leaf trees with high probability. Furthermore, the method can even be extended to work on trees with more than three leaves, using recursion. For example, consider the model tree given in Figure 1.1, where the branch lengths indicate the expected number of substitutions on the branch. Note that this model tree is ultrametric. Thus, under this model, the sequences at leaves $a$ and $b$ will be the most similar to each other of all the possible pairs of sequences at the leaves of the tree. Hence, to estimate this tree, we would first compare all pairs of sequences to find which pair is the most similar, and we’d select $a$ and $b$ as this pair. We’d then correctly infer that species $a$ and $b$ are siblings. We would then remove one of these two sequences (say $a$), and reconstruct the tree on what remains. Finally, we would add $a$ into the tree on the tree we construct on $b,c,d,e$, by making it a sibling to $b$.

This approach, which is a variant of the UPGMA (Sokal and Michener, 1958) method,
1.3 Estimating the Cavender-Farris-Neyman tree

is easily seen to converge to the true tree as the sequence length increases. That is, it is possible to make mistakes in the construction of the tree – but the chances of making a mistake decrease as the sequence length increases.

However, what if the evolutionary process isn’t clocklike? Suppose, for example, that we have a three-leaf CFN model tree with leaves $A$, $B$ and $C$, in which $A$ and $B$ are siblings. Suppose however that the amount of change on the edges leading to $B$ and $C$ are both extremely low, while the amount of change on the edge leading to $A$ is very high. Then, applying the technique described above would return the tree with $B$ and $C$ siblings – i.e., the wrong tree. In other words, this simplified version of UPGMA would converge to the wrong tree as the sequence length increases. This is clearly an undesirable property of a phylogeny estimation method! An example of a model tree where UPGMA and its variants would not construct the correct tree – even as the sequence length increases - is given in Figure 1.2; the probability of selecting $b$ and $c$ as the first sibling pair would increase to 1 as the sequence length increases, and so the methods would return the wrong tree.

Clearly, when there is no clock, then sequence evolution can result in datasets for which the inference problem seems to be much harder. In fact, without a molecular clock, the inference of rooted three-leaf trees is not possible – and more generally the estimation of
rooted trees is not possible. Instead, the best that can be hoped for is the estimation of the unrooted version of the rooted tree. Here, we show how to estimate the unrooted tree.

1.3.2 Estimating the unrooted CFN tree when evolution is not clock-like

We now discuss how to estimate the underlying unrooted CFN tree from sequences, without assuming clocklike evolution. Let \((T, \theta)\) be a CFN model tree on leaves \(s_1, s_2, \ldots, s_n\), so that \(T\) is the rooted binary tree and \(\theta\) gives all the edge parameters \(\lambda(e)\). Let \(\lambda_{i,j}\) denote the expected number of changes for a site on the path \(P_{i,j}\) between leaves \(s_i\) and \(s_j\) in the CFN model tree \(T\); it follows that

\[
\lambda_{i,j} = \sum_{e \in P_{i,j}} \lambda(e).
\]

Note that by definition, \(\lambda\) is the matrix of path distances in a tree, where the path distance between two leaves is the sum of the branch lengths and all branch lengths are positive. Matrices that have this property have special mathematical properties, and in particular are examples of “additive” matrices.

**Definition 1.2** An \(n \times n\) matrix \(M\) is *additive* if there is a tree \(T\) with leaves labelled 1, 2, \ldots, \(n\) and non-negative lengths (or weights) on the edges, so that the path distance between \(i\) and \(j\) in \(T\) is equal to \(M[i,j]\).

In other words, additive matrices correspond to edge-weighted trees in which all edge weights are non-negative; therefore, distance matrices arising from CFN model trees are necessarily additive. Furthermore, CFN model trees have strictly positive branch lengths, which additionally constrains the additive matrices corresponding to CFN model trees; we will use the properties of additive matrices, and in particular of additive matrices corresponding to trees with all positive branch lengths, to construct CFN trees from their additive matrices.

We will show that if we are given an additive distance matrix \(\lambda\) corresponding to a CFN model tree, then we can reconstruct the model tree and its branch lengths in polynomial time. Techniques to compute trees from CFN distance matrices (and even from noisy versions of CFN distance matrices) are presented in Chapter 5, and briefly summarized here.

First, let’s consider the case where the CFN tree \(T\) has only four leaves, \(s_1, s_2, s_3,\) and \(s_4\). Suppose we have the values of \(\lambda(e)\) for every edge in \(T\) and hence also the additive matrix \(\lambda_{i,j}\) of path distances in the tree. Without loss of generality, assume the tree \(T\) has an internal edge \(e_l\) that separates \(s_1\) and \(s_2\) from \(s_3\) and \(s_4\). Note that \(\lambda(e_l) > 0\), under the CFN model. Now, consider the three following pairwise sums:

- \(\lambda_{1,2} + \lambda_{3,4}\)
- \(\lambda_{1,3} + \lambda_{2,4}\)
- \(\lambda_{1,4} + \lambda_{2,3}\)
1.3 Estimating the Cavender-Farris-Neyman tree

Since the length of every edge is strictly positive, it is easy to see that the first of these three pairwise sums is strictly smaller than the other two, and the other two are equal. If the length of an edge can be zero, then it is possible to have all three pairwise sums being equal. However, for any additive matrix (one that equals leaf-to-leaf path distances in a tree with non-negative edge weights), the two largest pairwise sums will always be equal to each other; this property is called the **Four Point Condition**.

Hence, if we are given an additive $4 \times 4$ matrix that corresponds to a CFN model tree, then we can easily determine the tree topology that corresponds to the matrix (i.e., determine which of the three pairwise sums is the smallest, and use that one to define the split for the four leaves into two sets of two leaves each). We refer to this method as the **Four Point Method**.

As we will show in Chapter 5, the same technique can compute four-leaf trees when the input is a $4 \times 4$ matrix that is a noisy version of an additive matrix corresponding to model trees (CFN model trees, or trees under other statistical models). Furthermore, provided the noise level is low enough, the tree estimation will be correct. Perhaps less obviously, it is possible to correctly compute trees from $n \times n$ additive matrices (when all edge lengths are positive), and from noisy versions of these additive matrices, again providing that the noise is sufficiently low. Here we briefly describe how to do this.

1.3.3 Computing unrooted CFN trees from dissimilarity matrices

Additive matrices are special cases of a more general class of matrices that we will encounter in phylogenetic analysis, which we will refer to as **dissimilarity matrices**. Dissimilarity matrices are symmetric square matrices that are zero on the diagonal and non-negative off the diagonal; note that dissimilarity matrices are not assumed to satisfy the triangle inequality (that $d_{ij} \leq d_{ik} + d_{jk}$ for all $i, j, k$), and hence are not properly speaking “distance matrices”. We will begin by showing how to compute a tree from an additive matrix corresponding to the tree, and then show how to extend that approach to compute trees from dissimilarity matrices.

Suppose we have an $n \times n$ additive matrix $M$ and $n > 4$, and that $M$ corresponds to some unknown tree $T$ with strictly positive branch lengths. How can we construct the unrooted tree $T$ corresponding to this matrix?

The basic idea is to use the technique described earlier to compute the unrooted four-leaf tree $t(A)$ on every set $A$ of four leaves. Since the matrix is additive, each unrooted four-leaf tree $t(A)$ will be completely correct. Now, look at this set $X$ of four-leaf trees that you have constructed. In the tree $T$, there must be a pair of leaves $x, y$ that are siblings (meaning that they have a common parent vertex). For this pair of leaves, every tree in the set $X$ that contains both $x$ and $y$ must have $x$ and $y$ as siblings in the quartet tree. Conversely, if $u$ and $v$ are not siblings in $T$, then there are two other leaves $a, b$, such that the quartet tree on $a, b, u, v$ will put $a, u$ on one side, separated from $b, v$ on the other side. Thus, two leaves $x, y$ are siblings in $T$ if and only if they are siblings in every tree in $X$ in which both $x$ and $y$ appear.
Once you find a pair \( x, y \) of sibling leaves, remove one of those leaves (without loss of generality, remove \( x \)). If you have only four taxa left, then examine the tree on that set of four leaves, add \( x \) as a sibling to \( y \), and return this tree. Otherwise, you have at least five taxa, and you recursively construct a tree on \( S - \{x\} \) that is consistent with all the quartet trees. Then add \( x \) into that tree by making it a sibling to \( y \), and return the resultant tree.

In other words, if you know the additive matrix of leaf-to-leaf CFN model distances corresponding to the CFN model tree \((T, \theta)\), then you can construct the quartet trees for all four leaves, and from these quartet trees you can construct the unrooted tree topology \( T \).

What about matrices that aren’t additive? Because it is possible to correctly reconstruct four-leaf trees given noisy versions of additive matrices (as long as the noise is low enough), it follows that the same method will be able to construct unrooted tree topologies for CFN trees correctly given noisy versions of additive matrices, as long as the noise level is low enough. In fact, all we need is that the noise level is low enough that no quartet tree is incorrectly computed!

### 1.4 The Naive Quartet Method

The Naive Quartet Method was introduced in Erdos et al. (1999a), where it was called the “Naive Method”. The basic idea is extremely simple:

Step 1: Apply the Four Point Method to every four leaves; if any four-leaf subset fails to return a tree, return \textit{Failure}, and exit.

Step 2: Construct a tree that agrees with all the quartet trees computed in Step 1, if it exists, and otherwise return \textit{Failure}.

This two-step process has nice properties when the input matrix is additive and based on a tree \( T \) with non-zero branch lengths. In this case, the quartet trees that are computed in the first step are binary trees, and they are all compatible (and equal to the induced quartet trees in the tree \( T \)). Hence, in this case, the second step will return the tree \( T \). Also, as we showed, when the input matrix is very close to additive, then the first step still returns the true quartet tree for every four leaves, and so the second step will return the true tree \( T \).

On the other hand, when the input matrix is not close enough to additive, then this two-step process can fail to return anything! For example, the Four Point Method can fail to determine a unique quartet tree (if the smallest of the three pairwise sums is not unique), and the whole process can fail in that case. Or, even if the Four Point Method returns a unique tree for every set of four leaves, the set of quartet trees may not be compatible, and so the second step can fail to construct a tree on the full dataset. Thus, the two-step process will only succeed in returning a tree under fairly restricted conditions. For this reason, even though this two-step process for constructing trees has nice theoretical guarantees, it is not used in practice. We refer to this two-step process as the Naive Quartet Method, noting that the word “naive” is intentionally used to suggest that the method is really a mathematical construct rather than a practical tool.
The Naive Quartet Method is statistically consistent under the CFN model: Although the Naive Quartet Method can fail given some input dissimilarity matrices, we will show that the Naive Quartet Method is statistically consistent under the CFN model, which means that it will reconstruct the true tree with probability increasing to 1 as the number of sites increases. To show this, we begin by showing how we compute a dissimilarity matrix given a set of sequences that evolve down a model CFN tree.

Let $p(i, j)$ denote the probability that the leaves $s_i$ and $s_j$ have different states, and note that

$$\lambda_{i, j} = -\frac{1}{2} \ln(1 - 2p(i, j)).$$

If we knew all the $p(i, j)$ exactly, we could compute all the $\lambda_{i, j}$ exactly, and hence we would have an additive matrix for the tree $T$; this means we could reconstruct the model tree and its branch lengths perfectly. However, from finite sequences we cannot know any $p(i, j)$ exactly. Nevertheless, we can estimate these values from the data we observe, in a natural way. That is, given sequences $s_i$ and $s_j$ that evolve down the tree $T$ under this process, so that both $s_i$ and $s_j$ have the same length $k$, we can estimate $p(i, j)$ as the fraction of the sequence length in which $s_i$ and $s_j$ have different states. Put precisely, letting $H(i, j)$ be the Hamming distance between $s_i$ and $s_j$ (i.e., the number of positions in which they are different), then $\frac{H(i, j)}{k}$ is the fraction in which the two sequences are different. Furthermore, as $k \to \infty$, $\frac{H(i, j)}{k} \to p(i, j)$. Hence, we can estimate $\lambda_{i, j}$, the CFN model distance (also known as true evolutionary distance) between sequences $s_i$ and $s_j$, using the following formula:

$$\hat{\lambda}_{i, j} = -\frac{1}{2} \ln(1 - 2\frac{H(i, j)}{k}).$$

Note that as $k \to \infty$ that $\hat{\lambda}_{i, j} \to \lambda_{i, j}$. We call this the Cavender-Farris-Neyman distance correction, and the distances that we compute using this distance correction are the Cavender-Farris-Neyman (CFN) distances. Note that the CFN distance matrix computed using the Cavender-Farris-Neyman distance correction is a noisy version of the model Cavender-Farris-Neyman distance matrix, and that the estimated distances converge to the additive distance matrix as the number of sites increases. However, CFN distance matrices may not satisfy the triangle inequality (this isn’t trivial to see, but can be verified) but will be zero on the diagonal and symmetric. Hence, CFN distances produce dissimilarity matrices.

To say that $\hat{\lambda}_{i, j}$ converges to $\lambda_{i, j}$ for all $i, j$ as the sequence length increases means that for any $\epsilon > 0$ and $\delta > 0$, there is a sequence length $K$ so that the distance matrix $\hat{\lambda}$ will satisfy $L_{\infty}(\hat{\lambda}, \lambda) < \delta$ with probability at least $1 - \epsilon$ given sequence length at least $K$. Hence, with high probability, we can get arbitrarily close to an additive matrix for the model tree, given long enough sequences. This means that the model tree can be constructed correctly with high probability, given long enough sequences.

Thus, although the method we described for constructing rooted trees (that operates by finding siblings based on minimizing distances between sequences, and then recurses) will not work correctly without a strict molecular clock, it is possible to construct the unrooted
tree correctly (with high probability) using a more complex technique, provided that the sequence length is large enough. Furthermore, many methods have been developed to construct CFN trees from sequence data, and that have the theoretical guarantee of converging to the true tree as the sequence length increases.

1.5 Some comments about the CFN model

In the CFN model, we constrain the substitution probabilities $p(e)$ to be strictly between 0 and 0.5. The reason for this constraint is to ensure that the tree can be constructed correctly, given a large enough number of sites. Note that if $p(e) = 0$ for some edge $e$, then the edge $e$ cannot be detected by any method since no changes occur on the edge. If $p(e) = 0.5$ then the two sequences at the endpoint of the edge look random with respect to each other, and this makes reconstructing the tree with high probability impossible. This is why $p(e)$ is constrained to be strictly between 0 and 0.5.

In the CFN model, we also constrained all the sites to evolve exactly as the others. However, it is commonly assumed that the sites can evolve under different rates of evolution, and that each site draws its rate independently from a distribution of rates-across-sites. The meaning of “rates-across-sites” is that each rate gives a multiple for the expected number of changes. Thus, if site $i$ draws rate 2 and site $j$ draws rate 1, then site $i$ has twice as many expected changes as site $j$ on every edge of the tree. Typically, the distribution of rates across sites is modelled using the gamma distribution, but some other distributions (such as gamma plus invariable) are also sometimes used. Note that although the sites can have different rates, they draw their rates independently, and hence all sites evolve under the “same process”. This is called the i.i.d. (identical and independently distributed) assumption. Finally, given a particular gamma distribution, the entire stochastic model of evolution is fully described by the model tree topology $T$, the branch lengths, and the gamma distribution.

Biological data typically are not binary sequences, and instead are typically molecular sequences, either of nucleotides (which are over a four-letter alphabet) or amino acids (which are over a 20-letter alphabet). Statistical models of nucleotide and amino acid sequence evolution (discussed in Chapter 8) have also been developed, and methods for estimating trees under these more complex multi-state models have been developed to estimate under these models. Despite the increased complexity of the models and methods, for most of these models the theoretical framework and analysis for these more sophisticated methods are basically the same as that which we’ve described under the CFN model. Thus, even under more biologically realistic models it is possible to reconstruct the unrooted topology of the true tree with high probability, given sufficiently long sequences generated on the tree. This is called “statistical consistency.”
There are many phylogeny estimation methods that have been developed, some of which are statistically consistent under the standard statistical models of sequence evolution. One of the methods that has been used to construct trees is the UPGMA method alluded to earlier; UPGMA is an agglomerative clustering method that computes a distance between every pair of sequences, then selects the closest pair of sequences to be siblings, updates the matrix, and repeats the process until a tree is computed for the full dataset. Yet, as we have noted, UPGMA can fail to be statistically consistent under some model conditions.

Maximum parsimony is another approach that has been used to construct many trees. Maximum parsimony is best described as an optimization problem, which seeks to find a tree $T$ in which the input sequences are placed at the leaves of $T$ and additional sequences are placed at the internal nodes of $T$ so that the total treelength, defined to be the total number of changes over the entire tree, is minimized. Another way of defining maximum parsimony is that it is the Hamming Distance Steiner Tree Problem: the input is a set of sequences, and the output is a tree connecting these sequences (which are at the leaves) and with other sequences (i.e., the Steiner points) at the internal nodes, that minimizes the total of the Hamming distances on the edges of the tree. Since the Hamming distance between two sequences of the same length is the number of positions in which they are different, the total of the Hamming distances on the edges of the tree is the same as its treelength.

Finding the best tree under the maximum parsimony criterion is an NP-hard (Foulds and Graham, 1982) problem, and hence heuristics, typically based on a combination of hill-climbing and randomization to get out of local optima, are used to find good, though not provably globally optimal, solutions. Maximum parsimony heuristics have improved over the years, but can still be computationally very intensive on large datasets. However, suppose we could solve maximum parsimony exactly (i.e., find global optima); would maximum parsimony then be statistically consistent under the CFN model, or other models?

Unfortunately, maximum parsimony has been proven to be statistically inconsistent under the CFN model and also under standard DNA sequence evolution models, and may even converge to the wrong tree as the sequence length increases (Felsenstein, 1978).

Although UPGMA and maximum parsimony are both statistically inconsistent under standard DNA sequence evolution models, other methods have been developed that are statistically consistent under these models, and are commonly used in practice. Examples of these methods include polynomial time distance-based methods such as neighbor joining (Saitou and Nei, 1987) and FastME (Lefort et al., 2015). The Naive Quartet Method is statistically consistent under the CFN model, and also under standard nucleotide sequence evolution models, and its statistical consistency is extremely easy to prove. The Naive Quartet Method is also polynomial time, and so is a polynomial time statistically consistent method for estimating trees under standard sequence evolution models.

Other approaches, such as maximum likelihood (Neyman, 1971; Felsenstein, 1981), are statistically consistent if solved exactly, but due to computational difficulties (e.g., maximum likelihood is NP-hard (Roch, 2006)), are generally not solved exactly. Similarly,
Bayesian MCMC estimations of phylogenetic trees (Huelsenbeck et al., 2001), is statistically consistent under standard DNA sequence evolution models, but needs to run for a long time to have good statistical performance. Thus, maximum likelihood and Bayesian MCMC estimation of phylogenetic trees are generally much slower than distance-based estimation methods.

Based on this, one could presume that methods like neighbor joining and the Naive Quartet Method would dominate in practice, since they are polynomial time and statistically consistent.

1.7 Measuring error rates on simulated datasets

Phylogeny estimation methods are evaluated for accuracy, primarily with respect to the tree topology (as an unrooted tree), using both simulated and biological datasets. However, because the true evolutionary history of a biological dataset can rarely be known with confidence, most performance studies are based on simulated datasets. In a simulation study, a model tree is created, and then sequences are evolved down the tree. These sequences are then used to compute a tree, and the computed tree is compared to the model tree. If the sequence evolution process includes insertions and deletions, then a multiple sequence alignment is typically first computed before the tree is estimated. See Figure 1.3 for a graphical description of how a simulation study is performed.

Because the true tree and true alignment are rarely known on any biological dataset, simulation studies are the norm for evaluating phylogeny estimation methods, and are also frequently used to evaluate multiple sequence alignment methods. In a simulation study, a model tree is created, often using a simulation process where a tree is produced under a mathematical model for speciation (e.g., a birth-death process), and then sequences are evolved down the tree under a model that describes the evolutionary process. Often, these models will assume a substitution-only process (such as the Cavender-Farris-Neyman model for binary sequences that we discussed earlier, but also under models such as the Jukes-Cantor (Jukes and Cantor, 1969) and Generalized Time Reversible (Tavaré, 1986) models, which model DNA sequence evolution). However, when alignment estimation is also of interest, then other models are used in which sequences evolve with insertions, deletions, and sometimes other events. Thus, in one run of the simulation procedure, a set of sequences is generated for which we know the entire evolutionary history relating the sequences, and hence we know the true alignment. Once the sequences are generated, an alignment can be estimated from the unaligned sequences, and a tree can be estimated on the estimated alignment. The estimated alignment and estimated tree can be compared to the true (model) tree and true alignment, and the error can be quantified. By varying the model parameters, the robustness of the estimation methods to different conditions can be explored, and methods can be compared for accuracy.

There are many ways to quantify error in phylogeny estimation, but the most common one measures the distance between two trees in terms of the bipartitions induced by edges in the two trees that are in only one of the two trees. The bipartitions that are present in
the model tree but not the estimated tree are called false negative edges or false negatives (FN), for short, and the bipartitions that are present in the estimated tree but not the model tree are referred to as false positive edges or false positives (FP). Also, since branches and edges mean the same thing, false negative edges are also called false negative branches, and similarly for false positive branches and false positive edges. The Robinson-Foulds (RF) distance (also called the bipartition distance) between two trees is the number of non-trivial bipartitions that are present in one or the other tree but not in both trees. Each of these ways of quantifying error in an estimated tree can be normalized to produce a proportion between 0 and 1 (equivalently, a percentage between 0 and 100). For example, the FN error rate would be the percentage of the non-trivial model tree bipartitions that are not present in the estimated tree, and the FP error rate would be the percentage of the non-trivial bipartitions in the estimated tree that are not present in the model tree. Finally, the RF error rate is the RF distance divided by \( 2n - 6 \), where \( n \) is the number of leaves in the model tree; note that \( 2n - 6 \) is the maximum possible RF distance between two trees on the same set of \( n \) leaves.

Figure 1.4 provides an example of this comparison; note that the model tree (called the true tree in the figure) is rooted, but the inferred tree is unrooted. To compute the tree error, we unroot the true tree, and treat it only as an unrooted tree. Since both trees are binary (i.e., each non-leaf node has degree three), there are only two internal edges. Each of the two trees have the non-trivial bipartition separating \( S_1, S_2 \) from \( S_3, S_4, S_5 \), but each tree also has a bipartition that is not in the other tree. Hence, the bipartition distance between the two trees is 2, out of a maximum possible of 4, and so the Robinson-Foulds (bipartition) error rate is 50%. Note also that there is one true positive edge and one false positive edge in the inferred tree, so that the inferred tree has FN and FP rate of 50% (only internal edges count).

Performing simulation studies is a fundamental part of research in phylogenetics, and is helpful for understanding the performance of existing methods, and hence for designing new methods with improved performance. See Appendix C for a discussion about how to evaluate methods well, including issues such as how to simulate your data, how to vary parameters, how to select benchmarks, and how to report results.
Simulation Studies

Figure 1.3 A simulation study protocol.

produce the wrong tree with probability increasing to 1 as the sequence length increases under some conditions.

On the face of it, this would seem to suggest that UPGMA and Maximum Parsimony are both inferior to the Naive Quartet Method and Neighbor Joining. Thus, perhaps Maximum Parsimony should never be used instead of the Naive Quartet Method or Neighbor Joining.

Yet the conditions under which the Naive Quartet Method will return the true tree require that every quartet tree be computed without any error at all. As many have observed, some quartet trees can be very difficult to compute, even given sequences that have thousands of sites (Huelsenbeck and Hillis, 1993; Hillis et al., 1994). Furthermore, as the number of sequences in the dataset increases, attempting to compute the correct quartet tree for every set of four sequences would become more difficult. Hence, the Naive Quartet Method would seem to be a rather poor choice of method for phylogeny estimation for any large dataset, even though it is statistically consistent and runs in polynomial time. Indeed, the Naive Quartet Method may not even be useful on most moderate-sized datasets. In comparison, UPGMA, neighbor joining, and Maximum Parsimony always return a tree, and so will not have this kind of dramatic failure that the Naive Quartet Method has.

What does the theory suggest about the relative performance between Neighbor Joining and Maximum Parsimony? Or, put differently, since Neighbor Joining is polynomial time and statistically consistent whereas Maximum Parsimony is neither, does this mean that Neighbor Joining should be more accurate than Maximum Parsimony? The answer, perhaps surprisingly, is no: there are model conditions and sequence lengths where trees
1.8 Performance of methods on data

Quantifying Error

**Figure 1.4 How tree estimation error is calculated in a simulation study.** In a simulation study, the true tree is known, and so can be used to measure error in an estimated tree. Note that the true tree is rooted and the inferred tree is unrooted; the error calculation is based on the (non-trivial) bipartitions induced by the internal edges, and so the true tree is interpreted as an unrooted tree. Some of the edges in the two trees are labelled, but others are not. The edges that are not labelled induce bipartitions that are in both trees; all other edges define bipartitions for only one of the two trees. False positive (FP) edges are those that are in the estimated tree but not the model tree, while false negative (FN) edges are those that are in the model tree but not the estimated tree. In this example, one of the two internal edges in the inferred tree is a false positive, and the other is a true positive; hence the false positive rate is 50%. Similarly, although the true tree is rooted, when we treat it as an unrooted tree, one of its internal edges is a true positive and the other is a false negative; hence the false negative rate is 50%. The number of false positive plus false negative edges, divided by \(2n - 6\) (where \(n\) is the number of leaves in each tree) is the Robinson-Foulds (RF) error rate. When both trees are binary, the FN, FP, and RF rates are identical.

computed using Maximum Parsimony heuristics are substantially more accurate than trees computed using Neighbor Joining.

As an example of this phenomenon, Figure 1.5 presents a reproduction of Figure 5(b) from Nakhleh et al. (2002). The study examined a heuristic for maximum parsimony (MP), neighbor joining (NJ), a variant of neighbor joining called Weighbor (Bruno et al., 2000), and a method called DCM-NJ+MP (Nakhleh et al., 2001) (see Chapter 11). The results shown here are for simulated data that evolve down a model tree with 400 leaves, under varying rates of evolution from low (diameter = 0.2) to high (diameter = 2.0), where the diameter indicates the expected number of changes for a random site on the longest leaf-to-leaf path. The y-axis shows the Robinson-Foulds error rate (i.e., the normalized...
Robinson-Foulds distance), so the maximum possible is 1.0. Note that neighbor joining is less accurate than maximum parsimony under all the tested conditions.

This figure also shows other trends that are very interesting. First, Weighbor is very similar to neighbor joining for low diameters, but for high diameters Weighbor is clearly more accurate. The difference between Weighbor and neighbor joining is most noticeable for the highest diameter condition. In fact, Weighbor is designed explicitly to deal better with high rates of evolution, and it does this by considering the statistical model of evolution in a more nuanced way (in particular, by noting that large distances have high variance).

Another observation is that all methods have higher error at the lowest diameter (0.2) than at intermediate diameters (e.g., 0.4), but error generally increases as the diameter increases from 0.4 and 2.0. However, the amount of increase in error depends on the method, which neighbor joining showing the largest sensitivity and Weighbor showing much less sensitivity. Again, this difference between neighbor joining and Weighbor is due to how Weighbor treats large distances in its distance matrix.

The best performing method in this simulation study is DCM-NJ+MP. This method uses a divide-and-conquer approach as follows. First, given a dataset of sequences (each of the same length), DCM-NJ+MP computes a distance matrix (the same matrix that neighbor joining and Weighbor compute). DCM-NJ+MP then uses the distance matrix to compute a collection of triangulated graphs, which are graphs that have no simple induced cycles of size greater than three (Golumbic, 2004), where the vertices of these graphs represent the input sequences. Each of these triangulated graphs is then decomposed into a set of maximal cliques, a problem that is usually NP-hard but is polynomial time for triangulated graphs (Golumbic, 2004). Neighbor joining trees are then computed for each set of sequences defined by one of the maximal cliques. Since the maximal cliques overlap (i.e., they share vertices), trees on these cliques will share leaves. For each triangulated graph, then, a supertree is computed from the neighbor joining trees on the maximal cliques. The set of supertrees (one for each triangulated graph) is then scored for maximum parsimony, referring to the original sequence data. The tree with the best maximum parsimony score is then selected as the tree for the input set.

1.8.2 Designing methods for improved accuracy and scalability

DCM-NJ+MP is an example of a disk-covering method (DCM), where a base phylogeny estimation method (here, neighbor joining) is applied to subsets in a carefully constructed divide-and-conquer framework. While DCM-NJ+MP uses a rather complicated algorithmic approach, the result is an improvement in accuracy over the other methods. Similar methods based on nearly the same algorithmic approach include the heuristic versions (Nakhleh et al., 2001) of DCM1-NJ (Warnow et al., 2001) and DCM-Buneman (Huson et al., 1999a), two of the earliest “absolute fast converging methods” (Warnow et al., 2001), which are methods that are proven to recover the true tree with probability converging to 1 from sequence lengths that grow only polynomially in the number of leaves.

Other divide-and-conquer methods have been developed that use iteration as well as
divide-and-conquer, in which each iteration uses the tree computed in the previous iteration to perform a decomposition of the dataset into subsets, constructs trees on subsets using a preferred phylogeny estimation method, and then combines the trees into a tree on the full dataset. Examples of methods that have this flavor include SATé (Liu et al., 2009a, 2011) and PASTA (Mirarab et al., 2015), methods for co-estimating multiple sequence alignments and trees, and DACTAL (Nelesen et al., 2012), a method for estimating a tree without estimating an alignment. Similarly, the algorithmic strategy used in DACTAL has been used to improve the scalability and accuracy of MP-EST (Liu et al., 2010), a statistical method for estimating the species tree from multiple gene trees (Bayzid et al., 2014).

In other words, phylogeny estimation methods can be built using other phylogeny estimation methods, with the goal of improving accuracy and/or speed. The algorithmic techniques that produce improved performance include divide-and-conquer, iteration, and techniques for selecting trees from collections of trees. However, we are also interested in establishing theoretical guarantees under stochastic models of evolution. Therefore, a proper understanding of the graph theory involved in the divide-and-conquer strategies, and the stochastic models of evolution operating on the sequence data that are used to construct the phylogenetic trees (or phylogenetic networks, as the case may be), is also important. The rest of this text provides these foundations.

1.9 Summary

We began with a discussion of some basic (and fairly simple) methods for phylogeny estimation – UPGMA, Maximum Parsimony, neighbor joining, and the Naive Quartet Method – and how they perform under some simple statistical models of sequence evolution. We observed that these methods have very different theoretical guarantees, and that Neighbor Joining and the Naive Quartet Method are both statistically consistent under standard sequence evolution models while UPGMA and Maximum Parsimony are not. Yet, we also observed that Maximum Parsimony solved heuristically can be more accurate than Neighbor Joining, and that the Naive Quartet Method may be unlikely to return any tree at all for large datasets, until the sequence lengths are very large (perhaps unrealistically large). Hence, knowing that a method is statistically consistent and polynomial time does not mean that it is superior on data to another method that may not be statistically consistent.

Later chapters will return to this issue, but under increasingly complex and realistic models of evolution. For example, in Chapter 8, we will discuss the standard sequence evolution models that are used in biological systematics, and the statistical methods that are used to analyze data under these models. Since these models assume sequences evolve only under substitutions, Chapter 9 addresses phylogeny estimation and multiple sequence alignment when sequences evolve also with insertions and deletions. Chapter 10 discusses species tree estimation, under genome-scale evolution models in which gene trees evolve within species trees. Chapters 14-16 describes models of language evolution, and the methods that have been used to estimate trees under these models. In each of these chapters, we will explore the theoretical guarantees of methods as well as their performance (in terms
28

Brief introduction to phylogenetic estimation

(100) taxa (400) taxa

Figure 5: Accuracy as a function of the diameter under the K2P+Gamma model for fixed sequence length (500) and two numbers of taxa.

5.4 The Influence of the Model of Sequence Evolution

We reported all results so far under the K2P+Gamma model only. However, we explored performance under the JC (Jukes-Cantor) model as well. The relative performance of the methods we studied was the same under the JC model as under the K2P+Gamma model. However, throughout the experiments, the error rate of the methods was lower under the JC model (using the JC distance-correction formulas) than under the K2P+Gamma model of evolution (using the K2P+Gamma distance-correction formulas). This might be expected for the Weighbor method, which is optimized for the JC model, but is not as easily explained for the other methods. Figure 6 shows the error rate of NJ on trees of diameter 0.4 under the two models of evolution. NJ clearly does better under the JC model than under the K2P+Gamma model; other methods result in similar curves. Correlating the decrease in performance with specific features in the model is a challenge, but the results clearly indicate that experimentation with various models of evolution (beyond the simple JC model) is an important requirement in any study.

6 Conclusion

In earlier studies we presented the DCM-NJ+MP method and showed that it outperformed the NJ method for random trees drawn from the uniform distribution on tree topologies and branch lengths as well as for trees drawn from an ecological distribution, in which the trees are birth-death trees with a moderate deviation from ultrametricity. Here we have extended our result to include the Weighbor and

Figure 1.5 Tree error of four methods on datasets with 400 sequences, as a function of the evolutionary diameter. This figure appeared as Figure 5(b) in Nakhleh et al. (2002). It shows tree error, measured using the Robinson-Foulds error rate, for four different methods, as a function of the evolutionary diameter (expected number of changes of a random site across the longest path in the tree). Sequences were evolved under Kimura 2-parameter model trees (Kimura, 1980) with gamma distributed rates across sites, and distances between sequences were computed under the K2P model. Note that increasing the evolutionary diameter tends to increase the estimation error, that neighbor joining (NJ) is the least accurate method on these data, and that DCM-NJ+MP is the most accurate.

of accuracy) on data. In many cases, the theoretical guarantees established for methods provide insight into the conditions in which they will or will not work well, but in some cases there is a gap between theory and practice.

Note that this gap does not imply that the theory is wrong, but only that it does not predict performance very well. In other words, statistical consistency is a statement about asymptotic performance, and so addresses performance given unbounded amounts of data, and theoretical guarantees about asymptotic performance do not have any direct relevance to performance on finite data.

Predicting performance on finite datasets is a fabulously interesting theoretical question, but very little has been established about this. For example, there are some upper bounds that have been established for the sequence lengths that suffice for some methods to return the true tree with high probability under simple sequence evolution models (discussed in Chapter 13), and some lower bounds as well. But even here, the theory does not provide reliable insights into the relative performance of methods on datasets.

Simply put, it is very difficult to predict the performance of a phylogeny estimation
1.10 Review questions

1. Consider the Cavender-Farris-Neyman (CFN) model. What are the parameters of a CFN model tree? What do these parameters mean?

2. What does it mean to say that a method is statistically consistent for estimating the CFN model tree topology?

3. What is the CFN distance correction? Why is it used?

4. For a given set $S$ of binary sequences, each of the same length, will the matrix of pairwise Hamming distances satisfy the triangle inequality? Will the matrix of pairwise CFN distances satisfy the triangle inequality?

5. What is the definition of an “additive matrix”?

6. Is a square matrix in which all diagonal entries are 0 and all off-diagonal entries are 1 ultrametric? Is it additive?

7. What is the Four Point Condition?

8. What is the Four Point Method? If you were given a $4 \times 4$ dissimilarity matrix, would you know how to use the Four Point Method to construct a tree on the matrix?

9. Recall the Naive Quartet Method. What is the input, and how does the Naive Quartet Method operate on the input?

10. Given a model tree and an estimated tree, each on the same set of 5 leaves, what is the maximum possible number of false positive edges?

1.11 Homework problems

1. Suppose $e$ is an edge in a CFN model tree, and $p(e) = 0.1$. What is $\lambda(e)$?

2. Suppose you know $\lambda(e)$ for a given edge $e$ in a CFN model tree. Recall that $\lambda(e) = -\frac{1}{2} \ln(1 - 2p(e))$. Write $p(e)$ as a function of $\lambda(e)$.

---

1 The source of this quote is unknown; it may be Yogi Berra, Jan van de Snepscheut, Walter Savitch, or perhaps others.
3. Suppose you have a tree $T$ rooted at leaf $R$, and $R$ has two children, $X$ and $Y$, and each of these nodes has two children that are leaves. Hence, $T$ has four leaves: $A$ and $B$, which are below $X$, and $C$ and $D$, which are below $Y$. Draw $T$.

4. Suppose you are given a binary tree $T$ on $n$ leaves $s_1, s_2, \ldots, s_n$, with positive branch lengths. Show how to compute the set $Q(T)$ of quartet tree topologies induced by $T$ on four-taxon subsets of its leaf set in $O(n^4)$ time.

5. Make up a CFN model tree in which the branch lengths on the edges are all different. Now compute the matrix of the $4 \times 4$ distance matrix you get using the branch lengths you wrote down. (Hence your matrix should have values for $\lambda_{A,B}, \lambda_{A,C}, \lambda_{A,D}, \lambda_{B,C}, \lambda_{B,D}$, and $\lambda_{C,D}$.)
   - What is the largest distance in the matrix?
   - What is the smallest distance in the matrix?

6. Consider a rooted tree $T$ where $R$ is the root, the children of $R$ are $X$ and $Y$, the children of $X$ are $A$ and $B$, and the children of $Y$ are $C$ and $D$. Consider the CFN model tree with this rooted topology, where $p(R,X) = p(R,Y) = p(Y,C) = p(Y,D) = 0.1$, and $p(X,A) = p(X,B) = 0.4$
   1. Compute the values for $\lambda(e)$ for every edge $e$, and draw the CFN tree with these branch lengths.
   2. Compute the CFN distance of the root to every leaf. Is this distance the same for every leaf, or does it depend on the leaf?
   3. Write down the matrix $M$ of leaf-to-leaf CFN distances for this tree.
   4. What is the longest leaf-to-leaf path in this tree?
   5. What is the smallest value in the $M$?
   6. Are the two leaves with this smallest distance siblings in the tree?
   7. Write down the three pairwise sums. Which one is the smallest?
   8. Is the matrix additive?

7. Consider the same rooted tree $T$ as for the previous problem, but with $p(R,X) = p(R,Y) = p(Y,C) = p(X,A) = 0.1$, and $p(Y,D) = p(X,B) = 0.4$
   1. Compute the values for $\lambda(e)$ for every edge $e$, and draw the CFN tree with these branch lengths.
   2. Compute the CFN distance of the root to every leaf. Is this distance the same for every leaf, or does it depend on the leaf?
   3. Compute the matrix $M$ of leaf-to-leaf CFN distances.
   4. What is the longest leaf-to-leaf path in this tree?
   5. What is the smallest value in the matrix $M$? Are the two leaves with this smallest distance siblings in the tree?
   6. Write down the three pairwise sums. Which one is the smallest?
   7. Is the matrix additive?

8. Consider how $\lambda(e)$ is defined by $p(e)$.
   1. Compute $\lim_{p(e) \to 0} \lambda(e)$
2. Compute \( \lim_{p(e) \to 0} \lambda(e) \)
3. Graph \( \lambda(e) \) as a function of \( p(e) \), noting that \( 0 < p(e) < 0.5 \).
9. Let \( A, B \) and \( C \) be three binary sequences, each of length \( k \), and consider the values for \( \hat{\lambda}_{A,B}, \hat{\lambda}_{A,C}, \) and \( \hat{\lambda}_{B,C} \). Prove or disprove: for all \( A, B, C \), \( \hat{\lambda}_{A,B} + \hat{\lambda}_{B,C} \geq \hat{\lambda}_{A,C} \).
10. Let \( T \) be a CFN model tree with substitution probabilities \( p(e) \) for every edge \( e \) in \( T \). Let \( v \) and \( w \) be two vertices in \( T \) and let \( e_1, e_2, \ldots, e_m \) be the edges in the path from \( v \) to \( w \). Suppose you are given a binary tree \( T \) on \( n \) leaves \( s_1, s_2, \ldots, s_n \), with positive branch lengths. Show how to compute the set \( Q(T) \) of quartet tree topologies induced by \( T \) on four-taxon subsets of its leaf set in \( O(n^4) \) time.
11. Give an example of an \( 4 \times 4 \) normalized Hamming distance matrix \( H \) so that the Four Point Method applied to \( H \) yields a tree \( T \) that is different from the tree obtained by using the Four Point Method applied to Cavender-Farris-Neyman distances computed for \( H \).
12. Draw the rooted tree with edge weights that proves that a \( 4 \times 4 \) matrix in which all diagonal entries are 0 and all off-diagonal entries are 4 is ultrametric.
13. Suppose you have two trees \( T_1 \) and \( T_2 \), where \( T_1 \) is binary (fully resolved) but \( T_2 \) may have polytomies. If you treat \( T_1 \) as the true tree and \( T_2 \) as the estimated tree, is it necessarily the case that the Robinson-Foulds (RF) error rate is the average of the FN (false negative) error rate and the FP (false positive) error rate? If so, prove it, and otherwise give a counterexample. Robinson-Foulds error rate is the average of the
2
Trees

2.1 Introduction
A tree is just a special type of graph, with the constraint being that it is connected and acyclic; equivalently, a tree is a graph so that for every pair of vertices \( v, w \) in the graph there is a unique path between \( v \) and \( w \). Trees, and especially rooted trees, are used to represent evolutionary histories. In this context, they may be called “phylogenies”, “phylogenetic trees”, or “evolutionary trees”. The leaves of a rooted tree are the nodes that do not have any children, and these correspond to nodes of degree one in unrooted trees; the other nodes are called “internal nodes”. In a phylogenetic tree, the leaves represent the taxa of interest (generally extant species, but sometimes different individuals from the same species) and the internal nodes represent the ancestors of the taxa at the leaves. The most recent common ancestor (MRCA) of a set \( X \) of leaves in a rooted tree \( T \) is the node \( v \) that is a common ancestor of all nodes in \( X \), and that is further from the root of \( T \) than all other common ancestors.

Most statistical models of evolution assume that the model tree is a rooted binary tree, so that every node that is not a leaf has exactly two children. However, estimated trees will in general be unrooted and may not be binary. Thus, trees can be rooted or unrooted, and may be binary or may have nodes of high degree (called “polytomies”). This section is devoted to understanding the terminology regarding trees, learning how to move between rooted and unrooted versions of the same tree, how to determine whether two trees are the same or different, etc.

2.2 Rooted trees
We begin with a discussion of rooted trees. For a rooted tree \( T \) with leaf set \( S \), we draw the tree with the root \( r \) on top, on the bottom, on the left, or on the right – implicitly giving the edges an orientation (usually away from the root, towards the leaves). In this document, we’ll draw them as rooted at the top.

We begin with some basic definitions.

Definition 2.1 In a rooted tree, we can orient the edges in the direction of the root \( r \), so that all vertices other than \( r \) have outdegree one. Thus, for all nodes \( v \neq r \), there is a unique
2.2 Rooted trees

Figure 2.1 Two ways of drawing the same tree

vertex \( w \) such that \( v \rightarrow w \) is an arc (directed edge) in the tree. This vertex \( w \) is the parent of \( v \), and \( v \) is the child of \( v \). Two or more vertices sharing the same parent are siblings. A vertex without any children is called a leaf. A vertex with more than two children is a polytomy.

Thus, we can also define a binary tree as one that does not contain any polytomies!

**Definition 2.2** A rooted tree is said to be **binary** or **fully resolved** if it does not contain any polytomies. A rooted tree is said to be a **binary tree** if it does not contain any nodes with more than two children. If a tree is not binary, it is said to be **multifurcating**.

The representation of a polytomy can vary between different graphical representations. In Figure 2.2, we show two equivalent representations of the same branching process. One of these (on the left) is standard in computer science, and the other (on the right) is often found within biological systematics. Note that the horizontal lines do not necessarily correspond to edges.

Graphical representations of trees sometimes include branch lengths, to help suggest relative rates of change and/or actual amounts of elapsed time. The “topology” of the tree is independent of the branch lengths, however, and is generally speaking the primary interest of the systematist.

### 2.2.1 Newick notation for rooted trees

The first task is to be able to represent trees using Newick format: \( ((a,b),(c,d)) \) represents the rooted tree with four leaves, \( a, b, c, d \), with \( a \) and \( b \) siblings on the left side of the root, and \( c \) and \( d \) siblings on the right side of the root.

The same tree could have been written \( ((c,d),(a,b)) \), or \( ((b,a),(d,c)) \), etc. Thus, the graphical representation is somewhat flexible – swapping sibling nodes (whether leaves or
internal vertices in the tree) doesn’t change the tree “topology”. As a result, there are only three rooted trees on leaf-set \{a,b,c\}!

Similarly, the following Newick strings refer to exactly the same rooted tree as in Figure 2.3:

- \(((d,e),(c,(a,b)))\)
- \(((e,d),(a,b),c))\)
- \(((e,d),(c,(a,b)))\)

Similarly, there are exactly 8 different Newick representations for the rooted tree given in Figure 2.3:

- \(((a,b),(c,d))\)
- \(((b,a),(c,d))\)
- \(((a,b),(d,c))\)
- \(((b,a),(d,c))\)
- \(((c,d),(a,b))\)
- \(((c,d),(b,a))\)
- \(((d,c),(a,b))\)
- \(((d,c),(b,a))\)

The second fundamental task is to be able to recognize when two rooted trees are the same. Thus, when you don’t consider branch lengths, the three trees given in Figures 2.3 are different drawings of the same basic tree.

Sometimes the rooted tree you want to represent is not binary. To represent these trees using Newick notation is quite simple. For example, a rooted tree without any internal edges and six leaves \(a,b,c,d,e,f\), is represented by \((a,b,c,d,e,f)\). Similarly, a tree with
2.2 Rooted trees

three children $u, v, w$ off the root, and each of the children of the root has two more children; thus, $u$ has children $u_1$ and $u_2$, $v$ has children $v_1$ and $v_2$, and $w$ has children $w_1$ and $w_2$. To represent the tree, we would use $((u_1, u_2), (v_1, v_2), (w_1, w_2))$. Thus, we can also use Newick strings to represent non-binary trees.

2.2.2 The clade representation of a rooted tree

We begin with some basic terminology.
Definition 2.3  Let $T$ be a rooted tree in which every leaf is labelled by a distinct element from a set $S$. Thus, $\mathcal{L}(T) = S$, and is called the leafset of $T$. Any subset $A$ of $\mathcal{L}(T)$ that comprises the leafset of the subtree of $T$ rooted at some node $v$ of $T$ is called a clade of $T$, and is denoted $\mathcal{L}(T_v)$. The root of the tree $T$ is denoted by $r(T)$, and the set of vertices of $T$ is denoted by $V(T)$ (note that $V(T)$ includes the leafset $\mathcal{L}(T)$ and also the internal nodes).

We now show how to use the clades of a tree to compare it to other trees.

Definition 2.4  Let $T$ be a rooted tree on leaf-set $S$. We define the set $\text{Clades}(T) = \{ \mathcal{L}(T_v) : v \in V(T) \}$. Thus, $\text{Clades}(T)$ has all the singleton sets (each containing one leaf), a set containing all the taxa (defined by the root of $T$), and then a clade for every remaining vertex of $T$. The clades that appear in every tree on $S$ are called the trivial clades, and all other clades are called the non-trivial clades. Thus, all the singleton clades and the set $S$ are trivial clades.

Example 2.5  Consider the tree $T = ((a,b),(c,(d,e)))$. The trivial clades are $\{a\}, \{b\}, \{c\}, \{d\}, \{e\}$, and $\{a,b,c,d,e\}$; these appear in every possible tree on the leafset of $T$. The non-trivial clades are $\{a,b\}, \{a,b,c\}$, and $\{d,e\}$. Hence,

$$\text{Clades}(T) = \{\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{a,b\}, \{a,b,c\}, \{a,b,d,e\}, \{a,b,c,d,e\}\}.$$

Testing if two rooted trees are identical:  Determining if two rooted leaf-labelled trees are the same (with all leaves labelled distinctly) can be difficult if they are drawn differently. However, this is easy if you examine the clades! Thus, to determine if two trees $T$ and $T'$ are the same, you can write down the set of clades for the two trees, and see if the sets are identical. If $\text{Clades}(T) = \text{Clades}(T')$, then $T = T'$; otherwise, $T \neq T'$. For example, if you compare the trees in Figures 2.3, 2.4, and 2.5, you'll see that they all have the same set of clades. Thus, they are all identical.

2.2.3 Constructing a rooted tree from its set of clades

We now show how to compute a tree from its set of clades. To do this, consider the binary relation $R$ on the set $\text{Clades}(T)$, where $A, B \in R$ if and only if $A \subseteq B$. It is not hard to see that $R$ is a partial order, and so the set of clades of a tree, under this relation, is a partially ordered set.

Recall the definition of a Hasse Diagram for a partially ordered set. We now construct the Hasse Diagram for this partially ordered set. We make a graph with vertex set $\text{Clades}(T)$ and a directed edge from a node $x$ to a different node $y$ if $x \subseteq y$. Since containment is transitive, if $x \subseteq y$ and $y \subseteq z$, then $x \subseteq z$. Hence, if we have directed edges from $x$ to $y$, and from $y$ to $z$, then we know that $x \subseteq z$, and so can remove the directed edge from $x$ to $y$ without loss of information. This is the basis of the Hasse Diagram: you take the graphical representation of a partially ordered set, and you remove directed edges that are implied...
by transitivity. Equivalently, for a given subset $x$, you find the smallest subsets $y$ such that $x \subset y$, and you put a directed edge from $x$ to $y$.

As we will see, the Hasse Diagram formed for a set $\text{Clades}(T)$ is the tree $T$ itself. You can run the algorithm on an arbitrary set of subsets of a taxon set $S$, but the output may or may not be a tree.

**Example 2.6** Consider $A = \{\{a\}, \{a, b, c, d\}, \{a, d, e, f\}, \{a, b, c, d, e, f\}\}$. On this input, there are four sets, and so the Hasse Diagram will have four vertices. Let $v_1$ denote the set $\{a\}$, $v_2$ denote the set $\{a, b, c, d\}$, $v_3$ denote the set $\{a, d, e, f\}$, and $v_4$ denote the set $\{a, b, c, d, e, f\}$. Then, in the Hasse Diagram, we will have the following directed edges: $v_1 \rightarrow v_2$, $v_1 \rightarrow v_3$, $v_2 \rightarrow v_4$, and $v_3 \rightarrow v_4$. This is not a tree, since it has a cycle (even though this is only a cycle when considering the graph as an undirected graph).

**Theorem 2.7** Let $T$ be a rooted tree in which every internal node has at least two children. Then the Hasse Diagram constructed for $\text{Clades}(T)$ is isomorphic to $T$.

**Proof** We prove this by strong induction on the number $n$ of leaves in $T$. For $n = 1$, then $T$ consists of a single node (since every node has at least two children). When we construct the Hasse Diagram for $T$, we obtain a single node, which is the same as $T$.

The inductive hypothesis is that the statement is true for all positive $n$ up to $N - 1$, for some arbitrary positive integer $N$. We now consider a tree $T$ with $N$ leaves for which every internal node has at least two children. Since the root of $T$ has at least two children, we denote the subtrees of the root as $t_1, t_2, \ldots, t_k$ (with $k \geq 2$). Note that $\text{Clades}(T) = \bigcup_i \text{Clades}(t_i) \cup \mathcal{L}(T)$. Note also that the set of vertices for the Hasse Diagram on $T$ contains one vertex for $\mathcal{L}(T)$ and then each of the vertices for the Hasse Diagrams on the $t_i$, $i = 1, 2, \ldots, k$. Also, every directed edge in the Hasse Diagram on $T$ is either a directed edge in the Hasse Diagram on some $t_i$, or is the directed edge from $\mathcal{L}(t_i)$ to $\mathcal{L}(T)$. By the inductive hypothesis, the Hasse Diagram defined on $\text{Clades}(t_i)$ is isomorphic to $t_i$ for $i = 1, 2$, and hence the Hasse Diagram defined on $\text{Clades}(T)$ is isomorphic to $T$. 

### 2.2.4 Compatible sets of clades

Up to now, we have assumed we were given the set $\text{Clades}(T)$, and we wanted to construct the tree $T$ from that set. When the set of subsets is a subset of the set of clades of a tree, we say that the set of subsets is compatible, and otherwise we say it is not compatible.

**Definition 2.8** A set $\mathcal{X}$ of sets is said to be **compatible** if and only if there is a rooted tree $T$ with each leaf in $T$ given a different label, so that $\mathcal{X} \subseteq \text{Clades}(T)$.

Here we consider a related question: given a set $\mathcal{X}$ of subsets of a set $S$ of taxa, is there a tree $T$ so that $\mathcal{X} \subseteq \text{Clades}(T)$? To answer this, see what happens when you construct the Hasse Diagram for the set $\mathcal{X}$. 


Example 2.9  We begin with a simple example, $\mathcal{X}_1 = \{\{a,b,c\}, \{d,e,f\}, \{a,b\}\}$. Note that $\mathcal{X}_1$ contains three subsets and the set $S$ contains six elements. Thus, $\mathcal{X}_1$ does not contain the singleton sets, nor the full set of leaves, and so it is not possible for $\mathcal{X}_1$ to be equal to the set of clades of any tree; and as we observe, the Hasse Diagram we construct is not connected and so is not a tree. Therefore, we add all the trivial clades (the singletons and the full set of leaves) to $\mathcal{X}_1$ and obtain $\mathcal{X}_1'$. We then compute the Hasse Diagram on this set. Note that the result is a tree $T$, with Newick string $(((a,b),c),(d,(e,f)))$. This is not a binary tree, but it is a tree, and $\mathcal{X}_1 \subset \text{Clades}(T)$. However, there are other trees, such as $T'$ denoted by $(((a,b),c),(d,(e,f)))$, that also satisfy $\mathcal{X}_1 \subset \text{Clades}(T')$. Note that $T$ can be derived from $T'$ by contracting an edge in $T'$.

Example 2.10  Consider the set of subsets $\mathcal{X}_2 = \{\{a,b\}, \{b,e\}, \{c,d\}\}$. Note that $\mathcal{X}_2$ contains three sets and the set $S$ contains five elements. Also, as we saw for $\mathcal{X}_1$, $\mathcal{X}_2$ does not contain the singleton sets nor the full set of leaves, and so $\mathcal{X}_2$ is not a set of clades of any tree. We add all the trivial clades to $\mathcal{X}_2$ to obtain $\mathcal{X}_2'$, and construct the Hasse diagram for $\mathcal{X}_2'$. Note that $\{b\} \subset \{a,b\}$ and $\{b\} \subset \{b,e\}$. Hence, the Hasse Diagram for $\mathcal{X}_2'$ has a node with outdegree two – which is inconsistent with $X_2'$ being the subset of Clades($T$) for some tree $T$.

2.2.5 Hasse Diagram algorithm

These two examples suggest an algorithm that you use to determine if a set of clades it is compatible. We call the algorithm the Hasse Diagram algorithm, since it operates by computing a Hasse Diagram, and then checking to see if the Hasse Diagram is a tree.

The input will be a set $\mathcal{X}$ of subsets of a taxon set. The output will either be a rooted tree $T$ for which $\mathcal{X} \subseteq \text{Clades}(T)$, establishing that $\mathcal{X}$ is compatible, or $\text{Fail}$.

- Step 1: Compute $S = \cup_{X \in \mathcal{X}} X$ (i.e., all the elements that appear in any set in $\mathcal{X}$). Let $S = \{s_1, s_2, \ldots, s_n\}$. Define $\mathcal{X}' = \mathcal{X} \cup \{s_1\} \cup \{s_2\} \cup \ldots \cup \{s_n\}$; in other words, $\mathcal{X}'$ is the set of subsets of $S$ obtained by adding the full set $S$ and all the singletons to $\mathcal{X}$.
- Step 2: Construct the Hasse Diagram for $\mathcal{X}'$.
- Step 3: If the Hasse Diagram is a tree, then return $T$; otherwise return $\text{Fail}$.

Lemma 2.11  A set $\mathcal{X}$ of subsets is compatible if and only if for any two elements $X_1$ and $X_2$ in $\mathcal{X}$, either $X_1$ and $X_2$ are disjoint or one contains the other.

Proof  If a set $\mathcal{X}$ of subsets is compatible, then there is a rooted tree $T$ on leaf set $S$, in which every leaf has a different label, so that each element in $\mathcal{X}$ is the set of leaves below some vertex in $T$. Let $X_1$ and $X_2$ be two elements in $\mathcal{X}$, and let $v_1$ be the vertex of $T$ associated to $X_1$ and $v_2$ be the vertex associated to $X_2$. If $x_1$ is an ancestor of $x_2$, then $X_1$ contains $X_2$, and similarly if $x_2$ is an ancestor of $x_1$ then $X_2$ contains $X_1$. Otherwise neither is an ancestor of the other, and the two sets are disjoint. For the reverse direction, note that when all pairs of elements in set $A$ satisfies this property, then the Hasse Diagram will be a tree $T$ so that $\mathcal{X} = \text{Clades}(T)$.

$\square$
2.2 Rooted trees

The following corollary follows immediately, and will be very useful in algorithm design!

**Corollary** A set $\mathcal{X}$ of subsets of $S$ is compatible if and only if every pair of elements in $\mathcal{X}$ is compatible.

2.2.6 Difficulties in rooting trees

Although evolutionary trees are rooted, estimations of evolutionary trees are almost always unrooted, for a variety of reasons. In particular, unless the taxa (languages, genes, species, whatever) evolve under a “strong clock” (so that the expected number of changes is proportional to the time elapsed since a common ancestor), rooting trees requires additional information. The typical technique is to use an “outgroup” (a taxon which is not as closely related to the remaining taxa as they are to each other). The outgroup taxon is added to the set of taxa and an unrooted tree is estimated on the enlarged set. This unrooted tree is then rooted by “picking up” the unrooted tree at the outgroup. See Figure 2.4, where we added a fly to a group of mammals. If you root the tree at the fly, you obtain the rooted tree $(\text{cow}, (\text{chimp}, \text{human}))$, showing that chimp and human have a more recent common ancestor than cow has to either human or chimp.

The problem with this technique is subtle: while it is generally easy to pick outgroups, the less closely related they are to the remaining taxa, the less accurately they are placed in the tree. That is, very distantly related taxa tend to fit equally well into many places in the tree, and thus produce incorrect rootings. See Figure 2.5, where the outgroup (marked by “outgroup”) attaches into two different places within the tree on the remaining taxa. Note how the trees on the remaining taxa are different as rooted trees (when rooted at the outgroup), although identical as unrooted trees.
Figure 2.5 Two unrooted trees that differ only in the placement of the outgroup. If these trees were rooted at the outgroup, they would produce different rooted trees on the ingroup taxa $a, b, c, d$.

Furthermore, it is often difficult to distinguish between an outgroup taxon that is closely related to the ingroup taxa, and a taxon that is, in fact, a member of the same group which branched off early in the group’s history. For this reason, even the use of outgroups is somewhat difficult.

### 2.3 Unrooted trees

We begin with writing down rooted versions of unrooted trees, and then writing down unrooted versions of rooted trees.

#### 2.3.1 Newick notation for unrooted trees

First, the Newick format that is used to represent a rooted tree is also used to represent its unrooted version. In other words, every unrooted tree will have several Newick representations, for each of the ways of rooting the unrooted tree. Since phylogeny estimation methods almost universally produce unrooted trees, although the output of a phylogeny estimation procedure may be given in a rooted form, the particular location of the root is irrelevant and should be ignored.

Now that you know how to draw unrooted versions of rooted trees, we will do the reverse. You can generate rooted trees from an unrooted tree by picking up the tree at any edge, or at any node. You can even pick up the tree at one of its leaves, but then the tree is rooted at one of its own taxa – which we generally don’t do (in that case, we’d root it at the edge leading to that leaf instead, thus keeping the leaf set the same). Suppose we consider the unrooted tree given in Figure 2.6, which has four leaves: $a, b, c, d$, where $a$ and $b$ are siblings, and $c$ and $d$ are siblings. This tree has five edges and two internal nodes. If we root the tree at one of the internal nodes, we will get a rooted tree with three children, while rooting the tree at an edge gives a rooted tree in which all nodes have two children. More generally, if we root a binary unrooted tree (i.e., an unrooted tree in which all internal nodes have degree three) on an edge, we obtain a rooted binary tree.

**Definition 2.12**  Every node in an unrooted tree is either a leaf (in which case it has degree one) or an internal node. Two or more nodes with a common neighbor are siblings.
2.3 Unrooted trees

Two of the rooted trees consistent with the unrooted tree given in Figure 2.6 are provided in Figures 2.9 and 2.10.

2.3.2 The bipartitions of an unrooted tree

To determine if two unrooted trees are the same, we do something similar to what we did to determine if two rooted trees are the same. However, since the trees are unrooted, they do not contain clades (rooted subtrees). Therefore, we will work with bipartitions (partitions of the taxon set into two disjoint sets) rather than clades.

The bipartitions of an unrooted tree are formed by taking each edge in turn, and writing down the two sets of leaves that would be formed by deleting that edge. Note that when the edge is incident to a leaf, then the bipartition is trivial – it splits the set of leaves into
one set with a single leaf, and the other set with the remaining leaves. These bipartitions are present in all trees with any given leaf set. Hence, we will focus the discussion just on the non-trivial bipartitions.

For the tree in the previous section with four leaves $a, b, c,$ and $d$, there was only one non-trivial bipartition, splitting $a$ and $b$ on one side from $c$ and $d$ on the other. We denote this bipartition by $\{\{a, b\}|\{c, d\}\}$, or more simply by $(ab|cd)$. Note that we could have denoted this by $(cd|ab)$ or $(dc|ab)$, etc.; the order in which the taxa appear within any one side does not matter, and you can put either side first. Note also that we can omit commas, as long as the meaning is clear.

We summarize this discussion with the following definition:

**Definition 2.13** Given an unrooted tree $T$ with no nodes of degree two, the **bipartition encoding** of $T$, denoted by $C(T) = \{\pi(e) : e \in E(T)\}$, is the set of bipartitions defined by each edge in $T$, where $\pi(e)$ is the bipartition on the leaf set of $T$ produced by removing the edge $e$ (but not its endpoints) from $T$. If we restrict this set to the bipartitions formed by the internal edges of the tree $T$, we obtain $C_I(T)$.

### 2.3.3 Representing non-binary trees

Sometimes the unrooted tree we wish to represent is not fully resolved, which means it has nodes of degree greater than three. How do we represent such a tree? For example, consider the tree that has one internal node and four leaves, $a, b, c, d$. We can represent this simply by $(a, b, c, d)$. Note also that representing it by $(a, (b, c, d))$ yields the same unrooted tree. Similarly, what about a tree that has six leaves, $a, b, c, d, e, f$, and one internal edge that separates $a, b$, from $c, d, e, f$? We can represent this unrooted tree by $(a, b, (c, d, e, f))$, or any of the alternatives that also yield one single bipartition separating $a, b$ from the remaining leaves.

Sometimes, if the tree has only a single bipartition, we will simplify our representation
by just giving the bipartition; i.e., we represent the tree above by \( \{a,b\}\mid \{c,d,e,f\} \), or more simply by \( ab|cdef \). In other words, the representations for trees that appear in the mathematical literature are quite flexible. (Of course, representations of trees in software must be done very precisely, using the requirements for the software... but that is another matter.)

### 2.3.4 Comparing trees using their bipartitions

It is easy to see that we can write down the set of bipartitions of any given unrooted tree, and that two unrooted trees are identical if they have the same set of bipartitions. However, other relationships can also be inferred: for example, we can see when one tree refines another, by comparing their bipartitions. That is, if \( T \) and \( T' \) are two trees on the same leaf set, then \( T \) is said to refine \( T' \) (and conversely, we say that \( T' \) is a contraction of \( T \)) if \( T' \) can be obtained from \( T \) by contracting some edges in \( T \). In fact, \( T \) refines \( T' \) if and only if \( C(T') \subseteq C(T) \). (Note that using this definition, each tree refines itself, and is also a contraction of itself, since we can choose to contract no edges.)

**Definition 2.14** Given two trees \( T \) and \( T' \) on the same set of leaves (and each leaf given a different label), tree \( T \) is said to refine \( T' \) if \( T' \) can be obtained from \( T \) by contracting a set of edges in \( T \). We also express this by saying \( T \) is a refinement of \( T' \) and \( T' \) is a contraction of \( T \).

**Definition 2.15** An unrooted tree \( T \) is fully resolved if there is no tree \( T' \neq T \) that refines \( T \). Equivalently, \( T \) is fully resolved if all the nodes in \( T \) have degree 1 or 3. An unrooted tree that is fully resolved is also called a binary tree. (Note, however, that we also referred to rooted binary trees, so that “binary tree” has a slightly different meaning for rooted and unrooted trees.)

### 2.3.5 Constructing \( T \) from \( C(T) \)

Sometimes we are given a set \( A \) of bipartitions, and we are asked whether these bipartitions could co-exist within a tree (i.e., whether there exists a tree \( T \) so that \( A \subseteq C(T) \)). When this is true, the set of bipartitions is said to be compatible, and otherwise the set is said to be incompatible.

**Definition 2.16** A set \( A \) of bipartitions on the set \( S \) is compatible if there exists an unrooted tree \( T \) in which every leaf has a distinct label from a set \( S \), so that \( A \subseteq C(T) \).

To construct a tree from a compatible set \( A \) of bipartitions, we will use \( A \) to construct a set \( C \) of clades that will be compatible if and only if the set \( A \) is compatible. We will then run the Hasse Diagram Algorithm from Chapter 2.2.5 on the set \( C \). If \( C \) is a compatible set of bipartitions, this will return a rooted tree \( T \) realizing the set \( C \). Then, to construct the unrooted tree for \( A \), we will return \( T_u \), the unrooted version of \( T \). We will refer to \( T_u \) as the canonical tree for the set \( A \).
Figure 2.9 Unrooted tree on \{1...9\}, obtained by running the Hasse Diagram algorithm on the set \(A = \{(123|456789), (12345|6789), (12|3456789), (89|1234567)\}\); see Example 2.17.

To complete this description, we only need to say how we compute the set \(C\) of clades given \(A\). First, we add all of the missing trivial bipartitions (the ones of the form \(x|S - \{x\}\)) to \(A\). Then, pick any leaf (call it “r”) in the set to function as a root. This has the result of turning the unrooted tree into a rooted tree, and therefore turns the bipartitions into clades. For each bipartition \(A|B\), we write down the subset that does not contain \(r\), and denote it as a clade. We also include \(S\) (the full set of taxa) and \(\{x\}\) for each \(x \in S\) (the singleton sets). This is set \(C\) of clades we obtain from the set \(A\) of bipartitions.

**Example 2.17** We will determine if the set \(A\) of bipartitions given by \(A = \{(123|456789), (12345|6789), (12|3456789), (89|1234567)\}\) is compatible, and if so we will construct its canonical tree. First, we decide to root the tree at leaf 1. We look at each bipartition, and select the half of the bipartition that does not contain 1. Thus, we obtain the following set of clades:

\[
\{\{4, 5, 6, 7, 8, 9\}, \{6, 7, 8, 9\}, \{3, 4, 5, 6, 7, 8, 9\}, \{8, 9\}\}
\]

We then add the full set \(S\) and all the singleton sets, and construct a Hasse Diagram for this set of sets. We note that the Hasse Diagram has the property that every node has outdegree 1, and hence defines a rooted tree given by \((1, (2, (3, (4, 5, ((6, 7), (8, 9))))))\). Note that although we treat 1 as a root in order to form clades, this technique produces a tree in which 1 is a leaf and not the root. We then unroot this tree to obtain the tree given in Figure 2.9.

### 2.3.6 Testing compatibility of a set of bipartitions

What we have described is how to construct a tree from a set of compatible bipartitions. However, what if we want to determine if the set is compatible? In that case, we follow nearly the same algorithm, with the following modification. When we construct the Hasse
2.3 Unrooted trees

Diagram, we check that it creates a tree. If it does, then we return “Compatible” and otherwise we return “Not Compatible”.

It is easy to verify that this method returns the correct answer when the set is compatible. What about when the set is not compatible? We demonstrate this with an example.

Example 2.18 Suppose the set of bipartitions has two bipartitions \( ab|cd \) and \( ac|bd \). We root the bipartitions at leaf \( a \), and obtain the non-trivial clades \( \{c, d\}, \{b, d\}, \{b, c, d\} \). We add \( \{a, b, c, d\} \) and the singleton sets. When we compute the Hasse Diagram, we note that the graph has a cycle (as an undirected graph) on the vertices for clades \( \{d\}, \{c, d\}, \{b, d\} \), and \( \{b, c, d\} \). Hence, the Hasse Diagram is not a tree, and the algorithm returns “Not Compatible”.

Pairwise Compatibility ensures Setwise Compatibility: Just as we saw with testing compatibility for clades, it turns out that bipartition compatibility has a simple characterization, and pairwise compatibility ensures setwise compatibility.

Theorem 2.19 A set \( A \) of bipartitions on a set \( S \) is compatible if and only if every pair of bipartitions is compatible. Furthermore, a pair \( X = (X_1, X_2) \) and \( Y = (Y_1, Y_2) \) of bipartitions is compatible if and only if at least one of the four pairwise intersections \( X_i \cap Y_j \) is empty.

Proof We begin by proving that a pair of bipartitions is compatible if and only if at least one of the four pairwise intersections is empty. It is easy to see that a pair of bipartitions is compatible if and only if the clades produced (for any way of selecting the root) are compatible. So let’s assume that we set \( s \) to be the root (for an arbitrary element \( s \in S \), and that \( s \in X_1 \cap Y_1 \). Therefore, \( X \) and \( Y \) are compatible as bipartitions if and only if \( X_2 \) and \( Y_2 \) are compatible as clades. Therefore, \( X \) and \( Y \) are compatible as bipartitions if and only if one of the following statements holds:

- \( X_2 \subseteq Y_2 \)
- \( Y_2 \subseteq X_2 \)
- \( X_2 \cap Y_2 = \emptyset \)

If the first condition holds, then \( X_2 \cap Y_1 = \emptyset \), and at least one of the four pairwise intersections is empty. Similarly, if the second condition holds, then \( Y_2 \cap X_1 = \emptyset \), and at least one of the four pairwise intersections is empty. If the third condition holds, then directly at least one of the four pairwise intersections is empty. Thus, if \( X \) and \( Y \) are compatible as bipartitions, then at least one of the four pairwise intersections is empty.

For the converse, suppose that \( X \) and \( Y \) are bipartitions on \( S \), and at least one of the four pairwise intersections is empty; we will show that \( X \) and \( Y \) are compatible as bipartitions. Assume that \( X_1 \cap Y_1 \neq \emptyset \), and let \( s \in X_1 \cap Y_1 \). Hence, to show that \( X \) and \( Y \) are compatible as bipartitions it will suffice to show that \( X_2 \) and \( Y_2 \) are compatible as clades. Since \( X_1 \cap Y_1 \neq \emptyset \), the pair that produced the empty intersection must be one of the other pairs; i.e., one of the following must be true: \( X_1 \cap Y_2 = \emptyset, X_2 \cap Y_2 = \emptyset, \) or \( X_2 \cap Y_1 = \emptyset \). If \( X_1 \cap Y_2 = \emptyset \), then
Trees

$Y_2 \subseteq X_2$, and $X_2$ and $Y_2$ are compatible clades; thus, $X$ and $Y$ are compatible bipartitions. If $X_2 \cap Y_1 = \emptyset$, then a similar analysis shows that $X_2 \subseteq Y_2$, and so $X$ and $Y$ are compatible bipartitions. Finally, if $X_2 \cap Y_2 = \emptyset$, then directly $X_2$ and $Y_2$ are compatible clades, and so $X$ and $Y$ are compatible bipartitions.

Now that we have established that two bipartitions are compatible if and only if at least one of the four pairwise intersections is empty, we show that a set of bipartitions is compatible if and only if every pair of bipartitions is compatible. So let $s \in S$ be selected arbitrarily as the root, and consider all the clades (halves of bipartitions) that do not contain $s$. This set of subsets of $S$ is compatible if and only if every pair of subsets is compatible, by Theorem 2.11. Hence, the theorem is proven.

2.4 Constructing the Strict Consensus Tree

A common event in phylogeny estimation is that a set of trees is computed for a given dataset, and we will wish to compute a consensus of these trees.

Definition 2.20 The strict consensus tree of a set $T = \{T_1, T_2, \ldots, T_k\}$ of trees is the most resolved common contraction of the trees in $T$. Hence, $T$ is the strict consensus of $T$ if and only if every tree $T_i \in T$ refines $T$, and every other tree satisfying this property is a refinement of $T$. Furthermore, the strict consensus tree $T$ satisfies $C(T) = \cap_i C(T_i)$.

2.5 Quantifying error in estimated trees

The context in which we will be interested in trees is where we are estimating trees from data, but are hoping to come "close" to the true tree. Since the true tree is unknown, determining how close we have come is often difficult. However, for the purposes of this section, we will presume that the true tree is known, so that we can compare estimated trees to the true tree.

Let us presume that the tree $T_0$ on leaf set $S$ is the true tree, and that another tree $T$ is an estimated tree for the same leaf set. There are several techniques that have been used to quantify errors in $T$ with respect to $T_0$, of which the dominant ones are these:

False Negatives (FN): The false negatives are those edges in $T_0$ inducing bipartitions that do not appear in $C(T)$; these are also called “missing branches”. The false negative rate is the fraction of the total number of non-trivial bipartitions that are missing, or $\frac{|C(T_0) \setminus C(T)|}{|C(T_0)|}$.

False Positives (FP): The false positives in a tree $T$ with respect to the tree $T_0$ are those edges in $T$ that induce bipartitions that do not appear in $C(T_0)$. The false positive rate is the fraction of the total number of non-trivial bipartitions in $T$ that are false positives, or $\frac{|C(T) \setminus C(T_0)|}{|C(T)|}$.

Robinson-Foulds (RF): The most typically used error metric is the sum of the number of false positives and false negatives, and is called the Robinson-Foulds distance or
the **bipartition distance**. This distance ranges from 0 (so the trees are identical) to at most $2n - 6$, where $n$ is the number of leaves in each tree. To turn this into an error rate, that number is divided by the total number of bipartitions in the two trees.

Generally, evolutionary trees are presumed to be *binary*, so that all internal nodes have three neighbors (or, if rooted, then every internal node has two children). In this case, the number of internal edges in the tree is $n - 3$, and false negative error rates are produced by dividing by $n - 3$. When both the estimated and true trees are binary, then false negative and false positive rates are equal, and these also equal the Robinson-Foulds distance. The main advantage in splitting the error rate into two parts (false negative and false positive) is that many estimated trees are not binary. In this case, when the true tree is presumed to be binary, the false positive error rate will be less than the false negative error rate. Note also that the reverse can happen – the false negative error rate could be smaller than the false positive error rate – when the true tree is not binary. Also note that because Robinson-Foulds distances are normalized by dividing by $2n - 6$, they are not equal to the average of the false negative and false positive error rates. Also, the RF rate of a star tree (one with no internal nodes) is 50%, which is the same as the RF rate for a completely resolved tree that has half of its edges correct. Using the RF rate has been criticized because of this phenomenon, since it tends to favor unresolved trees.

**Observation 2.21** Let $T$ be the true tree, and $T_1$ and $T_2$ be two estimated trees for the same leaf set. If $T_1$ is a refinement of $T_2$, then the number of false negatives of $T_1$ will be less than or equal to that of $T_2$, and the number of false positives of $T_1$ will be at least that of $T_2$.

This observation will turn out to be important in understanding the relationship between the error rates of consensus trees (described in Chapter 6 and how they compare to the error rates of the trees on which they are based.

### 2.6 The number of binary trees on $n$ leaves

Since we are interested in estimating phylogenetic trees, knowing the number of possible binary trees on $n$ distinctly labelled leaves is of interest. Here we consider two different cases: rooted binary trees on $n$ leaves, and unrooted binary trees on $n$ leaves.

We first consider the unrooted case. For $n = 1, 2,$ or $3$, the answer is 1 – there is only one unrooted binary tree when $n \leq 3$. However, when $n = 4$, there are three possible trees. Furthermore, it is easy to see this algorithmically: to construct a tree on $n = 4$ leaves, $s_1, s_2, s_3,$ and $s_4$, take a tree $T$ on $n = 3$ leaves, and then add the remaining leaf by subdividing an edge in the tree $T$, and making $s_4$ adjacent to this newly introduced node. Thus, the number of possible trees on $n = 4$ leaves is equal to the number of edges in $T$. Since $T$ has three leaves, it has exactly three edges (you can see this by drawing it). Hence, there are three unrooted binary trees on four leaves.
Things become a bit more difficult for larger values of \( n \), but the same algorithmic analysis applies. Take a tree \( T \) on \( n - 1 \) leaves, pick an edge in \( T \) and subdivide it, and make \( s_n \) adjacent to the newly created node. The number of unrooted binary trees on \( n \) leaves is therefore equal to the product of the number \( t_{n-1} \) of unrooted binary trees on \( n - 1 \) leaves and the number \( e_{n-1} \) of edges in an unrooted binary tree on \( n - 1 \) leaves. It is not hard to see that \( e_{n-1} = 2(n - 1) - 3 = 2n - 5 \). Hence, \( t_n \) satisfies \( t_n = t_{n-1}(2n - 5) \). Thus, \( t_n = (2n - 5)!! = (2n - 5)(2n - 7) \ldots 3 \).

Now, we examine the number of different rooted binary trees on \( n \) leaves. To compute this, note that every rooted binary tree \( T \) on \( n \) leaves defines an unrooted binary tree \( T' \) (obtained by ignoring the root of \( T \)), and that every unrooted binary tree \( T' \) corresponds to \( 2n - 3 \) rooted binary trees formed by rooting the tree \( T' \) on one of its edges. Hence, the number of rooted binary trees on \( n \) leaves is \( (2n - 3)!! = (2n - 3)(2n - 5) \ldots 3 \).

### 2.7 Rogue taxa

Sometimes two trees are very different primarily (or even exclusively) in terms of where one leaf is placed. Such a taxon is called a “rogue taxon” in the biological literature. Because the inclusion of rogue taxa in a phylogenetic analysis can increase the error of the phylogenetic analysis, they are often removed from the dataset before the final tree is reported.

Causes for rogue taxa vary, but a common cause is having a distantly related outgroup taxon in the dataset. The sequences for such taxa can be extremely different from all other sequences in the dataset, so that there is close to no similarity beyond what two random sequences would have to each other. In the extreme case of using a random sequence, the taxon with the random sequence could fit equally well into any location of the tree, and hence its location cannot be inferred with any reliability. When a phylogenetic analysis explores multiple optimal or near-optimal trees for the dataset, this will mean that the profile (set of trees) computed for the dataset will include trees that differ substantially in the placement of the rogue taxon. A consensus tree of such a collection of trees, such as the strict consensus tree - which is the most refined common contraction of the set of trees in the profile - will be largely unresolved. We will return to the problem of detecting rogue taxa later in the textbook, after we discuss phylogeny estimation methods.

### 2.8 Induced subtrees

A comparison of two trees that differ only in terms of the placement of a rogue taxon (e.g., a plant that shows up within hominids) would best be done not through the use of FN and FP rates, but through other measures. To enable these more fine-tuned comparisons, we define the notion of “induced subtrees”. Later on we will talk about phylogeny reconstruction methods that operate by combining subtrees together, and there the concept of induced subtrees will also be helpful.

Suppose we are given a set \( X \) of taxa and an unrooted tree \( T \) with leaf set \( S \), and suppose
2.9 Some special trees

Some types of trees are used frequently as examples to illustrate different properties of algorithms. Here we describe two examples:

**Definition 2.22** The **caterpillar** tree on $n$ leaves $s_1, s_2, \ldots, s_n$ is $(s_1, (s_2, (s_3, \ldots)))$. This is also referred to as the **comb**. Note that the caterpillar tree has the maximum pairwise distance between any two leaves in the tree.

**Definition 2.23** The **completely balanced** tree on $2^n$ leaves is a rooted tree, where the root is the parent of the roots of two completely balanced trees on $2^{n-1}$ leaves. Note that in the completely balanced tree on $2^n$ leaves, the distance between any two leaves is minimized.

---

Figure 2.10 Tree $T$ and the subtree it induces on $a, b, c, d$

$X \subseteq S$. Now, consider the tree that is obtained by removing from $T$ all leaves that are not in $X$; this produces a tree that will have at least one node (and possibly many nodes) of degree two. Now, we modify $T$ further so that it has no nodes of degree two, as follows. If $T$ has any nodes of degree two, then it has at least one path $P = v_1, v_2, \ldots, v_t$ such that all of its internal nodes (i.e., $v_i$ for $i = 2, 3, \ldots, t-1$) have degree two, and its endpoints have degree greater than two. Find any such path $P$, and replace $P$ by $(v_1, v_t)$. Repeat this process until there are no degree two nodes. We refer to the final tree produced by this process as the **subtree of $T$ induced by $X$**, and more generally as an **induced subtree**.

When the tree $T$ is rooted, the process for computing the induced subtree is nearly the same, except that the path $P$ can never have the root as an internal node, and the root (which may well have degree two) is allowed to be an endpoint of $P$. Hence, the result of suppressing nodes of degree two maintains the location of the root.

Induced subtrees are often used when you have a tree $T$ (rooted or unrooted) but you are only interested in what the tree says about a particular subset of its leaf set. For example, $T$ could be on $a, b, c, d, e, f$, but you are only interested in the relationship between the taxa $a, b, c, d$. To understand what $T$ tells you about $a, b, c, d$, you construct the subtree induced by $T$ on $a, b, c, d$. See Figure 2.10 for an example of this process.
2.10 Further reading

2.11 Review questions

1. Are Newick representations of trees unique, or are there many Newick representations of any given tree?
2. How many clades are there (including the singleton clades and the full set of taxa) for a rooted binary tree on 10 leaves?
3. How many bipartitions are there in an unrooted binary tree on 10 leaves?
4. What is the running time to compute a rooted binary tree from its set of clades?
5. What is the largest possible Robinson-Foulds distance between two unrooted binary trees on the same set of \( n \) leaves?
6. What is the number of rooted binary trees on 10 leaves?
7. What is the number of unrooted binary trees on 10 leaves?
8. What is the strict consensus tree of a set of trees?

2.12 Homework problems

1. Draw the rooted tree that is given by \((f, ((a, b), (c, (d, e))))\).
2. Draw a rooted tree and give its Newick format representation.
3. Draw the rooted tree given by \((1, (2, (3, (4, (5, 6))))))\), and write down the set of clades of that tree.
4. Draw the same rooted tree using the different styles as described in the text.
5. For the rooted tree \( T \) given by \((a, ((b, c), (d, (e, f))))\),
   - write down at least three other Newick representations.
   - write down the set of clades, and indicate which of the clades is non-trivial.
6. Compute the Hasse Diagram on the partially ordered sets given by the following sets of clades and related by containment. Then draw the rooted tree for each set.
   - \( \{\{a, b\}, \{a, b, c\}, \{a, b, c, d\}, \{e, f\}, \{e, f, g\}\} \)
   - \( \{\{a, b, c\}, \{a, b, c, d\}, \{e, f\}, \{e, f, g\}\} \)
   - Which one of these trees is not binary?
7. Draw all rooted binary trees on leaf set \( \{a, b, c, d\}\). (Note that trees that can be obtained by swapping siblings are the same.)
8. Draw all rooted trees (not necessarily binary) on leaf set \( \{a, b, c, d\}\).
9. Give a polynomial time problem to determine if two Newick strings represent the same rooted tree. For example, your algorithm should return “YES” on the following pair of strings:
   - \( (a, (b, c)) \) and \( ((c, b), a) \)
   - and should return “NO” on
   - \( (a, (b, c)) \) and \( (b, (a, c)) \)
10. Draw the rooted and unrooted versions of the unrooted tree given by the following Newick string: \((a, (b, ((c, (d, e))))\).
11. Draw all the rooted versions of the unrooted tree \((x, (y, (z, w))))\), and give their Newick formats.
12. Draw the unrooted version of the trees given below, and write down the set \(C(T)\) of each tree \(T\) below. Are the two trees the same as unrooted trees?
   1. \((a, (b, (c, ((d, e), (f, g))))))\)
   2. \(((a, (b, (c, (d, e), (f, g))))))\)
13. Consider the two unrooted trees given below by their bipartition encodings. Draw them. Do you see how one tree can be derived from the other by contracting a single edge? Which one refines the other?
   \(T_1\) is given by \(C(T_1) = \{(ab|cdef), (abcd|ef)\}\)
   \(T_2\) is given by \(C(T_2) = \{(ab|cdef)\}\).
14. Draw two unrooted trees, so that neither can be derived from the other by contracting a set of edges.
15. Draw three different unrooted trees, \(T_1, T_2,\) and \(T_3\), on no more than 8 leaves, so that \(T_1\) is a contraction of \(T_2\), and \(T_2\) is a contraction of \(T_3\) (identically, \(T_3\) is a refinement of \(T_2\), and \(T_2\) is a refinement of \(T_1\)). Write down the bipartition encodings of each tree.
16. Apply the technique for computing unrooted trees from compatible bipartitions to the input given below, using leaf 3 as the root. After you are done, do it again but use a different leaf as the root. Compare the rooted trees you obtained using the different leaves as roots: are they different? Unroot the trees, and compare the two unrooted trees. Are they the same?
   Input: \{123|456789\}, \{12345|6789\}, \{12|3456789\}, \{89|1234567\}\}.
17. Compute the unrooted trees compatible with the following sets of bipartitions (use the algorithm that operates on clades, using the specified roots):
   \{\{ab|cdef\}, \{abc|def\}, \{abcd|ef\}\}.
   With root “b”. Then do this again using root c.
   Are the unrooted trees you get different or the same?
   \{\{ab|cdef\}, \{abc|def\}, \{abcd|ef\}\}, with root “d”.
   \{\{abc|def|ghi\}, \{abcd|e|f|ghi\}, \{abcdef|ghi\}\}, using any root you wish.
18. Give a polynomial time algorithm to determine if the unrooted trees defined by two Newick strings are the same. Your algorithm should return “YES” for the following pairs of strings:
   \{\{(a, b, (c, d))\} and \{(a, b), (c, d)\}\}
   \{\{(a, b, (c, d))\} and \{(a, (b, (c, d)))\}\}
   Your algorithm should return “NO” for
   \{\{(a, b, (c, d))\} and \{((b, d), (a, c))\}\}
19. Consider the unrooted tree given by \(\{1, ((2, 3), (4, (8, 9)), (5, (6, 7)))\}\). Root this tree at leaf 5, draw this rooted tree, and write the Newick string for the rooted tree you obtain.
20. Draw two binary unrooted trees on leafset \{a, b, c, d, e, f\} that induce the same tree on \{a, b, c, d, e\} but have no non-trivial bipartitions in common.

21. Suppose \(T_0\) is the true tree and \(T\) is the estimated tree. Which of the following statements are not possible, under the assumption that both \(T_0\) and \(T\) are unrooted trees on ten leaves, and that \(T_0\) is a binary tree on ten leaves (but \(T\) may not be a binary tree). If you think the statement is impossible, explain why. Else, give an example where it is true.

- There are 5 false negatives and 3 false positives.
- There are 3 false negatives and 5 false positives.
- There are 3 false negatives and 3 false positives.
- There are 8 false negatives and 2 false positives.
- There are 8 false negatives and 8 false positives.
- There are 7 false negatives and 1 false positive.
- There are 1 false negative and 7 false positives.

22. Answer the same questions as for the previous problem, but do not assume now that the true tree \(T_0\) is binary, but do require that \(T\) is binary.

23. Suppose the true tree \(T_0\) is an unrooted binary tree and you have estimated trees \(T_1\) and \(T_2\) on the same leafset, both unrooted. Suppose that \(T_1\) is a star tree (i.e., it has a single node in the center with all leaves adjacent to this node), and that \(T_2\) is fully resolved (binary).

1. What is the Robinson-Foulds (RF) rate of \(T_1\) with respect to the true tree?
2. For what trees \(T_2\) will \(T_1\) have a better RF rate than \(T_2\)?
3. What do you think of using the RF rate as a way of comparing trees? What alternatives would you give?

24. Let \(T_0\) be the unrooted tree given by splits \{123|456, 12|3456, 1234|56\}, and let \(T_1\) be an estimated tree. Suppose \(T_1\) is missing split 123|456, but has a single false positive 124|356. Draw \(T_1\).

25. Give an algorithm for the following problem:

- Input: unrooted tree \(T_0\) and two sets of bipartitions, \(C_1\) and \(C_2\), where \(C_1 \subseteq C(T_0)\) and \(C_2 \cap C(T_0) = \emptyset\).
- Output: tree \(T_1\) (if it exists) such that \(T_1\) has false negative set \(C_1\) and false positive set \(C_2\), when \(T_0\) is treated as the true tree. (Equivalently, \(C(T_1) = [C(T_0) - C_1] \cup C_2\).)

26. Let \(T\) be a caterpillar tree on \(n\) leaves (i.e., \(T = (s_1, (s_2, (s_3, \ldots, (s_{n-1}, s_n)) \ldots))\)) (see Definition 2.22). Now let \(\mathcal{T}\) be the set of trees on \(n + 1\) leaves formed by adding a new taxon, \(s_{n+1}\), into \(T\) in all the possible ways. What is the expected RF distance between two trees picked at random from \(\mathcal{T}\)?

27. Prove using induction that the number of edges in an unrooted binary tree on \(n\) distinctly labelled leaves is \(2n - 3\).

28. Consider the set \(\mathcal{T}_n\) of unrooted binary trees on leafset \(S = \{s_1, s_2, \ldots, s_n\}\). If you pick a tree uniformly at random from \(\mathcal{T}_n\), what is the probability that \(s_1\) and \(s_2\) are siblings in \(\mathcal{T}\)?
29. Consider a caterpillar tree $T$ on a set $S$ of $n$ taxa. Suppose there is a very “rogue” taxon, $x$. Consider the set $\mathcal{T}$ that contains all the trees formed by adding $x$ into $T$?

1. What is $|\mathcal{T}|$?
2. What is the strict consensus of all the trees in $\mathcal{T}$? (Give its bipartition set.)

30. Imagine the case where the CFN model tree $T$ has a very long branch to a leaf $x$, and otherwise all the branches are relatively short – short enough that the remainder of the tree (i.e., $T - \{x\}$) can be estimated with high accuracy from reasonable length sequences (e.g., length at most 1000 nucleotides). However, because the branch to $x$ is long, very long sequences are needed to estimate the tree that contains $x$. What do you think might happen during a phylogenetic estimation of $T$ from nucleotide sequences of length at most 1000 nucleotides? Suppose you were to explore the set of optimal and also near-optimal CFN maximum likelihood trees; what would you observe? Suppose you were to perform bootstrapping; what would you observe?

31. For each of the given unrooted trees, draw the subtree induced on $\{a, b, c, d\}$.

- $T$ has Newick format $(b, (a, (f, (c, (g, (d, e))))))$ (i.e., it is the caterpillar $b, a, f, c, g, d, e$).
- $T$ has the Newick format $(f, (a, (c, (g, (d, (b, e))))))$ (i.e., it is the caterpillar $f, a, c, g, d, b, e$).

32. Give two unrooted trees on $\{a, b, c, d, e, f, g\}$ that induce the same subtree on $\{a, b, c, d\}$ but which are different trees.

- Give two unrooted trees on $\{a, b, c, d, e, f, g\}$ that are identical on $\{a, b, c, d\}$ and different on $\{d, e, f, g\}$.
- Give two rooted trees on $\{a, b, c, d, e\}$ which are identical on $\{a, b, c\}$ but different on $\{d, e, f\}$. 
3

Constructing trees from true subtrees

3.1 Introduction

In many approaches to constructing phylogenetic trees, the input is a set $T$ of trees, each leaf-labelled by a subset of a set $S$, and the objective is to construct a tree $T$ on $S$ from $T$. In this chapter, we discuss methods for constructing trees when the set $T$ is compatible - which means that it is possible to construct a tree $T$ that agrees with every tree in $T$. In general, this will only happen when every tree in $T$ is the true species tree on its leaf set, and so we refer to this problem as “Constructing trees from true subtrees”. This is a simplification of the more general problem where the input trees may have some estimation error, which is the more realistic case (most estimated species trees have some error); this version of the problem is addressed in Chapter 7. Another variant of this problem is where the input trees are estimated (and perhaps correct) gene trees. Since gene trees can differ from the species tree due to biological processes such as incomplete lineage sorting (ILS) and gene duplication and loss (GDL), the estimation of a species tree from a set of gene trees presents its own challenges, which are discussed in Chapter 10.

3.2 Tree compatibility

One of the key concepts in this chapter is tree compatibility, both for rooted and unrooted trees. In Chapter 2, we defined the concept of compatibility in the context of clades and bipartitions; as we will see, these definitions extend naturally to rooted and unrooted trees, respectively.

3.2.1 Unrooted tree compatibility

We begin with unrooted trees. Recall that if we are given a tree $T$ on leafset $S$ and a proper subset $X$ of $S$, we can compute the tree $T|X$ induced by $T$ on $X$ (see Chapter 2.8). Now suppose we have two trees, $t$ and $T$, where $t$ has leafset $X$ and $T$ has leafset $S$, with $X \subseteq S$. Then we will say that $t$ is compatible with $T$ if $T|X$ is a refinement of $t$. Note that we do not require that $t = T|X$; what this means is that $t$ may be unresolved which can allow $C(T|X)$ to have additional bipartitions beyond those present in $C(t)$.
If we are given a set $\mathcal{T}$ of unrooted trees, each with leaves taken from a set $S$, we will say that the set is compatible if and only if there is a tree $T$ with leafset $S$ that is compatible with every tree in $\mathcal{T}$. Furthermore, when this tree exists, we will refer to it as the **compatibility supertree** for the set $\mathcal{T}$.

**Theorem 3.1**  
Let $\mathcal{T} = \{t_1, t_2, \ldots, t_k\}$ be a set of compatible fully resolved trees with $S_i$ the leafset of $t_i$. Then $\sum_{i=1}^{k} RF(T|S_i, t_i) = 0$, where $T$ is a compatibility supertree for $\mathcal{T}$.

**Proof**  
Note that two binary trees on the same leafset that are compatible must be identical. Hence, since every tree $t_i \in \mathcal{T}$ is fully resolved, $t_i = T|S_i$ for $i = 1, 2, \ldots, k$. The rest follows.

#### 3.2.2 Rooted tree compatibility

Just as we said that a set $\mathcal{T}$ of unrooted trees is compatible if there is a compatibility supertree (i.e., a tree on the full set of taxa that is compatible with every tree in $\mathcal{T}$), the same statement can be made for rooted trees. However, to make this precise we need to say what we mean for two rooted trees $t$ and $T$ to be compatible, when the leafset $X$ of $t$ is a subset of the leafset $S$ of $T$. The only difference between rooted and unrooted trees is that to determine if $t$ and $T$ are compatible, we examine the *rooted* subtree induced in $T$ by $X$.

### 3.3 The ASSU algorithm: constructing rooted trees from rooted triples

We now present an algorithm for constructing a rooted tree from its set of “rooted triples”, where by “rooted triple” we mean a rooted three-leaf tree. We indicate the rooted triple on $a, b, c$ in which $a$ and $b$ are more closely related by $(a, b)|c$, by $ab|c$, or by any of the equivalent representations. We will also discuss the closely related problem of determining whether a set of rooted triplets is compatible with some tree, and constructing it if so.

**Algorithm for determining if a set of rooted triples is compatible, and constructing the compatibility tree (if it exists).** Suppose we are given a set $X$ of rooted triples, and we wish to know if $X$ is compatible, which means that there is a tree $T$ on which all the rooted triples in $X$ agree. Furthermore, when the set $X$ is compatible, we wish to return some tree $T$ on which all the rooted triples agree. The first algorithm for this problem was developed by Aho et al. (1978), and is widely known in phylogenetics. For the sake of brevity, we will often refer to this algorithm as the **ASSU** algorithm, after the four authors Aho, Sagiv, Szymanski, and Ullman.

The input to the ASSU algorithm is a pair $(S, Trip)$, where $S$ is a set of taxa and $Trip$ is a set of rooted three-leaf trees on $S$, with at most one tree for any three leaves; furthermore, we assume that every tree in $Trip$ is fully resolved (i.e., of the form $((a, b), c)$).

- Group the set $S$ of taxa into disjoint sets, as follows. We make a graph where the vertices are the elements of the set $S$ and we add an edge $(a, b)$ for every $((a, b), c) \in Trip$. 

Constructing trees from true subtrees

- If the graph is connected, then return \textit{Failure}, and exit - no tree is possible. Otherwise, let $C_1, C_2, \ldots, C_k$ ($k \geq 2$) be the connected components of the graph. For each connected component $C_i$,
  - Let \textit{Trip}_i be the set of triplets in \textit{Trip} that have all their leaves in $C_i$.
  - Recurse on $(C_i, \text{Trip}_i)$, and let $t_i$ be the compatible rooted tree, if it exists. (If no tree exists, we will have rejected the input and returned \textit{Failure}.)

Make the roots of the $t_1, t_2, \ldots, t_k$ all children of a common root, and return the constructed rooted tree. Note that this tree has all the taxa in $S$ as leaves.

This surprisingly simple algorithm is provably correct, and runs in polynomial time. A simple proof by induction on the number of leaves establishes correctness, and is in Aho et al. (1978).

3.4 Constructing unrooted binary trees from quartet subtrees

3.4.1 Notation

\textbf{Definition 3.2} Given an unrooted tree $T$ on $n$ distinctly labelled leaves, we denote by $Q(T)$ the set of all quartet trees on leaves in $T$, and by $Q_r(T)$ the set of all binary quartet trees in $Q(T)$. Hence, if $T$ is binary, then $Q_r(T) = Q(T)$. There are several ways to represent a binary quartet tree, each of which has been used in the literature. For example, the binary quartet tree on $a, b, c, d$ that splits $a, b$ from $c, d$ can be represented by $ab|cd, (ab)(cd), (ab, cd), (a, b|c, d), \text{ or } ((a, b), (c, d))$. We represent the tree on $a, b, c, d$ that has no internal edges by $(a, b, c, d)$, and refer to it as the \textbf{star tree}.

It is easy to see that every unrooted binary tree $T$ is defined by $Q(T)$; as we will see, some proper subsets of $Q(T)$ can also uniquely define $T$.

3.4.2 The All Quartets Method

We begin with the Quartet Compatibility problem:

- Input: Set $Q$ of quartet trees
- Output: Tree $T$ such that $Q \subseteq Q(T)$ (if such a tree $T$ exists) or \textit{Fail}

The Quartet Compatibility problem is NP-complete, even when all the trees in $Q$ are binary (Steel, 1992b), but some special cases of the problem can be solved in polynomial time, as we will show. In particular, the case where all the trees in $Q$ are binary and $Q$ has at least one tree on every four leaves in a set $S$ can be solved in polynomial time, using the All Quartets Method, which we now describe.

The \textit{All Quartets Method} takes as input a set $Q$ of quartet trees (each of them fully resolved), with one tree for every four leaves. The output is either a tree $T$ such that $Q(T) = Q$ or else \textit{failure}. 

3.4 Constructing unrooted binary trees from quartet subtrees

We call this algorithm the All Quartets Method, because it assumes that the input is the set of all quartet trees for an unknown tree \( T \); indeed, while the method can be applied to proper subsets of \( Q(T) \), there are no guarantees that the method will correctly answer whether the input quartet trees are compatible under this more general condition. However, when the input contains a tree on every four leaves, then we can prove that the All Quartets Method is correct.

All Quartets Method:
- Step 1: If \(|S| = 4\), then return the tree in \( Q \). Else, find a pair of taxa \( s_i, s_j \) that are always grouped together in any quartet that includes both \( s_i \) and \( s_j \). If no such pair exists, return “No compatibility tree”, and exit. Otherwise, remove \( s_i \).
- Step 2: Recursively compute a tree \( T' \) on \( S - \{s_i\} \).
- Step 3: Return the tree created by inserting \( s_i \) next to \( s_j \) in \( T' \).

Example 3.3 Consider the unrooted tree \( T = (1, (2, (3, (4, 5)))) \) and its set of quartet trees \( Q(T) = \{12|34, 12|35, 12|45, 13|45, 23|45\} \). Note that taxa 1 and 2 are always grouped together in all the quartets that contain them both, but so also are 4 and 5. On the other hand, no other pair of taxa are always grouped together. If we remove taxon 1, we are left with the single quartet on 2, 3, 4, and 5. The unrooted quartet tree on that set is 23|45. We reintroduce the leaf for 1 as sibling to 2, and obtain the unrooted tree given by \( (1, (2, (3, (4, 5)))) \).

3.4.3 Inferring quartet trees from other quartet trees

Recall that the All Quartets Method will construct \( T \) given \( Q(T) \), the set of quartet trees induced by \( T \). This suggests a method for constructing a tree, in which unrooted trees are estimated on four leaves at a time, and then combined into a tree on the full dataset using the All Quartets Method. This approach will produce the true tree, but only if all of the estimated trees are correct – even one single error will make the entire approach fail.

Since some of the estimated quartet trees might be incorrect, we may wish to try to compute a tree from a proper subset of its quartet trees – ones that look like they may be correctly computed. Here we present one attempt at solving this problem, in which we use just a subset \( Q \) of the possible quartet trees and try to infer the remaining quartet trees. If we succeed, then we can apply the All Quartets Method to the final set of quartet trees, and construct the tree \( T \).

The basic idea is to repeatedly compare two quartet trees at a time from the input set \( Q \), to see if any additional quartet trees can be inferred from the quartet trees. These rules are designed so that whenever the trees in \( Q \) are true, then the added trees must also be true. In other words, we assume that the input set \( Q \) satisfies \( Q \subseteq Q(T) \) for some (unknown) tree \( T \), and we ensure that any added quartet trees are also in \( Q(T) \). In the rules below, we will consider \( ab|cd \) to be the same quartet tree as \( ba|cd \), \( ba|dc \) and \( ab|dc \).

- Rule 1: If \( ab|cd \) and \( ac|de \) are in \( Q \subseteq Q(T) \), then \( ab|ce, ab|de \) and \( bc|de \) are also in...
Constructing trees from true subtrees

\( Q(T) \). Hence, if any of these three quartet trees are missing from \( Q \), we can add them to \( Q \).

- Rule 2: If \( ab|cd \) and \( ab|ce \) are in \( Q \subseteq Q(T) \), then \( ab|de \) must be in \( Q(T) \). Hence, if \( ab|de \) is missing from \( Q \), we can add \( ab|de \) to \( Q \).

It is easy to see that these rules are valid, and so if the input set \( Q \) contains only correct quartet trees (meaning true quartet trees for some unknown tree \( T \)), then the quartet trees that are added are also correct quartet trees for that unknown tree \( T \). These two rules are dyadic (also called “second order”) rules, in that they are based on combining two quartet trees to infer additional quartet trees.

The dyadic closure of a set \( Q \) of quartet trees is computed by repeatedly applying Rules 1 and 2 to pairs of quartet trees until no additional quartet trees can be inferred. The final set of quartet trees is the dyadic closure of \( Q \), and is denoted by \( cl_2(Q) \). The discussion above clearly shows the following:

**Theorem 3.4** Suppose \( Q \subseteq Q(T) \) for some tree \( T \). Then \( cl_2(Q) \subseteq Q(T) \).

Now suppose we were lucky, and \( cl_2(Q) = Q(T) \); then we can construct \( T \) using the All Quartets Method. However, if \( Q \) is too small a set, then \( cl_2(Q) \) may not be equal to \( Q(T) \). In the next section, we investigate how big \( Q \) has to be, in order for \( cl_2(Q) = Q(T) \).

### 3.4.4 Constructing a tree from a subset of its quartet trees

We begin by defining “short quartet trees” of a tree \( T \). As we will see, if \( Q \) contains all the short quartet trees and no incorrect quartet trees, then \( cl_2(Q) = Q(T) \).

**Definition 3.5** Let \( T \) be a binary tree and \( w : E(T) \rightarrow \mathbb{R}^+ \) be the positive edge weighting of \( T \). The deletion of an internal edge \( e \in E(T) \) (and its endpoints) creates four subtrees, \( A, B, C, \) and \( D \). Let \( a, b, c, d \) be four leaves nearest to \( e \) from these four subtrees; hence, \( a \in A, b \in B, c \in C \) and \( d \in D \). The definition of “nearest” is based on the path length, and takes the edge weights into account. Then \( a, b, c, d \) is a short quartet around \( e \), and the quartet tree induced on \( a, b, c, d \) by \( T \) is called a short quartet tree. Since there can be more than one nearest leaf in a given subtree to the edge \( e \), there can be more than one short quartet around \( e \). The set of all short quartets over all internal edges of \( T \) is called the set of short quartets of \( T \) and is denoted \( Q_{\text{short}}(T) \), while the set of short quartet trees over all internal edges of \( T \) is called the set of short quartet trees of \( T \) and is denoted \( Q_{\text{short}}^*(T) \).

**Example 3.6** Consider the caterpillar tree \( \{1, \{2, (3, \ldots, (99, 100)\ldots\} \} \). There are 97 internal edges of the tree, each of which contributes at least one short quartet. A careful inspection of this tree shows that the set \( Q_{\text{short}}(T) \) has 99 quartets. For example, \( Q_{\text{short}}(T) \) includes \( \{1, 2, 3, 4\}, \{2, 3, 4, 5\}, \{1, 3, 4, 5\}, \) and \( \{3, 4, 5, 6\} \). Now consider the set \( Q_{\text{short}}^*(T) \) of trees on the short quartets of \( T \). A little examination will show that \( T \) is the only tree on the same leafset that can contain all the short quartet trees. For example, Rule 1, applied to
3.5 Testing compatibility of a set of trees

12|34 and 23|45, produces 13|45, 12|45, and 12|35. In other words, applying Rule 1 produced the five quartet trees induced by 1, 2, 3, 4, 5. Indeed, if we compute $c_2(Q_{\text{short}}(T))$, we will obtain $Q(T)$.

In other words, the following theorem can be proven:

**Theorem 3.7** (From Erdos et al. (1997)) If $Q_{\text{short}}(T) \subseteq Q \subseteq Q(T)$ for some binary unrooted tree $T$, then $c_2(Q) = Q(T)$. In other words, under the assumption that $T$ is the true tree, then if $Q$ has no incorrect quartet trees and also contains all the short quartet trees of $T$, then the dyadic closure of $Q$ will include the true tree on every four leaves in $T$, and nothing beyond that.

What this theorem shows is that if we were lucky enough to find such a set $Q$ of quartet trees (one that has no incorrect quartet trees, and yet contains all the short quartet trees), we could use the dyadic closure to infer $Q(T)$, and then construct the tree $T$ using the All Quartets Method. This is a nice theoretical result, but it is not yet clear how we could use this to infer a tree from a set of sequences. In Chapter 13.1.2, we present methods, such as the Dyadic Closure Method (Erdos et al., 1999b), that build on the theory we have outlined here.

We close with the following corollary, which highlights the utility of the short quartet trees.

**Corollary** (From Erdos et al. (1997)) Let $T$ be a binary unrooted tree and let $Q_{\text{short}}(T)$ be the set of short quartet trees for $T$ for some edge-weighting of $T$. If $T'$ is a tree on the same leafset as $T$ and $Q_{\text{short}}(T) \subseteq Q(T')$, then $T' = T$.

**Proof** Assume that $T$ and $T'$ are on the same leaf set, and that $Q_{\text{short}}(T) \subseteq Q(T')$. We begin by noting that Theorem 3.7 implies that $c_2(Q_{\text{short}}(T)) = Q(T)$. Now, since $Q_{\text{short}}(T) \subseteq Q(T')$, by Theorem 3.4, $c_2(Q_{\text{short}}(T)) \subseteq Q(T')$. Hence, $Q(T) \subseteq Q(T')$. Since $T$ and $T'$ are on the same leaf set, $Q(T)$ and $Q(T')$ have the same cardinality. Hence, it must follow that $Q(T) = Q(T')$, and so $T = T'$.

3.5 Testing compatibility of a set of trees

We now describe a different type of tree construction problem, where we wish to know if the input set $X$ of source trees is compatible, which means that there is a tree $T$ such that $T$ agrees with every source tree in $X$. This tree, when it exists, is called a compatibility supertree for $X$.

**Example 3.8** Consider the set $X$ consisting of unrooted source trees, $(ab|cde), (bc|def)$, and $(cd, eg)$. This set of source trees is compatible, since the caterpillar tree (see Definition 2.22) $(a, (b, (c, (d, (e, (f, g)))))))$ is a compatibility supertree.

Because quartet compatibility is NP-complete (Steel, 1992b), determining if a set of unrooted trees is compatible is NP-complete.
Constructing trees from true subtrees

Theorem 3.9  The Unrooted Tree Compatibility Problem – determining if a set $X$ of unrooted trees, each leaf-labelled by elements from $S$, is compatible – is NP-complete, even if all the trees are binary (fully resolved).

As noted before, some special cases of the Unrooted Tree Compatibility can be solved in polynomial time. For example, we already know that the All Quartets Method can construct a tree $T$ from its set $Q(T)$ of quartet trees, and furthermore that the All Quartets Method can be used to determine if a set $X$ of quartet trees is compatible when $X$ contains a tree on every four taxa. Hence, if $X$ is a set of unrooted trees and every four taxa are in at least one tree in $X$, then we can determine if $X$ is compatible in a straightforward, if brute-force, way: we replace every tree $t$ in $X$ by its set $Q(t)$, and thus make $X$ into a set of quartet trees that contains a tree on every four taxa. We can then apply the All Quartets Method to the set of quartet trees we have created to determine if the quartet trees are compatible.

However, suppose that we are given a set $X$ of rooted leaf-labelled trees and we want to know if there is a rooted tree $T$ that induces each of the trees in $X$. To answer this question, we can encode each of the rooted leaf-labelled trees in $X$ by its set of rooted triplet trees, and then run the ASSU algorithm (described earlier) on the resultant set of rooted triplet trees. If the output is a rooted tree that induces all the rooted triplet trees, then it follows that the set $X$ is compatible. The other possible outcome is that the algorithm fails to return a tree (because during at least one of the recursive calls, the graph has a single connected component); in that case, the rooted triplet trees are not compatible, and hence the set $X$ is incompatible. In other words, it is easy to see that testing a set of rooted binary trees for compatibility is a polynomial time problem. We summarize this as follows:

Theorem 3.10  The Rooted Binary Tree Compatibility Problem – determining if a set $X$ of rooted binary trees, each leaf-labelled by elements from $S$, is compatible – can be solved in polynomial time.

The extension of this problem to rooted trees that can have polytomies is also solvable in polynomial time, but the proof of this is left to the reader.

3.6 Further reading

In this chapter we described several quartet-based methods for tree estimation; in each of these cases, we assumed that the set of quartet trees is computed using some technique, and the objective is to construct a tree that is consistent with the quartet trees. Yet, since quartet trees are not always perfectly estimated, these tree estimation methods can fail to construct any tree. In later chapters, we will return to quartet-based methods for tree estimation, and address this inference problem when the input trees are presumed to have some error.

3.7 Review questions

1. What is a “rooted triple”?
3.8 Homework problems

2. For each problem below, state whether it is solvable in polynomial time, NP-hard, or of unknown computational complexity:
   - Determining if a set of rooted triples is compatible.
   - Determining if a set of unrooted quartet trees is compatible.
   - Determining if a set of rooted leaf-labelled trees is compatible.
   - Determining if a set of unrooted leaf-labelled trees is compatible.

3. If $T$ is an unrooted leaf-labelled tree, what does $Q(T)$ refer to?

4. What does $ab|cd$ refer to?

5. What is the All Quartets Method? Does it run in polynomial time?

6. Suppose you are given a set $Q$ of unrooted quartet trees that contains a tree for some but not all of the different sets of four species taken from a species set $S$. Can you use the All Quartets Method to test for compatibility of the set $Q$?

7. Suppose you are given a set $R$ of rooted triplet trees that contains a rooted tree for some but not all of the different sets of three species taken from a species set $S$. Can you use the ASSU method to test for compatibility of the set $R$?

3.8 Homework problems

1. Make up a rooted tree on six leaves, and write down all its rooted triples. Then make up another rooted tree on the same six leaves, and write down all its rooted triples. How many rooted triples do your trees disagree on?

2. Make up two rooted trees on at least five leaves that differ in exactly one rooted triple.

3. 1. Write down the set $X$ of rooted triples for the caterpillar tree given by $(1, (2, (3, (4, 5))))$.
   2. Apply the ASSU algorithm to this set of rooted triples. What do you find?

4. Is it possible to have a compatible set $X$ of rooted triplets for which some pair of leaves $i, j$ is not separated in any rooted triplet in which they both appear, but where $i$ and $j$ are not siblings in any tree that is compatible with the set of rooted triplets? If so, provide the example, and otherwise prove it is impossible.

5. Suppose we modify the ASSU algorithm as follows. We compute the equivalence relation, and if there is more than two equivalence classes, $C_1, C_2, \ldots, C_k$ (with $k > 2$) we make two subproblems, $C_1$ and $C_2 \cup C_3 \cup \ldots \cup C_k$. Otherwise, we don’t change the algorithm. Does this also solve rooted triplet compatibility? (Prove or disprove.)

6. Prove that the ASSU algorithm correctly solves the problem of determining if a set $Trip$ of rooted, fully resolved, three-leaf trees is compatible. (Hint: use induction.)

7. Consider input sets $T$ of rooted trees, each on a subset of taxon set $S$, and suppose some of them have polytomies. We will consider the polytomies to be soft, meaning that we do not consider them to imply any constraint on the tree on the full set $S$. We would like to find a tree $T$ on the full taxon set that is compatible with every tree in $T$, meaning that it will either agree with the trees in $T$ or refine the trees when restricted to the same leafset.

1. Show how to use the ASSU algorithm so that it solves this problem.
2. Prove your algorithm correct.

8. Suppose that we allow triplet trees to represent hard polytomies. For example, we would use \((a, b, c)\) to indicate that the compatibility tree (if it exists) induces the unresolved tree \((a, b, c)\). Suppose that ASSU ignores these triplet trees. Give an example of an input set \(\text{Trip}\) of triplet trees that is allowed to have these hard polytomies, and show that the ASSU algorithm will not correctly solve the compatibility problem on \(\text{Trip}\). Thus, either \(\text{Trip}\) should be compatible but the ASSU algorithm should say it is not, or vice versa.

9. Consider input sets \(\text{Trip}\) of rooted triplet trees, each on a subset of taxon set \(S\), and suppose some of them are polytomies (i.e., of the form \((a, b, c)\)). Suppose we consider the triplet tree \((a, b, c)\) to be a hard polytomy, meaning that it imposes a constraint on the tree \(T\) on \(S\) to induce the tree \((a, b, c)\) on \(\{a, b, c\}\), rather than just be compatible with it. In other words, we would like to find a tree on the full taxon set that induces each triplet tree in \(\text{Trip}\).

1. Modify the ASSU algorithm so that it solves this problem.
2. Prove your algorithm correct.

10. In the text, we stated that the ASSU algorithm is polynomial time. Provide a running time analysis, where the input is a set of \(k\) rooted triplet trees drawing their leaves from set \(S\) of \(n\) taxa.

11. Make up an unrooted tree on at least 5 leaves, and write down all its unrooted quartet trees.

12. Make up two different unrooted trees on the same leaf set, but try to make them disagree on as few unrooted quartet trees as possible. How many do they disagree on?

13. Construct a tree on leaf set \(\{a, b, c, d, e, f\}\) that induces each of the following quartet trees:
   - \((ab|cd)\),
   - \((ab|ce)\),
   - \((ac|de)\),
   - \((bc|de)\),
   - \((ab|de)\),
   - \((ab|ef)\),
   - \((ab|df)\),
   - \((ab|ef)\),
   - \((ac|df)\),
   - \((ac|ef)\),
   - \((ad|ef)\)

14. Recall the All Quartets Method, which constructs a tree on a leafset \(S\) given a set \(X\) of quartet trees that contains a quartet tree on every set of four taxa from \(S\), or else returns \(\text{Fail}\). Suppose we are given a set \(A\) of quartet trees that contains at most one quartet tree for every set of four taxa (i.e., it can fail to have a tree on one or more sets of four taxa).
3.8 Homework problems

Give an example of a set $A$ that is compatible, and so $A \subseteq Q(T)$ for some binary tree $T$, but where the All Quartets Method can fail to construct $T$ from $A$.

15. Recall that the All Quartets Method is designed to solve the Quartet Compatibility problem when the input is a set of fully resolved (i.e., binary) trees, with exactly one tree on every set of four taxa. Prove that the All Quartets Method is correct for such inputs. (Hint: use induction.)

16. Consider the case where the unrooted tree $T$ is not binary, and so can have a node of degree greater than three. Give an example of such a tree $T$, so that when the All Quartets Method is applied to $Q(T)$ it fails to recover the tree $T$.

17. Modify the All Quartets Method so that it will correctly handle inputs $Q$ that contain quartet trees with hard polytomies (i.e., of the form $(a, b, c, d)$).

18. Suppose we have a set $X$ of unrooted binary trees, and we encode each tree $T \in X$ by its set $Q(T)$ of quartet trees. Prove or disprove: The set $X$ is a compatible set of unrooted trees if and only if $\bigcup_{T \in X} Q(T)$ is a compatible set of quartet trees.

19. Suppose we have a set $X$ of rooted binary trees, and we encode each tree $T \in X$ by its set $R(T)$ of rooted triplet trees. Prove or disprove: The set $X$ is a compatible set of rooted trees if and only if $\bigcup_{T \in X} R(T)$ is a compatible set of rooted triplet trees.

20. Suppose we have a set $X$ of unrooted binary trees, and we want to test if the set is compatible. What can you say about using the All Quartets Method on the set $\bigcup_{T \in X} Q(T)$? Will it solve this problem?

21. Suppose we have a set $X$ of rooted binary trees, and we want to test if the set is compatible. What can you say about using the ASSU algorithm on the set $\bigcup_{T \in X} R(T)$? Will it solve this problem?

22. Consider the Split Constrained Quartet Support problem. How would you define the input set $X$ of allowed bipartitions so that the solution to the problem gave an optimal tree over all possible binary trees on the taxon set?

23. Suppose you have a collection $T$ of unrooted trees, not necessarily binary, all with exactly the same leafset $\{1, 2, 3, \ldots, n\}$. Suppose that the set $T$ is compatible. What is the maximum size of $T$ (expressed as a function of $n$).

24. Suppose you have a collection $T$ of unrooted binary trees, each of them different, all with exactly the same leafset $\{1, 2, 3, \ldots, n\}$. Suppose that the set $T$ is compatible. Express the maximum size of $T$ as a function of $n$.

25. Consider the following three unrooted trees:
   - $T_1 = (1, (3, (5, (6, 7))))$
   - $T_2 = (1, (2, ((4, 8), (3, 7))))$
   - $T_3 = (2, ((4, (3, 5)), 1))$

   Answer the following questions:
   1. Are these unrooted trees compatible? Justify your answer.
   2. Root all the three trees at leaf 1, and draw the rooted versions of these trees. Are these rooted trees compatible? Justify your answer.
26. Recall that the definition of short quartets depends on the edge-weighting in a tree. Suppose $T$ is a rooted binary tree and the edge weights $w(e)$ produce an ultrametric distance matrix. Let $r$ be the root of $T$ and let its children be $u$ and $v$. Now unroot the tree $T$ to obtain an unrooted tree $T_u$. The root of $T$ is suppressed, so that $u$ and $v$ are now adjacent, and the weight of the edge $(u, v)$ is $w(u, r) + w(r, v)$. Prove or disprove each of the following statements:

- For any such $T_u$ with edge-weighting $w$, every four leaves in the tree $T$ are a short quartet.
- For any such $T_u$ with edge-weighting $w$, every two leaves in the tree $T$ are in some short quartet.
4

Constructing trees from qualitative characters

4.1 Introduction

In essence, there is really one primary type of data used to construct trees – **characters**. An example of a character in biology might be the nucleotide (A, C, T, or G) that appears in a particular location within a gene, the number of legs (any positive integer), or whether the organism has hair (a Boolean variable). In linguistics we find similar variety in characters; for example, a character could be the cognate class for a basic meaning, it could be a binary character indicating whether or not a language has undergone a sound change, the particular way the language handles some aspect of its morphology, or the presence or absence of some typological feature. In each of these cases, a character divides the input set into disjoint subsets so that the leaves of the tree (which could be molecular sequences, biological species, or languages) within each subset are equivalent with respect to that character. We will refer to the objects placed at the leaves as “taxa” and the different sets defined by the character as the **character states**. In other words, a character is an equivalence relation on the set $S$ of taxa, and the different equivalence classes are the different character states. However, characters can also be described by functions from the taxon set $S$ to the set of character states.

The number of states the character can take is an important aspect for modelling purposes. In biological phylogenetics, many characters are based on sites within a molecular sequence alignment, and thus have a maximum number of possible states (four for DNA or RNA, and 20 for amino acids). Morphological features can be multi-state (and perhaps even have an unbounded number of possible states), but many morphological features are just based on the presence or absence of a given characteristic, and hence are explicitly binary.

In linguistic phylogenetics, characters similarly range in the number of state, but many are multi-state and may have an unbounded number of possible states. For example, if cognate classes for a given semantic slot are used, then the number of possible states is unbounded. On the other hand, syntactic characters and phonological characters are based on the presence or absence of a feature, and so are binary.

Mathematically, most models for the evolution of characters down trees assume that character state changes occur due to substitution. However, not all evolution is treelike,
so that some characters can evolve with reticulation. Horizontal gene transfer (HGT) is common in some organisms (e.g., bacteria), and hybridization (whereby two species come together to make a new species) is also frequent for some organisms (e.g., plants). There are similar challenges in linguistic phylogeny, where words can be borrowed (i.e., transmitted) between lineages. Borrowing is frequent for lexical characters under some circumstances, and can also occur for simple phonological characters, but is unlikely for morphological features.

Character-based methods are the basis of nearly all phylogenetic estimation methods, since characters form the way the input is nearly always described. If the evolutionary process operating on the characters can be modelled adequately, then phylogeny estimation - whether of a tree or of a phylogenetic network - can be performed using statistical methods, such as maximum likelihood or Bayesian MCMC. However, when no statistical model is available (or the statistical models are unreliable), then simple methods that are not based on explicit models of evolution can be used to estimate trees. In this chapter, we will discuss the estimation of trees from character data using two such simple methods – maximum parsimony and maximum compatibility. In later chapters, we will discuss models of character evolution, and statistical methods for estimating trees under these models.

### 4.2 Terminology

Suppose we have \( n \) taxa, \( s_1, s_2, \ldots, s_n \) described by \( k \) characters, \( c_1, c_2, \ldots, c_k \). This input is typically provided in an \( n \times k \) matrix \( M \), with the taxa occupying rows and different characters occupying the columns. In this case, the entry \( M_{ij} \) is the state of the taxon \( s_i \) for character \( c_j \). We can also represent this input by just giving the \( k \)-tuple representation for each taxon.

When the substitution process produces a state that already appears anywhere else in the tree, this is said to be homoplastic evolution (or, more simply, homoplasy). Back-mutation (reversal to a previous state) and parallel evolution (appearance of a state in two separate lineages) are the two types of homoplasy. When all substitutions create new states that do not appear anywhere in the tree, the evolution is said to be homoplasy-free. Furthermore, when the tree fits the character data so that no character evolves with any homoplasy, then the tree is called a perfect phylogeny.

While homoplasy-free evolution may not be the rule, individual characters can evolve without homoplasy. Therefore, we will say that a character is compatible on a tree if it evolves on the tree without homoplasy. We will also say that a set of characters is compatible if there is a tree on which all of the characters evolve without homoplasy, and so are compatible on the tree.

In some cases (e.g., some morphological features in biology, and some phonological or morphological characters in linguistics), the characters are given with a known tree structure relating the character states. For example, for some morphological features, represented by either presence (marked as 1) or absence (marked as 0) of the feature, it is known that the ancestral state is 0 and that the derived state is 1. Some multi-state characters may
also have a clear tree structure, where the ancestral state is known, and the progression
between the states is also known. We refer to these as directed characters, and directed
binary characters for those binary characters for which the ancestral state and derived
state are known. Homoplasy-free directed characters provide substantial information about
the phylogeny, provided that the assumptions (that the evolution is homoplasy-free and the
process between the states is known) are valid.

In this chapter we discuss two approaches that have been used to construct trees from
undirected characters: maximum parsimony and maximum compatibility. The parsimony
score (also called the length) of a character on a tree is the number of times the character
has to change state on the tree. Finding a tree \( T \) that has the smallest total parsimony score
for an input character matrix is called the maximum parsimony problem. The maximum
compatibility problem is similar, but seeks the tree on which the maximum number
of characters are compatible, which means they evolve without any homoplasy. The rela-
tionship between parsimony score and compatibility (homoplasy-free evolution) is that a
character is homoplasy-free on a tree \( T \) if and only if it changes \( r - 1 \) times, where \( r \) is the
number of states of the character at the leaves of the tree.

We begin with maximum parsimony, which is much more commonly used in biological
phylogenetics than maximum compatibility. However, maximum compatibility, and the
related problem of determining if a perfect phylogeny exists, and computing it when it
does, is relevant to the analysis of phylogenies within species, using SNP (single nucleotide
polymorphism) data, as well as in linguistic phylogenetics, and so is also of interest.

### 4.3 Tree construction based on Maximum Parsimony

Maximum parsimony (MP) is an optimization problem in which a tree is sought for an
input character matrix (the same type of input as is provided to maximum compatibility),
for which the total number of character state changes is minimized.

We begin this discussion by making a precise statement of what is meant by the number
of state changes of a character \( c \) on a tree \( T \). We are given a tree \( T \) and character state
assignments to the leaves of \( T \). Our objective is to find character state assignments to the
internal nodes of \( T \) so that the number of edges that have different states at the edge’s
endpoints is as small as possible. This is the “fixed tree parsimony problem”, which is also
called the “small parsimony problem”.

It should be clear that this definition has nothing to do with whether the tree \( T \) is rooted
or not, so we will simplify this discussion by assuming \( T \) is not rooted. Hence, the fixed
tree parsimony problem can be formally stated as follows:

#### Parsimony score of a fixed tree

- **Input:** unrooted tree \( T \) with leafset \( S \), and character \( c : S \rightarrow \{1, 2, \ldots, r\} \).
- **Output:** assignment of character states to the internal nodes of \( T \) so as to minimize the
  number of edges \( e = (u, v) \) where \( c(u) \neq c(v) \).
As we will see, a simple dynamic programming due to Fitch (1971) and Hartigan (1973), and hence referred to as the “Fitch-Hartigan” algorithm, will find the parsimony score and an optimal labelling of the internal nodes.

### 4.3.1 The Fitch-Hartigan algorithm for fixed-tree maximum parsimony

The input to the small parsimony problem is a tree $T$ whose leaves have assigned states for a character $c$.

The algorithm operates by rooting the tree (arbitrarily) on an edge, and then has two stages. In the first stage it computes the parsimony score for the character on the tree, and in the second stage it computes an assignment of character states to the internal nodes to achieve that score. Thus, the dynamic programming algorithm gives the optimal labelling as well as the maximum parsimony score.

- Root the tree $T$ on an edge, thus producing a rooted binary tree $T'$ with root $r$. Note that $r$ is not an internal node in $T$, but that otherwise all the nodes of $T'$ are nodes in $T$.
- For all the leaves $x$, let $A(x)$ denote the state at the leaf $x$ for character $c$.
- Initialize score to be 0.
- Starting at the nodes $v$ which have only leaves as children, and moving up the tree (towards the root), do the following:
  - If $v$ has children $w$ and $x$, and if $A(w) \cap A(x) \neq \emptyset$, then set $A(v) = A(w) \cap A(x)$. Else, set $A(v) = A(w) \cup A(x)$, and increment score by 1.
- When you reach the root, $r$, pick an arbitrary state in $A(r)$ to be its state; this completes Stage 1 of the algorithm.
- Stage 2 is the downward traversal, beginning at the root. For every node $y$ that is visited, starting at the root and moving towards the leaves, pick a state for $y$, as follows:
  - If the parent of $y$ has been assigned a state that is in $A(y)$, then set the state for $y$ to the same state as was assigned to its parent. Otherwise, pick an arbitrary element in $A(y)$ to be its state.

At the end of this two-stage process (one up the tree, and one down), you will have assigned states to each node in the tree. Note that during the upwards phase, the final state for $v$ is selected from $A(v)$. Thus, any node $v$ for which $|A(v)| = 1$ has only one possible assignment in an optimal labelling of the internal nodes. The converse, however, is not necessarily true. As we will see, there are cases where $|A(v)| > 1$ and yet every optimal labelling of $T$ has $v$ assigned to the same specific state.

**Example 4.1** Suppose that $T$ is the binary tree $(a, (b, (c, d)))$, and assume that character $\alpha$ is defined by $\alpha(d) = 0$ and $\alpha(s) = 1$ for all $s \neq d$. Root the tree on the edge leading to $a$ (so that you get the rooted tree corresponding to the Newick string given above). We will name the internal nodes of this rooted tree with $r$ (for the introduced root), $p$ for the parent of $c$ and $d$, and $q$ for the MRCA of $b, c, d$. 
Stage 1: The variable \( \text{score} \) is initialized to 0. When you run the dynamic programming algorithm, the leaves are processed first, and we obtain \( A(a) = A(b) = A(c) = \{0\} \) and \( A(d) = \{1\} \). The first internal node that is visited is \( p \), the parent of leaves \( c \) and \( d \). Since \( A(c) \cap A(d) = \emptyset \), we set \( A(p) = A(c) \cup A(d) \), and obtain \( A(p) = \{0, 1\} \); we also increment \( \text{score} \) so that it is now 1. We then compute \( \text{score} \) so that it is now 1. We then compute \( \text{parsimony score} \). Note that there is only one edge on which there is a change, and it presents some properties of the algorithm that are applicable in general. For example, the parsimony score of the character on the tree is 1.

Stage 2: We now assign states to the nodes of the tree, beginning at the root. Since \( |A(r)| = 1 \), we set \( c(r) = 0 \), the unique element in \( A(r) \). We then visit the children of \( r \), which are \( a \) and \( q \). Since \( a \) is a leaf, its character state is already assigned. To assign a state to node \( q \), we check to see if \( c(r) = 0 \in A(q) \). Since \( A(q) = \{0\} \), we set \( c(q) = 0 \). We then visit the children of \( q \), which are \( b \) and \( p \). Since \( b \) is a leaf, we do not need to process it. To assign a state to node \( p \), we check to see if \( c(q) = 0 \in A(p) \). Since \( A(p) = \{0, 1\} \), we set \( c(p) = 0 \). We then visit the children of \( p \), but they are both leaves, and so we are done with Stage 2.

At this point all the nodes of the tree have been assigned states, and we can calculate the parsimony score. Note that there is only one edge on which there is a change, and it \((p, d)\). Hence, the parsimony score of the character on the tree is 1.

This example is a very simple illustration of the Fitch-Hartigan algorithm, but also presents some properties of the algorithm that are applicable in general. For example, the parsimony score (indicated by the variable \( \text{score} \)) of any character on any tree is the number of nodes \( v \) for which \( A(v) \) is computed by taking the union of \( A(y) \) and \( A(z) \), where \( y \) and \( z \) are the children of \( v \). That is, otherwise we would have \( A(v) = A(y) \cap A(z) \), and the character state assignment obtained during the downwards pass (Stage 2) would pick the same state for all three nodes, \( v, y \) and \( z \). Thus, the parsimony score can be computed at the end of Stage 1, without needing to do Stage 2.

Note also that Stage 2 is necessary if the actual states of the internal nodes are desired. For example, although \( |A(p)| = 2 \), in the downward pass there was only one option for how to set \( c(p) \). Hence, the optimal character states at internal nodes can only be determined after both stages are complete – they are not determined during the first pass.

**Theorem 4.2**  For all trees \( T \) and characters \( c \) defined at the leaves of \( T \), the parsimony score computed by the Fitch-Hartigan dynamic programming is correct, and the assignment of states to the internal nodes of \( T \) achieves the reported parsimony score. Furthermore, the algorithm runs in \( O(nr) \) time, where there are \( n \) leaves in \( T \), and \( c \) is an \( r \)-state character.

The proof of the correctness of the algorithm is omitted, but the running time analysis of this algorithm is straightforward. Computing \( A(v) \) for a node \( v \) only depends on being able to compute the intersection and union of sets, each of which has cardinality at most \( r \).
Constructing trees from qualitative characters

(where \( r \) is the maximum number of states of any character on \( T \)). Since the characters can be processed independently, the following theorem is easy to prove:

**Theorem 4.3** The parsimony score of a character on a tree \( T \) can be computed in \( O(nkr) \) time, where there are \( n \) leaves and \( k \) characters, with each character having at most \( r \) states.

### 4.3.2 The Sankoff algorithm for fixed tree maximum parsimony

The parsimony problem is somewhat more complicated if the substitution costs vary, depending on the particular substitution. For example, there are two different types of nucleotides – purines (which are A and G) and pyrimidines (which are C and T). Substitutions that change a purine into a purine, or a pyrimidine into a pyrimidine (which are called **transitions**) are considered more likely than substitutions that change a purine into a pyrimidine or vice-versa (which are called **transversions**). Therefore, one variant of maximum parsimony would treat these two types of substitutions differently, so that transitions would have lower cost than transversions, but any two transitions or any two transversions would have the same cost.

More generally, suppose you have an alphabet \( \Sigma \) with \( r \) letters, and so you represent the substitution cost as a symmetric \( r \times r \) matrix \( M \), where \( M[x, y] \) is the cost of substituting \( x \) by \( y \). Clearly \( M[x, x] \) should be 0. If all entries off the diagonal are the same, then this is identical to unweighted maximum parsimony, and the previous algorithm works. But what if the entries off the diagonal are different?

As we will see, this problem can be solved using a straightforward dynamic programming algorithm, due to Sankoff (1975a). We present it here, along with a proof of its correctness, because it illustrates the power of dynamic programming algorithms when working with trees.

Let \( t \) be an unrooted binary tree with leaves labelled by sequences of length \( k \), all drawn from \( \Sigma^k \). We root \( t \) on an edge, thus producing a rooted binary tree \( T \), in which only the leaves are labelled by sequences. We consider a single character (site) at a time.

We define the following variables:

- For every vertex \( v \) in \( T \), we let \( A(v) \) denote the state at \( v \). Thus, \( A(v) \) is defined by the input for each leaf \( v \), but will be set during the algorithm for the remaining nodes.
- For every vertex \( v \) in the rooted tree \( T \), and for every letter \( x \) in \( \Sigma \), we define \( \text{Cost}(v, x) \) to be the minimum parsimony cost of the subtree \( T_v \) over all possible labels at the internal nodes of \( T_v \), given that we label \( v \) by \( x \).

How do we set \( \text{Cost}(v, x) \)? If \( v \) is a leaf, then we set \( \text{Cost}(v, x) = 0 \) if \( A(v) = x \), and otherwise we set \( \text{Cost}(v, x) = \infty \). Then, if \( v \) is a node that is not a leaf, and if we have already computed \( \text{Cost}(w, x) \) for all nodes \( w \) in the subtree below \( v \), and for all letters \( x \) in \( \Sigma \), we can compute \( \text{Cost}(v, x) \) as follows. Let \( w \) and \( w' \) be the two children of \( v \). Then:

\[
\text{Cost}(v, x) = \min \{ \text{Cost}(w, y) + M[x, y] : y \in \Sigma \} + \min \{ \text{Cost}(w', y) + M[x, y] : y \in \Sigma \}
\]
4.3 Tree construction based on Maximum Parsimony

To see how this works, suppose \( v \) has two children \( w \) and \( w' \) and they are both leaves. In this case, \( \text{Cost}(w, x) = \infty \) if \( A(w) \neq x \), and otherwise \( \text{Cost}(w, x) = 0 \). Hence, \( \text{Cost}(v, x) = M[x, A(w)] + M[x, A(w')] \), which is what we want.

Now consider the case where one child \( w \) of \( v \) is a leaf and the other \( w' \) is not. Then
\[
\text{Cost}(v, x) = M[x, A(w)] + \min\{\text{Cost}[w', y] + M[x, y] : y \in \Sigma\}.
\]

What this means is that the smallest cost you can get for the tree \( T_v \) given that you label \( v \) by \( x \) is obtained for some way of labelling the child \( w' \) with a letter in \( \Sigma \). Suppose \( y \) is the best way of labelling \( w' \), given that we’ve constrained the label at \( v \) to be \( x \). Now consider the total cost of the entire subtree \( T_v \); this is computed by summing the costs on the edges. The cost of the edge \((v, w)\) is simply \( M[x, A(w)] \), and the cost of the edge \((v, w')\) is simply \( M[x, A(w')] = M[x, y] \). The sum of the costs of the edges in the subtree \( T_{w'} \) is then \( \text{Cost}(w', y) \), since we’ve said that the label at \( w' \) is \( y \).

The case where both children of \( v \) are not leaves can be analyzed similarly, showing that the formula is correct. Therefore, the algorithm would compute \( \text{Cost}(v, x) \) for all nodes \( v \) and all letters \( x \) as you go from the bottom of the tree up to the root. Therefore, you should not calculate \( \text{Cost}(v, x) \) until you have calculated \( \text{Cost}(w, y) \) for all nodes \( w \) below \( x \) and for all letters \( y \in \Sigma \). To determine the parsimony score of the tree, you calculate \( \min\{\text{Cost}(r, x) : x \in \Sigma\} \), where \( r \) is the root of the tree.

An optimal label at the root \( r \) will be \( x_0 \) such that \( \text{Cost}(r, x_0) \) is the parsimony score of the tree. However, to label the remaining nodes, you will need some additional calculations. Suppose that as you go up the tree, calculating \( \text{Cost}(v, x) \) for each node \( v \) and letter \( x \), you record at least one pair of values for \( y \) and \( y' \) such that \( \text{Cost}(v, x) = \text{Cost}(w, y) + \text{Cost}(w', y') + M[v, y] + M[v, y'] \). Then, to set optimal labels for the internal nodes of the rooted tree \( T \), you first set the label \( x_0 \) for the root \( r \). Then, you visit the two children \( w, w' \) of \( r \). Since you have recorded the pair of values \( y \) and \( y' \) associated to \( \text{Cost}(r, x_0) \), you set \( A[w] = y \) and \( A[w'] = y' \). Having set these labels, you can then continue down the tree and set the labels for every internal node.

Thus, the second phase of the algorithm in which you set the labels at the internal nodes can be performed in \( O(n) \) time, where \( n \) is the number of leaves in the tree, provided that during the first phase you have recorded the additional information.

**Theorem 4.4** The Sankoff algorithm correctly computes the maximum parsimony score of a binary tree \( T \), and does so in \( O(nkr^2) \) time, where \( T \) has \( n \) leaves, and the input matrix has \( k \) characters, each with character drawing its states from \( \Sigma = \{1, 2, \ldots, r\} \).

**Proof** We prove this for a rooted tree \( T \), which defines the order in which we will compute all the \( \text{Cost}(v, x) \) variables. Let \( \text{Cost}^*(v, x) \) denote the best achievable score for the tree \( T_v \) given that \( v \) is labelled by \( x \); we will prove that \( \text{Cost}(v, x) = \text{Cost}^*(v, x) \) by induction on the length \( L(v) \) of the longest path from \( v \) to any leaf in the tree below it. Note that in the dynamic programming algorithm, we compute \( \text{Cost}(v, x) \) first for the leaves \( v \), then for vertices \( v \) whose children are leaves, etc., so that the vertices are processed according to \( L(v) \) values.
The base case is $L(v) = 0$; i.e., for the leaves of $T$. Note that $\text{Cost}(v,x) = \text{Cost}^*(v,x)$ for all leaves $x$ and states $x$. Now consider a vertex $v$ with children $w$ and $w'$. Therefore, $\max\{L(w), L(w')\} < L(v)$, and so by induction we know that $\text{Cost}(w,x) = \text{Cost}^*(w,x)$ and $\text{Cost}(w',x) = \text{Cost}^*(w',x)$ for all $x \in \Sigma$. Now let $y \in \Sigma$. It is trivial to see that $\text{Cost}(v,y) \geq \text{Cost}^*(v,y)$, so all we need to do is to prove that $\text{Cost}(v,y) \leq \text{Cost}^*(v,y)$. Assume $\text{Cost}^*(v,y)$ is obtained by labelling $w$ and $w'$ by $z$ and $z'$, for some states $z$ and $z'$ in $\Sigma$. Then

$$\text{Cost}^*(v,y) = H(y,z) + H(y,z') + \text{Cost}^*(w,z) + \text{Cost}^*(w',z')$$

and so

$$\text{Cost}^*(v,y) = H(y,z) + H(y,z') + \text{Cost}(w,z) + \text{Cost}(w',z').$$

By the definition of how $\text{Cost}(v,y)$ is computed, $\text{Cost}(v,y) \leq \text{Cost}^*(v,y)$. Hence, $\text{Cost}^*(v,y) = \text{Cost}(v,y)$ for all states $y \in \Sigma$. Hence the algorithm correctly computes $\text{Cost}(v,y)$ for all vertices $v$ and for all states $y \in \Sigma$.

The last step of the algorithm sets the parsimony score of the tree by selecting $y$ so that $\text{Cost}(r,y)$ is minimized, where $r$ is the root of the tree; since $\text{Cost}(r,z)$ is correctly computed for all $z \in \Sigma$, this proves that the algorithm correctly computes the parsimony score of the tree. Backtracking to label the internal nodes of the tree with the optimal states is straightforward.

The running time analysis is easy: the variables we need to compute are $\text{Cost}(v,y)$ for all $v \in V(T)$ and for all $y \in \Sigma$; hence there are $O(nr)$ variables to compute. When we compute $\text{Cost}(v,y)$ in the dynamic programming algorithm, we have to compute all $\text{Cost}(w,z) + H(z,y)$ and $\text{Cost}(w',z) + H(z,y)$, where $w$ and $w'$ are the children of $v$ and where $z \in \Sigma$. Hence, we have to do $O(r)$ work to compute $\text{Cost}(v,y)$ after we have computed $\text{Cost}(w,z)$ and $\text{Cost}(w',z)$ for all $z \in \Sigma$. Hence the total running time to compute all the $\text{Cost}(v,y)$ variables is $O(nr^2)$ for each character. Then, to compute the best score at the root only takes an additional $O(r)$ time. Backtracking adds $O(n)$ time. Hence the total time for a single character is $O(nr^2)$. Since the characters can be processed independently, the total time is $O(nr^2)$.

4.3.3 Finding maximum parsimony trees

What about finding the best tree, rather than computing the score of a given tree? This is the maximum parsimony problem, also known as the “large parsimony problem”.

Maximum parsimony problem

**Input:** Matrix $M$ with $n$ rows and $k$ columns, where $M_{ij}$ denotes the state of taxon $s_i$ for character $c_j$.

**Output:** Tree $T$ on leafset $\{s_1, s_2, \ldots, s_n\}$ with the smallest total number of changes for character set $\{c_1, c_2, \ldots, c_k\}$.
4.4 Constructing trees from compatible characters

Unfortunately, while the small maximum parsimony problem is polynomial time, the large maximum parsimony problem is NP-hard (Foulds and Graham, 1982). Furthermore, exhaustive search or branch-and-bound solutions are limited to small datasets. Fortunately, effective search heuristics exist that enable reasonable analyses on large datasets (with hundreds or even thousands of taxa). Of course, these heuristics are not guaranteed to solve the optimization problem exactly, but many seem to produce trees that are possibly close in score and topology to the optimal solution, when run long enough. However, large datasets may be beyond the reach, even in terms of near-optima, using current software.

Finding good maximum parsimony trees is of substantial interest in the biology community, since many phylogenies are computed using software optimized for this problem. The challenges involved in developing better methods are discussed in Chapter 11.

4.4 Constructing trees from compatible characters

4.4.1 Constructing trees from compatible binary characters

Suppose we have a set of binary characters that we assume have evolved without any homoplasy on some unknown tree $T$, and we wish to infer $T$ from the data. Under this assumption, we will show we can infer an unrooted tree on which all the characters are compatible. Consider one such character $c$, and the tree $T$ on which it has evolved. Because $c$ is homoplasy-free and has only two states, there must be an edge in $T$ that separates the leaves with one state from the leaves with the other state. In other words, $C(T)$ must have a bipartition $\pi$ that produces the same split as $c$. Therefore, given a set of homoplasy-free binary characters, the problem of constructing an unrooted tree consistent with the characters is equivalent to constructing an unrooted tree from a set of bipartitions. This is a problem we studied in Chapter 2, and which can be solved in polynomial time. We describe this algorithm here, for the sake of completeness.

**Algorithm to compute an unrooted tree from compatible binary characters:**

- Input: Set $\mathcal{C}$ of compatible binary characters
- Output: Tree $T$ on which the characters in $\mathcal{C}$ are compatible.

- Let $S$ denote the leafset of $T$. Select one taxon to be the root, and let its state for each character be the ancestral state of that character.
- For every binary character $c$ in the input, denote the set of taxa with the derived state by $S_c$.
- Let $S$ denote the leafset of $T$. Compute the Hasse Diagram for the set $\{A_c : c \in \mathcal{C}\} \cup S \cup \{S\}$. This will produce a rooted tree $T$ with leafset $S$, and that contains each $A_c$ as a clade.
- Return the unrooted version of $T$.

Note that this algorithm produces a tree that may not be fully resolved. However, any other tree that is consistent with the set of binary characters will be a refinement of this
Constructing trees from qualitative characters

Hence this tree is a **minimum compatibility tree** for the set of binary characters. In other words, this tree will be a *common contraction* of all trees that are consistent with the character set. This is a strong statement, and it allows us to explore (and succinctly characterize) the solution space.

Note that the algorithm we described could be modified to test whether the set of characters is compatible, by returning *Fail* if the Hasse Diagram does not produce a tree. Also, the algorithm cannot be used when the state of some character for some taxa is unknown. Note that the equivalence between binary characters are bipartitions yields the following:

**Theorem 4.5** A set $C$ of binary characters is compatible if and only if every pair of binary characters is compatible.

**Example 4.6** Suppose that the input is given by

- $A = (1, 0, 0, 0, 1)$
- $B = (1, 0, 0, 0, 0)$
- $C = (1, 0, 0, 1, 0)$
- $D = (0, 0, 0, 0, 0)$
- $E = (0, 1, 0, 0, 0)$
- $F = (0, 1, 1, 0, 0)$

In this case, there are two non-trivial characters (defined by the first and second positions), but the third through fifth positions define trivial characters. When we apply this algorithm, we pick one taxon as the root. Since the choice of root doesn’t matter, we’ll pick $A$ as the root. The clades under this rooting are: $\{D, E, F\}$ (for the first character), and $\{E, F\}$ (for the second character). We then add the clade $\{B, C, D, E, F\}$ (i.e., everything but the root taxon) and all the singletons. When we apply the algorithm for constructing trees from clades to this set, we get $(A, (B, C, (D, (E, F)))))$. When we unroot this tree, we note that it has a node of degree four, and so is not fully resolved.

Now, suppose we have a data matrix in which we have some missing entries, and we would like to know if it is possible to assign values of 0 or 1 to the missing entries in the character matrix so that the resultant data matrix has a perfect phylogeny. Furthermore, if we can assign the values to make the characters compatible, we would also want to demonstrate this by providing perfect phylogeny for the matrix.

**Example 4.7** In the following input, “?” means that the state is unknown.

- $A = (0, 0, 0)$
- $B = (0, 1, 1)$
- $C = (1, ?, 1)$
- $D = (1, 0, ?)$
- $E = (?, 0, 0)$
We would like to know whether it is possible to set the various missing entries so that the result is a set of compatible characters (i.e., a set of five binary sequences that have a tree on which all the characters are compatible). The answer for this input is yes, as we can use the following assignments of states to the missing values:

- \( A = (0, 0, 0) \)
- \( B = (0, 1, 1) \)
- \( C = (1, 0, 1) \)
- \( D = (1, 0, 1) \)
- \( E = (0, 0, 0) \)

We know this is compatible, because the tree given by \((A, (E, (B, (C, D))))\) is compatible with these characters (i.e., it is a perfect phylogeny).

Example 4.8 By contrast, there is no way to set the values for the missing entries (indicated by ?) in the following matrix to 0 or 1, in order to produce a tree on which all the characters are compatible:

- \( A = (0, 0, ?) \)
- \( B = (0, 1, 0) \)
- \( C = (1, 0, 0) \)
- \( D = (1, ?, 1) \)
- \( E = (?, 1, 1) \)

Figuring out that these characters are incompatible, no matter how you set the missing data, is not that trivial. But as there are only three missing values, you can try all \(2^3 = 8\) possibilities. A more elegant analysis that does not require examining all possible settings is possible, and left to the reader.

More generally, however, answering whether an input with missing data admits a perfect phylogeny is NP-complete, even when only two states otherwise appear (Bolaender et al., 1992; Steel, 1992b).

4.4.2 Constructing trees from compatible multi-state characters

Related to the problem of handling missing data is the question of whether a set of multi-state characters are compatible (i.e., that they evolve down some common tree without any homoplasy). Unfortunately, while determining if a perfect phylogeny exists for binary characters is solvable in linear time (Gusfield, 1991a), determining if a perfect phylogeny exists for multi-state characters is NP-complete (Bolaender et al., 1992; Steel, 1992b). Furthermore, while the compatibility of two multi-state characters can be determined in polynomial time (Warnow, 1993), pairwise compatibility no longer ensures setwise compatibility, even for three-state characters ( Warnow, 1993). Algorithms to determine whether a set of multi-state characters is compatible, and to compute their perfect phylogenies when they exist, have been developed (Kannan and Warnow, 1997; McMorris et al., 1994; Dress and
Steel, 1992), but these are either limited to special cases, or have running times that grow exponentially in some parameter (e.g., the maximum number of states per character, the number of characters, or the number of taxa) in the input.

Although perfect phylogenies are an idealized construct, there are conditions in which nearly perfect phylogenies are expected to be achievable. Examples of such conditions include datasets of individuals in the same species and language phylogenies, and in those cases, methods that can compute perfect phylogenies on subsets of the datasets can be useful.

4.5 Tree construction based on Maximum Compatibility

The Maximum Compatibility problem seeks the tree on which the largest number of characters are compatible. As with Maximum Parsimony, there are two variants of the problem: the “small” maximum compatibility problem, which is about computing the compatibility score of a fixed tree, and the “large” maximum compatibility problem, which is about finding the best tree (i.e., the tree with the largest compatibility score). As we will see, the small maximum compatibility problem is solvable using a small variation on the algorithm for the small maximum parsimony problem. Also, like maximum parsimony, the maximum compatibility problem is also NP-hard.

4.5.1 Algorithm for the small compatibility problem

The problem we address here is to compute the compatibility score of a set of characters on a fixed tree, i.e.:

Input: Matrix \( M \) with \( n \) rows and \( k \) columns (so that \( M_{ij} \) is the state of taxon \( s_i \) for character \( c_j \)), and a tree \( T \) with leaves labelled by the different species, \( s_1, s_2, \ldots, s_n \).

Output: The number of characters that are compatible on \( T \).

To address this question, we can take each character \( c \) in turn, and see if whether we can label the nodes of the tree with character states for \( c \) so that \( c \) has no homoplasy (back-mutation or parallel evolution). Equivalently, we wish to set the states of the character for the internal nodes of the tree in such a way that for each state of the character, the nodes of the tree that exhibit that state are connected.

We begin with a basic observation that allows us to use the algorithm for computing the maximum parsimony score of a character to determine if it is compatible:

Observation 4.9 A character \( c \) is compatible on tree \( T \) if and only if its parsimony score on \( T \) is \( r - 1 \), where \( r \) is the number of states exhibited by \( c \) at the leaves of \( T \).

Proof If the character is compatible on \( T \), then it must be able to be extended to the internal nodes so that there is no homoplasy. Without loss of generality, suppose that \( c \) attains states \( 1, 2, \ldots, r \) at the leaves, and that the state at the root is 1. Every substitution
4.5 Tree construction based on Maximum Compatibility

has to produce a new state, and so there must be exactly \( r - 1 \) edges on which there are substitutions. Hence, \( c \) must have parsimony length \( r - 1 \). Conversely, if \( c \) has parsimony length \( r - 1 \), then the assignment of states to the internal nodes must be homoplasy free, since \( c \) has \( r \) states at the leaves.

It is therefore easy to establish the following corollary:

**Corollary**  Determining if a character \( c \) is compatible on a tree \( T \) can be solved in \( O(nr) \) time using the algorithm for computing the parsimony score on a tree, where \( r \) is the number of states exhibited by \( c \) at the leaves of \( T \) and \( n \) is the number of leaves in \( T \). Hence, the compatibility score of a set of \( k \) characters on a tree \( T \) can be computed in \( O(nkr) \) time.

On the other hand, testing the compatibility of a character on a tree \( T \) can be determined by eye, if the tree is small enough. For a given internal node \( v \) in the tree, if \( v \) lies on a path between two leaves having the same state \( x \), we assign state \( x \) to node \( v \). If this assignment doesn’t have any conflicts – that is, as long as we don’t try to assign two different states to the same node – then the character can evolve without any homoplasy on the tree, and otherwise, homoplasy-free evolution is not possible on this tree. In fact,

**Theorem 4.10**  Computing the compatibility score of a tree \( T \) on \( n \) leaves with respect to \( k \) characters can be computed in \( O(nk) \) time.

The proof of this statement is left to the reader.

4.5.2 Finding maximum compatibility trees

The maximum compatibility problem attempts to find the tree on which the largest number of characters are compatible:

**Maximum Compatibility**

**Input:** Matrix \( M \) with \( n \) rows and \( k \) columns (so that \( M_{ij} \) is the state of taxon \( s_i \) for character \( c_j \))

**Output:** Tree \( T \) on the leafset \( S = \{s_1, s_2, \ldots, s_n\} \) on which the number of characters in \( C = \{c_1, c_2, \ldots, c_k\} \) that are compatible is maximized.

Finding the tree with the largest compatibility score is an NP-hard problem, even for the special case where all the characters are binary (Day and Sankoff, 1986). We sketch the proof for binary characters.

**Theorem 4.11**  Maximum Compatibility is NP-hard, when the input is a set of binary characters.

**Proof**  We will prove that the decision problem for Binary Character Maximum Compatibility is NP-complete. The decision problem has two inputs: the data matrix defining a set \( S \) of taxa in terms of a set \( C \) of binary characters, and an integer \( B \). We will provide a
Karp reduction from Maximum Compatibility of binary characters to Max Clique, which is NP-hard. Let $\mathcal{C}$ be a set of binary characters defined on a set $S$ of taxa, and let integer $B$ be given. We will create a graph $G(\mathcal{C})$ so that $\mathcal{C}$ has a subset of $B$ compatible characters if and only if $G(\mathcal{C})$ has a clique of size $B$.

The vertices in $G(\mathcal{C})$ will be the characters $c \in \mathcal{C}$, and the edges will be those pairs $(v_c, v_{c'})$ such that $c$ and $c'$ are pairwise compatible. It is easy to see that if $\mathcal{C}$ has a set of $B$ compatible characters, then the corresponding vertices in $G(\mathcal{C})$ are a clique of size $B$. If $G(\mathcal{C})$ has a clique of size $B$, then the corresponding characters in $\mathcal{C}$ are pairwise compatible. Since pairwise compatibility of binary characters ensures setwise compatibility, it follows that $\mathcal{C}$ has a set of $B$ compatible characters. Hence, binary character maximum compatibility is NP-hard.

Note that this was a linear reduction, so that binary character maximum compatibility has the same approximability results as maximum clique! Unfortunately, maximum clique is one of the hardest problems to approximate, as shown in Zuckerman (2006).

**Example 4.12** Consider the following set of taxa, represented by three characters.

- $A = (0,0,0)$
- $B = (0,0,3)$
- $C = (1,1,0)$
- $D = (1,1,1)$
- $E = (2,1,0)$
- $F = (2,2,4)$

This set of characters is compatible, since $(A, (B, (E, (F, (C, D)))))$ is a perfect phylogeny for the input.

**Example 4.13** The next example is more interesting, because there is no perfect phylogeny:

- $A = (0,1,0)$
- $B = (0,0,0)$
- $C = (1,0,0)$
- $D = (1,1,1)$

Note that the third character is compatible on every tree, but the first two characters are incompatible with each other. Therefore, any tree can have at most one of these first two characters compatible with it, for a total of two compatible characters. One of those trees is given by $((A,B),(C,D))$, and the other is $((A,D),(B,C))$. The third possible unrooted binary tree on these taxa is $((A,C),(B,D))$, but it is incompatible with both these characters.
4.6 Treatment of missing data

Input matrices to maximum parsimony and maximum compatibility often contain missing data, indicated by symbols such as “?”. (As we will see later, multiple sequence alignments also contain letters other than nucleotides or amino acids, and the dashes in the alignments are also often treated as missing data.) In a maximum parsimony or maximum compatibility analysis, these missing data entries are typically replaced by character states so as to obtain the best possible score; the output, however, will include the symbols representing the missing data.

An alternative treatment of missing data replaces all the missing entries with the same new state, and then seeks a tree that optimizes the criterion. This approach can produce a different result.

Finally, in some analyses, all columns that are considered to have too missing data are removed before the tree is computed. The decision about how much missing data is “too much” also varies.

4.7 Informative and uninformative characters

Both maximum parsimony (MP) and maximum compatibility (MC) are computationally intensive if solved exactly. Thus, finding ways to speed up methods to try to solve MP and MC can be helpful. Here, we consider the question of whether we can eliminate some of the characters from the input, without changing the solution space. In other words, given input matrix $M$ (where $M_{ij}$ is the state of the taxon $s_i$ for the $j^{th}$ character), you would like to know whether removing some specific character (say character $x$) has any impact on the tree that is returned. For example, suppose we wish to solve maximum parsimony (MP).

Since removing a character amounts to removing one column in the matrix, this would be the same as saying “If we define matrix $M-x$ to be the matrix obtained by taking $M$ and removing column $x$, when is it guaranteed that the set of optimal parsimony trees for $M-x$ are optimal parsimony trees for $M$? If we can prove that it is safe to delete the column $x$, then removing it could reduce the running time of searching for optimal maximum parsimony trees (and similarly for maximum compatibility). A character that has no impact on tree estimation using maximum parsimony methods (when solving MP exactly) is called “parsimony uninformative”. We formally define this as follows:

**Definition 4.14** Let $x$ be a character defined on set $S$ of species. Then $x$ is **parsimony uninformative** if and only if for all matrices $M$ for $S$ the set of optimal parsimony trees on $M$ is identical to the set of optimal parsimony trees on $M+x$, where $M+x$ denotes the matrix obtained by adding column $x$ to $M$.

As a consequence, the set of optimal parsimony trees will not change by removing a parsimony uninformative site from any alignment. All other characters are called “parsimony informative”. Removing parsimony uninformative characters can result in a speed-up in the search for optimal trees (especially if there are many such characters). Equally importantly,
thinking about which characters are parsimony informative or not will help you understand the different impact of different characters on phylogeny estimation using maximum parsimony.

The same property can be asked about any phylogeny estimation method, obviously, and so we can ask whether a character is “compatibility-informative”. It is not hard to see the following:

**Lemma 4.15** A character is parsimony-informative and compatibility-informative if and only if it has at least two “big states”, where a state is “big” if it has at least two taxa in it.

**Proof** Suppose a character $c$ exhibits $r$ states on the taxon set $S$. We will begin by showing that $c$ is parsimony and compatibility uninformative if it does not have two or more big states. There are two possible cases for this: $c$ has exactly one big state (and so all other states are singletons), or it has only singleton states. In either case, let $i$ be one of the largest states for $c$ (thus, $i$ is the unique big state, or $i$ is a singleton state). Given a tree $T$ to be scored, we put $i$ as the state for every internal node in $T$. Note that $c$ is compatible on the tree $T$ using this labelling for the internal nodes, and that the tree has parsimony score $r-1$. Since $r-1$ is the best possible parsimony score for any tree with $r$ states appearing at the leaves and the tree $T$ was arbitrary, this means that all trees have the same parsimony and compatibility score for this character. Therefore, removing $c$ from the set of characters will not change the relative ordering of trees with respect to either parsimony or compatibility. This proves one direction of the theorem.

For the other direction, let $c$ be a tree with at least two big states, $i$ and $j$. Hence, there is a quartet of taxa $u, v, x, y$ for which $c(u) = c(v) = i$ and $c(x) = c(y) = j$. Now, consider the case where $C = \{c\}$, so that $C$ only has this one character. If we remove $c$ from $C$ we obtain the empty character set, for which all trees are equally good with respect to both parsimony and compatibility. Hence, all we need to show is that there are at least two trees that have different parsimony and compatibility scores with respect to $c$. It is easy to see that there is at least one tree $T$ on which $c$ is compatible, and so it has compatibility score 1 and parsimony score $r-1$. Furthermore, we can define a second tree $T'$ formed by beginning with quartet tree $ux|yz$, and then attaches all the other leaves arbitrarily to this quartet tree. Since $ux|yz$ is incompatible with $c$, it follows that $c$ will not be compatible on the tree $T'$, and so its parsimony score will be greater than $r-1$ and its compatibility score will be 0. Thus, $c$ distinguishes $T$ and $T'$ with respect to both parsimony and compatibility.

Other methods can be used to estimate trees, of course, and so the definition of what constitutes “informative” has to be based on the method.

**Example 4.16** Consider the following set of four DNA strings, $u, v, w, x$:

- $u = $ACAAAAAG
- $v = $ACTTTTCG
- $w = $TTTTTTTG
4.8 Further reading

The question of whether a perfect phylogeny exists for a character matrix defining a set of taxa by a set of qualitative characters, and constructing the perfect phylogeny when it exists, is an approach to phylogeny estimation that was introduced in LeQuesne (1969), and then elaborated on in a series of papers by various authors. This problem, which is known as the Character Compatibility problem or the Perfect Phylogeny problem, is equivalent to a graph-theoretic problem, which asks whether it is possible to add edges to a given vertex-colored graph $G$ so that it becomes triangulated, without adding edges between vertices of the same color. These two formulations of the problem have led to a collection of algorithms, some combinatorial and others graph-theoretic, that collectively address the fixed-parameter variants of the two problems (i.e., algorithms when the number of states per character is bounded or when the number of characters is bounded). For an overview of the early history of this problem (up until 1993), see Warnow (1993).

4.9 Review questions

1. What is a perfect phylogeny?
2. Define homoplasy and give an example of a biological characteristic that clearly evolved homoplastically.
3. Give some examples of evolution that are not treelike.
4. What is a character? What is a binary character?
5. Suppose you have a binary character matrix, so the rows represent species and the columns represent characters, and every entry of the matrix is either a 0 or a 1. What does it mean to say that the character set is compatible? How computationally difficult is it to determine if the character set is compatible?
6. Suppose you have a multi-state character matrix, so the rows represent species and the columns represent characters, but the entries of the matrix can have any integral value. What does it mean to say that the character set is compatible? How computationally difficult is it to determine if the character set is compatible?

7. How computationally difficult is it to test whether a character is compatible on a tree?

8. State the maximum compatibility and maximum parsimony problems. How computationally difficult is it to solve each problem (i.e., are these problems in \( P \), \( NP \)-hard, or of unknown computational complexity)?

9. Suppose you are given a tree \( T \) and there are DNA sequences (each of length \( k \)) at the leaves of the tree. What would you do to calculate the parsimony score of these sequences on the tree?

10. Define "parsimony informative", and give an example of a binary character that is not parsimony informative.

### 4.10 Homework problems

1. Suppose we are given the following input of four taxa described by six-tuples (i.e., six characters), where each character is binary. We let 0 denote the ancestral state and 1 denote the derived state. Construct the rooted tree that is consistent with these characters evolving without homoplasy.
   - \( a = (1, 1, 0, 0, 1, 0) \)
   - \( b = (1, 0, 1, 0, 1, 0) \)
   - \( c = (0, 0, 0, 1, 0, 0) \)
   - \( d = (0, 0, 0, 0, 1, 1) \)

2. Take the data matrix from the previous problem, and add in the root sequence, \( r \), given by \( r = (0, 0, 0, 0, 0, 0) \). Thus, you now have a matrix with five taxa, \( a, b, c, d, r \), defined by six characters (one character for each position in the 6-tuple). Divide this matrix into two pieces: the first three characters, and the last three characters. Construct the minimally resolved unrooted tree that is compatible with each submatrix. How are these trees different? Are they fully resolved, or do they have polytomies? Compare them to the compatibility tree you obtained on the full matrix. Now, treat the tree on the full matrix as the “true tree”, and compute the False Negative and False Positive rates for these two trees. What do you find?

3. Construct an unrooted tree that is consistent with the following input of four taxa described by four binary characters, under the assumption that all characters evolve without homoplasy. (You may not assume that any particular state is ancestral on any character.)
   - \( a = (0, 0, 1, 1) \)
   - \( b = (1, 0, 0, 1) \)
   - \( c = (1, 1, 0, 1) \)
   - \( d = (1, 0, 1, 0) \)
4. For the tree $T$ given by $((a, (b, (c, (d, (e, f)))))))$, determine for each of the characters (columns in the following tuple representation) whether it could have evolved on the tree $T$ without any homoplasy:

- $a = (0, 0, 0, 0, 1)$
- $b = (0, 1, 1, 0, 0)$
- $c = (1, 0, 0, 1, 1)$
- $d = (1, 2, 0, 1, 0)$
- $e = (2, 0, 2, 0, 1)$
- $f = (2, 3, 2, 0, 1)$

5. For the following input, show how to set the entries given with “?” so as to produce a compatible matrix:

- $A = (0, 1, 0, ?)$
- $B = (0, 1, 1, 0)$
- $C = (0, 0, 1, 0)$
- $D = (1, 0, 1, 1)$
- $E = (1, 0, ?, 1)$

Explain how you derived your solution.

6. In the text, we said that there was no way to set the values for the missing entries in the following matrix, in order to produce a tree on which all the characters are compatible:

- $A = (0, 0, ?)$
- $B = (0, 1, 0)$
- $C = (1, 0, 0)$
- $D = (1, ?, 1)$
- $E = (?, 1, 1)$

Prove this assertion.

7. Suppose $T$ and $T'$ are two trees on the same leafset, and $T'$ refines $T$.

- Prove or disprove: if character $c$ is compatible on $T$ then it is compatible on $T'$.
- Prove or disprove: if character $c$ is compatible on $T'$ then it is compatible on $T$.

8. The maximum parsimony problem asks us to find a tree that has the best maximum parsimony score with respect to a matrix $M$. Suppose we consider the following problem, “binary tree maximum parsimony”: Given a matrix $M$, find a binary tree that optimizes maximum parsimony.

1. Is it possible for a solution to the “binary tree maximum parsimony” problem to not be optimal for the standard maximum parsimony problem?
2. Consider the same question but restated in terms of maximum compatibility and “binary tree maximum compatibility”. Does your answer change?

9. Consider the set of six taxa described by two multi-state characters, $A = (0, 0), B = (1, 2), C = (0, 2), D = (2, 1), E = (1, 1)$, and $F = (1, 0)$, and the tree on the taxa given by: $(((A, B), C), (D, (E, F))))$. 

• Apply the parsimony algorithm to assign states to each node for each of the two characters. What is the parsimony score of this tree?
• For which nodes of the tree is the character state of either character determined, and for which nodes is it optional?
• Give two different character state assignments to the nodes to produce the minimum number of changes.

10. Consider the set of sequences (but ignore the tree provided) given for the input given in the previous problem. Find an optimal (unrooted) tree topology $T$ on this set. (Do this without trying to score all possible trees - think about the best achievable score for this specific dataset.) Are either of the characters compatible on $T$? If not, find an optimal MP tree for this input for which at least one character is compatible.

11. Suppose $T$ and $T'$ are two trees on the same leafset, and $T'$ refines $T$. Prove that the parsimony score of $T'$ is at most that of $T$.

12. Suppose the tree is given by $((A, (B, (C, (D, E)))))$, and that we have three homoplasys-free characters on these taxa given by:
   - $A = (0, 0, 1)$
   - $B = (0, 1, 1)$
   - $C = (0, 0, 0)$
   - $D = (1, 0, 0)$
   - $E = (1, 0, 0)$

   Assume that 0 is the ancestral state and 1 the derived state for each of these characters. Determine the edges in the tree that could contain the root.

13. Let $S = \{s_1, s_2, \ldots, s_n\}$ be a set of binary sequences of length $k$ and let $T$ be a binary tree on the same leafset. Which of the following is the correct running time of the dynamic programming algorithm for computing the parsimony score of $T$ with this set of sequences at the leaves?
   - $\Theta(nk)$
   - $\Theta(2^k)$
   - $\Theta(2^k n)$
   - $\Theta(n^2 k)$

14. Suppose $M$ is an input matrix for maximum parsimony, so $M$ assigns states for each character to all the taxa in a set $S$. Suppose $M'$ the result of removing all characters from $M$ that are identical on all taxa (i.e., characters $c$ such that $c(s) = c(s')$ for all $s, s'$ in $S$). Prove or disprove: $M$ and $M'$ have the same set of optimal trees under maximum parsimony.

15. Suppose $M$ is an input matrix for maximum parsimony and $M'$ the result of removing all characters from $M$ that have different states on every taxon (i.e., i.e., characters $c$ such that $c(s) \neq c(s')$ for all $s \neq s'$ in $S$). Prove or disprove: $M$ and $M'$ have the same set of optimal trees under maximum parsimony.
16. Let $M$ be an input matrix to maximum parsimony (MP), and let $M'$ be the result of removing all parsimony uninformative characters from $M$. Thus, $M'$ has a subset of the columns of $M$. By the definition of parsimony uninformative, the trees that are returned by an exact MP solution on $M'$ will be the same as the maximum parsimony trees returned for $M$. However, suppose you use the characters to define “branch lengths” in some output tree (as there can be many), as follows. You use maximum parsimony to calculate ancestral sequences, and then you use Hamming distances to define the branch lengths on the tree.

1. Is it the case that branch lengths you compute on a given tree $T$ must be the same for $M$ as for $M'$? (In other words, can branch length estimations change?)
2. If you use normalized Hamming distances instead of Hamming distances, does your answer to the previous question change?

17. Consider the following input matrix to maximum parsimony:

- $a = (0, 1, 0, 0, 0)$
- $b = (0, 0, 1, 1, 1)$
- $c = (0, 0, 2, 3, 2)$
- $d = (0, 2, 0, 1, 1)$
- $e = (1, 2, 0, 1, 1)$
- $f = (0, 0, 3, 2, 1)$

Write down all the trees that have the best maximum parsimony scores on this input, and explain how you obtain your answer. Do not solve this by looking at all possible trees on $\{a, b, c, d, e, f\}$.

18. Is it the case that maximum compatibility and maximum parsimony always return the same set of optimal trees? If so prove it, and otherwise find a counterexample.
Distance-based tree estimation methods

5.1 Introduction

The first step in a distance-based method for tree estimation is the calculation of a distance matrix for the input set (typically, a set of aligned molecular sequences). Once this step is completed, a tree can be computed from the distance matrix. Thus, distance-based tree estimation is really a two-step process, and many versions of the two-step process are polynomial time. Hence, distance-based tree estimation methods often are fast in practice, and have a computational advantage over most other techniques for computing phylogenetic trees.

As discussed in Chapter 1, for the approach to have good statistical properties, the estimated distances should be defined so that they converge (as the sequence length increases) to an additive matrix for the true tree. The first step, therefore, depends on the specific assumptions of the evolutionary model.

Gene tree estimation is one of the applications of distance-based methods. In this context, the input is a sequence alignment, and distances between the sequence are computed using an assumed model of sequence evolution, such as the Generalized Time Reversible (GTR) model (Tavaré, 1986). This approach is discussed in Chapter 8.

Distance-based tree estimation can also be used for estimating phylogenetic trees from other types of data. For example, species trees can be computed using multiple gene trees, where gene trees can differ from each other (and from the species tree) due to biological processes such as incomplete lineage sorting, horizontal gene transfer, and gene duplication and loss. Methods for computing distances between species have been developed that are statistically consistent under the multi-species coalescent model, and so can be used to estimate species trees in the presence of gene tree discord due to incomplete lineage sorting; Chapter 10 describes this framework.

Phylogenetic trees on whole genomes can also be computed, using genomic architectures (e.g., the order and copy number of genes within genomes). Methods for computing distances between genomes based on rearrangements (inversions, transpositions, and inverted transpositions) and other types of events (e.g., duplications, fissions, and fusions) have been developed and used to compute genome-scale trees, and are described in Chapter 13.3. Some of these methods have even been used to understand the evolution of the
Canterbury Tales (Spencer et al., 2003)! Phylogenies have also been computed for languages, such as the Indo-European family of languages, using distance-based methods. These methods are described in Chapter 14.

In this chapter, we will assume that the first step has been completed, so that the input is a matrix $D$ of estimated pairwise distances between different taxa, and the objective is a tree on the taxa. In later chapters, we will then return to distance-based methods, and show how they can be used in conjunction with methods for computing distances in these different contexts (the estimation of individual gene trees from sequence data, species trees using multiple loci, genome-scale trees, and linguistic phylogenies).

5.2 Basics of distance-based tree estimation

The input to a distance-based method is a set $S = \{s_1, s_2, \ldots, s_n\}$ of taxa, typically represented by molecular sequences. Given this set, a distance-based method operates as follows:

- Step 1: compute pairwise distances between every pair of taxa in the set $S$, thus producing an $n \times n$ matrix $d = (d_{ij})$ (where $d_{ij}$ is the “distance” between taxa $s_i$ and $s_j$), and
- Step 2: compute a tree $T$ (and perhaps its edge weights) from matrix $d$.

The matrices computed in step one will be symmetric (i.e., $d_{ij} = d_{ji}$ for all $i, j$) and zero on the diagonal (i.e., $d_{ii} = 0$ for all $i$). However, these distances generally will not satisfy the triangle inequality, defined as follows:

**Definition 5.1** An $n \times n$ matrix $d$ is said to satisfy the **triangle inequality** if, for all $i, j, k$, $d_{ik} \leq d_{ij} + d_{jk}$.

A matrix that satisfies all three conditions (symmetric, zero on the diagonal, and triangle inequality) is called a “distance matrix”. Since phylogenetic distances may not satisfy the triangle inequality, to be completely mathematically rigorous, we should not refer to them as distance matrices, and instead should only refer to them as “dissimilarity” matrices. However, the literature in phylogeny estimation abuses the term, and refers to dissimilarity matrices as “distance matrices”. Therefore, we will continue this abuse, and refer to the matrices in this way. However, please be aware that the matrices may not satisfy the triangle inequality, and hence any proofs regarding distance-based phylogeny estimation cannot assume this property.

Once the distance matrix is computed, then there are many methods used to construct trees from distance matrices, many of which are guaranteed to produce trees with the same topology as the model tree under some circumstances. Thus, distance-based estimation techniques depend on statistical techniques for calculating distances, and on appropriate computational methods for computing trees from distance matrices, in order to have guarantees. Finally, distance-based methods typically run in polynomial time, which makes the use of distance-based estimation methods attractive in practice.
5.2.1 Distance-based tree estimation methods

5.3 UPGMA

One of the original ways of computing trees is UPGMA, which stands for “Unweighted Pair Group Method with Arithmetic Mean” (Sokal and Michener, 1958). UPGMA computes a rooted tree from an input distance matrix in an agglomerative fashion. In the first iteration, it finds a pair of taxa \(x, y\) that have the smallest distance, and makes them siblings. These two are then replaced by the cluster \(\{x, y\}\), and the distance from \(\{x, y\}\) to every other taxon \(z\) is defined to be the average of \(d(x, y)\) and \(d(x, z)\); i.e., \(d(\{x, y\}, z) = \frac{d(x, z) + d(y, z)}{2}\). In subsequent iterations, the elements are clusters that are either singletons (the original taxa) or sets of two or more taxa. In each iteration, the pair of clusters \(A\) and \(B\) with the minimum distance is selected to be the next sibling pair and the clusters are merged. The distance matrix is then updated by reducing its dimension by one (removing rows and columns for \(A\) and \(B\) and replacing it with a row for \(A \cup B\)) with \(d(A \cup B, C) = \frac{|A|d(A, C) + |B|d(B, C)}{|A| + |B|}\). The process is repeated until all the taxa are merged into a single cluster. Note that this technique produces a rooted tree. Variations of this technique can be considered where other ways of updating the distance matrix are used (e.g., consider the simple variant described in Chapter 1), but the basic algorithmic structure is to find the pair of clusters that are closest and replace them with a new cluster.

We begin with an example of UPGMA applied to a case where the distances obey a strong clock and so produce an ultrametric matrix. Figure 5.1 gives an ultrametric matrix, and Figure 5.2 gives the rooted tree realizing that matrix. If we had applied UPGMA to the matrix in Figure 5.1, we would obtain the tree in Figure 5.2. Hence, UPGMA would have been correct on this input.

Not all distances obey a strict molecular clock, and UPGMA can fail when the input matrix is not sufficiently clocklike. Consider, for example, the tree given in Figure 5.3. This tree has lengths on each edge, and thus defines a distance between every pair of
5.3 UPGMA

Figure 5.2 Rooted tree realizing the ultrametric matrix from Figure 5.1. Note that the distance from the root to every leaf is the same.

Figure 5.3 Additive matrix and its edge-weighted tree leaves obtained by adding up the lengths of each edge. Note that the pair that minimizes the distance is $L_1, L_2$, but that these are not siblings! Thus, when UPGMA is applied to the matrix for this tree, it will produce the \textit{wrong} tree.

UPGMA has a very specific way of updating the distance matrix, and variants of UPGMA can update the matrix using simpler techniques. For example, in Chapter 1, we described a variant where we updated the matrix by just removing the row and column corresponding to one of the two taxa being made into siblings. Clearly the specific technique used to update the distance matrix will impact the final tree. However, \textit{any} agglomerative method that makes the taxon pair with a minimum pairwise distance siblings will fail on an input distance matrix that has its nearest taxa not being siblings. Also, for any variant of UPGMA, just knowing that it has that algorithmic design allows you to determine the first sibling pair (up to ties). Therefore, if the dataset has only three leaves, you can determine the rooted tree that will be constructed, and if the dataset has four leaves, then you can determine the \textit{unrooted} tree it will produce. Therefore, for at least some model conditions, proving that UPGMA or some variant of UPGMA produces the wrong tree is generally
easy: if the first sibling pair it produces isn’t a true sibling pair, the result will be incorrect, no matter what the subsequent steps are.

5.4 Additive Matrices

Given a tree $T$ with non-negative edge weights, and given two leaves $x$ and $y$ in $T$, we define the distance $D(x, y)$ between $x$ and $y$ to be the sum of the weights of the edges on the path between $x$ and $y$. It is easy to see that the matrix $D$ defined in this manner is a true distance matrix (i.e., it is a dissimilarity matrix that satisfies the triangle inequality). Any distance matrix that can be derived in this manner is said to be additive. Furthermore, given any additive matrix, there is a unique tree with strictly positive branch lengths that fits the matrix exactly, obtained by collapsing all the edges that have zero length. We will refer to this tree as the minimally resolved tree realizing the additive matrix. One of the most well known theorems about additive matrices was established in Buneman (1974b):

**Theorem 5.2 The Four Point Condition:** A $n \times n$ matrix $M$ is additive if and only if it satisfies the “four point condition” for all four indices $i, j, k, l$, which is that the median and largest of the following three values are the same:

- $M_{i,j} + M_{k,l}$
- $M_{i,k} + M_{j,l}$
- $M_{i,l} + M_{j,k}$

One direction of the proof of this theorem is easy—if the matrix is additive, then there is a tree $T$ with non-negative edge weights realizing the additive matrix. In that case, given the four indices, $i, j, k, l$, you can assume (without loss of generality) that the tree has an edge separating $i, j$ from $k, l$. In that case, $M_{i,j} + M_{k,l}$ is not greater than the other two sums, and the other two sums have the same total score. Furthermore, if the edge between $i, j$ and $k, l$ has positive length, then $M_{i,j} + M_{k,l}$ is strictly smaller than the other two sums.

Thus, there are two possible outcomes when we examine an additive matrix and compare the three pairwise sums for four indices, $i, j, k, l$: all pairwise sums are identical or there are two different values. How can we use this to infer the topology of the minimally resolved tree realizing the additive matrix on this set of four leaves?

The answer is simple. If the smallest of the three pairwise sums is $M_{i,j} + M_{k,l}$ and it is strictly smaller than the largest two, then $i/j/k/l$ is the quartet tree on $\{i, j, k, l\}$. Otherwise, all three pairwise sums are identical. When the three pairwise sums are equal, any tree that realizes the additive matrix either induces a star tree on $\{i, j, k, l\}$ or induces a resolved tree on $\{i, j, k, l\}$ in which the internal edge has length 0. Note that when the additive matrix corresponds to a binary tree in which the branch lengths are all positive, then for all four leaves, the three pairwise sums will not be identical, and the resolved quartet tree topology can be inferred using the pairwise sum that is smallest.

So suppose that we have an additive matrix corresponding to an edge-weighted tree where all the edge weights are positive. Obviously, the tree can be inferred from its additive
matrix, since we can construct the correct quartet tree on every four leaves, and then use the All Quartets Method to combine these quartet trees into the tree on the full dataset.

However, what about estimating trees from distance matrices that are not additive? Here we show a method that can always estimate a tree on four-leaf datasets, whether or not the matrix is additive.

5.5 Estimating four-leaf trees: The Four Point Method

Given four taxa, \(i, j, k, l\), and given the dissimilarity matrix \(d\) on the four taxa, the Four Point Method (FPM) operates as follows:

- **Step 1:** Compare the three pairwise sums \(d_{i,j} + d_{k,l}\), \(d_{i,k} + d_{j,l}\), and \(d_{i,l} + d_{j,k}\), and find the pairwise sums that have the smallest total.

- **Step 2:** If there are two or more pairwise sums with the same smallest total (i.e., the minimum is not unique), then return a star tree. Else, without loss of generality, assume \(d_{i,j} + d_{k,l}\) has the smallest value, and return the tree \(i j | kl\).

**Theorem 5.3** Let \(M\) be a \(4 \times 4\) additive matrix corresponding to a tree \(T\) on four leaves with strictly positive branch lengths. Then the Four Point Method applied to \(M\) returns tree \(T\).

**Definition 5.4** Given a binary tree \(T\) leaf-labelled by taxon set \(S\), and real-valued edge weights defined by the function \(w : E(T) \to \mathbb{R}\), we will define \(E_I\) to be the set of internal edges in the tree, and \(f = \min_{e \in E_I} \{w(e) : e \in E_I\}\).

Note that \(f > 0\) for all trees \(T\) with positive branch lengths.

**Lemma 5.5** Let \(T\) be an edge-weighted tree with \(n \geq 4\) leaves such that \(w(e) \geq f > 0\) for all internal edges \(e\). Let \(M\) be the additive matrix for this edge-weighted tree, and let \(d\) be an \(n \times n\) matrix satisfying \(L_\infty(d, M) < f/2\). Select any four leaves \(q = \{i, j, k, l\}\) and consider the submatrix \(d(q)\) of \(d\) induced by quartet \(q\). Then the Four Point Method applied to distance matrix \(d(q)\) will return the tree induced by \(T\) on leaf set \(q\).

**Proof** Recall that the Four Point Method returns the quartet tree \(i j | kl\) if and only if \(d_{i,j} + d_{k,l}\) is strictly less than the other two pairwise sums formed using these four indices (and that when two or more pairwise sums are tied for having the smallest values, then the Four Point Method returns a star tree). Hence, we just need to prove that when \(L_\infty(d, M) < f/2\) then \(d_{i,j} + d_{k,l}\) is the smallest of the three pairwise sums, under the assumption that \(M_{i,j} + M_{k,l}\) is the smallest of its three pairwise sums.

Note that \(M_{i,j} + M_{k,l} = M_{i,k} + M_{j,l} - 2F = M_{i,l} + M_{j,k} - 2F\), where \(F\) is the length of the path separating the pairs \(i, j\) from \(k, l\) in the edge-weighted true tree \(T\) associated with \(M\). Therefore, \(F \geq f\), where \(f\) is the length of the shortest edge in \(T\). Now, since \(L_\infty(d, M) = \delta < f/2\), it follows that the gap between \(d_{i,j} + d_{k,l}\) and the other two pairwise sums can only be reduced by \(4\delta\) (adding \(\delta\) to each of \(M_{i,j}\) and \(M_{k,l}\) and subtracting \(\delta\) from each of the other four entries). Therefore, if \(4\delta < 2F\), it follows that the Four Point Method will
return the correct tree on \(i, j, k, l\). Since we require that \(\delta < \frac{f}{2}\) and \(f \leq F\), it follows that the \(4\delta < 2F\). Hence, the Four Point Method will correctly return the true quartet tree on any quartet \(i, j, k, l\) whenever \(L_{\infty}(d, M) < f/2\).

\[\square\]

5.6 Quartet-based methods

In the previous section, we presented the Four Point Method, and showed that we can use it to construct quartet trees, and in Chapter 3.4, we presented several methods for combining quartet trees into a tree on the full dataset. The combination of these techniques suggests phylogenetic tree estimation methods based on combining these approaches, the subject of this section.

5.6.1 The Naive Quartet Method

We begin with the simplest quartet-based method for phylogeny estimation. The input is an \(n \times n\) dissimilarity matrix \(d\), and we assume \(n \geq 5\).

Step 1: For every four indices \(i, j, k, l\), use the Four Point Method (FPM) on matrix \(d\) restricted to the rows and columns for \(i, j, k, l\). If the output of the FPM for some set of four indices is \textit{Failure}, then return \textit{Failure}. Otherwise, the FPM returns a resolved quartet tree for every set of four indices.

Step 2: Apply the All Quartets Method (from Chapter 3.4.2) to test if the quartet trees are compatible. If they are compatible, then return the tree \(T\) that agrees with all the quartet trees; else return \textit{Failure}.

This approach was introduced in Erdos et al. (1999c), where it was called the “Naive Method”; here, we call it the Naive Quartet Method, to emphasize the use of quartet trees.

\textbf{Theorem 5.6} (From Erdos et al. (1999c)) Let \(d\) be the input \(n \times n\) dissimilarity matrix and \(D\) be the matrix corresponding to a tree \(T\) with positive edge lengths. Assume \(L_{\infty}(d, D) < f/2\), where \(f > 0\) is the minimum length of any internal edge of \(T\). Then the Naive Quartet Method applied to \(d\) will return \(T\).

\textbf{Proof} By Lemma 5.5, the Four Point Method will return the correct quartet tree topology on every four leaves in \(T\). Then, since all the quartet trees are correct, the All Quartets Method will return the true tree \(T\) from the set of quartet trees computed by the Four Point Method. \(\square\)

What we have shown is that the Naive Quartet Method will be accurate on a small enough neighborhood of any additive matrix that corresponds to the true tree with positive branch lengths. What does the Naive Quartet Method do when the input matrix \(d\) is not close enough to an additive matrix corresponding to the true tree? Unfortunately, it is not easy to state the outcome for the Naive Quartet Method given a matrix \(d\) that is not
close to an additive matrix for the true tree. However, since getting even one quartet tree incorrect can produce an incompatible set of quartet trees and the All Quartets Method will return \textit{Failure} whenever the quartet trees are incompatible, Naive Quartet Method is very vulnerable to errors in the input dissimilarity matrix.

The neighborhood is defined by the $L_{\infty}$ distance between matrices, and the \textit{radius} of this neighborhood is $f/2$, where $f$ is the minimum length of any internal edge in the tree. This property of the Naive Quartet Method is called its safety radius. We formalize this, so we can use this term in when we consider other methods:

\textbf{Definition 5.7} Let $(T,w)$ be an edge-weighted tree defining additive matrix $A$, and let $\Phi$ be a distance-based phylogeny estimation method. Then the \textit{safety radius} of $\Phi$ is the largest value $x$ such that whenever the input dissimilarity matrix $d$ satisfies $L_{\infty}(d,A) < x$, then $\Phi(d)$ returns an additive matrix $A'$ corresponding to a weighted version of tree topology $T$.

\section*{5.6.2 The $Q^*$ method and related techniques}

Any method for combining quartet trees can be used to estimate a tree from the set $Q$ of quartet trees computed using the Four Point Method, or any other technique. Hence, distance-based phylogeny estimation can be addressed through many variations on this theme: compute quartet trees using the input dissimilarity matrix, and then combine the quartet trees using a quartet amalgamation method.

The $Q^*$ method (Berry and Gascuel, 1997), also known as the Buneman Tree, is the most well known of these methods. The first step of the $Q^*$ method is the calculation of the set $Q$ of estimated quartet trees using the Four Point Method. The second step seeks a maximally resolved tree $T$ and a subset $Q^* \subseteq Q$ such that the set $Q_r(T)$ of resolved quartet trees in $Q(T)$ is equal to $Q^*$. It is easy to see that the tree $T$ always exists, since the star tree is a feasible solution (with $Q^* = \emptyset$), and Buneman (1971) proved that there is a unique such tree. The unique maximally resolved tree $T$ satisfying $Q_r(T) \subseteq Q$ is called the Buneman Tree or the $Q^*$ tree, and can be calculated in polynomial time using a greedy technique: arbitrarily order the leaves, and then construct the tree by adding the next leaf into the previous tree, collapsing edges if necessary, to maintain the required property that no edge in the tree be incompatible with any of the quartet trees in $Q$. A more careful implementation, described in Berry and Gascuel (1997), shows that the $Q^*$ Tree can be computed in $O(n^3)$ time.

The $Q^*$ Tree method is superior to the Naive Quartet Method, in that it is guaranteed to return a tree on every input. However, the $Q^*$ tree is also generally highly unresolved, since any quartet tree that violates a bipartition will cause that bipartition to not appear in the $Q^*$ tree. In other words, as the dissimilarity matrix $d$ deviates from additivity, the Four Point Method can fail to correctly compute quartet trees, and this will reduce the resolution in the $Q^*$ Tree. Therefore, extensions of the $Q^*$ tree method (Berry et al., 1999; Berry and Bryant, 1999; Berry et al., 2000; Della Vedova et al., 2002; Faghhi and Brown, 2010;
Bryant and Moulton, 1999) have been developed that will produce trees with bipartitions that are contradicted by a bounded number of quartet trees in Q.

Yet these quartet-based methods have not generally been as accurate as the better distance-based estimation methods. For example, St. John et al. (2003) evaluated several quartet-based methods, including the Buneman Tree, Quartet Cleaning, and Quartet Puzzling (Strimmer and von Haeseler, 1996), and compared them to neighbor joining. On the datasets St. John et al. (2003) examined, they found that neighbor joining had much better accuracy than any of the quartet-based methods. Most importantly, St. John et al. (2003) showed that using neighbor joining to compute a tree on the entire dataset and then using the quartet trees induced by the neighbor joining tree produced more accurate quartet trees than using the Four Point Method. This observation suggests that estimating quartet trees independently of the other taxa loses valuable information, and suggests that quartet-based methods for tree estimation may not provide the same level of topological accuracy, compared to methods that analyze the full dataset.

5.7 Neighbor joining

Neighbor joining (Saitou and Nei, 1987) is probably the most well known and the most widely used of all distance-based methods. We will begin with a description of how it operates, using a modification due to Studier and Keppler (1988). As we will see in later sections, neighbor joining is one of the most accurate methods in use today, and there is great interest in understanding why it works well (Steel and Gascuel, 2006).

Neighbor joining is an agglomerative technique, and so operates using iteration, building the tree from the bottom-up. The input is an $n \times n$ dissimilarity matrix $d$. In the first iteration, the $n$ leaves are all in their own clusters; in subsequent iterations, each cluster is a set of leaves, but the clusters are disjoint. At the beginning of each iteration, the taxa are partitioned into clusters, and for each cluster we have a rooted tree that is leaf-labelled by the elements in the cluster. During the iteration, a pair of clusters is selected to be made siblings; this results in the trees for the two clusters being merged into a larger rooted tree by making their roots siblings. When there are only three subtrees, then the three subtrees are merged into a tree on all the taxa by adding a new node, $r$, and making the roots of the three subtrees adjacent to $r$. Thus, neighbor joining produces an unrooted tree. So far, neighbor joining sounds like UPGMA, which we know is statistically inconsistent under some model conditions. Neighbor joining avoids this by deciding which pair of clusters to make into siblings using a more sophisticated strategy, as we now show.

The input is an $n \times n$ dissimilarity matrix, $d$, and we will refer to the leaves of the tree as “taxa”. If $n = 3$, then a star tree is computed. The algorithm repeats until the number of taxa that are left is at most 3, at which point it returns the tree produced by making the three leaves into a star tree.
5.8 Distance-based methods as functions

Step 1: Compute the $n \times n$ matrix $Q$, defined by

$$Q_{i,j} = (n - 2)d_{i,j} - \sum_{k=1}^{n} (d_{ik} + d_{jk}),$$

where $n$ is the number of taxa.

Step 2: Find the pair $i, j$ minimizing $Q_{i,j}$. Without loss of generality, we will call that pair $a, b$. Make the rooted trees associated with taxa $a$ and $b$ siblings, and call the root of the tree you form $u$.

Step 3: Update the distance matrix by deleting the rows and columns for $a$ and $b$, and including a new row and column for $u$, and set $d_{u,k} = \frac{d_{uk} + d_{uk} - d_{i,j}}{2}$ for all $k \neq u$. Decrement $n$ by 1.

Theorem 5.8 Neighbor joining runs in $O(n^3)$ time. Furthermore, given an additive matrix $D$ corresponding to a tree $T$ with positive edge weights $w(e)$, neighbor joining will return the tree $T$. In addition, if the input is a dissimilarity matrix $d$ with $L_{\infty}(d, D) < f/2$, where $f = \min \{ w(e) : e \in E_\text{I}(T) \}$ (where $E_\text{I}(T)$ denotes the set of internal edges of $T$), then neighbor joining will return $T$. Hence, neighbor joining is statistically consistent under the Cavender-Farris-Neyman (CFN) model, when used with CFN distances.

Proof The running time analysis is easily obtained from Saitou and Nei (1987), but the guarantee that neighbor joining will return the true tree given a dissimilarity matrix that is within $f/2$ (where $f$ is the shortest length of any internal edge in the true tree) of an additive matrix for the true tree comes from Atteson (1999). The proof of statistical consistency then follows since CFN distances converge to the model tree distances as the number of sites per sequence increase. Hence, as the sequence length increases, then $L_{\infty}(d, D) < f^2$ with probability converging to 1. See also Bryant (2005); Gascuel (1997) for alternative proofs for the statistical consistency of neighbor joining.

Note that the proof of statistical consistency for the CFN model can be easily extended to any model in which statistically consistent distance estimation is possible. Hence, neighbor joining is also statistically consistent under all the standard models, including the GTR and General Markov models.

5.8 Distance-based methods as functions

We have described distance-based tree estimation as operating in two steps, where the first step computes a dissimilarity matrix, and the second step uses that matrix to compute a tree. However, it is often helpful to describe distance-based methods as functions that map dissimilarity matrices to trees with positive branch lengths, and hence to additive matrices. We formalize this as follows.

Definition 5.9 A distance-based method $\Phi$ is a function that maps dissimilarity matrices to additive matrices, i.e.: the input is an $n \times n$ dissimilarity matrix $d$ and the output is an $n \times n$ additive matrix $D = \Phi(d)$. 
In order to prove theorems about statistical consistency for distance-based methods, we will generally require the following two properties:

**Identity on additive matrices**  \( \Phi \) is the identity on additive matrices (i.e., \( \Phi(D) = D \) for all additive matrices \( D \)).

**Continuous on neighborhoods of additive matrices** For all additive matrices \( D \), there is some \( L_\infty \)-neighborhood of \( D \) on which the distance-based method \( \Phi \) is continuous. That is, for all additive matrices \( D \) and any \( \delta > 0 \), there is an \( \varepsilon > 0 \) such that for all dissimilarity matrices \( d \) satisfying \( L_\infty(d, D) < \varepsilon \) it follows that \( L_\infty(\Phi(d), \Phi(D)) < \delta \).

The next two theorems show that distance-based methods that satisfy these two properties will be statistically consistent under the CFN model, and more generally under any model in which statistically consistent distance-based estimation is possible.

**Theorem 5.10** Consider an \( n \times n \) additive matrix \( D \) corresponding to a tree \( T \) with edge-weighting \( w \). Let \( f > 0 \) be the length of the shortest internal edge in \( T \), and let \( D' \) be another \( n \times n \) additive matrix such that \( L_\infty(D, D') < \frac{f}{2} \). Then \( D' \) corresponds to the same tree \( T \) with some other edge weighting \( w' \).

**Proof** Since \( L_\infty(D, D') < \frac{f}{2} \), \( D \) and \( D' \) induce the same set of quartet trees; hence they define topologically identical trees.

**Theorem 5.11** Any distance-based method \( \Phi \) that is the identity on additive matrices and continuous on an \( L_\infty \)-neighborhood of every additive matrix is statistically consistent under the CFN model.

**Proof** Suppose \( \Phi \) is continuous on a neighborhood of every additive matrix. Let \((T, w)\) be a CFN model tree (so that \( w \) is the branch length function), let \( D \) be the additive matrix corresponding to the model tree, and let \( f > 0 \) be the minimum length of any edge in the model tree \( T \). Since \( \Phi \) is continuous on some neighborhood of every additive matrix, there is some \( \varepsilon > 0 \) so that whenever matrix \( d \) satisfies \( L_\infty(d, D) < \varepsilon \) then \( L_\infty(\Phi(d), \Phi(D)) < \frac{f}{2} \). By Theorem 5.10, the additive matrix \( \Phi(d) \) defines the same tree topology as \( T \).

There is nothing special about the CFN model in this theorem; indeed, the theorem holds for any stochastic model for which the model tree topology can be defined by an additive matrix and all edges in the model tree have positive length (see Chapter 8). Furthermore, most distance-based methods have these two properties – being the identity on additive matrices, and being continuous on neighborhoods of additive matrices. Hence, most distance-based methods can be proven statistically consistent under the CFN model, and other models as well.

### 5.9 Optimization problems

What we have described so far are methods that run in polynomial time, and that are not explicitly attempting to solve any optimization problem. However, some other methods have
5.9 Optimization problems

been developed that are explicit attempts to find either optimal solutions or approximate solutions to optimization problems.

A natural optimization problem that has been suggested is to find the additive matrix \( D \) that is as close as possible to the input dissimilarity matrix \( d \), with respect to some way of defining distances between matrices, such as \( L_1 \), \( L_2 \), and \( L_\infty \):

- \( L_1(d, D) = \sum_{ij} |d_{ij} - D_{ij}| \)
- \( L_2(d, D) = \sqrt{\sum_{ij} (d_{ij} - D_{ij})^2} \)
- \( L_\infty(d, D) = \max_{ij} |d_{ij} - D_{ij}| \).

Finding an additive matrix that minimizes one of these distances is equivalent to finding an edge-weighted tree whose additive matrix has the minimum distance. Hence, these problems are equivalent to tree estimation problems.

Finding the edge-weighted tree that is as close as possible to the input dissimilarity matrix with respect to the \( L_2 \) distance is also known as the “ordinary least-squares” (OLS) problem. The “weighted least-squares” (WLS) problem is a variant on this approach, and is based on the observation that large estimated pairwise distances have higher variance than small estimated pairwise distances. In WLS, the distance function is given by \( WLS(d, D) = \sqrt{\sum_{ij} \frac{(d_{ij} - D_{ij})^2}{v_{ij}}} \), where \( v_{ij} \) is the variance of \( d_{ij} \). WLS-based approaches were originally suggested in Fitch and Margoliash (1967); Felsenstein (1997).

Yet, finding the best tree (i.e., the best additive matrix) with respect to any of these criteria is NP-hard (Day (1987) showed this for \( L_1 \) and \( L_2 \), and Agarwala et al. (1998) showed this for the \( L_\infty \) norm). Agarwala et al. (1998) also showed that finding an arbitrarily close approximation is NP-hard, but provided an \( O(n^3) \) time 3-approximation algorithm for the \( L_\infty \)-nearest tree.

Because the optimization problems are NP-hard, heuristics have been developed. These heuristics typically operate by finding good branch lengths on a given tree with respect to the desired criterion, and then searching for trees with better branch lengths (i.e., ones that produce better scores). As a result, the problem of scoring a tree is important. Consider then the following type of problem:

Input: \( n \times n \) dissimilarity matrix \( d \) and \( n \)-leaf tree \( T \)

Output: weights \( w(e) \) on the edges of \( T \) so as to produce an additive matrix \( D \) minimizing \( dist(d, D) \)

The problem depends on how \( dist(d, D) \) is defined, and could be (for example) any of the distances we have discussed so far. Fortunately, finding optimal edge weights of a tree with respect to the standard criteria can be performed in polynomial time (Desper and Gascuel, 2005). For example, for the OLS problem (ordinary least-squares), Vach (1989) described an \( O(n^3) \) algorithm, and Gascuel (2000) and Bryant and Waddell (1998) gave \( O(n^2) \) algorithms. Desper and Gascuel (2005) also provides formulae for optimal branch lengths under the OLS criterion.
5.10 Minimum Evolution

The minimum evolution approach (originally proposed in Kidd and Sgaramella-Zonta (1971)) to phylogeny encompasses a collection of methods, having the following two basic steps. Given an $n \times n$ dissimilarity matrix $d$, a collection of edge-weighted trees is produced using some method; then from this set, the tree $(T, w)$ with the minimum total branch length (where the total branch length is $\sum_{e \in E(T)} w(e)$) is returned. See Desper and Gascuel (2005) for a wonderful discussion about the mathematics involved in these problems, and the history of minimum evolution (ME) approaches.

An ME method thus requires that each tree be given with weights on the edges; while this can be done in any desired way, the most typical ways are to use one of the functions described in Chapter 5.9. For example, given a tree $T$ and an input dissimilarity matrix $d$, the optimal weights on the edges could be based on minimizing the $L_2$ (ordinary least-squares, or OLS) or the weighted $L_2$ (weighted least-squares, or WLS) distances. As discussed above, finding optimal branch lengths of a given tree can be performed exactly in polynomial time for the OLS criterion. However, finding the tree with the minimum total branch length is NP-hard under the OLS criterion (Bastkowski et al., 2016) and under a variant of the WLS criterion called “balanced minimum evolution” (Fiorini and Joret, 2012).

ME methods, therefore, tend to be heuristics without necessarily provable guarantees about solving their optimization problems. However, we can ask whether an exact solution to an ME method - however it is defined - is statistically consistent. Unsurprisingly, this depends on how the branch lengths are defined. Thus, Rzhetsky and Nei proved that ME used with OLS branch lengths is statistically consistent (if solved exactly), but Gascuel et al. (2001) showed that ME with WLS branch lengths is not statistically consistent. However, the balanced minimum evolution criterion used in the FastME algorithm is statistically consistent (Desper and Gascuel, 2004). The updated FastME software (Lefort et al., 2015) has an improved search strategy over the earlier versions, and runs in $O(n^2 \log n)$ time.

5.11 Comparing distance-based methods

In this chapter we have presented several polynomial time distance-based methods, including the Naive Quartet Method, the Buneman Tree method and its variants, Neighbor Joining, the approximation algorithm of Agarwala et al. (1998) for the $L_\infty$-nearest tree, and the FastME algorithm. Furthermore, all of these methods are statistically consistent under standard sequence evolution models. How do they compare with each other?

We begin this discussion by establishing the safety radius of each these methods, as well as for an exact algorithm for the $L_\infty$-nearest tree (see Definition 5.7), since showing that a method has a finite safety radius proves that the method is statistically consistent under a very large set of models (including the CFN model, the General Markov Model, and
5.11 Comparing distance-based methods

since all that is needed is for the model to permit statistically consistent estimation of pairwise distances.

**Theorem 5.12** The safety radius of the Naive Quartet Method, the Buneman Tree method, and the neighbor joining method is $\frac{f}{2}$, where $f$ is the shortest length of any internal edge in the model tree. Hence, all these methods are both statistically consistent under the CFN model of sequence evolution.

Proof The proof that the Naive Quartet Method has safety radius $\frac{f}{2}$ was provided earlier, and whenever the Naive Quartet Method is accurate then the Buneman Tree method (and its variants) will also be accurate. Hence, the Buneman tree method has safety radius at least $\frac{f}{2}$. The proof that the neighbor joining method has safety radius $\frac{f}{2}$ was provided in Atteson (1999). No distance-based method can have safety radius greater than $\frac{f}{2}$ (Lemma 2 in Erdos et al. (1999c)). Hence, all these methods have safety radius exactly $\frac{f}{2}$, and are statistically consistent under the CFN model of sequence evolution.

**Theorem 5.13** An exact algorithm for the $L_\infty$-nearest tree problem has a safety radius of $\frac{f}{4}$, where $f$ is the length of the shortest internal edge in the model tree. Hence, an exact algorithm for the $L_\infty$-nearest tree problem is statistically consistent under the CFN model of sequence evolution.

Proof Suppose $\Phi$ is an exact algorithm for the $L_\infty$-nearest tree problem. Suppose that $d$ is an $n \times n$ dissimilarity matrix, that $D$ is an $n \times n$ additive matrix corresponding to the model tree $(T, w)$, and that $L_\infty(d, D) < \frac{f}{4}$, where $f > 0$ is the minimum length of any internal edge in $T$. Then $\Phi(d) = D'$ is an additive matrix corresponding to $(T, w')$, for some edge-weighting $w'$. Because $\Phi$ is an exact algorithm for the $L_\infty$-nearest tree problem, $L_\infty(D', d) \leq L_\infty(d, D) < \frac{f}{4}$. By the triangle inequality,

$$L_\infty(D', D) \leq L_\infty(D', d) + L_\infty(d, D).$$

Therefore,

$$L_\infty(D, D') \leq \frac{f}{4} + \frac{f}{4} = \frac{f}{2}.$$

By Theorem 5.10, $D$ and $D'$ correspond to the same tree topology. Hence, given $D'$ we can construct the model tree topology $T$, and so $\Phi$ has a safety radius of at least $\frac{f}{4}$. To see that the safety radius cannot be greater than $\frac{f}{4}$, it suffices to note that for every additive matrix $A$ corresponding to a binary tree in which the shortest internal branch has length $f$ and for all $\varepsilon > 0$ there is another additive matrix $A'$ defining a different tree topology satisfying $L_\infty(A, A') \leq \frac{f}{4} + \varepsilon$ (Lemma 2 in Erdos et al. (1999c)).

**Theorem 5.14** The polynomial time 3-approximation algorithm provided in Agarwala et al. (1998) for the $L_\infty$-nearest tree has a safety radius of $\frac{f}{8}$ where $f$ is the minimum length of any internal edge in the model tree. Hence, the 3-approximation algorithm of Agarwala et al. (1998) for the $L_\infty$-nearest tree problem is statistically consistent under the CFN model of sequence evolution.
The proof follows along the same lines as that for Theorem 5.13, and is omitted.

The prior theorems have established that the Naive Quartet method, the Buneman Tree, neighbor joining, an exact algorithm for the $L_\infty$-nearest tree problem, and the 3-approximation algorithm by Agarwala et al. (1998) all have finite safety radii that can be expressed in terms of the shortest internal branch length in the model tree. Can we use this information to infer anything about their performance in practice?

We begin with neighbor joining, the Buneman Tree, and the Naive Quartet Method. By Theorem 5.12, these three methods have the same safety radius; yet these methods perform very differently in practice. For example, the requirement that all quartet trees be correctly computed means that the Naive Quartet Method generally returns Failure, except on very small datasets or on datasets with extremely long sequences. Similarly, while the Buneman Tree is basically a better method than the Naive Quartet Method, it tends to produce highly unresolved trees, especially on large datasets (St. John et al., 2003). In contrast, the neighbor joining method generally produces more accurate trees (i.e., ones with lower missing branch rates) than the Buneman Tree, and so is the best of these three methods (St. John et al., 2003; Huson et al., 1999a).

What about the other methods? According to Theorem 5.14, the safety radius of the Agarwala et al. (1998) algorithm is one fourth that of the Naive Quartet Method, the Buneman Tree Method, and neighbor joining, and so the Naive Quartet Method ought to be more accurate than all three of these methods. Yet, Huson et al. (1999a) compared the Agarwala et al. (1998) algorithm to neighbor joining and the Buneman Tree on simulated datasets and found that neighbor joining had the best accuracy, followed by Agarwala et al. (1998), and finally by the Buneman Tree.

In other words, a finite safety radius enables statistical consistency guarantees to be established, but does not directly imply how accurate a method will be on data. Similarly, comparing the safety radii of two methods does not directly imply anything about the relative performance on data between the methods. In particular, the safety radius indicates how close the dissimilarity matrix has to be to the model tree additive matrix in order for the method to produce the true tree without any error, and performance on data inevitably is about partial accuracy rather than exact accuracy.

Studies comparing FastME to neighbor joining have shown that FastME often produces topologically more accurate trees, and under a range of different scenarios (Desper and Gascuel, 2004; Sy Vinh and von Haeseler, 2005; Wang et al., 2006; Vachaspati and Warnow, 2015), suggesting both that the balanced minimum evolution criterion is a helpful one and that the search strategy in the FastME software is effective.

It is therefore interesting to consider the inquiry into why neighbor joining works well. While we have described neighbor joining method as an iterative heuristic that begins with $n$ disconnected leaves and then gradually connects them all, neighbor joining can also be described as a method that begins with a star tree, and then greedily refines around the polytomies, gradually turning a star tree into a binary tree. Steel and Gascuel (2006) used this formulation to show that neighbor joining can be seen as always selecting the next pair of nodes to make siblings on the basis of a balanced minimum evolution principle.
Other studies also show a connection between neighbor joining and the balanced minimum evolution principle (Haws et al., 2011; Mihaescu and Pachter, 2008). The connection to the balanced minimum evolution principle may help explain why neighbor joining does as well as it does in comparison to many other distance-based methods. However, neighbor joining is not as accurate as FastME. The explanation may be that because neighbor joining uses a greedy heuristic, it cannot undo any steps it makes, while FastME does do a heuristic search to find improved solutions.

5.12 Further reading

In this chapter, we have described distance-based tree estimation as having an input set of sequences that evolved down an unknown model tree so that the sequences are all the same length, computing a distance matrix for the sequences, and then computing a tree from the sequences. This is the simplest version of distance-based tree estimation. However, distance-based tree estimation occurs in other contexts, and we briefly describe these here.

First, biological sequence datasets typically have sequences with different lengths, as the result of processes such as insertions and deletions (jointly called “indels”). When sequences have different lengths, the first step in a phylogeny estimation is the calculation of the “multiple sequence alignment”, discussed in Chapter 9. Because calculating multiple sequence alignments is a computationally difficult problem, approaches to constructing trees that do not require the multiple sequence alignment be computed have also been developed. These “alignment-free” methods typically operate by computing pairwise distances between sequences, often based on the distribution of substrings of a fixed length within each sequence (Vinga and Almeida, 2003; Chan and Ragan, 2013; Chan et al., 2014). However, most of the alignment-free methods do not have statistical guarantees under sequence evolution models that include indels. However, Daskalakis and Roch (2010) provide an interesting method that they prove is statistically consistent for estimating the tree topology under the Thorne-Kishino-Felsenstein 1991 model (also known as the TKF91 model) proposed in Thorne et al. (1991), in which sequence evolution includes indels as well as substitutions.

Distance-based tree estimation is sometimes used for estimating species trees when the input is a set of trees, each estimated for a potentially different subset of the species. One of the ways this happens is when the input is a set of trees estimated for different genes (i.e., gene trees), and where the taxon set of each gene can be different. Because gene trees can be different from the species tree, due to biological processes such as incomplete lineage sorting (ILS), methods that take gene tree heterogeneity into account have been developed (see Chapter 10). One of the types of statistically consistent methods operates by computing pairwise distances between species based on the set of gene trees, and then applies standard methods, such as neighbor joining or FastME, to the resultant matrix.

However, in this context, it can happen that some pair of species $s_i$ and $s_j$ are never in the same gene tree, and when this happens then the distance matrix will have no value for the distance between $s_i$ and $s_j$. When the distance matrix has missing entries, then standard
distance-based methods, such as neighbor joining and FastME, cannot be applied. Computing trees from matrices with missing entries is very difficult, and only a few methods have been developed for this problem (Criscuolo and Gascuel, 2008). Supertree methods (see Chapter 7), which construct trees by combining trees on subsets, also sometimes are based on distances, and also face this kind of challenge. This is one of the interesting and basically unsolved problems in phylogenetics.

While many distance-based methods are easily shown to be statistically consistent under some models of evolution, it is not as easy to establish the convergence rate, or the sequence length that suffices (or is required) for accuracy with high probability. In particular, given arbitrary upper and lower bounds on the branch lengths, we may wish to know if the method will recover the true tree with high probability from polynomial length sequences, or whether exponential length sequences are needed. Methods that can recover the true tree with high probability from polynomial length sequences are called “absolute fast converging” (afc) methods, and are discussed in Chapter 13.1. See also Erdos et al. (1999b,c); Warnow et al. (2001); Nakhleh et al. (2001); Daskalakis et al. (2006); Roch (2008, 2010) for some afc distance-based methods.

5.13 Review questions

1. What is the triangle inequality?
2. What is a dissimilarity matrix?
3. What are p-distances?
4. What is the Cavender-Farris distance formula?
5. What is meant by saying that a matrix is additive?
6. What is the Four Point Condition?
7. What is the Four Point Method?
8. What is the Naive Quartet Method?
9. What is the technique for computing the length of the internal branch in the quartet tree on four taxa, given an additive matrix for the four taxa?
10. Are Hamming distances computed on sequences necessarily additive?
11. What is the meaning of safety radius?
12. Suppose \( d \) and \( D \) are two \( n \times n \) dissimilarity matrices, and that \( D \) is additive. Give a condition on \( L_{\infty}(d,D) \) under which we are guaranteed that \( NJ(d) \) and \( NJ(D) \) define the same tree topology.
13. State three distance-based optimization problems and their computational complexity (i.e., are they NP-hard, solvable in polynomial time, or of unknown computational complexity).
14. Agarwala et al. presented an approximation algorithm for some optimization problem related to distance-based phylogeny estimation. What problem was that? Is their approximation algorithm statistically consistent under the CFN model? What about the Jukes-Cantor model?
15. What is meant by “alignment-free” tree estimation?

5.14 Homework problems

1. Compute the CFN distance matrix between all pairs sequences in the set
   • $s_1 = 0011010111$
   • $s_2 = 0011000111$
   • $s_3 = 0011111111$

2. Consider the set $\{0, 1, 2, \ldots, 15\}$ and represent them as binary numbers with ten digits (i.e., $8 = 00001000$). Treat these as binary sequences, generated by some CFN model tree. What is the largest CFN distance between any two binary sequences in this set? (Hint: do not compute all pairwise distances.)

3. What is the largest CFN distance possible between two binary sequences $s$ and $s'$ of the same length $k$, under the constraint that they differ in strictly less than $\frac{k}{2}$ positions?

4. Prove or disprove: For all pairs of binary sequences $s, s'$ of the same length $k$ that differ in strictly less than $\frac{k}{2}$ positions, the CFN distance between $s$ and $s'$ is at least the Hamming distance between $s$ and $s'$.

5. Prove or disprove the following statement: For all datasets of binary sequences of the same length $k$ such that all pairs of sequences in the set differ in strictly less than $\frac{k}{2}$ positions, the CFN distance matrix will satisfy the triangle inequality.

6. In the problems preceding this, we have constrained the set of binary sequences of length $k$ to differ pairwise in strictly less than $\frac{k}{2}$ positions. Why did we do this?

7. Consider the matrix in Figure 5.3 from the text. Apply UPGMA to the matrix. What is the unrooted tree that you obtain? Does it equal the tree given in that figure?

8. Let $T$ be a CFN model species tree on four leaves $A$, $B$, $C$, and $D$, with unrooted topology $AB|CD$. Let $\lambda(e)$ denote the CFN branch length of edge $e$. Let $e_I$ denote the single internal edge in $T$, and let $e_x$ denote the edge incident with leaf $x$. Assume that $\lambda(e_I) = 2, \lambda(e_A) = 0.1, \lambda(e_B) = 0.2, \lambda(e_C) = 2.1$ and $\lambda(e_D) = 3.2$.
   1. Prove or disprove: UPGMA on Hamming distances is statistically consistent for estimating the unrooted tree topology for $T$
   2. Prove or disprove: UPGMA on CFN distances is statistically consistent for estimating the unrooted tree topology for $T$

9. Draw an edge-weighted tree $T$ with at least five leaves and all branches having positive weight. Derive its additive matrix. Check that the four point condition applies for at least two different quartets of leaves.

10. Compute the Hamming distance matrix for the set of four taxa, $L = \{L_1, L_2, L_3, L_4\}$, given below (each described by four binary characters). Is the distance matrix additive? If you apply the UPGMA method to this distance matrix, what do you get? If you apply the Four-Point Method to the matrix, what do you get? What is the solution to maximum parsimony on this input of four taxa? What is the solution to maximum compatibility? Are these characters compatible?
• $L_1 = (0, 1, 0, 1, 0)$
• $L_2 = (0, 0, 0, 0, 0)$
• $L_3 = (1, 0, 0, 0, 0)$
• $L_4 = (1, 0, 1, 0, 1)$

11. Let $D$ be an additive matrix corresponding to binary tree $T$ with positive edge-weighting $w$. Let $D'$ be the matrix corresponding to the edge-weighted tree obtained by changing the edge of some internal edge $e$ to 0. Does $D'$ still satisfy the four-point condition?

12. Consider those matrices that correspond to path distances in edge-weighted trees $T$ with positive branch lengths, but where $T$ may not be binary (i.e., the trees can have polytomies). Prove or disprove: matrices computed this way satisfy the four-point condition.

13. Prove or disprove: If $C$ is a set of characters (not necessarily binary) that evolve without any homoplasy on a tree $T$, then the Hamming distance matrix is additive.

14. Suppose $D$ and $D'$ are two additive matrices, both corresponding to the same tree topology, but using (perhaps) two different edge weightings. Prove or disprove: for all constants $c > 0$, $cD + D'$ is also additive.

15. Give an example of two $n \times n$ additive matrices $D$ and $D'$, but where $D$ corresponds to an edge-weighted binary tree $T$ with positive branch weights, and $D'$ is additive and so corresponds to a unique tree $T'$ (which may not be binary) with positive branch lengths where $T \neq T'$, and where $L_{\infty}(D, D') = f/2$, where $f$ is the minimum length of any internal edge in $T$.

16. Give a $\Theta(n^2)$ algorithm to compute the $n \times n$ additive distance matrix defined by a tree on $n$ leaves with positive weights on the edges. Hint: use dynamic programming.

17. Suppose $l_1, l_2, \ldots, l_k$ are non-negative integers, and $M_{ij}$ is a matrix defined by $M_{ij} = 0$ if $i = j$ and otherwise $M_{ij} = l_i + l_j$. Is $M$ additive? Prove or disprove.

18. Recall the definition of an additive matrix and the Four Point Method. Is it ever possible for some set of four taxa that the two smallest of the three pairwise sums will be identical but strictly smaller than the largest pairwise sum, given an additive matrix? If so, give an example, and otherwise prove this cannot happen.

19. Suppose that branch lengths of a tree can be negative. Are there any constraints you can infer about the three pairwise sums in this case?

20. Give a $\Theta(n^4)$ algorithm to compute the set $Q(T)$, where $T$ is tree on $n$ leaves with positive edge weights on the edges.

21. Give an $O(n^2 + k)$ algorithm to solve the following problem. The input is a tree $T$ with positive edge weights, and a list $L$ of $k$ four-leaf subsets, and the output is the list of quartet tree topologies on every four-leaf subset in $L$. Hint: use $O(n^2)$ time for the preprocessing to enable each four-leaf subset to be answered in $O(1)$ time.

22. Same as the previous problem, but change the output so that quartet trees have weights on all the edges defined by the additive matrix corresponding to the input edge-weighted tree.

23. Let $D$ be an $n \times n$ additive matrix corresponding to binary tree $T$ and positive edge-weighting $w$ with $f$ the minimum length of any internal edge. Let $D'$ be an $n \times n$ additive
5.14 Homework problems

matrix such that $L_{\infty}(D, D') > f/2$. Is it possible for $D$ and $D'$ to define the same tree topology? If so, give an example of such a pair of matrices $D$ and $D'$, or else prove this is impossible.

24. For the additive matrix you produced in the previous problem, compute the tree for every quartet of taxa by applying the Four Point Method. Then apply the Naive Quartet method to the set of quartets. Verify that you produce the same tree.

25. Take any additive matrix and change one entry. Determine if the new matrix is additive. If not, prove it is not by producing the four leaves for which the four-point condition fails. If yes, prove that is by producing the edge-weighted tree that realizes the new matrix.

26. Prove that every additive matrix uniquely defines a unique tree $T$ (not necessarily binary) with positive branch lengths.

27. Show how to modify the Naive Quartet Method to construct the unique tree corresponding to an edge-weighted (but not necessarily binary) tree with positive branch lengths.

28. Suppose you have an additive matrix $M$ but some of the entries are missing (or, equivalently, replaced by "?"). Your task is to see if you can figure out the missing entries. Consider the special case where there is only one pair $i, j$ for which $M_{i,j}$ and $M_{j,i}$ are both missing. Give an algorithm to infer $M_{i,j}$ from the remaining data, and prove it correct.

29. Consider the Naive Quartet Method applied to pairwise Hamming distances, and call this the NQM(Hamming) method. For binary characters, what characters are uninformative for the NQM(Hamming) method?

30. Write down an edge-weighted (with positive weights) tree with five leaves, and compute the additive matrix for the tree. Apply neighbor joining (by hand) to the matrix, and compare the tree you get to the tree you started with.

31. Take the additive matrix you computed for the previous problem, and compute $f$, the minimum length of any internal edge in the tree. Add or subtract something less than $f/2$ from each entry to produce another matrix. Run neighbor joining on the new matrix. Do you get the same tree topology?

32. Consider the function $\Phi$ that maps dissimilarity matrices $d$ to $2d$ (i.e., $\Phi(d)$ is the matrix $d'$ such that $d'_{i,j} = 2d_{i,j}$).

- Does $\Phi$ map dissimilarity matrices to additive matrices?
- How could you use $\Phi$ to estimate the tree topology?

33. Suppose we know that $\Phi_1$ and $\Phi_2$ are both distance-based methods that satisfy the identity property on additive matrices and are continuous on a neighborhood of every additive matrix. Does the composition of these two methods also satisfy both properties?

34. Suppose $\Phi$ maps dissimilarity matrices $d$ to $2D$, where $D$ is the additive matrix that is closest to $d$ under the $L_{\infty}$ metric. Does $\Phi$ satisfy one, both, or neither of the two desired properties (identity on additive matrices, and continuous on a neighborhood of every
additive matrix)? If we use CFN distances, is $\Phi$ a statistically consistent technique for estimating CFN trees?


36. Suppose $\Phi$ is an exact solution to the $L_1$-nearest tree problem, where the input is a dissimilarity matrix $d$ and the output is an additive matrix $D$ that minimizes $L_1(d, D) = \sum_{ij} |d_{ij} - D_{ij}|$. Is $\Phi$ a statistically consistent method for estimating CFN model trees, if applied to CFN distances? If so, why? If not, why not?

37. Suppose $\Phi$ is a log-$n$-approximation algorithm to the $L_1$-nearest tree problem (see previous problem). Is $\Phi$ a statistically consistent method for estimating CFN model trees, if applied to CFN distances? If so, why? If not, why not?

38. Prove the following: If $C$ is a set of binary characters that evolve without homoplasy on a tree $T$, then the Hamming distance matrix is additive.

39. Consider sequences that evolve down a tree $T$ with $n$ leaves, and let $\text{length}(e)$ denote the number of changes that occur on edge $e$. Consider the $n \times n$ matrix $\text{LENGTH}$ defined by this way of defining branch lengths. Prove that the matrix is additive.

40. An edge of a tree on which no changes occur (so that the sequences at the endpoints of the edge are identical) is called a “zero-event” edge. How do zero-event edges impact phylogeny estimation? Can zero-event edges be recovered?

41. Consider the case where a set $S$ of binary sequences evolve down a binary tree $T$ without any homoplasy.

- Prove or disprove: $T$ is an optimal solution to maximum parsimony for this dataset.
- Prove or disprove: $T$ is an optimal solution to maximum compatibility for this dataset.
- What is the relationship between $T$ and the strict consensus of the set of all maximum parsimony trees?
- What is the relationship between $T$ and the strict consensus of the set of all maximum compatibility trees?

42. Consider the case where a set $S$ of binary sequences evolve down a binary tree $T$ without any homoplasy, and there are no zero-event edges in $T$ with respect to $S$; hence, every edge has at least one change on it.

- Give an exact characterization to the set of maximum parsimony trees on the set $S$.
- Give an exact characterization to the set of maximum compatibility trees on the set $S$.
- What is the result of applying the Naive Quartet Method to the Hamming distance matrix for $S$?

43. Consider the case where a set $S$ of binary sequences evolve down a binary tree $T$ without any homoplasy, but the tree $T$ does have zero-event edges with respect to $S$. Let $E_0 \subseteq E(T)$ denote the zero-event edges in $T$. Consider $T_0$, the tree obtained by collapsing all the zero-event edges in $T$.

- Give an exact characterization to the set of maximum parsimony trees on the set $S$. 

• Give an exact characterization to the set of maximum compatibility trees on the set $S$.
• What is the result of applying the Naive Quartet Method to the Hamming distance matrix for $S$?

44. Suppose that you have a set $S$ of aligned binary sequences, each of length 100, that you are told have evolved down some unknown CFN model tree. You would like to compute a distance-based tree on $S$, and so you try to compute the CFN distance matrix. However, when you do this, you find a pair of sequences $x, y$ which differ in 50 or more positions (out of the total 100). What will you do? When will this kind of problem happen? Would you suggest changing the CFN distance calculation to handle such inputs, or would you suggest not using distance-based tree estimation methods? What, in general, would you consider doing to construct a tree under such circumstances? (This is a research question - there are many possible reasonable answers.)
6
Consensus and Agreement Trees

6.1 Introduction

In this chapter we discuss techniques for analyzing sets of trees. Depending on how the set of trees was computed and the objective in the analysis, different techniques are needed.

For example, when a maximum parsimony analysis is performed, many equally good trees may be found, all having the same “best” score. Similarly, when a Bayesian MCMC (see Chapter 8.7) analysis is performed, then a random sample of the trees is examined. Sometimes, many different types of methods are used on the same data, and for each analysis a set of trees is obtained. Then, from the full set of trees, each of which has been estimated on the same data, again some kind of point estimate is sought. In each of these cases, a “consensus method” is used to provide a point estimate of the tree from the full set of trees.

An alternative objective might be to find those subsets of the taxon set on which the trees in the profile all agree; this kind of approach does not produce a point estimate of the true tree, but can be used to identify the portion of the history on which all trees agree, and also (potentially) problematic taxa. We refer to this type of approach as an “agreement method”.

By definition, agreement methods and consensus methods require that the trees in the profile each have exactly the same set of leaves. When this condition does not hold, then a different type of method must be used. For example, sometimes the trees in the profile are each an estimate of a gene tree, rather than an estimate of the species tree. Because gene trees can be different from the species tree (and hence from each other) due to biological processes such as gene duplication and loss, horizontal gene transfer, or incomplete lineage sorting (Maddison, 1997), the trees in the profile may not be estimates of the same underlying species tree. When this happens, the use of consensus methods or supertree methods can be misleading, because these do not take biological causes for gene tree discord into account. On the other hand, a wide variety of methods, some of which are quite simple, can be quite accurate and have theoretical guarantees of converging to the true species tree, even in the presence of gene tree heterogeneity.

In this chapter, we discuss consensus and agreement methods, which are methods for analyzing datasets of trees under the assumption that all the trees are estimated species trees with the same leafset $S$. We discuss supertree methods in Chapter 7, which extend
6.2 Consensus trees

When two or more trees are given on the same leaf set, we may also be interested in computing *consensus trees*. In general, these consensus methods are applied to unrooted trees (and we will define them in that context), but they can be modified so as to be applicable to rooted trees as well. Here, we will focus on the ones that are the most frequently used in practice: the *strict consensus*, *majority consensus*, and the *greedy consensus*. However, we also present two other consensus methods that have interesting properties. The first is the *compatibility tree*, which only exists when all the trees in the input are compatible, and the second is the Asymmetric Median Tree (Phillips and Warnow, 1996). We will assume that the input is a set $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ of unrooted trees, and that each tree $T_i$ has leafset $S$.

6.2.1 Strict consensus

To construct the strict consensus, we write down the bipartitions that appear in *every* tree in the input set $\mathcal{T}$. The tree that has exactly that set of bipartitions is the "strict consensus". Note that the strict consensus is unique, and is a contraction of every tree in the input.

**Definition 6.1** Given a set $\{T_2, T_2, \ldots, T_k\}$ of unrooted trees, each on the same leafset, the *strict consensus* tree $T$ is the tree that contains exactly the bipartitions that appear in all the trees. Therefore, $C(T) = \cap_{i=1}^k C(T_i)$.

By definition, the strict consensus tree will not be fully resolved except when all the trees in the profile are topologically identical.

6.2.2 Majority consensus

To construct the majority consensus, we write down the bipartitions that appear in *more than half* the trees in the profile. The tree that has exactly those bipartitions is called the "majority consensus". It is interesting to note that the majority consensus tree $T_{maj}$ is the tree $T$ that maximizes $\Sigma_{i=1}^k |C(T) \cap C(T_i)|$, and equivalently is the tree $T$ that minimizes the total Robinson-Foulds distance to the input trees, i.e., $\Sigma_{i=1}^k RF(T, T_i)$. For this reason, the majority consensus tree is also called a *median tree* (Barthélemy and McMorris, 1986).

**Definition 6.2** Given a set $\{T_2, T_2, \ldots, T_k\}$ of unrooted trees, each on the same leafset, the *majority consensus* tree $T$ is the tree that contains exactly the bipartitions that appear in more than half of the trees.
Observation 6.3 The majority consensus tree is unique, and is either identical to the strict consensus tree or refines it.

Note that the majority consensus tree may not be fully resolved; however, unlike the strict consensus tree, the majority consensus tree can be fully resolved even when the trees in the profile are different from each other.

It is easy to see that the majority consensus tree can be computed in polynomial time. A randomized linear time algorithm to compute the majority consensus tree was developed in Amenta et al. (2003).

6.2.3 Greedy consensus tree

We now define the greedy consensus tree, by showing how to compute it. To construct the greedy consensus, we order the bipartitions by the frequency with which they appear in the profile. We then start with the majority consensus, and then "add" bipartitions (if we can), one by one, to the tree we’ve computed so far.

When we attempt to add a bipartition \(A|B\) to a tree \(T\), we are asking whether we can find a refinement of \(T\) that contains the bipartition \(A|B\). If \(T\) already contains this bipartition the answer is "Yes", and we do not need to change \(T\) at all. If \(T\) does not contain the bipartition, then we are trying to modify \(T\) (by finding the refinement, if it exists) that contains all the bipartitions in \(T\) and then exactly one more – \(A|B\). Since it may not be possible to add \(A|B\) to \(T\), this refinement may not exist. However, when it exists, the unique minimal refinement containing \(A|B\) can be constructed in \(O(n)\) time (Gusfield, 1991b).

We stop either when we construct a fully resolved tree (because in that case no additional bipartitions can be added), or because we finish examining the entire list. Note that the order in which we list the bipartitions will determine the greedy consensus – so that this particular consensus is not uniquely defined for a given profile of trees (we give such an example below). On the other hand, the strict consensus and majority consensus do not depend upon the ordering, and are uniquely defined by the profile of trees.

Observation 6.4 The greedy consensus is either equal to the majority consensus or it refines it, since it has every bipartition that appears in the majority consensus. Therefore, the greedy consensus is also called the extended majority consensus.

While the greedy consensus tree will be at least as resolved as the majority consensus, it may also not be fully resolved.

6.2.4 Compatibility trees

Recall that a tree \(T\) refines another tree \(t\) if \(t\) can be obtained from \(T\) by collapsing some of the edges in \(t\). This definition extends to a set \(\mathcal{T}\) of trees, by saying that \(T\) is a common refinement of the trees in \(\mathcal{T}\) if \(T\) refines every tree \(t \in \mathcal{T}\).
Definition 6.5 A set $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ of trees, all with the same leafset, is compatible if there is a tree that refines every tree $T_i$, $i = 1, 2, \ldots, k$. When $\mathcal{T}$ is compatible, then it has a unique minimal common refinement $T$, and we refer to $T$ as the compatibility tree for $\mathcal{T}$.

We will often be interested in determining whether a set $\mathcal{T}$ of trees is compatible, and in computing a common refinement of the set of trees when they are.

Example 6.6 As an example, consider the trees $T_1$ and $T_2$, defined as follows:

- $T_1$ given by $C(T_1) = \{(abc|de\ f\ g)\}$, and shown in Figure 6.6(a)
- $T_2$ given by $C(T_2) = \{(abcd|e\ f\ g), (abcede|fg)\}$, and shown in Figure 6.6(b)

We can see they are compatible, because $T_3$, shown in Figure 6.6(c), is a common refinement of each of the trees. Since they are compatible, there should be a minimal common refinement, and we would like to construct it. Note that $C(T_1) \cup C(T_2)$ must be a compatible set of bipartitions, since otherwise there would not be any common refinement. Let $T$ be the tree satisfying $C(T) = C(T_1) \cup C(T_2)$. Note that $T$ is a common refinement of $T_1$ and $T_2$, and that every tree that is a common refinement of $T_1$ and $T_2$ must refine $T$. Hence, $T$ is the minimal common refinement of $T_1$ and $T_2$ and is the compatibility tree.

Theorem 6.7 If a set $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ of trees, where each $T_i$ is leaf-labelled by the same set $S$ of taxa, has a compatibility tree $T$, then $C(T) = \cup_i C(T_i)$. Hence, a set $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ of trees on the same leaf set is compatible if and only if the set $\cup_i C(T_i)$ is compatible.

More generally, to see if a set $\mathcal{T}$ of trees is compatible, we write down their bipartition sets, and then we apply the algorithm for constructing trees from bipartitions from Chapter 2.3.5 to the union of these sets. If $\mathcal{T}$ is compatible, the output will be a tree $T$ that is the compatibility tree; otherwise, the algorithm will fail to return a tree, which will indicate that $\mathcal{T}$ is not compatible.

6.2.5 Asymmetric Median Tree

The asymmetric median tree was proposed in Phillips and Warnow (1996) to address the inability of the median tree approach to construct a compatibility tree when the input set of trees is compatible. For example, consider $T_1$, $T_2$, and $T_3$ on leafset $a, b, c, d, e, f$, each having only one internal edge. Thus, each tree is defined by its single non-trivial bipartition, as follows: $T_1$’s nontrivial bipartition is $ab|\text{def}$, $T_2$’s nontrivial bipartition is $abc|\text{def}$, and $T_3$’s nontrivial bipartition is $abcd|\text{ef}$. The three trees are compatible since $T = (a, (b, (c, (d, (e, f))))))$ is a common refinement of $T_1$, $T_2$ and $T_3$. However, the median tree, which minimizes the total Robinson-Foulds distance to the three trees, is the star tree (i.e., it has no internal edges), and has total distance of 3. All other trees have a larger total Robinson-Foulds distance, and the compatibility tree has a total distance of 6.

The reason that a compatibility tree is not a median tree is that the Robinson-Foulds
Consensus and Agreement Trees

(a) Tree $T_1$

(b) Tree $T_2$

(c) Tree $T_3$

Figure 6.1 Three trees on the same leafset; tree $T_3$ is a common refinement of trees $T_1$ and $T_2$, and so demonstrates that $T_1$ and $T_2$ are compatible.

distance between the consensus tree and the source trees penalizes edges that are in the consensus tree and that appear in some but not all source trees. Thus, even if a source tree is compatible with all the other source trees, any bipartitions in that source tree that are not in strictly more than half the source trees cannot be in the compatibility tree.

The asymmetric median tree was formulated in Phillips and Warnow (1996), in an attempt to address this weakness of the majority consensus problem. Instead of penalizing the edges in the consensus tree, it only counts edges in the source tree that do not appear in
6.2 Consensus trees

Precisely, the asymmetric median tree is the tree \( T \) on the same leafset as the source trees that minimizes \( \sum_{i=1}^{k} |C(T_i) \setminus C(T)| \). Thus, if the set \( \mathcal{T} \) is compatible, then the asymmetric median tree is the compatibility tree.

The asymmetric median tree of two trees can be computed in polynomial time, but computing the asymmetric median tree of three or more trees is NP-hard (Phillips and Warnow, 1996). In contrast, the median tree (also called the majority tree) can be computed in polynomial time. Thus, although the asymmetric median tree has some theoretical advantages over the median tree, it is computationally more intensive to compute.

6.2.6 Comparisons between consensus trees

We now explore the differences between these consensus trees.

**Example 6.8** We give three different trees on the same leaf set, defined by the non-trivial bipartitions for each tree:

- \( T_1 \) given by \( C_I(T_1) = \{ (12|3456), (123|456), (1234|56) \} \)
- \( T_2 \) given by \( C_I(T_2) = \{ (12|3456), (123|456), (1235|46) \} \)
- \( T_3 \) given by \( C_I(T_3) = \{ (12|3456), (126|345), (1236|45) \} \)

The bipartitions are:

- \((12|3456)\), which appears three times
- \((123|456)\), which appears twice
- \((1234|56)\), which appears once
- \((1235|46)\), which appears once
- \((1236|45)\), which appears once
- \((126|345)\), which appears once

Using the definition of the strict consensus tree, we see that the strict consensus has only one bipartition, \((12|3456)\), and that the majority consensus has two bipartitions – \((123|456)\) and \((12|3456)\). Note that the greedy consensus tree depends upon the ordering of the remaining four bipartitions (since all appear exactly once, they can be ordered arbitrarily), and so there can be more than one greedy consensus tree. In fact, there are 24=4! possible orderings of these bipartitions! However, we will only show the results for three of these.

- **Ordering 1:** \((1234|56), (1235|46), (1236|45), (126|345)\). For this ordering, we see that we can add \((1234|56)\) to the set we have so far, \((12|3456), (123|456)\), to obtain a fully resolved tree. Note that this is equal to \( T_1 \).
- **Ordering 2:** \((126|345), (1236|45), (1234|56), (1235|46)\). For this ordering we see that we cannot add the bipartition \((126|345)\) to the set we have so far. However, we can add \((1236|45)\), to obtain a fully resolved tree. This final tree is given by \((1, (2, (3, (6, (4, 5)))))\). This is not among the trees in the input.
Consensus and Agreement Trees

- Ordering 3: $(126|345), (1235|46), (1234|56), (1236|45)$. For this ordering, we cannot add $(126|345)$, but we can add the next bipartition, $(1235|46)$. When we add this, we obtain a fully resolved tree that is equal to $T_2$.

**Example 6.9** We now consider how the asymmetric median tree and the majority tree can differ. Suppose the input set $\mathcal{T} = \{t_1, t_2, t_3\}$, each on leaf set $S = \{1, 2, 3, 4, 5, 6\}$, and that these trees are given by the following bipartitions:

- $C(T_1) = \{(123|456)\}$
- $C(T_2) = \{(12|3456)\}$
- $C(T_3) = \{(1236|45)\}$

The majority consensus of these three trees would be a star tree (i.e., the tree without any internal edges), since there is no bipartition that appears in the strict majority of these trees. However, the three trees are compatible, since their compatibility tree is $((1, (2, (3, (6, (4, 5))))))$. Furthermore, the compatibility tree, when it exists, is always the optimal solution to the asymmetric median tree. Hence, the asymmetric median tree on this input is a fully resolved tree, while the majority consensus tree is a completely unresolved tree! Recall that the majority consensus tree is a median tree, using the Robinson-Foulds distance to measure the distance between two trees. Hence, the median tree and the asymmetric median tree can be very different.

### 6.3 Agreement subtrees

Agreement subtrees are trees on a subset of the shared taxon set that represent the shared features in the profile. The two main techniques of this type are the **maximum agreement subtree (MAST)** and the **maximum compatibility subtree (MCST)**.

#### 6.3.1 Maximum Agreement Subtree

The maximum agreement subtree (MAST) problem is as follows. Given a set $\mathcal{T}$ of input trees (which may be rooted or unrooted), the objective is the largest set $X$ of the leafset $S$ so that the trees in $\mathcal{T}$ agree on $X$ (i.e., they all induce the same tree on $X$).

The MAST problem was posed in Finden and Gordon (1995). The computational complexity of the MAST problem depends on the number of trees and their maximum degree. The first polynomial time algorithm to compute the MAST of two trees (whether rooted or not) was presented in Steel and Warnow (1993), which included an $O(n^2)$ algorithm to compute the MAST of two bounded degree rooted trees. Subsequently, an $O(n^{1.5} \log n)$ algorithm for computing the MAST of two unbounded-degree trees was found (Farach and Thorup, 1994). Polynomial time algorithms for three or more bounded-degree trees were presented in Amir and Keselman (1994), along with an NP-hardness proof for MAST on three unbounded degree rooted trees.
6.3.2 Maximum Compatibility Subtree

The objective of the maximum compatibility subtree (MCST) problem (also known as the maximum refinement subtree problem (Hein et al., 1996)) is to find the largest subset $X$ of the leaf set $S$ so that the input trees are compatible when restricted to $X$. Thus, the MCST problem is similar to the MAST problem. The motivation for the problem is that many estimated trees have low support branches (edges), and when these branches are collapsed the estimated trees have polytomies. These polytomies are considered “soft” polytomies, because what they indicate is uncertainty about the phylogeny as opposed to a statement that the evolution occurred with a multi-way speciation event. Therefore, if the input source trees are modified by collapsing low support branches, the objective is to find a subset of the taxa on which all the trees have a common refinement.

It is trivial to see that when all the trees in the input are binary, then the MAST of the set is identical to the MCST of the set. However, when the input trees can contain polytomies, then these two trees can be different. Furthermore, the number of leaves in the MCST is always at least as big as the number of leaves in the MAST. The MCST problem was introduced in Hein et al. (1996), where it was shown to be solvable in polynomial time for two bounded-degree trees, and NP-hard when one of the two trees has unbounded degree. See Ganapathy and Warnow (2001) for polynomial time fixed-parameter algorithms to compute the MCST of three or more trees and Ganapathy and Warnow (2002) for approximation algorithms for computing the size of the complement of the MCST.

6.4 Further reading

The consensus tree and agreement tree methods we have presented are just a small sample of a large number of such methods. For a survey of some of the consensus tree literature up to 1991, see Bryant (1991).

More recent developments include methods that operate by encoding each source tree as a set of triplet trees (if the source trees are rooted) or quartet trees (if the source trees are unrooted), and then applying methods for combining these subset trees together. An example of this approach is the the “local consensus tree” (Kannan et al., 1995), method, which assumes a rule is provided for combining rooted triplet trees and then constructs a tree, if it exists, consistent with that rule. Quartet-based approaches are generally more popular, since they avoid the problem of rooting the source trees. developed; see Chapter 3 for examples of quartet-based tree estimation methods that can be used as consensus methods. By construction, however, quartet-based methods are expensive, since they require the calculation of all $\Theta(n^4)$ quartet trees (where $n$ is the number of species), which makes their use limited to small datasets.

6.5 Review questions

1. Suppose you have ten trees on the same leafset.
Consensus and Agreement Trees

- Define the strict consensus tree, and describe a method for how to calculate the strict consensus tree.
- Define the majority consensus tree, and describe a method for how to calculate the majority consensus tree.
- Define the greedy consensus tree, and describe a method for how to calculate the greedy consensus tree.

2. Suppose you have a collection of binary trees (each of them different), all on the same leafset \{1, 2, \ldots, n\}, with \(n > 4\). Suppose that \(T\) is a compatibility tree for the set. How many trees can be in the collection?

3. What is the maximum agreement subtree (MAST) problem? What is the computational complexity of MAST on two trees?

4. What is the maximum compatibility subtree (MCST) problem? What is the computational complexity of MCST on two trees?

5. Suppose \(T\) and \(T'\) are two different trees, each on the same leafset. Can the MCST and MAST of \(T\) and \(T'\) be identical? Can they be different? Suppose that they have different numbers of leaves; which one must have more leaves? Suppose \(T\) and \(T'\) are binary; can the MCST and MAST be different?

6. Suppose you have a binary tree \(t\) on 10 leaves and you create two trees \(T\) and \(T'\) by adding in a leaf \(x\) into \(t\) in two different places. What is the size of the MAST of \(T\) and \(T'\)?

6.6 Homework problems

1. Consider the following three trees, each on set \(S = \{1, 2, 3, 4, 5, 6\}\).
   - \(T_1\) given by \(C(T_1) = \{(12|3456), (123|456), (1234|56)\}\)
   - \(T_2\) given by \(C(T_2) = \{(12|3456), (123|456), (1235|46)\}\)
   - \(T_3\) given by \(C(T_3) = \{(12|3456), (126|345), (1236|45)\}\)

   Is it possible to order the bipartitions of this set so as to produce \(T_2\) as a greedy consensus? If so, provide one such ordering. If not, explain why not.

2. Suppose you have an arbitrary set \(\mathcal{T}\) of trees on the same leaf set, and you compute the strict, majority, and greedy consensus trees. For each of the following pairs of trees, suppose they are different; must one of them refine the other? If so, which one, and why?
   - The strict consensus and majority consensus
   - The greedy consensus and the majority consensus
   - The strict consensus tree and an arbitrary tree in \(\mathcal{T}\)
   - The majority consensus and an arbitrary tree in \(\mathcal{T}\)

3. Give two different compatible unrooted trees on the same leaf set, and present their minimal common refinement.
4. Give two different trees on the same leaf set, neither of which is fully resolved, and which are not compatible.

5. Describe a polynomial time algorithm to compute the compatibility tree of two unrooted trees, and implement it. (Remember that the compatibility tree is the minimally resolved tree that is a common refinement of the two input trees.)

6. Let unrooted \( T_0 \) given by \( \{a, (b, (c, (d, e), (f, g)))\} \) denote the true tree.

1. For each unrooted tree below, draw the tree, and write down the bipartitions that are false positives and false negatives with respect to \( T_0 \).
   
   - \( T_1 = \{f, (g, (a, (b, (c, (d, e))))))\} \).
   - \( T_2 = \{g, (f, (c, (d, (e, (a, b))))))\} \).
   - \( T_3 = \{g, (f, (a, (b, (c, (d, e))))))\} \).

2. Draw the strict, majority, and greedy consensus trees for these three trees \( T_1, T_2, \) and \( T_3 \). Compute the false negatives and false positives (with respect to \( T_0 \)) for these consensus trees.

7. Consider an arbitrary unrooted binary true tree and let \( \mathcal{T} \) be a set of estimated unrooted trees. Suppose you compute the strict consensus, majority consensus, and greedy consensus of these trees. Now compute the false negative error rates of these three consensus trees, and compare them to each other and also to the false negative error rates of the trees in the set \( \mathcal{T} \). What can you deduce? Do the same thing for the false positive error rates.

8. Give two unrooted trees, \( T_1 \) and \( T_2 \), that are compatible, and their unrooted compatibility tree \( T_3 \). Treat \( T_3 \) as the true tree, and compute the False Negative and False Positive rates of \( T_1 \) and \( T_2 \) with respect to \( T_3 \). What do you see?

9. Consider a caterpillar tree \( T \) on a set \( S \) of \( n \) taxa. Suppose there is a very “rogue” taxon, \( x \). Consider the set \( \mathcal{T} \) that contains all the trees formed by adding \( x \) into \( T \).

   1. What is \( |\mathcal{T}| \)?
   2. What is the strict consensus of all the trees in \( \mathcal{T} \)? (Give its bipartition set.)
   3. Draw the majority consensus when \( n = 4 \).
   4. Draw the majority consensus when \( n = 5 \).
   5. Draw the majority consensus when \( n = 6 \).
   6. Can you generalize the observations seen here to provide a rule for the majority consensus trees, as a function of \( n \)?

10. Consider the caterpillar tree \( T \) on 6 taxa, and let \( \mathcal{T} \) be the set of trees produced by adding an additional taxon, \( x \), into \( T \) in each of the possible ways. What is the maximum agreement subtree of the trees in \( \mathcal{T} \)?

11. Suppose you have a set \( X \) of trees, each on the same set of taxa, and you are fortunate enough to be able to label each edge as a true positive or a false positive with respect to the true tree \( T \). Suppose you contract every false positive edge in each tree in \( X \), and consider the resultant set \( X' \). For each of the following pairs of trees \( T_1 \) and \( T_2 \), indicate if one tree in the pair must refine the other tree (and if so, which one). Also indicate if
the number of leaves in one tree must be at most the number of leaves in the other one (and if so, which one).

- Let $T_1$ be an MCST (maximum compatible subtree) of the trees in $X'$, and $T_2$ be an MCST of the trees in $X$.
- Let $T_1$ be an MCST of the trees in $X$, and let $T_2$ be a MAST of the trees in $X$.
- Let $T_1$ be an MCST of the trees in $X'$, and let $T_2$ be a MAST of the trees in $X'$.
- Let $T_1$ be a MAST of the trees in $X'$, and let $T_2$ be the true tree.
- Let $T_1$ be an MCST of the trees in $X'$, and let $T_2$ be the true tree.

12. Suppose you have a set $X$ of trees, each on the same set of taxa, and the MCST and MAST for $X$ have different numbers of taxa. Which one of these has more taxa?
7

Supertrees

7.1 Introduction

The basic objective of most supertree studies is the assembly of a large species tree from a set of species trees that have been estimated on smaller sets of taxa. Indeed, it is generally believed that construction of the Tree of Life, which will encompass millions of species, will require supertree methods, because no software will be able to deal with the computational challenges involved in such a difficult task. More recently, however, new uses of supertree methods have been discovered, especially in the context of divide-and-conquer strategies. This chapter examines both applications of supertree methods.

Traditionally, supertree methods were used to combine trees computed by different researchers that had already been estimated for different taxon sets. In this case, the person constructing the supertree has no control over the inputs, neither how the different subset trees were constructed nor how the taxon sets of the different subset trees overlap. Furthermore, the person constructing the supertree may not have easy access to the data (e.g., sequence alignments) on which the subset trees were constructed.

A modern and more interesting use of supertree methods is in the context of a divide-and-conquer strategy to construct a very large tree, or to enable a statistically powerful but computationally intensive method to be applied to a larger dataset. In such a strategy, a large set of taxa is divided into overlapping subsets, trees are estimated (using the desired method) on the subsets, and the estimated subset trees are combined into a tree on the full set of taxa using a supertree method.

Divide-and-conquer techniques for constructing large trees have many desirable features: (1) the subsets can be made small enough that expensive methods can be used to construct trees on them, (2) different methods can be used on each subset, thus making it possible to better address heterogeneity within the full dataset, and (3) the subsets can be created so as to have desirable overlap patterns. The first two of these features tend to increase the accuracy of the estimated subset trees, while the third feature can make it easier to construct an accurate supertree from the subset trees. We will return to the topic of divide-and-conquer strategies and how to use them to construct large trees under a variety of scenarios in Chapter 11. For now, just be aware that supertree methods are more than
just ways of assembling large trees from other trees; they are key ingredients in developing methods to enable powerful but expensive methods to run on ultra-large datasets.

The input to the supertree method is a set of source trees, called a profile. In the idealized condition, all the source trees are compatible with each other, and the objective will be to construct a compatibility supertree. However, since source trees are estimated trees, they are likely to have some estimation error, and no compatibility supertree will exist. Therefore, rather than trying to find a compatibility supertree (see Chapter 3.5), the main objective is to find a supertree that is somehow as close as possible, with respect to some criterion, to the input source trees. Examples of optimization problems that have been used to define supertrees include Matrix Representation with Parsimony (MRP) (Baum and Ragan, 2004), Matrix Representation with Likelihood (MRL) (Nguyen et al., 2012), Robinson-Foulds Supertree (RFS), and Maximum Quartet Support Supertree (MQS), described below. Each of these optimization problems is formulated in terms of unrooted source trees, but have equivalent formulations for rooted source trees. Supertree software implementing specific approaches (some of which are based on optimality criteria) include MinCut Supertree (Semple and Steel, 2000), Modified MinCut Supertree (Page, 2002), MinFlip Supertree (Chen et al., 2003, 2006) and the MRF Supertree (Burleigh et al., 2004).

Unfortunately, all the optimization problems mentioned above, and in fact nearly all optimization problems that have been posed for supertree construction, are NP-hard. Therefore, heuristics rather than exact algorithms are used to find good solutions to these optimization problems. Thus, supertree methods are, like most problems in phylogenetics, understood by the theoretical properties of the optimization problems on which they are based, by the details of their implementations, and by how well they perform on data.

For the rest of this chapter, we will assume that the profile given as input to the supertree problem is \( T = \{ t_1, t_2, \ldots, t_k \} \). We will let \( S_i \) denote the leafset of tree \( t_i \), and we let \( S = \bigcup S_i \). Most of the supertree problems we will discuss involve profiles of unrooted trees; however, we will also (briefly) describe some methods for computing supertrees when the profile contains rooted trees.

### 7.2 Compatibility Supertrees

We begin with a definition of the compatibility supertree problem:

- **Input:** \( \mathcal{T} = \{ t_1, t_2, \ldots, t_k \} \), where \( t_i \) has leafset \( S_i \) and is an unrooted tree
- **Output:** tree \( T \), if it exists, such that \( T|S_i \) refines \( t_i \) for each \( i = 1, 2, \ldots, k \). Equivalently, \( C(t_i) \subseteq C(T|S_i) \) for every \( i \), and \( \sum_{i=1}^{k} |C(t_i) \setminus C(T|S_i)| = 0 \).

Note that a compatibility supertree may not exist, but that when a profile has a compatibility supertree, it can have more than just one.

**Example 7.1** Let \( t_1 = ((a,b),(c,d)) \) and \( t_2 = ((a,b),(c,e)) \). Then \( (a,(b,(c,(d,e)))) \) and \( (a,(b,(d,(c,e)))) \) are both compatibility supertrees for the profile \( \{ t_1, t_2 \} \).
Determining if a compatibility supertree exists is NP-complete since quartet compatibility is NP-complete.

7.3 Asymmetric Median Supertrees

In Chapter 6.2.5, we described a consensus method called the Asymmetric Median Tree and that was related to determining if a profile of source trees was compatible. Since consensus methods require that the profile contain trees that have the same set of leaves, we extend that approach to allow for profiles of trees that are on different taxon sets, and we call this the **Asymmetric Median Supertree**:

- **Input**: profile \( T = \{ t_1, t_2, \ldots, t_k \} \) of unrooted trees, with \( t_i \) having leafset \( S_i \)
- **Output**: tree \( T \) with leafset \( S = \bigcup_{i=1}^{k} S_i \) that minimizes \( \sum_{i=1}^{k} |C(t_i) \setminus C(T)| \).

**Lemma 7.2** Let \( T \) be a profile of unrooted source trees, and suppose that \( T \) is compatible so that a compatibility supertree exists. Then \( T \) is a compatibility supertree for \( T \) if and only if \( T \) is an asymmetric median supertree for \( T \). Furthermore, any compatibility supertree \( T \) satisfies \( \sum_{i=1}^{k} |C(t_i) \setminus C(T)| = 0 \).

Computing the asymmetric median supertree is also NP-hard, since a compatibility supertree would be an optimal solution (if it exists), and determining if a compatibility supertree exists is NP-complete. Note that we do not require that the source trees be bifurcating. If we add that condition, then any compatibility supertree \( T \) will satisfy \( T|S_i = t_i \) for each \( i = 1, 2, \ldots, k \), and so will also satisfy \( RF(T|S_i, t_i) = 0 \) for each \( i = 1, 2, \ldots, k \).

This leads to the statement of the Robinson-Foulds Supertree problem, the subject of the next section.

7.4 Robinson-Foulds Supertrees

The Robinson-Foulds Supertree (Bansal et al., 2010; Chaudhary et al., 2012; Chaudhary, 2015; Kupczok, 2011), is a supertree \( T \) that minimizes the total Robinson-Foulds (RF) distance between \( T \) and the source trees. Since the leafset of a source tree will in general be a proper subset of \( S \), we need to define how the Robinson-Foulds distance is computed between two trees in the context of supertree construction.

**Definition 7.3** Let \( T \) have leafset \( S \) and let \( t \) have leafset \( S' \subseteq S \). Then \( RF(T, t) \) is the Robinson-Foulds distance between \( T|S' \) and \( t \).

**Example 7.4** Let \( T = (1, ((6,7),(4,(3,(5,2)))))) \) and \( t = (5,((3,2),(6,7))) \). \( S' \), the leafset of \( t \), is \( \{2,3,5,6,7\} \). To compute the Robinson-Foulds distance between \( T \) and \( t \) we modify \( T \) by deleting the taxa in \( S\setminus S' \), and then suppressing nodes of degree two; this gives us \( T|S' \), which is \(( (6,7),(3,(5,2))) \). The non-trivial bipartitions of \( T|S' \) are \((235|67)\) and \((25|367)\). The non-trivial bipartitions of \( t \) are \((235|67)\) and \((23|567)\). Therefore, \( RF(T,t) = 1 \).

The Robinson-Foulds supertree problem is:
Supertrees

- Input: profile $\mathcal{T} = \{t_1, t_2, \ldots, t_k\}$ of unrooted source trees
- Output: tree $T_{RFS}$ that minimizes the total Robinson-Foulds distance to the source trees, i.e.
  \[ T_{RFS} = \arg\min_T \sum_{i=1}^k RF(T, t_i). \]

The Robinson-Foulds Supertree problem is a popular approach to supertree estimation, and several methods have been developed to try to find good solutions to the problem. Examples of these methods include PluMiST (Kupczok, 2011), Robinson-Foulds Supertrees (Bansal et al., 2010), and MulRF (Chaudhary, 2015).

The Robinson-Foulds supertree and the Asymmetric Median supertree are clearly similar, because both are based on bipartitions, but the Asymmetric Median supertree does not penalize the supertree for containing bipartitions that appear in the source trees, while the Robinson-Foulds supertree does. Thus, the two problem formulations can give different results, even on the same input. However, the two problems are related to the compatibility supertree problem, as we now show:

**Theorem 7.5** Let $\mathcal{T} = \{t_1, t_2, \ldots, t_k\}$ be a set of trees where $t_i$ has leafset $S_i$, and let $S = \bigcup_i S_i$. Then $T$ is a compatibility supertree for $\mathcal{T}$ if and only if $T$ has leafset $S$ and $|C(t_i) \setminus C(T|S_i)| = 0$ for all $i = 1, 2, \ldots, k$. Hence, when a compatibility supertree exists, then $T$ is an asymmetric median supertree if and only if $T$ is a compatibility supertree. Furthermore, if each $t_i$ is a fully resolved tree, then any compatibility supertree $T$ satisfies $RF(T, t_i) = 0$ for all $i = 1, 2, \ldots, k$. Conversely, if $T$ is a tree on leafset $S = \bigcup_i S_i$ that satisfies $|C(t_i) \setminus C(T|S_i)| = 0$ for all $i$, then by definition $\mathcal{T}$ is a compatible set of trees, and $T$ is a compatibility supertree for $\mathcal{T}$.

We will use this analysis to prove that finding the optimal Robinson-Foulds supertree is NP-hard.

**Theorem 7.6** The Robinson-Foulds Supertree problem is NP-hard.

**Proof** Suppose that $\mathcal{A}$ is an algorithm that solves the Robinson-Foulds Supertree problem in polynomial time; thus, $\mathcal{A}(\mathcal{T})$ is a Robinson-Foulds supertree for the profile $\mathcal{T}$. We will show that we can use $\mathcal{A}$ to solve the Unrooted Tree Compatibility problem in polynomial time, for the case where the input trees are all binary. Since Unrooted Tree Compatibility is NP-complete even when the input trees are all binary, this will prove that the Robinson-Foulds Supertree problem is NP-hard.

Let $\mathcal{T} = \{t_1, t_2, \ldots, t_k\}$ be a profile of unrooted binary trees, with $S_i$ the leafset of $t_i$ for $i = 1, 2, \ldots, k$. We run $\mathcal{A}$ on the profile $\mathcal{T}$ to obtain $T = \mathcal{A}(\mathcal{T})$; thus, $T$ is an optimal solution to the Robinson-Foulds Supertree problem. We then compare $T|S_i$ to $t_i$ for each $i = 1, 2, \ldots, k$. If $T|S_i$ refines $t_i$ for every $i = 1, 2, \ldots, k$, then $T$ is a compatibility supertree, and otherwise $T$ is not a compatibility supertree. This is easily checked in polynomial time. Since we assume that each $t_i$ is binary (i.e., fully resolved), then if $T|S_i$ refines $t_i$ it follows
that $T|S_i$ is identical to $t_i$. If $T$ passes this test for every $i = 1, 2, \ldots, k$, then we will say that $T$ is compatible, and otherwise we will say that $T$ is not compatible.

To prove that we have solved the Unrooted Tree Compatibility problem (for the case where every tree in the input profile is fully resolved), we just need to show that the profile $T$ is compatible if and only if $T = \alpha(T)$ passes this test for every $i$. So suppose $T$ passes the test for every $i = 1, 2, \ldots, k$. Then $T$ is a compatibility supertree for $T$, and so the profile $T$ is compatible. Now suppose that the profile $T$ is compatible, but that $T$ is not a compatibility supertree for $T$. Since $T$ is compatible, there is a compatibility supertree, $T^*$. Note that $T^*|S_i = t_i$ (i.e., $RF(T^*, t_i) = 0$) for $i = 1, 2, \ldots, k$, and so $T^*$ is an optimal solution to the Robinson-Foulds Supertree problem. Since $\alpha$ is an exact algorithm for the Robinson-Foulds Supertree problem, $T^*$ and $T$ must both have the same score under the Robinson-Foulds Supertree criterion. Hence, $RF(T, t_i) = 0$ for each $i$ as well, and so $T$ is also a compatibility supertree for $T$, contradicting our assumption. Hence, the profile $T$ is compatible if and only if $T = \alpha(T)$ is a compatibility supertree.

Since $\alpha$ runs in polynomial time, we have shown that if the Robinson-Foulds Supertree problem can be solved in polynomial time then the unrooted tree compatibility problem can also be solved in polynomial time, when all the source trees are binary. Since unrooted tree compatibility is NP-complete, even when all the source trees are binary, this shows that the Robinson-Foulds Supertree problem is NP-hard.

Note that the key part of the proof is that when the input profile is compatible the compatibility tree would be an optimal solution to the Robinson-Foulds supertree problem. Therefore, the same kind of argument can be used to establish that any supertree problem where an optimal solution would be a compatibility supertree is NP-hard. We note this here, since nearly every supertree problem we discuss henceforth will have this property: 

**Corollary** Let $X$ be a supertree optimization problem where the input is a profile of unrooted source trees. If an exact algorithm for $X$ would return a compatibility supertree (when it exists), then $X$ is NP-hard. Hence, the Asymmetric Median Tree problem is NP-hard.

### 7.5 Matrix Representation with Parsimony (MRP)

The Matrix Representation with Parsimony (MRP) (Baum and Ragan, 2004) supertree problem is by far the most well known (and most popular) supertree optimization problem. Although several variants of MRP have been posed, which differ depending on whether the input trees are rooted or unrooted, we will focus on the version where the input trees are unrooted.

The input to MRP is a profile $\{t_1, t_2, \ldots, t_k\}$ of unrooted trees where $t_i$ has leafset $S_i$. From this set of trees, we compute a matrix, called the “MRP matrix”, defined by the concatenation of the matrices obtained for each of the trees in the profile. Thus, to define the MRP matrix for the profile, it suffices to show how we define the MRP matrix for a single tree $t_i$. 


Supertrees

The matrix for source tree \( t_i \) on taxon set \( S_i \) has a row for every element of \( S = \bigcup_i S_i \) and a column for every internal edge in \( t_i \). To define the column associated to the internal edge \( e \) in \( t_i \), we compute the bipartition on the leafset \( S_i \) defined by removing \( e \) from \( t_i \), and we arbitrarily assign 0 to the leaves on one side of this bipartition and 1 to the other side. We then assign \(?\) to every \( s \in S \setminus S_i \). Thus, for \( |S| = n \), then each edge in \( t_i \) is represented by an \( n \)-tuple with entries drawn from \( \{0, 1, ?\} \). For each such edge we create a column defined by its \( n \)-tuple, and we concatenate all these columns together into a matrix that represents the tree \( t_i \). After computing all the matrices for all the trees, we concatenate all the matrices together into one large matrix, which is called the “MRP matrix”. Note that in this matrix, every element \( s \in S \) is identified with its row. The number of columns is the sum of the number of internal edges among all the trees. Since each tree can have up to \( n - 3 \) internal edges, this means that the number of columns is \( O(nk) \), where \( k \) is the number of source trees and \( n \) is the number of species.

Under the MRP criterion, we seek the tree that optimizes the Maximum Parsimony criterion with respect to the input MRP matrix. Since the MRP matrix will in general have \(?\)s, we need to explain how these are handled. Let \( M \) be the MRP matrix computed from the profile. Since \( M \) may have \(?\)s, we consider the set \( \mathcal{M} \) of all matrices \( M' \) that can be formed by replacing the \(?\) entries by 0 or 1; hence, if \( M \) has \( p \) entries that are \(?\), then \( |\mathcal{M}| = 2^p \). In other words, \( \mathcal{M} \) is the set of binary matrices that agree with \( M \).

Recall that \( MP(T, M) \) is the maximum parsimony score of tree \( T \) for the matrix \( M \). We denote by \( MRP(T, \mathcal{T}) \) the MRP score of a tree \( T \) with respect to the profile \( \mathcal{T} \). Then, \( MRP(T, \mathcal{T}) = \min \{ MP(T, M') : M' \in \mathcal{M} \} \). In other words, we are seeking the best way of replacing all the question marks (\(?\)s) by zeros and ones so that the result gives us the best possible maximum parsimony score.

Thus, the MRP problem is really the Maximum Parsimony problem on the MRP matrix, with the understanding of how “missing data” (as represented by \(?\)s) are handled by Maximum Parsimony. Because maximum parsimony is NP-hard, even when the input is a set of binary characters, MRP is NP-hard (Foulds and Graham, 1982). Here we present a proof of this statement.

**Theorem 7.7** Let \( \mathcal{T} = \{ t_1, t_2, \ldots, t_k \} \) be an input of unrooted source trees. If \( \mathcal{T} \) is compatible, then the MRP matrix defined on this input has a perfect phylogeny, and any optimal solution to MRP will be a compatibility supertree. Furthermore, the MRP problem is NP-hard.

**Proof** When the source trees are compatible, every column in the MRP matrix (which corresponds to a bipartition in some source tree) will be compatible with every compatibility supertree, in the sense that the \(?\)s can be replaced by 0s and 1s to correspond to some edge in the compatibility supertree. In other words, the columns in the MRP matrix are compatible partial binary characters (where “partial binary characters” are characters with 0s, 1s, and \(?\)s), and a perfect phylogeny exists for the matrix (see Chapter 4.1). Conversely, if the MRP matrix is compatible, then a compatibility supertree exists for the profile. Since par-
7.6 Matrix Representation with Likelihood (MRL)

The supertree optimization problem, Matrix Representation with Likelihood (MRL), was introduced in Nguyen et al. (2012). MRL as is nearly identical to the MRP problem, but instead the objective being a maximum parsimony tree for the MRP matrix, the objective is a maximum likelihood tree for the MRP matrix, under the CFN (Cavender-Farris-Neyman) model. The CFN model is the symmetric binary model of sequence evolution where the state at the root is equiprobable to be 0 or 1; therefore, the choice of 0 or 1 is randomized to ensure that the state at the root is selected randomly. As with the MRP supertree methods, MRL treats ?s as missing data.

As shown in Nguyen et al. (2012), using RAxML to solve MRL resulted in highly accurate supertrees, that were typically as accurate or more so than using good heuristics for maximum parsimony on the MRP matrix. Thus, in Nguyen et al. (2012), MRL seems to be at least as accurate as MRP as a supertree method.

7.7 Maximum Quartet Support Supertrees

7.7.1 Problem formulation

We begin by defining the quartet support between a tree $t$ and a larger tree $T$, under the assumption that the leafset of $t$ is a subset of the leafset of $T$. Recall that $Q(t)$ and $Q(T)$ denote the set of quartet trees of $t$ and $T$, respectively. We define the quartet support of $t$ for $T$ to be $|Q(t) \cap Q(T)|$. Then, the quartet support of a profile $\mathcal{T} = \{t_1, t_2, \ldots, t_k\}$ for the tree $T$ is $\sum_{i=1}^{k} |Q(t_i) \cap Q(T)|$. Equivalently, the Maximum Quartet Support supertree $T_{MQS}$ is the tree

$$T_{MQS} = \arg\max_T \sum_{i=1}^{k} |Q(t_i) \cap Q(T)|.$$

Finding the Maximum Quartet Support supertree is NP-hard, since the compatibility supertree (if it exists) would be an optimal solution to the Maximum Quartet Support supertree problem, and determining whether a profile of trees is compatible is itself NP-complete (Steel, 1992b).

Heuristics for this problem could encode each source tree as a set of quartet trees, and then use heuristics to try to construct trees that satisfy as many of the input quartets as possible; examples of such methods include Quartet Puzzling algorithm (Strimmer and von Haeseler, 1996), Weight Optimization (Ranwez and Gascuel, 2001), Short Quartet Puzzling (Snir et al., 2008), Quartets Max Cut (Snir and Rao, 2010), and QFM (Reaz et al., 2014). Although this optimization problem is NP-hard, approximation algorithms can be developed, and when the set contains a tree on every quartet, then a polynomial time approximation scheme (PTAS) is also possible (Kearney et al., 1999).
An interesting variant allows the input quartet trees to be equipped with arbitrary positive weights \(w(q)\) and seeks the tree \(T\) with the maximum total quartet weight (i.e., maximizing \(\sum_{q \in Q(T)} w(q)\)). Since the unweighted version is NP-hard, the weighted version problem is also NP-hard. Heuristics for maximum weighted quartet support have been proposed, including Weighted Quartets Max Cut (Avni et al., 2015).

### 7.7.2 Split-Constrained Quartet Support Supertrees

One way of handling NP-hardness is to constrain the search space, and then solve the problem exactly within that constrained space. This kind of approach has been used very effectively with quartet-based optimization problems, and is the subject of this section. Specifically, we define the **Split-Constrained Quartet Support Problem** as follows:

- **Input**: Function \(w\) that assigns non-negative weights to all quartet trees on a set \(S\), and set \(X\) of bipartitions of \(S\).
- **Output**: Unrooted binary tree \(T\) such that \(Q(T)\) has maximum total weight among all unrooted binary trees \(t\) that satisfy \(C(t) \subseteq X\). In other words, \(T\) maximizes \(\sum_{t \in Q(T)} w(t)\) among all trees that draw their bipartitions from \(X\).

Note that when \(X\) is the set of all possible bipartitions on \(S\), there is no constraint on the set of trees \(T\) that can be considered, and so the Split-Constrained Quartet Support problem is just the Maximum Quartet Compatibility problem. However, for other settings for \(X\), the constraint on the set of possible trees can be very substantial.

The Split-Constrained Quartet Support problem can be solved in polynomial time using dynamic programming. To do this, we first define a nearly identical problem (the Clade-Constrained Quartet Support problem) where we seek a rooted tree instead of a rooted tree, and we constrain the set of clades the rooted tree can have.

We will say that quartet tree \(ab | cd\) supports the unrooted tree \(T'\) if \(ab | cd \in Q(T')\). Then, since every rooted tree \(T\) defines an unrooted tree \(T_u\), and so we will say that a quartet tree \(ab | cd\) supports the rooted tree \(T\) if \(ab | cd\) supports \(T_u\).

The input to the Clade-Constrained Quartet Support problem is a set \(C\) of subsets of \(S\), and a non-negative function \(w\) on the set of all possible quartet trees of \(S\). The objective is a rooted binary tree \(T\) such that \(\text{Clades}(T) \subseteq C\) and \(T\) has maximum quartet support among all rooted binary trees that satisfy this constraint. In other words, letting \(R_{C,S}\) denote the set of rooted binary trees on taxon set \(S\) that draw their clades from \(C\), then the Clade-Constrained Quartet Support tree is

\[
T_{\text{CCQS}} = \arg \max_{T \in R_{C,S}} \sum_{t \in Q(T)} w(t)
\]

We can construct a rooted binary tree \(T\) that has the best Clade-Constrained Quartet Support using dynamic programming. After we find a rooted binary tree with the best Clade-Constrained Quartet Support, we will unroot the rooted tree, thus producing an unrooted binary tree with the best quartet support.
To see how this might work, let \( \mathcal{F} \) be the set of source trees and \( X \) the set of allowed bipartitions given as input to the Bipartition Constrained Quartet Support problem. To construct \( C \), we include every half of every bipartition in \( X \), and then we add the singletons (i.e., the elements of \( S \)) and the full set \( S \). Note that for every bipartition \( A\mid B \) in \( X \), the set \( C \) will contain both \( A \) and \( B \).

If \( R_{CS} = \emptyset \), then there are no feasible solutions to the Clade-Constrained Quartet Support problem, and also no feasible solutions to the Split-Constrained Quartet Support problem, and we return \( \text{Fail} \). Otherwise, suppose \( T \in R_{CS} \) and let \( t \in Q(T_\emptyset) \). Let \( v \) be the (unique) lowest node \( v \) in \( T \) (i.e., the node that is furthest from the root of \( T \)) for which at least three of \( t \)'s leaves are below \( v \). We will say that the quartet tree \( t \) is mapped to the node \( v \) with this property.

Let \( v \)'s children be \( v_1 \) and \( v_2 \). Since \( T \in R_{CS} \), the sets of leaves below \( v \), \( v_1 \), and \( v_2 \) are all elements of \( C \). Furthermore, since the set \( A \) of leaves below \( v \) is an element of \( C \), then if \( v \) is not the root of \( T \), then the set \( S \setminus A \) is also in \( C \). In other words, the node \( v \) defines a tripartition of the leafset \( S \) into three sets of allowed clades, \( (A_1, A_2, S \setminus A) \). The root is also associated to a tripartition \( (A_1, A_2, \emptyset) \), with \( A_1 \) and \( A_2 \) the leaves of the two subtrees off the root. Just as we talked about mapping quartet trees to nodes, we can say that a quartet tree is induced by the tripartition associated to the node. Furthermore, we will say that a quartet tree is induced by the tripartition it is associated to.

Note that quartet \( ab\mid cd \) maps to \( (U, V, W) \) if and only if the following properties hold:

1. \( ab\mid cd \) is induced by the tripartition \( (U, V, W) \)
2. If the set \( \{a, b, c, d\} \) does not split 2-2 among \( U \) and \( V \), then two of its leaves are in \( U \) and one is in \( V \), or vice-versa.

Thus, we can determine if \( ab\mid cd \) maps to a given tripartition just by looking at the tripartition, and we do not need to consider the tree as a whole. Therefore, given any tripartition of \( S \) into \( (U, V, W) \) (where \( W = S \setminus (U \cup V) \)), we can compute the set of quartet trees that map to the tripartition, and hence the total weight of all quartet trees that map to the tripartition. We will denote this by \( QS(U, V, W) \). Since every quartet tree that supports a binary rooted tree \( T \) is mapped to exactly one node in \( T \), we can write \( \text{Support}(T) = \sum_{v \in V(T)} QS(U_v, V_v, W_v) \), where \( v \) defines tripartition \( (U_v, V_v, W_v) \). We generalize this by letting \( QScore(T,v) \) be the total quartet support at all the nodes in \( T \) at or below \( v \). Then

\[
QScore(T,v) = QScore(T,v_1) + Qscore(T,v_2) + QS(A_1, A_2, A_3),
\]

where \( v_1 \) and \( v_2 \) are the children of \( v \), \( A_i \) is the set of leaves below \( v_i \) for \( i = 1, 2 \), and \( A_3 = S \setminus (A_1 \cup A_2) \).

We will use these concepts by computing, for every allowed clade \( A \), the best possible quartet support score achievable on any rooted binary tree on \( A \), which will be the total support contributed by quartet trees that map to nodes in the tree on \( A \). Letting \( Qscore(A) \) denote this best possible score, we obtain \( QScore(A) = 0 \) for any clade \( A \) where \( |A| \leq 2 \). Otherwise, we look over all ways of dividing \( A \) into two sets \( A_1 \) and \( A_2 \) where each \( A_i \) is an allowed clade, and we set \( QScore(A) = QScore(A_1) + QScore(A_2) + QS(A_1, A_2, A_3) \), where
Supertrees

$A_3 = S \setminus A$. Note that this calculation requires that we compute $QScore(A)$ for each clade $A$ in order of increasing size, and that we precompute the $QS(A_1, A_2, A_3)$ values. In other words, we have formulated a dynamic programming solution to the Clade-Constrained Quartet Support problem!

- Given set $X$ of allowed bipartitions, compute set $C$ of allowed clades, and include the full set $S$ and all the singleton sets.
- Order the set $C$ by cardinality, from smallest to largest, and process them in this order.
- Compute $QS(U, V, W)$ for all tripartitions $(U, V, W)$ where $U, V,$ and $W$ are each non-empty allowed clades, and $U \cup V \cup W = S$.
- For clades $A \in C$ where $|A| \leq 2$, set $QScore(A) = 0$. For all larger $A \in C$, compute $QScore(A)$ in order from smallest to largest, setting $QScore(A) = \max \{ QScore(A_1) + QScore(A_2) + QS(A_1, A_2, S \setminus A) : A_i \in C, A = A_1 \cup A_2 \}$.
- Return $QScore(S)$ to find the maximum quartet support of any $T \in R_{C,S}$. To construct the optimal tree, use backtracking through the DP matrix.

The running time of this algorithm is clearly polynomial. A careful implementation would yield $O(n^4 |X| + n^2 |X|^2)$ time, where $n$ is the number of species (Bryant and Steel, 2001).

**Lemma 7.8** For all sets $C$ of allowed clades and all $A \in C$, $QScore(A)$ is the maximum number of quartet trees $t$ that can be satisfied by any rooted tree $T$ on leafset $A$ where at least three of the leaves of $t$ are in $A$.

**Proof** The proof follows from the observation that every quartet tree that is induced in $T$ is mapped to exactly one node in any rooted tree. Suppose the quartet tree $t$ is mapped to the root of the tree $T$, where the root defines the tripartition $A_1, A_2, S \setminus A$. In this case, $t$ is counted exactly once in $QS(A_1, A_2, S \setminus A)$. Otherwise, $t$ is mapped to a node below $v_1$ or $v_2$, where $v_1$ and $v_2$ are the two children of the root of $T$. Suppose without loss of generality that $t$ is mapped to a node below $v_1$; then by induction $t$ is counted once in $QScore(A_1)$, where $A_1$ is the clade below $v_1$. The lemma follows.

As a result, the following theorem is easy to prove:

**Theorem 7.9** The algorithm for clade-constrained quartet optimization finds an optimal solution in $O(|X|^2 nk)$ time, where the input has $k$ trees on $n$ taxa. Hence the split-constrained quartet optimization problem can be solved in $O(|X|^2 nk)$ time.

A variant of this constrained search problem has an input set $\mathcal{T}$ of source trees (each leaf-labelled by a subset of $S$), and sets the weight of a quartet tree $ab|cd$ to be the number of these source trees that induce $ab|cd$. For this specific problem an optimal solution to the Split-Constrained Quartet Support problem can be found in $O(nk |X|^2)$ time, where $n$ is the number of species and $k$ is the number of source trees (Mirarab and Warnow, 2015). This
7.8 The Strict Consensus Merger

7.8.1 Overview of the Strict Consensus Merger

The Strict Consensus Merger (SCM) tree is a supertree method that is used in both SuperFine (Swenson et al., 2012), a supertree method described in the next section, and DACTAL (Nelesen et al., 2012), an iterative divide-and-conquer technique for improving the scalability and accuracy of phylogeny estimation methods, described in Chapter 11.5.2. Because of its general utility, we describe the SCM tree here. For additional details, see Huson et al. (1999a,b); Warnow et al. (2001).

The Strict Consensus Merger (originally described in Huson et al. (1999a)) is a polynomial time supertree method. Unlike most supertree methods, the tree computed by the Strict Consensus Merger is designed for use as a constraint tree, which will need to be subsequently refined into a binary tree. More generally, the SCM tree is generally high unresolved, and can in some cases have large polytomies.
7.8.2 Computing the Strict Consensus Merger of two trees

We begin by describing how the Strict Consensus Merger (SCM) of two trees is performed; see Figure 7.8.1. The input is two source trees, $S_1$ and $S_2$.

- **Step 1**: We compute $B$, the common leafset of $S_1$ and $S_2$. We then compute the strict consensus tree of $S_1|B$ and $S_2|B$, and refer to this as the “backbone”. We modify $S_1$ and $S_2$, if necessary, to make them induce the backbone on $B$; thus, any conflict between $S_1|B$ and $S_2|B$ results in collapsing edges in $S_1$ and $S_2$. After this step completes, we refer to the pair of source trees as $S'_1$ and $S'_2$. Note that any conflict between $S_1|B$ and $S_2|B$ causes at least one polytomy in the SCM tree.

- **Step 2**: The taxa that are not in the backbone are now added to it. Each edge in the backbone corresponds to an edge or a path of more than one edge in each of $S'_1$ and $S'_2$. Note that $S'_i$ contributes additional taxa to an edge $e$ in the backbone if and only if $e$ corresponds to a path of more than one edge in $S'_i$. When only one of the two trees $S'_1$ and $S'_2$ contribute additional taxa to the edge, then those taxa are added directly, maintaining the topological information in the source tree. However, if both trees contribute taxa to the same edge, then this is a “collision”, and the additional taxa are added in a more complicated fashion, as we now describe. In the presence of a collision involving edge $e$, the paths corresponding to edge $e$ are collapsed to a path with two edges in both $S'_1$ and $S'_2$, producing trees $S''_1$ and $S''_2$. Note that now $S''_i$ has a single rooted subtree to contribute to the backbone at the edge $e$. The edge $e$ is subdivided by the addition of a new vertex $v_e$, and the two rooted subtrees (one from each of $S''_1$ and $S''_2$) are attached to vertex $v_e$, by identifying their roots with $v_e$. Note that each collision creates a single additional polytomy in the SCM tree.

7.8.3 Computing the Strict Consensus Merger of a set of trees

To compute the SCM of a set of trees, an ordering on the set of pairwise mergers must be defined. Different proposals have been made for how to define the ordering, with the objective being that the ordering should maintain as much resolution as is possible; thus, collisions should be avoided. One approach that has been used in the SuperFine (Swenson et al., 2012) supertree method (which uses the SCM in its protocol) is to pick the pair of trees to merge that maximizes the number of shared taxa; however, other approaches can be included.

Given an ordering of pairwise mergers, the SCM of a set of trees proceeds in the obvious way. After each pairwise merger is computed, the set of subset trees is reduced by one, and the process repeats. At the end, a tree is returned that has all the taxa; this is the SCM tree. The SCM of two trees is easily computed in polynomial time, since the strict consensus of two trees is a linear time operation.
7.8.4 Theoretical properties of the SCM tree

If the set of trees is compatible, then there will never be a conflict in any pairwise merger; however, it is still possible for a pair of compatible trees to have a collision, which will result in the SCM tree having a polytomy. In other words, the SCM of a set of compatible binary trees may not be fully resolved. In particular, the SCM tree is not guaranteed to solve the Unrooted Tree Compatibility Problem. This is not surprising, since the Unrooted Tree Compatibility Problem is NP-hard (Steel, 1992a), and hence a polynomial time algorithm cannot be expected to solve it.

On the other hand, there are conditions on the subtrees under which the SCM tree is guaranteed to be the unique compatibility supertree if the subtrees are compatible. Thus, the SCM method solves the Unrooted Tree Compatibility Problem under some conditions. See Chapter 13.1 for more about this.

**Theorem 7.10**  Let \( \mathcal{T} \) be a profile of \( k \) trees on set \( S \) of \( n \) taxa. The SCM of \( \mathcal{T} \) can be computed in time that is polynomial in \( k \) and \( n \).

It is easy to see that the SCM can be computed in polynomial time, and the running time analysis is left to the reader.

7.8.5 Performance in practice

By design the SCM is very conservative, and so will collapse edges in the presence of any conflict, and also in the presence of any collisions. As a result, the SCM tends to be highly unresolved. Indeed, the topological error in an SCM tree is high - but the error is in the form of false negatives (i.e., missing branches) instead of false positives.

7.9 SuperFine: a meta-method to improve supertree methods

7.9.1 Overview of SuperFine

SuperFine (Swenson et al., 2012) is a general technique that can be used with any supertree method, and operates in two steps (see Fig. 7.9.2).

- Step 1: the source trees are combined into a constraint tree (which tends to be incompletely resolved) using the Strict Consensus Merger (SCM).
- Step 2: The SCM tree is resolved into a binary tree using the selected supertree method.

Thus, SuperFine is a meta-method that is designed to work with a selected supertree method, and we refer to SuperFine run with supertree method \( M \) by “SuperFine+M”. SuperFine has been tested with MRP, MRL, and QMC (Quartets MaxCut) (Swenson et al., 2012; Nguyen et al., 2012; Swenson et al., 2011), and shown to improve the accuracy and scalability of these base supertree methods.

In Chapter 7.8, we described the Strict Consensus Merger (SCM) and how the SCM tree is computed. We also noted that the SCM tree is typically highly unresolved as a result of
collisions (where two source trees can be combined in more than one way) and conflicts (where two source trees are incompatible). Here we show how the SCM tree is refined into a binary tree, one polytomy at a time, using SuperFine.

7.9.2 SuperFine: refining each polytomy in the SCM tree

The key technique in SuperFine is to recode the topological information in each source tree using a new but smaller taxon set, and then SuperFine runs its selected supertree method on these recoded source trees. The outcome of this process is a supertree on the new taxon set, which is then used to refine the polytomy. This refinement step is applied to each polytomy in turn.

Here we describe how this recoding is done. Let $v$ be a polytomy in the SCM tree with degree $d > 3$. First, we relabel every leaf in the SCM tree using labels $\{1, 2, \ldots, d\}$, according to which of the $d$ subtrees off the vertex $v$ it belongs to. This produces a relabelling of the leaves of every source tree in $T$ with $\{1, 2, \ldots, d\}$. Now, if two sibling leaves $x, y$ in a source tree $t \in T$ have the same label $L$, we label their common neighbor by $L$ and remove both $x$ and $y$ from $t$. We repeat this process until no two sibling leaves have the same label. As proven in Swenson et al. (2012), when this process terminates, the source tree $t$ will have at most one leaf of each label. We refer to this process as “recoding the source trees for the polytomy $v$”, and the modified source trees are referred to as “recoded source trees”. Note that each recoded source tree has at most $d$ leaves. Thus, when the polytomy degree is relatively small (compared to the original dataset), this produces recoded source trees that are much smaller than the original source trees.

To refine the SCM tree at the polytomy $v$, we apply the selected supertree method to the recoded source trees, and obtain a tree $T(v)$ on leafset $\{1, 2, \ldots, d\}$. We then use $T(v)$ to refine the SCM tree at node $v$. Note that when $d = \text{deg}(v)$ is sufficiently small, this step can be quite fast. Furthermore, the refinements around the different polytomies can be performed in parallel (or sequentially, but in any order that is desired), since the different refinements do not impact each other.

7.9.3 Performance in practice

As shown in Swenson et al. (2012); Nguyen et al. (2012); Swenson et al. (2011), SuperFine is generally very fast even when run sequentially, and efficient parallel implementations have also been developed. Figure 7.4 (from Swenson et al. (2012)) shows a comparison of several supertree methods, including MRP (the most well known supertree method), SuperFine using heuristics for MRP as the base supertree method, and concatenation using maximum likelihood (CA-ML). While CA-ML had the best accuracy (i.e., lowest Robinson-Foulds topological error rate), the supertree method with the best accuracy was SuperFine+MRP. Furthermore, although all methods could complete on the 100-taxon datasets, fewer could complete on the 500-taxon datasets, and only CA-ML, SuperFine+MRP, and MRP could complete on the 1000-taxon datasets. A comparison of
the running times of these methods is shown in Figure 7.5. Note that the difference in running times is very large on the 500- and 1000-taxon datasets, where CA-ML takes more than 1440 minutes, MRP takes 180-240 minutes, and SuperFine takes between 90-180 minutes. Studies evaluating SuperFine with heuristics for MRL (Nguyen et al., 2012) and with Quartets MaxCut (QMC) (Snir and Rao, 2010) have also shown similar results (Nguyen et al., 2012; Swenson et al., 2011). Thus, SuperFine is a meta-method that uses divide-and-conquer to improve the accuracy and speed of supertree methods.

7.10 Further reading

Quartet-based supertree methods: Since unrooted trees can be encoded as quartet trees, quartet-based tree construction methods can be used as supertree methods. For example, the Quartets MaxCut method (Snir and Rao, 2010) has been studied as a supertree method in Swenson et al. (2011), and shown to have good accuracy.

Supertrees for rooted source trees: Optimization problems for supertree construction when the input is a set of rooted trees have also been explored. One such approach encodes each rooted source tree as a set of rooted triples, and then seeks a rooted tree that maximizes the number of triplet trees (from the rooted source trees) that agree with it. This is called the Maximum Triplet Support problem. Equivalently, this can be formulated as finding a subset of the triplets that are setwise compatible.

The Maximum Triplet Support problem is NP-hard (Bryant, 1997), and so heuristics have been developed to find good solutions to the problem. The most well known of these heuristics is MinCutSupertree (Semple and Steel, 2000), which the authors describe as “a recursively optimal modification of the algorithm described by Aho et al.” In other words, MinCutSupertree is a modification of the Aho, Sagiv, Szymanski, and Ullman (ASSU) algorithm Aho et al. (1978), which we discussed and used earlier, so that it can be run on incompatible source trees. A modification to the MinCutSupertree method was developed in Snir and Rao (2006), and found to be more accurate than MinCutSupertree.

Distance-based supertree estimation: Another class of supertree methods takes advantage of the fact that the source trees for the supertree problem typically define distance matrices of some type, so that supertree estimation methods can use the distance matrices to infer the final supertree. For example, estimated phylogenetic trees are often based on maximum likelihood, and so come with branch lengths that reflect the expected number of changes on the branch; thus, a maximum likelihood tree defines an additive matrix for its taxon set.

As Willson (2004) observed, the use of these “distance matrices” is a valuable source of information in supertree construction, since they naturally enable the estimation of branch
lengths in the supertree, something that is not possible when just using the source trees alone.

Here we describe a classical distance-based supertree problem, **Matrix Representation with Distances** (MRD) from Lapointe and Cucumel (1997). The input is a set of $k$ dissimilarity matrices, each on a subset of the taxon set $S$. We seek an additive matrix on the full species set that minimizes the total distance to the input matrices; in other words, we seek a median tree with respect to some way of measuring distances between matrices. Common ways of measuring distances use the $L_\infty$, $L_1$, and $L_2$ norms, but other norms can also be used. Note also that we require that the output be an additive matrix, so that we can use it to define the supertree topology and branch lengths. If we wish to compute a rooted supertree with branch lengths reflecting elapsed time, then we will require that the output be an ultrametric matrix.

Distance-based supertree estimation is not new (e.g., Willson (2004); Criscuolo et al. (2006); Lapointe et al. (2003); Lapointe and Cucumel (1997)), but none of the current methods has become widely in use. However, Build-with-Distances (Willson, 2004) came close to MRP and in some cases was more accurate (Brinkmeyer et al., 2011), leading the authors to conclude that distance-based supertree estimation might have potential to replace MRP.

As discussed in Chapter 5, there is a substantial literature on related problems, where the input is a single dissimilarity matrix $M$ and the objective is an additive matrix or an ultrametric matrix that is optimally close to $M$, under some metric between distance matrices. Some of these problems are solvable in polynomial time, others are NP-hard but can be approximated, and some are hard to approximate; see Ailon and Charikar (2005); Fakcharoenphol et al. (2003); Agarwala et al. (1996) for an entry to this literature.

**Statistical properties of supertree methods:** The supertree methods presented so far are heuristics for NP-hard optimization problems. As such, understanding their statistical properties is quite challenging, since characterizing the conditions under which the heuristics are guaranteed to find globally optimal solutions to their criteria is difficult. However, suppose that each of the optimization problems could be solved exactly – i.e., suppose that globally optimal solutions could be found. Could we say anything about the probability of recovering the true supertree? To answer this question, we pose this as a statistical estimation problem in which the (unknown) true supertree is used to generate a sequence of source trees under some random process.

Suppose that the source trees are on subsets of the full taxon set, and are generated by a random process defined by a model species tree on the full set of taxa. For example, the model could assume that a source tree is the tree induced by the model species tree on a random subset of the taxon set. Under this model of source tree generation, all the source trees are compatible, and the true species tree is a compatibility supertree. Since the random process generates all subtrees with non-zero probability, the model species tree is identifiable (i.e., the model species tree has non-zero probability, and no other tree on the full taxon set has non-zero probability). Furthermore, any method that is guaranteed to
return a compatibility supertree for the input set is statistically consistent under this model. Thus, exact solutions for many supertree optimization problems (e.g., MRP, Robinson-Foulds supertree, and Maximum Quartet Support supertree) will be statistically consistent methods for species tree estimation under this model.

However, when source trees are estimated species trees, then estimation error, and possibly gene tree heterogeneity due to biological factors such as incomplete lineage sorting and gene duplication and loss, are also part of the generative model. Several models have been proposed and maximum likelihood and Bayesian methods have been developed for these models (Steel and Rodrigo, 2008; Bryant and Steel, 2009; Martins et al., 2016; Ronquist et al., 2004; Cotton and Wilkinson, 2009).

Although the Robinson-Foulds Supertree will not be a maximum likelihood supertree under these models, under some conditions it will provide a good solution to the ML supertree under one of the exponential models described in Steel and Rodrigo (2008) (see discussion in Bryant and Steel (2009)). Therefore, the local search heuristics for the RF Supertree that have been developed, such as PluMiST (Kupczok, 2011), Robinson-Foulds Supertrees (Bansal et al., 2010), and MulRF (Chaudhary, 2015), can be considered heuristics for the ML Supertree problem.

7.11 Review questions

1. Define the MRP problem, and explain how to write down the MRP matrix.
2. Suppose you have a set of 100 binary trees, each tree has 10 species, and the total number of species is 100. How many rows and columns are in the MRP matrix?

7.12 Homework problems

1. Suppose $X$ is a set of compatible unrooted trees on different sets of leaves. What can you say about the solution space to MRP on input $X$?
2. Suppose you have 1000 trees, each with 100 leaves, and 5000 taxa overall. How big is the MRP matrix?
Figure 7.2 Schematic representation of the algorithmic strategy of SuperFine+MRP. Source trees $S_1$-$S_4$ are combined pairwise to produce a Strict Consensus Merger (SCM) tree, which retains only internal branches that are compatible with all of the source trees. Each polytomy in the SCM tree is then refined by running Matrix Representation with Parsimony (MRP) on modified source trees (see text).
Figure 7.3 Schematic representation of the second step of the algorithmic strategy of SuperFine+MRP, in which we refine the SCM tree produced in the first step. The steps here refer to the SCM tree $T'$, polytomy $u$, and source trees shown in Figure 1. a) The deletion of the polytomy $u$ from the tree $T'$ partitions $T'$ into four rooted trees, $T_1, T_2, T_3, T_4$. b) The leaves in each of the four source trees are relabeled by the index of the tree $T_i$ containing that leaf, producing relabeled source trees $S_{r1}, S_{r2}, S_{r3}, S_{r4}$. For example, the relabeled version of $S_4 = ac|bd$ is $S_{r4}^{c} = 12|34$. c) Each $S_{r}^c$ is further processed by repeatedly replacing sibling nodes with the same label, until no two siblings have the same label; this results in trees $S_{c1}, S_{c2}, S_{c3}, S_{c4}$. d) The MRP matrix is shown for the four source trees, including only the parsimony informative sites; thus, $S_{c3}$ does not contribute a parsimony informative site and is excluded. e) The result of the MRP analysis on the matrix given in (d). f) The tree resulting from identifying the root of each $T_i, i = 1, 2, 3, 4$, with the node $i$ in the tree from (e).
Figure 7.4 [From Swenson et al. (2012)] Robinson-Foulds (RF) error rates (mean with standard error bars) for SFIT, PhySIC, MRP, Q-imputation, SuperFine+MRP, and CA-ML for three different taxon sizes, as a function of the scaffold factor. Subfigure (a) shows results for 100 taxa, (b) shows results for 500 taxa, and (c) shows results for 1000 taxa.
Figure 7.5 [From Swenson et al. (2012)] Average running times (logarithmic scale), including the time needed to compute source trees, for SFTT, Q-imputation, CA-ML, PhySIC, MRP, and SuperFine+MRP for three different taxon sizes, as a function of the scaffold factor. For the supertree methods, running times shown include the time required to generate ML source trees using RAxML. Subfigure (a) shows results for 100 taxa, (b) shows results for 500 taxa, and (c) shows results for 1000 taxa. The curves for PhySIC, MRP, and SuperFine+MRP overlap for the 100-taxon datasets.
PART TWO
ANALYZING MOLECULAR DATA
8

Statistical gene tree estimation methods

8.1 Introduction to statistical estimation in phylogenetics

Phylogeny estimation is often posed as a statistical inference problem, where the taxa evolve down a tree via a stochastic process. Statistical estimation methods take advantage of what is known (or hypothesized) about that stochastic process in order to produce an estimate of the evolutionary history. That estimate can include a range of hypotheses – starting with the underlying tree, and perhaps also the location of the root, the time at the internal diversification events, the rates of evolution on each branch of the tree, etc. When we consider phylogeny reconstruction methods as statistical estimation methods, many statistical performance issues arise. For example: is the method guaranteed to construct the true tree (with high probability) if there is enough data? How much data does the method need to obtain the true tree with high probability? Is the method still relatively accurate if the assumptions of the model do not apply to the data that are used to estimate the tree?

Markov models of evolution form the basis of most computational methods of analysis used in phylogenetics, and can be used to describe how qualitative characters with any number of states evolve. The simplest of these are for two states, reflecting the presence or absence of a trait. But more commonly, these models are used for nucleotide or amino acid sequences, and so have 4 or 20 states, depending on the type of data. They can also be used (less commonly) for codon models, in which case they have 64 states (Goldman and Yang, 1994; Yang et al., 2000).

In Chapter 1, we described the Cavender-Farris-Neyman (CFN) model of binary sequence evolution, and a simple method to estimate the CFN tree from binary sequences. We continue the discussion of phylogeny estimation under the CFN model, and describe more sophisticated methods for phylogeny estimation under this model. We then address sequence evolution models that are applicable to nucleotide evolution. As we will see, the mathematical theorems and algorithmic approaches are very similar to those developed for phylogeny estimation under the CFN model.
8.2 Models of molecular sequence evolution

Because there are four nucleotides, A,C,T, and G, 4-state Markov models are used to describe nucleotide sequence evolution. Thus, while the CFN model could be described with very few parameters (the tree topology and the probability of change on each edge), for a DNA or RNA sequence evolution model, the $4 \times 4$ substitution matrix for each edge must be described. Similarly, for an amino acid sequence evolution model, since there are 20 amino acids, the $20 \times 20$ substitution matrices must be described.

The Jukes-Cantor model: The Jukes-Cantor (JC69) model (Jukes and Cantor, 1969) is the 4-state version of Cavender-Farris (CFN). Like the CFN model, the JC69 model is for a single site; to extend it to model how a sequence evolves down a tree, the sites of the sequence are assumed to evolve identically and independently under the same process.

Under the JC69 model, the state at the root (i.e., which nucleotide is picked) is drawn uniformly from the set of nucleotides, and if a change of state occurs on an edge then the new nucleotide is drawn uniformly from the remaining three nucleotides. Hence, a JC69 model is fully described by its tree $T$ and the numeric parameters on the edges of the tree that define the stochastic model of evolution. As with the Cavender-Farris-Neyman model, these numeric edge parameters can be expressed as the probability of change or as the expected number of changes on the edge. If we use probabilities of change, then we require that $0 < p(e) < 0.75$ for every edge. As with the Cavender-Farris-Neyman model, the Jukes-Cantor model can be described using branch length parameters, $\lambda(e)$, with $\lambda(e) = -\frac{3}{4} ln(1 - \frac{4}{3} p(e))$, so that $\lambda(e)$ denotes the expected number of changes of a random site on the edge $e$. Note that since $p(e) > 0$ it follows that $\lambda(e) > 0$.

Extending the JC69 model: Note that the Jukes-Cantor model states that (1) the sites evolve identically and independently down a tree, (2) all nucleotides are equiprobable at the root, and (3) if a site changes on an edge, it changes to each of the other nucleotides with equal probability. These properties make it analytically easy to estimate the tree from sequences that evolve under this model. However, what happens if we relax some of these assumptions?

It is not hard to see that changing the second assumption (that the nucleotides are equiprobable at the root) will not make inference more difficult (at worst, minor changes to estimation methods are needed), as long as each nucleotide has positive probability of being selected. However, the other two assumptions have larger impact on estimation. For example, if the first assumption (i.e., i.i.d. site evolution) is relaxed, then the unrooted model tree may not be identifiable from the distribution defined by the model tree (see, for example, Chang (1996a)). Therefore, most estimation methods are based on models that enforce this assumption. Finally, the third assumption – that all nucleotide substitutions are equiprobable – can be substantially relaxed without losing identifiability.

Over the period of 1980 to 1993, many more complex models were developed. The first of these was the Kimura 2-parameter model (referred to as both the K2P and the
8.2 Models of molecular sequence evolution

K80 model), which allowed for substitution rates to differ between nucleotides depend-
ing on the whether the change was to a nucleotide of the same or different type (purine
or pyrimidine, Kimura (1980)). Other elaborations include the F81 (Felsenstein, 1981),
HKY85 (Hasegawa et al., 1985), and TN93 (Tamura and Nei, 1993) models. However, all
of these models are subsumed by the Generalised Time-Reversible (GTR) model, intro-
duced in Tavaré (1986), and which has become the main model under which aligned DNA
sequences are analyzed. An even more general model than the GTR model is the General
Markov Model, introduced in Steel (1994a), and which is also identifiable. Finally, the No
Common Mechanism (NCM) model (Tuffley and Steel, 1997) was also introduced, but is
not identifiable. We describe the Kimura two-parameter model, the Generalised Time Re-
versible (GTR) Model (Tavaré, 1986) and the General Markov Model (Steel, 1994a) here,
and the No Common Mechanism model in Chapter 8.9. See Hillis et al. (1996); Li (1997);
Yang (2014) for more information about these and other DNA sequence evolution models.

Kimura two-parameter model: The Kimura two-parameter (K2P) model was developed
to allow for substitution probabilities that depended on whether the change of nucleotide
was between two purines (As and Gs), between two pyrimidines (Ts and Cs), or between
a purine and a pyrimidine. Thus, the K2P model has two free parameters, while the JC69
model has only one free parameter. The K2P and JC69 models are often used in simulation
studies, but not as frequently used in phylogenetic estimation as the GTR model.

The Generalized Time Reversible Model: The Generalized Time Reversible (GTR) model
is the most commonly used nucleotide sequence evolution model for phylogenetic estima-
tion purposes. The GTR model assumes there is a $4 \times 4$ substitution matrix $\mathbf{M}$ that defines
the probability of changing from one nucleotide to another nucleotide. This matrix can be
quite general, but $\det(\mathbf{M}) \neq 0, -1, 1$ is required in order to ensure identifiability.

The General Markov Model: A more general model is the General Markov Model (Steel,
1994a), which allows each edge $e$ to have its own substitution matrix $\mathbf{M}(e)$, as long as
each $\mathbf{M}(e)$ satisfies $\det(\mathbf{M}(e)) \neq 0, 1, -1$. Like the GTR model, the distribution of states
at the root can be fairly arbitrary, as long as each state has strictly positive probability.
Thus, the General Markov (GM) model contains the GTR model as a special case where
the substitution matrices across the different edges are all identical (i.e., $\mathbf{M}(e) = \mathbf{M}(e')$ for
all pairs of edges $e, e'$).

Amino acid sequence evolution models: Amino-acid sequence evolution is also modelled
using similar techniques, but here we would have $20 \times 20$ substitution matrices instead of
the $4 \times 4$ matrices. Unlike the nucleotide case, however, the $20 \times 20$ substitution matrices
are not estimated from the data. Instead of attempting to estimate all the parameters for a
given dataset, substitution matrices are computed based on some set of biological datasets,
and then used for subsequent studies. In other words, these amino acid substitution matrices
have no free parameters.
The earliest amino acid models were developed in Dayhoff et al. (1978), which were based on empirical frequencies observed in a relatively small database of aligned amino acid sequences. More recent amino acid models, such as the JTT (Jones et al., 1992) and WAG (Whelan and Goldman, 2001) models, have been developed using much larger databases of aligned amino acid sequences. To choose between different amino acid models, a biologist generally uses a statistical test, such as prottest (Abascal et al., 2005), that evaluates the fit of each of the different models to the biological dataset he or she wishes to analyze.

### 8.3 Statistical identifiability and consistency

Statistical identifiability is an important concept related to Markov models. We say that a parameter (such as the tree topology) of the Markov model is identifiable if the probability distribution of each character of the patterns of states at the leaves of the tree is sufficient to determine that parameter. Thus, some parameters of a model may be identifiable while others may not be. For example, the unrooted tree topology is identifiable under the General Markov Model (and hence under all its submodels, such as the GTR model), but the location of the root in the model tree is not.

Statistical consistency is another important concept, but is a property of a method rather than of the model. That is, a method is statistically consistent under a model if the method converges to the correct answer as the amount of data increases. Thus, if a method is statistically consistent under a model, then the model is identifiable. Conversely, if the model is not identifiable, then no method can be statistically consistent under the model. Note that statistical consistency or inconsistency cannot be established using a simulation; instead a formal proof is required.

### 8.4 Distance-based estimation

Distance-based tree estimation has two steps: first a matrix of pairwise distances is computed, and then a method is used to compute a tree for the distance matrix. In Chapter 5, we discussed methods for the second step, and we note that some of these methods (such as neighbor joining) have the nice property that they are guaranteed to return a tree \( T \) whenever the distance matrix \( d \) is close enough to an additive matrix \( D \) defining the tree \( T \).

In this chapter, we show how to compute a matrix of pairwise distances given a set of aligned sequences. These computations are done using formulae that are specifically designed for the models of sequence evolution we have discussed above, and have the guarantee that as the sequence length increases, the matrix of estimated pairwise distances will converge to an additive matrix defining the model tree. Therefore, under the assumption that the sequences are generated by under the same model of evolution, the two-step process is statistically consistent. Furthermore, as we will see, the calculation of these distances is polynomial time. We conclude this section with a comparison between existing distance-based methods.
8.4 Distance-based estimation

8.4.1 Statistical methods for computing distances

Computing distances under the CFN model: Let \( S = \{s_1, s_2, \ldots, s_n\} \) be a set of sequences, each of length \( k \), that have evolved down a Cavender-Farris-Neyman (CFN) model tree \( T \). We let \( \lambda_{i,j} \) denote the expected number of changes of a random site on the path in \( T \) between \( s_i \) and \( s_j \). Note that \( \lambda \) is an additive matrix for \( T \). We estimate the model distance between \( s_i \) and \( s_j \) using

\[
\hat{D}_{i,j} = -\frac{1}{2} \ln(1 - 2 \frac{H(i,j)}{k}),
\]

where \( H(i,j) \) denotes the Hamming distance between \( s_i \) and \( s_j \). Furthermore, \( \hat{\lambda}_{i,j} \) converges, as \( k \to \infty \), to \( \lambda_{i,j} \). This is called the CFN distance correction.

Computing distances for the Jukes-Cantor model: Let \( S = \{s_1, s_2, \ldots, s_n\} \) be a set of sequences, each of length \( k \), that have evolved down a Jukes-Cantor model tree \( T \). We let \( \lambda_{i,j} \) denote the expected number of changes of a random site on the path in \( T \) between \( s_i \) and \( s_j \). Note that \( \lambda \) is an additive matrix for \( T \). We estimate the model distance between \( s_i \) and \( s_j \) using

\[
\hat{\lambda}_{i,j} = -\frac{3}{4} \ln(1 - 4 \frac{H(i,j)}{3k}),
\]

where \( H(i,j) \) denotes the Hamming distance between \( s_i \) and \( s_j \). Furthermore, \( \hat{\lambda}_{i,j} \) converges, as \( k \to \infty \), to \( \lambda_{i,j} \). This is called the Jukes-Cantor distance correction.

Computing distances for the General Markov model: The calculation of pairwise distances for the General Markov model is more complicated than for the Jukes-Cantor or CFN models. For a given pair of sequences \( s_i \) and \( s_j \) that have evolved down a General Markov model tree \( T \), let \( \hat{f}_{ij}(\alpha, \beta) \) denote the relative frequency of \( s_i \) having state \( \alpha \) and \( s_j \) having state \( \beta \). Then let \( F_{ij} \) denote the \( 4 \times 4 \) matrix (with rows and columns indexed by the four possible character states) using the \( \hat{f}_{ij}(\alpha, \beta) \) values for its entries. Let \( d_{ij} = -\log \det(F_{ij}) \). Then the matrix \( d \) converges as the sequence length increases to an additive matrix for \( T \). This is called the “logdet” distance calculation Steel (1994a). Note that the matrix that this logdet distance calculation converges to, although additive on the true tree, is not the matrix of expected numbers of changes for a random site between each pair of sequences. This makes the logdet distances different from the Jukes-Cantor distance, and from the Cavender-Farris-Neyman distance.

8.4.2 Statistical properties of distance-based methods

In Chapter 5, we presented a collection of methods that can construct trees when given distance matrices as input. Examples of such methods included the Naive Quartet Method, Neighbor Joining, the method of Agarwala et al. for 3-approximating the \( L_\infty \)-nearest tree, and the Dyadic Closure Method. Each of these methods is robust - to varying extents - to
error in the estimated distance matrix, in the following sense. For each of these methods, there is a positive safety radius $\delta$, that can depend on the method and the model gene tree, so that when the input distance matrix $d$ satisfies $L_{\infty}(d, D) < \delta$ where $D$ is an additive matrix defined by an edge-weighting of the model gene tree $T$, then the method is guaranteed to return the tree $T$.

The existence of a positive safety radius (i.e., that $\delta > 0$) for a distance-based method $M$ ensures that $M$ is statistically consistent under any sequence evolution model for which estimated pairwise distances converge in probability to the model pairwise distances as the sequence length increases. Thus, since the estimated distances computed using the CFN distance calculation converge in probability to the model CFN distances, each of these methods is statistically consistent under the CFN model. The same statement is true for the General Markov Model, since logdet distances converge in probability to additive matrices for the General Markov model tree. And, since the General Markov Model contains all the standard DNA sequence evolution models (including the Jukes-Cantor, Kimura 2-parameter, and other models), this means that the distance-based methods are statistically consistent under the standard DNA sequence evolution models.

The safety radius for neighbor joining and the Naive Quartet Method are both $f^2$, where $f$ is the length of the shortest internal edge in the model tree $T$. Thus, if the input distance matrix is $\hat{\lambda}$ and $\lambda$ is an additive matrix for the model tree with $f$ the length of shortest internal edge, then whenever $L_{\infty}(\hat{\lambda}, \lambda) < f^2$ the Naive Quartet and the neighbor joining methods applied to matrix $d$ will both return tree $T$. In other words, exact accuracy is guaranteed for both the Naive Quartet Method and neighbor joining, whenever the matrix of estimated distances is within $f^2$ of the additive matrix for the model tree.

The safety radius for the Agarwala et al. algorithm is $f^8$. Just as for the Naive Quartet method and for neighbor joining, this means that the Agarwala et al. algorithm is guaranteed to return the true tree whenever the matrix of estimated distances is within $f^8$ of the additive matrix for the true tree. Although this guarantee suffices to establish statistical consistency for the Agarwala et al. algorithm, it is a stricter condition than was established for neighbor joining and the Naive Quartet Method to be accurate.

This analysis shows that all three methods are statistically consistent under any statistical model of sequence evolution (such as the General Markov Model) for which estimated distances converge in probability to an additive distance matrix for the true tree. In other words, to prove that a distance-based method is statistically consistent under a model, you only need to show that it has some tolerance to errors in the estimated distance matrix, and that the model distances can be estimated in a statistically consistent manner. The first condition (error tolerance in estimated distances) depends on the distance method, but the second condition (that estimated distances converge to model tree distances) depends on the sequence evolution model.
8.5 Calculating the probability of a set of sequences on a model tree

The models of evolution that we have been working with all assume that the sites evolve independently and identically down some model tree. Hence, the probability of a set of sequences given in a multiple sequence alignment, and without any gaps in the sequences, is just the product over all the sites of the probability of the pattern for that site. Furthermore, the models we have discussed are time-reversible; hence, the location of the root has no impact on the result of the calculation.

This simple observation allows us to compute the probability of a sequence dataset given a model gene tree. For example, in the context of the Jukes-Cantor model, the input would be a Jukes-Cantor model tree, which would have a binary tree topology $T$ and the probability of change on each edge; more complex models would require additional parameters, such as a $4 \times 4$ DNA substitution matrix governing the tree.

Now, suppose we have just a single site. Then, if we look at all the possible assignments of states to the internal nodes of the tree, we can compute the probability of those states arising under the model tree, and add up those probabilities. This calculation uses brute-force to correctly calculate the probability of the single site on the tree. For a model tree with $n$ leaves, there are $n-1$ internal nodes in the tree (since the model tree is rooted and binary). Hence, assuming the characters have $r$ states (e.g., $r=2$ for the CFN model and $r=4$ for the Jukes-Cantor and GTR models), there are $r^{n-1}$ ways of assigning states to each of the internal nodes. Then, for each of these $r^{n-1}$ possible assignments, the probabilities of all the state changes on the edges must be calculated and multiplied together, each calculation producing the probability of observing exactly those states at all the nodes of the tree. Finally, each of these $r^{n-1}$ values must be added together. Overall, this approach, although it works, is very expensive, even for the simplest case where $r=2$.

Felsenstein (1981) introduced a dynamic programming algorithm to compute the probability of a single site on a model tree, much along the lines of the dynamic programming algorithm used for score a tree with respect to maximum parsimony. This algorithm, often referred to as “Felsenstein’s Peeling Algorithm”, runs in $O(r^2n)$ algorithm to compute the probability of a single site for a given model tree with $n$ leaves and where the characters have $r$ states, under the assumption that the calculation of any state transition can be performed in $O(1)$ time. Hence, for $k$ sites, the running time is $O(knr^2)$. This means the calculating the probability of binary sequences on a given CFN model tree requires $O(kn)$ time, and so does calculating the probability of DNA sequences on a given GTR model tree.

8.6 Maximum Likelihood

Maximum likelihood phylogeny estimation is a very popular technique used for phylogeny estimation in systematics. We present this approach in the context of estimating trees under the Cavender-Farris-Neyman (CFN) model.

Recall that a CFN model tree consists of a rooted binary tree $T$ and the numerical pa-
rameters for the model, which are just the probabilities of change on the edges of the tree. Thus, a CFN model tree is described as a pair \((T, \theta)\), where \(\theta\) is the set of substitution probabilities on the edges of the tree.

Under the CFN model, maximum likelihood estimation takes as input a set \(S\) of sequences, each of the same length, and seeks the CFN model tree \((T^*, \theta^*)\) that maximizes \(Pr(S|T^*, \theta^*)\), the probability of generating the observed sequence data \(S\).

Because the CFN model is time-reversible, moving the root from one edge to another edge within a tree \(T\) does not change the probability of generating the observed data, and hence there are always multiple optima that at a minimum can differ by the location of the root. For this reason, rather than returning a rooted model tree, the result of a maximum likelihood analysis is an unrooted model tree. (This is also the reason that other phylogeny estimation methods, except for a few distance-based methods that assume a molecular clock, return unrooted trees.)

Note that we have described ML under the CFN model, but that the description applies therefore to any of the statistical models of evolution we have discussed so far. Thus, ML under the Jukes-Cantor (JC69) model would seek the JC69 model tree \((T, \theta)\) that maximized the probability of generating the observed sequence data, and similarly ML under the Generalized Time Reversible (GTR) model would seek the GTR model tree that maximized the probability of generating the observed sequence data. The only thing that changes as we substitute one model for another in these different formulations is what we mean by \(\theta\). For the CFN and JC69 models, these are only the substitution probabilities; however, for the GTR model we would also seek the \(4 \times 4\) substitution matrix that governs the evolution across the tree. Similarly, for the General Markov (GM) model, we would seek the substitution matrices for each edge. Even more complex models (that go beyond the GM model) can be considered, and ML can be extended to address estimation under these models.

Since ML requires the estimation of all the model parameters, as the model becomes more parameter-rich, ML estimation becomes more computationally intensive. Indeed, there is also the possibility of over-fitting if there are an excessive number of model parameters. Furthermore, while the maximum likelihood scores may be distinct for different trees, the differences in scores between different trees are sometimes small enough to not be considered significant; for this reason, the problem of finding the optimal as well as the near-optimal trees is sometimes the objective. Finally, even though maximum likelihood is statistically consistent for estimating phylogenetic trees under the CFN model, it does not follow that the ML tree will be the true (model) tree on any given dataset.

**Computational complexity of Maximum Likelihood.** Maximum likelihood (ML), if run exactly (so that optimal solutions are found), is statistically consistent under the JC69, GTR, and GM models (Chang, 1996b). However, finding an optimal ML tree is an NP-hard problem (Roch, 2006), and so heuristics are used instead of exact solutions. As with any NP-hard phylogeny estimation problem, because the number of trees on \(n\) leaves is exponential in \(n\), examining all tree topologies is infeasible, and heuristic searches are
used to explore tree space. However, heuristic searches for maximum likelihood are more complicated than heuristic searches for maximum parsimony, because even scoring a tree is computationally intensive! That is, while Felsenstein’s Peeling Algorithm provides a way to compute the probability of a set of sequences given a model tree, this assumes that we know the set \( \theta \) of numeric parameters for the model tree. As it turns out, the estimation of numeric parameters on a tree is also addressed heuristically, and may not find global optima (Steel, 1994b). Hence, ML phylogeny estimation is complicated and computationally intensive, even for the simplest sequence evolution models.

### 8.7 Bayesian methods

Bayesian methods are similar to maximum likelihood methods in that they also calculate likelihoods of trees based upon explicit parametric mathematical models of evolution. However, while maximum likelihood methods are trying to find the best model tree (the one that has the largest probability of generating the observed data), Bayesian methods try to sample from the distribution of model trees proportionally to their likelihoods. The result is that Bayesian methods produce a set of trees rather than a single tree, and they use this set to evaluate the statistical support for different evolutionary hypotheses. However, Bayesian methods are even more computationally intensive in practice than maximum likelihood, with the result being that they are rarely used on very large datasets.

Bayesian methods are usually implemented using Markov Chain Monte Carlo (MCMC) techniques. In this type of approach, they perform a “random walk” through model tree space (where the tree and the associated parameters of evolution are provided), by computing the probability of producing the observed data for each model tree it visits. The probability of accepting the new model tree depends on whether the probability has increased or decreased, but if it is accepted then the MCMC chain continues from the new model tree.

The Bayesian MCMC walk is guaranteed to converge to the stationary distribution on model trees, as long as the MCMC walk satisfies the “detailed balance property”, which essentially says that \( \pi(i)p_{ij} = \pi(j)p_{ji} \), where \( \pi(x) \) is the probability of state \( x \) and \( p_{xy} \) is the probability of moving to state \( y \) from state \( x \) (here, \( x, y, i, \) and \( j \) all represent model trees). Thus, the design of Bayesian methods requires some care, both in terms of how model trees are visited and the probabilities of accepting a new model tree, in order for the detailed balance property to hold. However, if the detailed balance property holds and if the MCMC walk lasts long enough, then it will start sampling model tree topologies according to their true probabilities, as defined by the sequence evolution model and observed data.

After the MCMC chain has run long enough to reach stationarity, a sample of the model trees it visits is then taken from the trees visited after the burn-in period (i.e., after the chain has run long enough to reach the stationary distribution). If a point estimate of the tree topology is desired, then a summary of the distribution may be computed using several different techniques. For example, a majority consensus tree of the trees in the sample can be computed. In a Bayesian analysis, the frequency in which a tree topology appears in the
sample is its posterior probability; hence, another common summary statistic is the maximum a posteriori (MAP) tree, which is the tree topology that appears the most frequently in the sample (Rannala and Yang, 1996). A third approach assembles a set of tree topologies, starting with the most frequently observed trees and then decreasing, until a desired threshold is achieved (e.g., where 95% of the probability distribution is in the set), and then computes a point estimate based on this subset of trees. These point estimates can then be used as an estimated phylogeny, and the support for the branches of the phylogeny can similarly be computed based on the observed distribution. Finally, if the Bayesian method is run properly (i.e., so that it reaches the stationary distribution), then its point estimates will be statistically consistent under the Jukes-Cantor model as well as under more general models, such as the General Time Reversible model (Steel, 2010).

8.8 Statistical properties of Maximum Parsimony and Maximum Compatibility

So far we have shown that distance-based methods (e.g., the Naive Quartet Method) can be statistically consistent under some stochastic models of evolution such as the Cavender-Farris-Neyman model, the Jukes-Cantor model, and the General Time Reversible Model. We also noted (although we did not prove) that maximum likelihood and Bayesian methods are also statistically consistent under these models. We now turn to the maximum parsimony and maximum compatibility problems, and their statistical properties under stochastic models of sequence evolution. Are these statistically consistent under any of these models of evolution?

Here the story is not so positive. Felsenstein (1978) gave an example of a very simple four-leaf model tree on which maximum parsimony would be inconsistent. Worse, maximum parsimony would converge to the wrong tree as the number of sites increased. This result also applied to maximum compatibility, so that on this tree maximum compatibility would also converge to the wrong tree. This negative example is a cautionary note about these two methods, and had a very substantial impact on biologists’ choice of which tree estimation method to use to analyze their datasets.

**Brief sketch of Felsenstein’s proof that maximum parsimony is inconsistent.** Consider the CFN model tree with tree topology uv|wx, where the probability of change on the edges incident with u and w are both very large (say, 0.49999) and all other edges have very low probabilities of change (say, 0.00001). We describe this by saying that the tree has three very short branches (where \( p(e) \) is very small) and two very long branches (where \( p(e) \) is very large).

Recall that we only need to consider the parsimony informative sites when calculating the maximum parsimony tree(s). For a four-taxon tree, the parsimony informative sites have to split two/two. Hence, every parsimony informative site must have one of the following forms:

- Case 1: \( u = v = 0 \) and \( w = x = 1 \) or \( u = v = 1 \) and \( w = x = 0 \).
8.9 The No Common Mechanism Model

- Case 2: \((u = w = 0 \text{ and } v = x = 1)\) or \((u = w = 1 \text{ and } v = x = 0)\)
- Case 3: \((u = x = 0 \text{ and } v = w = 1)\) or \((u = x = 1 \text{ and } v = w = 0)\)

There are only three possible unrooted binary trees on \(u,v,w,x\), given by 
\[T_1 = uv|wx, T_2 = uw|vx, \text{ and } T_3 = ux|vw.\]
Note that any parsimony informative site that is in Case 1 will have cost 1 on \(T_1\) and cost 2 on the other trees. Hence, to understand how maximum parsimony will behave on this model tree, all we need to know is which parsimony informative site has the highest probability of occurring under this CFN model tree. Since all but two edges have very low substitution probabilities, it is easy to see that the most probable parsimony informative sites will have changes on both of the long edges and no changes on the short branches. The probability of such an event is close to 1/4. In contrast, to produce a different parsimony informative pattern would require a change on at least one short branch, which would have much lower probability. Hence, the parsimony informative pattern with the highest probability will have changes on the edges incident with \(u\) and \(w\) and no changes elsewhere, and so will be in Case 2. Therefore, the tree that maximum parsimony will return with probability increasing to 1 as the number of sites increases is \(T_2\). However, the model tree is actually \(T_1!\) This means that maximum parsimony is not only statistically inconsistent, but also that it is positively misleading, under this CFN model.

It is trivial to see that maximum compatibility behaves identically to maximum parsimony on four-taxon CFN trees, and so is also positively misleading under this CFN tree. Therefore, both maximum parsimony and maximum compatibility are positively misleading for some CFN model trees.

Note however that this does not mean that they are positively misleading for all CFN model trees! For example, both maximum parsimony and maximum compatibility will be statistically consistent under a CFN model tree \(uv|wx\) where the internal edge is very long and all other edges are very short, because the most probable parsimony informative sites will support the model tree. Therefore, statistical inconsistency is only a statement that there are some model conditions in which the method is not statistically consistent. Also note that establishing statistical consistency or inconsistency requires a mathematical proof, and cannot be established using a simulation.

8.9 The No Common Mechanism Model

In most of the models of DNA sequence evolution we have discussed so far, there is a single substitution matrix \(M\) that governs the entire tree. This assumption is relaxed in the General Markov Model, in which each edge has its own substitution matrix. However, the variation between different sites for these models is constrained by having each site draw its rate from a distribution. What this means, in essence, is that if site \(i\) evolves twice as quickly as site \(j\) on one branch of the tree, then it evolves twice as quickly on every branch of the tree. Violations of this assumption are referred to as heterotachy (although heterotachy can take other forms as well).
Tuffley and Steel (1997) generalized this further, by assigning an independent substitution matrix to each combination of edge and site. Under this model, the evolutionary process on every edge and site is independent of what happens on any other edge and site. Under the assumption that all character state changes are equiprobable (as in the CFN and Jukes-Cantor models) and that the state at the root is selected at random, the evolutionary process is fully specified by the substitution probabilities $p_{e,i}$ that specify the probability of a change on edge $e$ for site $i$, with $0 \leq p_{e,i} \leq \frac{r-1}{r}$. Tuffley and Steel called this the No Common Mechanism Model (NCM).

**Theorem 8.1** (From Tuffley and Steel (1997)) Let $S$ be a set of $r$-state sequences. Then tree $T$ is an optimal maximum likelihood tree under the No Common Mechanism Model if and only if $T$ is an optimal solution to maximum parsimony.

Note that the NCM model for binary sequences contains the CFN model as a special case. Now suppose a Felsenstein Zone model CFN tree $(T, \Theta)$ for which maximum parsimony will converge to the wrong tree topology. If we estimate ML trees on data generated by $(T, \Theta)$ under the No Common Mechanism (NCM) model, then on every dataset we examine we will return the MP tree. Since MP is positively misleading under $(T, \Theta)$, ML under the NCM model will also be positively misleading. In other words, Tuffley and Steel (1997) showed

**Theorem 8.2** Maximum likelihood under the NCM model is not statistically consistent, and can be positively misleading.

### 8.10 Beyond statistical consistency: performance on data

As we have noted, some phylogeny estimation methods are statistically consistent under the General Time Reversible (GTR) model, and others are not. For example, the Naive Quartet Method is statistically consistent under the GTR model, but maximum parsimony is not. On the other hand, even maximum likelihood is not statistically consistent under the NCM model. Thus, statistical consistency depends on the model of evolution – so that with sufficiently complex models, even good techniques may not have the desired property of being statistically consistent.

Saying that a method is statistically consistent says nothing in essence about how well it will perform on data, since you don’t know how much data it needs to return the true tree with high probability. Thus, another issue of practical importance is the amount of data that a method needs to reconstruct the true tree with high probability. This issue is often estimated using simulation, but mathematical analyses of this question have also been performed.

A good example of a study comparing methods on finite data is the “Hobgoblin of Phylogenetics” paper (Hillis et al., 1994), which presented model conditions and sequence lengths under which some statistically inconsistent methods were more accurate than some
8.11 Estimating branch support

A very common concern is figuring out how reliable each branch is within the tree $T$ computed on a sequence alignment. Depending on the method for performing the phylogenetic analysis, different approaches can be used to assess branch support. However, one basic approach that can be used with any phylogenetic estimation method is non-parametric bootstrapping.

Here, the input sequence alignment is used to generate a large number (e.g., 100 or more) “bootstrap replicate” datasets. A bootstrap replicate is a data matrix with the same dimensions of the original matrix, but where the columns of the bootstrap replicate are obtained by sampling with replacement from the original data matrix. As a result, some columns from the original matrix will appear not at all, some will appear exactly once, and others will appear more than once. After the bootstrap replicate datasets are obtained, a phylogeny is estimated on each bootstrap replicate dataset using the same method as was used to estimate a tree on the original dataset. This produces a set of bootstrap trees (one per replicate dataset).

The set of bootstrap trees can be used in different ways to explore support (or, conversely, uncertainty). The most common use of bootstrapping is to characterize the support for each edge in the tree $T$. Specifically, if an edge $e$ in $T$ defines a bipartition $A|A'$ (where $A' = S \setminus A$) on the leafset $S$, then we look at the bootstrap trees and determine the fraction of those trees that also have edges defining the same bipartition; that fraction is the bootstrap support for the edge $e$.

The interpretation of bootstrap support values is complicated, since high bootstrap support may not indicate high probability of accuracy, but in general edges with low support values (below 50%) are considered unreliable, and edges with support values above 95% are considered reliable. In the middle region, where support values are between 50% and 95%, opinions differ as to the reliability of edges. Bootstrap support is commonly used when the phylogenetic analysis method is maximum parsimony, maximum likelihood, or distance-based methods such as neighbor joining. However, when the phylogenetic analysis is a Bayesian MCMC method, then another technique is typically used. Recall that the Bayesian MCMC methods operate by performing a random walk through tree space, and a random sample of the model trees that are visited (after “burn-in”) is saved. That set of trees is then used to produce a distribution on tree topologies (i.e., what fraction of the model trees in the set have a particular tree topology), as well as a distribution on bipartitions (i.e., what fraction of the model trees in this set have a particular bipartition). If a single point estimate tree is desired, then typically the maximum \textit{a posteriori} (MAP) tree is returned, which is the tree topology that appears most frequently in the set; however, a consensus tree (e.g., a majority consensus or a greedy consensus) is also sometimes returned. The branch
supports on the tree are obtained by using the percentage of the trees in the set that induce the same bipartition, and are called the \textit{posterior probabilities} for each edge. Note the similarity between how branch support is computed for both techniques – bootstrapping and Bayesian MCMC – the only difference is how the set of trees is computed.

In general, Bayesian support values (posterior probabilities) tend to be higher than bootstrap support values, so that interpreting branch support needs to take this into consideration.

\section*{8.12 Other gene tree estimation problems}

So far we have talked about how to estimate a single tree from a set of sequences. However, in practice biologists want to know much more than just the best tree, what they can have confidence in, what the competing hypotheses might be, and have some sense for why there is uncertainty in the estimation (if possible).

As we have just discussed, branch support can be defined using various techniques. More generally, the distribution on tree topologies that is produced by either bootstrapping or the posterior distribution generated during the Bayesian MCMC walk through tree space can be used to obtain support for various topology-based hypotheses, such as whether a given set of species is monophyletic, or whether a given set of four species has a particular quartet tree.

However, biologists may wish to be able to estimate other model tree parameters, such as the branch lengths, the location of the root, the GTR substitution matrix, etc. Or they may wish to estimate dates at internal nodes, and possibly even infer character states at the internal nodes. Here we discuss how these estimations are performed.

\textit{Estimating branch lengths:} Methods like maximum likelihood and distance-based methods such as neighbor joining, which produce estimates of the numerical model parameters (which are usually expressed in terms of lengths), automatically generate branch length estimates.

\textit{Inferring ancestral states:} A fundamental problem of great interest is estimating the characteristics of the ancestral taxa. Recall that the output of the phylogenetic analysis is a tree (sometimes rooted, but not always), with sequences labelling the leaves, and sometimes also branch lengths. The internal nodes of the tree represent ancestral species, for which we do not have any sequence data. To infer sequences for these ancestral species, various techniques can be applied.

In parsimony analyses, these ancestral sequences are inferred so as to minimize the total number of changes; note that in some cases, there is a unique state for a given position within the alignment at a given node, but often there a multiple states that can be assigned without increasing the parsimony score.

In a maximum likelihood analysis, the estimation of ancestral sequences is performed
probabilistically, with respect to the estimated model parameters. Thus, the maximum likelihood analysis can be used to infer the probability of each nucleotide for each position in the sequence alignment for each node in the tree. In general, the inference of character states is easier close to the leaves than far from the leaves.

Locating the root: The stochastic models we have discussed so far are time-reversible, which has the consequence that it is not possible to determine the location of the root. However, when a molecular clock is assumed to hold, then the root location can be determined from the sequence data (in the sense that as the sequence length goes to infinity, then with high probability the correct location of the root can be estimated).

Under models of trait evolution, however, sometimes there is additional information about how the trait evolves that makes it possible to identify the location of the root. In an extreme case, the “ancestral” state of the trait is known and homoplasy (including parallel evolution and back mutation) is forbidden. For example, suppose we have five taxa, $A, B, C, D, E$, and three characters on taxa, given by:

- $A = (0, 0, 0)$
- $B = (0, 1, 0)$
- $C = (0, 0, 1)$
- $D = (1, 0, 1)$
- $E = (1, 0, 1)$

If we assume that 0 is the ancestral state, and 1 the derived state, and there is no homoplasy in these characters, then the set of possible rooted trees consistent with these data is quite limited. In particular, the unrooted tree topology must be $(A, (B, (C, (D, E))))$, and there are only two edges in the tree that could contain the root.

More generally, however, even with morphological traits, knowing the ancestral form is often difficult, and ensuring that there is no homoplasy is even more difficult; many seemingly homoplasy-free characters have been revealed to have evolved in parallel.

Hence, the process of inferring the location for the root generally operates in the following way: first, a tree is estimated for the dataset, and then the feasible locations for the root are determined. The first step (estimating the tree) typically does not use any assumption about a molecular clock, and instead is often just a maximum likelihood analysis (or similar approach) that produces an unrooted tree. For the second step, one frequent approach is to use the “midpoint” rooting, whereby the longest leaf-to-leaf path in the tree is computed (using branch lengths produced in the first step to compute the leaf-to-leaf path lengths), and the tree is rooted on the edge containing the midpoint of the longest path. This approach in a sense inherently attempts to minimize the deviation from the molecular clock.

Another approach to rooting an unrooted tree is to include a carefully selected outgroup in the dataset. For example, if the primary interest is estimating the tree on primates, then using a non-primate mammal that is closely related can be helpful. For example, if a tree is
Statistical gene tree estimation methods

estimated on the primates plus the rodent, then the tree can be rooted on the branch leading to the rodent.

Estimating dates at internal nodes: Sometimes researchers are particularly interested in estimating dates at internal nodes of the tree. To do this, the first step involves estimating a tree and its branch lengths, two tasks that are relatively reasonably well addressed. The second step involves combining those branch length estimations with dates at certain nodes of the estimated tree provided by external evidence.

8.13 Further reading

8.14 Review questions

1. State the CFN model. What is a CFN model tree?
2. Suppose we have a CFN model tree with branch substitution probabilities given by $p(e)$, as $e$ ranges over the branches of the tree. Show how to define $\lambda(e)$.
3. What is the safety radius of neighbor joining?
4. State the Jukes-Cantor (JC69) model. What is a JC69 model tree? What is the JC69 distance correction?
5. State the Generalized Time Reversible (GTR) model.
7. What is the relationship between the GM, GTR, and JC69 models?
8. What is the logdet distance correction? What models is it applicable to?
9. What is meant when we say that a method $M$ is statistically consistent for estimating a GTR model tree?
10. Can statistical consistency be proven using a simulation study?
11. What is meant when we say that a method $M$ is statistically inconsistent for estimating a GTR model tree?
12. Can statistical inconsistency be proven using a simulation study?
13. What is meant by saying that a method $M$ is positively misleading for estimating a GTR model tree?
14. Suppose you are given a JC69 model tree and a set of sequences at the leaves. What is the computational complexity of computing the probability of the sequences at the leaves?
15. What is the JC69 Maximum Likelihood problem? What is its computational complexity?
16. Comment on the use of ML under the GTR model to analyze data that have evolved under the JC69 model. Is this statistically consistent?
17. Is maximum parsimony statistically consistent under the JC69 model? If not, what does it mean to say this?
18. What is the No Common Mechanism (NCM) model for binary sequences? Is maximum parsimony statistically consistent under the binary sequence NCM model? Is maximum likelihood statistically consistent under the binary sequence NCM model?
19. What is the relationship between the NCM and CFN models?
20. What techniques are used to compute branch support on estimated trees?

8.15 Homework problems

1. Prove or disprove: Every GTR model tree is a JC69 model tree.
2. Prove or disprove: Every JC69 model tree is a GTR model tree.
3. Prove or disprove: Every CFN model tree is a JC69 model tree.
4. Prove or disprove: Every GM model tree is a GTR model tree.
5. Prove or disprove: Every GTR model tree is a GM model tree.
6. True or False? If a method M is statistically consistent under the JC69 model, then it is also statistically consistent under the GTR model.
7. Consider the following algorithm for estimating JC69 model trees from sequence data. Given a set of sequences, we compute JC69 distances for the sequences. We then check to see if the distance matrix is additive; if it is, we return the tree $T$ corresponding to the additive distance matrix, and otherwise we return a random tree. Prove or disprove: this is a statistically consistent method under the JC69 model.
8. Suppose we are given sequence dataset $S$ generated by an unknown Jukes-Cantor model tree. We compute logdet distances, and then run neighbor joining on the distance matrix we obtain. Is this a statistically consistent method?
9. Suppose we are given sequence dataset $S$ generated by an unknown GTR model tree. We compute Jukes-Cantor distances, and then run neighbor joining on the distance matrix we obtain. Is this a statistically consistent method?
10. Suppose you have the CFN tree $T$ with topology $((A,B),(C,D))$ with every edge having $p(e) = 0.1$, and rooted at $A$ (note that in this tree, the root has only one child).
   - Compute the probability that $B = C$.
   - Compute the probability that $A = C$.
11. For the same CFN tree as in the previous problem, compute the probability that $A = B = C = D = 0$.
12. Consider a CFN model tree $T$ with topology $((A,B),(C,D))$. Treat this as a rooted tree, with $A$ being the root, and thus having five edges. Suppose the internal edge is labelled $e_I$, and we set $p(e_I) = 0.1$, and $p(e) = 0.001$ for all the other edges. Compute the probability of the following events:
   - $A = B = 0$ and $C = D = 1$
   - $A = C = 0$ and $B = D = 1$
   - $A = D = 0$ and $B = C = 1$
13. Let \( S = \{ s_1, s_2, \ldots s_n \} \) be a set of binary sequences of length \( k \), and let \((T, P)\) be a rooted CFN tree on the same leafset, where \( T \) is the model tree topology and \( P \) is the set of substitution probabilities on the edges. Which of the following is the correct running time of the dynamic programming algorithm for computing the \( \Pr(S|(T, P)) \)?

- \( \Theta(n^2k) \)
- \( \Theta(2^n k) \)
- \( \Theta(2^k n) \)
- \( \Theta(n^2 k) \)

14. In this problem we will define a set of different CFN model trees on the same tree topology, \(( (A, B), (C, D) )\) but with different edge parameters. We let \( e_I \) be the internal edge separating \( A, B \) from \( C, D \), and let \( e_x \) be the edge incident with leaf \( x \) (for \( x=A, B, C, D \)). The trees are then defined by the edge parameters \( p(e) \) for each of these edges, with these \( p(e) \) given as follows:

- For \( T_1 \), we have \( p(e_A) = p(e_C) = .499 \), and \( p(e) = 0.0001 \) for the other edges \( e \).
- For \( T_2 \), we have \( p(e_I) = .499 \) and \( p(e) = .01 \) for the other edges \( e \).

Think about what kinds of character patterns you would see at the leaves of the trees, and answer the following questions:

1. Of the three parsimony-informative character patterns, identify which one(s) would appear most frequently for tree \( T_1 \).
2. Of the three parsimony-informative character patterns, identify which one(s) would appear most frequently for tree \( T_2 \).
3. For each of these model trees, do you think maximum parsimony would be statistically consistent? Why?
4. For each of these model trees, do you think UPGMA on CFN distances would be statistically consistent? Why?
5. For each of these model trees, do you think neighbor joining on CFN distances would be statistically consistent? Why?

15. Consider CFN model trees, all with the same tree topology, \(( (A, B), (C, D) )\), but with different edge parameters. We let \( e_I \) be the internal edge separating \( A, B \) from \( C, D \), and let \( e_x \) be the edge incident with leaf \( x \) (for \( x=A, B, C, D \)). The trees are then defined by the edge parameters \( p(e) \) for each of these edges, with these \( p(e) \) given as follows:

- For \( T_1 \), we have \( p(e_A) = p(e_C) = .499 \), and \( p(e) = 0.0001 \) for the other edges \( e \).
- For \( T_2 \), we have \( p(e_I) = .499 \) and \( p(e) = .01 \) for the other edges \( e \).
- For \( T_3 \), we have \( p(e) = .499 \) for all edges \( e \).
- For \( T_4 \), we have \( p(e) = .0001 \) for all edges \( e \).

1. Suppose one of these CFN trees generated a dataset of four sequences, and you had to guess which one generated the data. Suppose the dataset consisted of four sequences \( A, B, C, D \) of length 100 that were all identical, which would you choose?
2. Same question as above, but suppose the dataset consisted of four sequences \( A, B, C, D \) of length 10, where
8.15 Homework problems

- \( A = 0100100111 \)
- \( B = 0000000000 \)
- \( C = 0010101001 \)
- \( D = 0000000000 \)

16. Suppose we are given sequence dataset \( S \) generated by an unknown Jukes-Cantor model tree, and we analyze the sequences using GTR maximum likelihood (solving the problem exactly). Will this be a statistically consistent method? (More to the point, if we estimate the tree under GTR using a statistically consistent method for GTR, such as maximum likelihood, but the data are generated by a JC69 model tree, is this a statistically method?)

17. Recall the Cavender-Farris-Neyman (CFN) model, and consider three methods: maximum likelihood under CFN, maximum parsimony, and UPGMA on CFN distances.
   1. Consider invariant characters (i.e., characters that assign the same state to all the taxa). For each of the methods given above, say whether the invariant characters are informative, and explain your reasoning.
   2. Consider characters that are different on every taxon. For each of the methods above, say whether these characters are informative, and explain your reasoning.

18. Consider a CFN model tree \( T \) given by \( ((A,B),(C,D)) \). Treat this as a rooted tree, with \( A \) being the root, and thus having five edges. Suppose the internal edge is labelled \( e_I \), and we set \( p(e_I) = .4 \), and \( p(e) = 0.001 \) for all the other edges. Would maximum parsimony be statistically consistent on this model tree? Why?

19. Consider the following type of character evolution down a rooted binary tree \( T \), in which every node is labelled by a unique integer (which may be positive, negative, or zero); note this means that in a tree with \( n \) leaves, there are \( 2n - 2 \) distinct labels. We do not assume that the label of a node is larger or smaller than its parent node, but we do assume that the label at the root is 0. The state of the character at the root is always 0. Every edge \( e \) in the tree \( T \) has a substitution probability \( p(e) \) with \( 0 < p(e) < 1 \). On an edge \( e = (x,y) \), with \( x \) the parent of \( y \), the character changes its state with probability \( p(e) \); if it changes state, then the new state is \( y \). As with other models we’ve studied, if there are multiple sites that evolve down the same tree, we assume that the substitution probabilities \( p(e) \) govern all the sites, but can differ between edges. We also assume that the labels at the nodes are part of the model tree, and so are the same for all characters that evolve down the tree.
   1. Suppose the rooted model tree \( T \) has topology \( (a,(b,c)) \). Let the parent of \( b \) and \( c \) be labelled by 3, and let \( a \) be labelled by 5, \( b \) be labelled by 2 and \( c \) be labelled by 4. Recall that the root is always labelled by 0.
      - Suppose that a character evolves down this model tree but never changes its state. What are the character states at the leaves \( (a,b,c) \) for this character?
      - Suppose that the character evolves down this model tree and changes exactly once - on the edge from the root to \( a \); what are the character states at the leaves for this character?
Statistical gene tree estimation methods

Suppose the character evolves down this model tree and changes exactly once - on the edge from the root to the parent of b and c. What are the character states at the leaves for this character?

Suppose the character evolves down this model tree and changes state on every edge of the tree. What are the character states at the leaves of the character?

2. Suppose the following four sequences evolve down some unknown model tree of this type:
   - \( u = (3, 0, 1) \)
   - \( v = (3, 0, 5) \)
   - \( w = (0, 8, 2) \)
   - \( x = (0, 8, 4) \)

   What is the tree topology, and what are the labels at the nodes of the tree? ( Recall we already know that the root label is 0.)

3. Suppose the following five sequences evolve down some unknown model tree of this type:
   - \( A = (4, 2, 0, 3, 1) \)
   - \( B = (4, 2, 0, 3, 6) \)
   - \( C = (0, 2, 0, 3, 7) \)
   - \( D = (0, 0, 0, 3, 8) \)
   - \( E = (0, 0, 5, 5, 9) \)
   - \( F = (0, 0, 5, 5, 10) \)

   What is the tree topology, and what are the labels at the nodes of the tree?

4. Suppose the following three sequences are given to you. Is it possible that they evolve down some unknown model tree of this type?
   - \( A = (4, 0) \)
   - \( B = (4, 2) \)
   - \( C = (0, 2) \)

   If so, present the tree; otherwise prove this cannot be the case.

5. Describe a polynomial time statistically consistent method to infer the model tree topology from the site patterns. What is the running time of your algorithm? (Don’t just say “polynomial”.) What is your justification for saying it is statistically consistent under this model?
Multiple sequence alignment

9.1 Introduction

Since tree estimation is typically done using methods that explicitly refer to sequence evolution models based only on substitutions, the first step in a phylogenetic estimation generally begins by estimating a multiple sequence alignment (MSA) on the set of sequences. Once the MSA is computed, a tree can then be computed on the alignment; see Figure 9.1 for this two-phase process.

As we have seen, there are many methods for estimating trees from gap-free data. However, because multiple sequence alignments almost always contain dashes, phylogeny estimation methods must be modified to be able to analyze alignments with dashes. Typically this is performed by treating the dashes as “missing data”, but sometimes the dashes are treated as an additional state in the sequence evolution model, or sites that contain dashes are eliminated from the alignment before a tree is computed. The different treatments of sequence alignments can result in quite different theoretical and empirical performance.

The focus of this chapter is how we estimate MSAs from unaligned sequences. As we will see, MSA estimation is challenging, especially for large datasets. Furthermore, errors in multiple sequence alignments can result in errors in downstream analyses, such as phylogeny estimation, and so how the sequence alignment is computed is important. In this chapter, we describe the techniques that are used to produce multiple sequence alignments, and the impact multiple sequence alignment has on phylogeny estimation.

9.2 Evolutionary history and sequence alignment

The standard models of sequence evolution (e.g., the Jukes-Cantor model, the Generalized Time Reversible Model, and the General Markov Model) all describe the evolutionary process as exclusively involving substitutions, so that the length of the sequence does not change over evolutionary time. Yet, biological datasets typically have sequences of different length, which clearly violates this assumption. Events such as insertions and deletions (jointly called “indels”) change the length of the sequences, and must be accounted for in a phylogenetic analysis. To explain how this is accounted for, we begin by showing how
Two-phase tree estimation:
First align, then construct the tree

S1 = AGGCTATCACCTGACCTCCA
S2 = TAGCTATCACGACCGC
S3 = TAGCTGACCGC
S4 = TCAGGACCGACA

S1
S4
S2
S3

Figure 9.1 Two-phase phylogeny estimation. In the standard two-phase approach, a multiple sequence alignment is first computed, and then a tree is computed on the alignment. There are many ways to produce multiple sequence alignments, and many ways to estimate trees on multiple sequence alignments, but this basic two-step process is the standard technique for estimating trees from unaligned sequences. Other approaches that go directly from unaligned sequences to trees have also been considered, and are discussed later in the text.

a pair of sequences that are related by evolution can be “aligned”, using the evolutionary process that separates them.

Figure 9.2 shows how one sequence evolves into another sequence through a combination of insertions, deletions, and substitutions, and the pairwise alignment that reflects the evolutionary history. Note that the true pairwise alignment depends on the true evolutionary history – and that because the true history is generally unknown, the true pairwise alignment cannot be known for sure but must instead be estimated. But, if the true history were known, then the true pairwise alignment would be defined by it.

Note that each column in the pairwise alignment of two sequences contains either one or two nucleotides. When it contains two nucleotides, then the two nucleotides are related to each other by a substitution process, which means that the two nucleotides are homologous. Note also that two nucleotides that are homologous may or may not be identical. In fact, evolution from a common ancestral nucleotide via a sequence of substitutions is referred to as “positional homology”.

Note also that two nucleotides that are homologous may or may not be identical. In fact, evolution from a common ancestral nucleotide via a sequence of substitutions is referred to as “positional homology”. 
A multiple sequence alignment is similarly a representation of the evolutionary history relating the sequences in terms of the homologies between the letters in the input sequences. In other words, two nucleotides drawn from two different sequences are homologous if they are both descended from a common nucleotide via substitutions. The challenge in inferring a multiple sequence alignment (or even a pairwise sequence alignment) is that the true evolutionary history is difficult to infer, even when it is the (supposedly) simpler task of just inferring homologies between letters in the input molecular sequences.

However, if the true evolutionary history is known, then the true multiple sequence alignment can also be inferred, as follows. Suppose that we are given the true tree, the sequences at every node in the tree, and how each sequence has evolved from its parent sequence. Because we know how each sequence evolves from its parent, we know the true pairwise alignment on each edge of the true tree. Then, if letters \(x\) and \(y\) are in the same column because of edge \(e\), and letters \(y\) and \(z\) are in the same column because of edge \(e'\), then letters \(x, y,\) and \(z\) are all in the same column. Thus, transitivity of the pairwise relations defined for each pairwise alignment gives a multiple sequence alignment.

### 9.3 Evaluating multiple sequence alignments

As with phylogeny estimation, the evaluation of multiple sequence alignments is challenging due to the inherent difficulty in knowing the true evolutionary history that relates a pair of sequences. However, we may know the true alignment because we performed a simulation study and recorded all the events, or \(A\) could be an alignment based on experimentally confirmed structural features of the molecules (e.g., secondary structures or tertiary structures of RNAs and proteins). In both cases, we will treat the alignment \(A\) as the “true” alignment, and evaluate error of estimated alignments by comparing them to \(A\).

Just as with techniques to quantify the error in tree estimation, we consider two types of error – false negatives and false positives. To do this, we represent a multiple sequence alignment by its set of homology pairs, which are any two letters that appear in the same column within the alignment. Note that we distinguish between different occurrences of the same nucleotide or amino acid, based on where it appears in the sequence. Thus, the sequence \(S = \text{ACATTA}\) has three copies of A and these are not considered identical. Now suppose that \(S' = \text{TACA}\) and that the true alignment between these \(S\) and \(S'\) is the alignment shown in Table 9.1. Suppose we have an estimated alignment, shown in Table 9.2.

Both alignments have a column with two A’s, but they aren’t using the same copies of this nucleotide. Hence, the two homology pairs (one from each alignment) are not identical. The number of true positive homology pairs is therefore just the total number of homology pairs that both alignments share. The false positive homology pairs in an estimated alignment are the ones it produces that aren’t in the true alignment, and the false negative homology pairs are the ones in the true alignment that are missing from the estimated alignment. The number of these true positives, false positives, and false negatives can be turned into rates by dividing by the appropriate number of homology pairs. Although this is described in terms of two sequences within a pairwise alignment, the definition extends
Multiple sequence alignment

The true multiple alignment
- Reflects historical substitution, insertion, and deletion events
- Defined using transitive closure of pairwise alignments computed on edges of the true tree

Figure 9.2 Evolution and the true multiple sequence alignment. The top sequence evolves into the bottom sequence via the deletion of the substring GGTG, the substitution of a T for a C, and the insertion of a T. This corresponds to the pairwise alignment on the right. Note that two letters are placed in the same column only when they have a common history. Thus, the substring GGTG in the top string is above dashes in the bottom string, and indicates that deletion event. Similarly, the red T is above the blue C, to indicate that they have a common history. (The use of color here is only to help illustrate the points; nucleotides don’t otherwise have colors!)

Table 9.1 $S = ACATTA$ evolves into $S' = TACA$ by (1) deleting the first two letters (A and C) and the second T (thus producing ATA), (2) changing the remaining T into a C (thus producing ACA), and (3) putting a T at the front (thus producing TACA). The alignment shown is the true alignment of $S$ and $S'$ reflecting this evolutionary history.

Table 9.2 Estimated alignment between $S$ and $S'$. 

![Diagram of sequence alignment with annotations for deletion, substitution, and insertion events.](image-url)
to multiple sequence alignments, but still depends on homology pairs. These false negative and false positive rates are then values between 0 and 1, and are error metrics. Since they depend on the sum-of-pairs scores, they are referred to as the SPFN and SPFP rates, respectively (Mirarab and Warnow, 2011).

Sometimes the SP-score is used, which is either the number of true positives or the fraction of the number of homology pairs in the true alignment that appear in the estimated alignment. Used in this way, the SP-score is an accuracy measure and not an error metric.

The total column (TC) score is related to this representation, but seeks the columns that are exactly identical with respect to the implied homology pairs. This is a popular metric for evaluating alignment accuracy, but since it depends on matching everything in a column the TC score can be very low - especially for large datasets, or datasets that are highly heterogeneous. Note that large values for the TC score are good (as it is an accuracy measure), while low values for the SPFN and SPFP rates are good (as these are error metrics). We summarize this as follows:

**Definition 9.1** Let $A$ be the reference multiple sequence alignment, and $A'$ an estimated multiple sequence alignment, both of the same set $S$ of unaligned sequences. Each alignment can be represented by its set of pairs of homologous letters, and these sets can be compared.

- The **SPFN rate** is the fraction of the truly homologous pairs (in $A$) that are not present in $A'$; this is the false negative rate.
- The **SPFP rate** is the fraction of the homologous pairs in $A'$ that are not present in $A$; this is the false positive rate.
- The **TC score** is the number of columns that are identical (including gaps) in the two alignments.

Thus, the SPFN and SPFP rates represent error rates, but TC is an accuracy measure.

### 9.4 Edit distances and how to compute them

A very common way to compute a pairwise alignment of two sequences is to define a cost for a substitution and a cost for an insertion or deletion (which typically depends on the length of the insertion or deletion), and then use these costs to define the cost of a pairwise alignment. Under this approach, the objective would be a pairwise alignment with the minimum cost.

As an example, suppose that substitutions and insertions and deletions of single nucleotides all have unit cost, and the input pair of sequences is $S = AAT$ and $S' = CAAGG$. There are several transformations of $S$ into $S'$ of minimum cost. For example, we could insert $C$ in front of $s$, obtaining $S_1 = CAAT$. We could then change the $T$ into a $G$, obtaining $S_2 = CAAG$. Finally, we could add one more $G$, obtaining $S' = CAAGG$. Since each step had unit cost, the cost of this transformation of $S$ into $S'$ is 3.

It is not too hard to see that there is no transformation of $S$ into $S'$ that uses fewer than
Table 9.3 Optimal pairwise alignment of two sequences when all substitutions and single letter indels have unit cost.

<table>
<thead>
<tr>
<th>S</th>
<th>-</th>
<th>A</th>
<th>A</th>
<th>T</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>S'</td>
<td>C</td>
<td>A</td>
<td>A</td>
<td>G</td>
<td>G</td>
</tr>
</tbody>
</table>

three steps, and so this is a minimum cost transformation of \( S \) into \( S' \). However, there is another transformation with the same cost, and so for this particular cost function and pair of sequences, there is no unique minimum transformation. Since this transformation also defines an evolutionary process relating \( S \) and \( S' \), it has a pairwise alignment associated to it, shown in Table 9.3.

Finding the minimum cost transformation of \( S \) into \( S' \) was easy for this case, but what if the sequences are much longer? Say, for example, \( S = AATAGATCGAATTAG \) and \( S' = CATTAGATTGAAACATTAGTACA \)? You can quickly convince yourself that finding a best transformation (where each insertion, deletion, and substitution has unit cost) by trial and error is painful, and that even if you could find one for this particular pair, you wouldn’t want to do this if the two sequences had hundreds of nucleotides in them. Fortunately, there are methods that have been developed to find minimum cost transformations between two sequences, which are very fast, and can be easily understood.

### 9.4.1 Needleman-Wunsch

The dynamic programming algorithm we will present in this section is essentially the same as the one proposed in Needleman and Wunsch (1970), but is described for the simplest case and in terms of edit distances.

**Edit distances under a linear gap model.** We will begin with the simplest case of pairwise alignment (representing the minimum cost edit transformation) where the cost of an indel of length \( L \) is \( cL \) for some constant \( c > 0 \); thus, an indel of 10 nucleotides in a row costs as much as 10 indels in a row. This is called the “linear cost” model; it’s not particularly realistic, but it’s a good starting point for understanding how algorithms work to compute these optimal pairwise alignments.

Assume also that you have an alphabet \( \Sigma \) (e.g., \( \Sigma = \{A,C,T,G\} \) for DNA sequences), and a substitution cost matrix that tells you the cost of substituting a given letter in \( \Sigma \) by some other letter. We’ll assume for now that if there is no change, then there is no cost.

Note that any sequence \( X \) can be transformed into another sequence \( Y \) by some edit transformation (e.g., delete everything in \( X \), then insert everything in \( Y \)). However, what we want to compute is the minimum cost edit transformation between \( X \) and \( Y \). Under the assumptions of a linear gap model, the calculation of the minimum cost edit transformation is pretty easy – using dynamic programming. But before we introduce the dynamic programming solution to this problem, we will consider a related problem of computing
the cost of an edit transformation implied by a given pairwise alignment between $X$ and $Y$. For example, suppose every indel costs 2 and every substitution costs 1; what is the cost of the pairwise alignment shown in Table 9.4? What if substitutions cost 2 and indels cost 1?

Now suppose you are given two sequences $X$ and $Y$ in a pairwise alignment. There are many edit transformations that are consistent with the given pairwise alignment, of course, because the order of the events changes the transformation. However, if all indels are always of length 1, then modulo the order of events that occur, there is a canonical edit transformation that is implied by the pairwise alignment – start at the left end of the alignment and perform the events implied by the sites, from left to right, until you reach the right end of the pairwise alignment.

We continue with some definitions that will make this exposition easier to follow.

**Definition 9.2** Let $S$ be a string of length $n$, with $S = s_1 s_2 s_3 \ldots s_n$, so that $s_i$ is the $i$th letter in $S$. The $i$th prefix of $s$ is the string $S_i = s_1 s_2 \ldots s_i$, and we let $S_0 = \lambda$ denote the empty string, which is (vacuously) a prefix of all strings. Hence, in a pairwise alignment between two sequences $S$ and $S'$, the first $j$ positions in the alignment define an alignment between a prefix of $S$ and a prefix of $S'$. (Note that we use lower case letters to refer to the letters in the string and upper case letters to refer to the string itself.)

Furthermore, the cost of the edit transformation (i.e., the edit distance) is the sum of the costs of the edit transformations defined for each individual site in the pairwise alignment. Thus, in the pairwise alignment shown in Table 9.3, the sites indicate the following events that transform sequence $S = AAT$ into sequence $Y = CAAGG$

- the first site indicates an insertion of $s'_1 = C$
- the second and third sites indicate no changes (no indels and no substitutions),
- the fourth site indicates a substitution of $s_3 = T$ by $s'_4 = G$
- the fifth site indicates an insertion of $s'_5 = G$

Because every event has unit cost, the cost of this pairwise alignment is 3. It is easy to see that this is an optimal pairwise alignment of the two strings.

Note that doing the edit transformation in this way also defines pairwise alignments and their associated canonical edit transformations from each prefix of $S$ into a prefix of $S'$. Thus, the first site defines an optimal pairwise alignment of $S_0$ and $S'_1$, the first two sites together define an optimal pairwise alignment of $S_1$ and $S'_2$, the first three sites together define an optimal pairwise alignment of $S_2$ and $S'_3$, etc.

It is also not hard to see that under the assumption of a linear gap model (so that a gap of length $L$ has the same cost as $L$ indels of length 1), each optimal pairwise alignment of two strings is obtained by extending an optimal alignment of prefixes of the two strings.
We can use these ideas and terminology to define a dynamic programming algorithm to compute the edit cost between two strings. We begin with the simplest problem - where all substitutions and single indels have unit cost, and where an indel involving \( L \) letters costs the same as \( L \) indels of length 1.

**Definition of the subproblems.** Let \( X \) and \( Y \) of length \( n \) and \( m \), respectively. Let \( \text{Cost}(i, j) \) denote the edit distance between \( X_i \) and \( Y_j \), where \( X_i \) denotes the \( i \)th prefix of \( X \) and \( Y_j \) denotes the \( j \)th prefix of \( Y \). We let \( 0 \leq i \leq n \) and \( 0 \leq j \leq m \), and hence will need to compute \((n+1)(m+1)\) values.

**The base case.** The base case is \( i = j = 0 \), which denotes the cost of transforming an empty string into an empty string; it is easy to see that \( \text{Cost}(0, 0) = 0 \).

**The recursive definition.** Suppose we want to compute \( \text{Cost}(i, j) \) and we have computed all “smaller” subproblems. Hence, in particular, we have computed \( \text{Cost}(i, j-1), \text{Cost}(i-1, j), \text{Cost}(i-1, j-1) \). Thus, we know the edit distance between \( X_i \) and \( Y_{j-1} \), between \( X_{i-1} \) and \( Y_j \), and between \( X_{i-1} \) and \( Y_{j-1} \). Although we don’t know any optimal edit transformation is between \( X_i \) and \( Y_j \), we know that they exist. Furthermore, given one such optimal edit transformation, its pairwise alignment has as its final site one of the following patterns:

- Case 1: \( x_i \) and \( y_j \) are aligned together in the final site. Note that in this case the other sites (before this last site) define a pairwise alignment of \( X_{i-1} \) and \( Y_{j-1} \).
- Case 2: \( x_i \) is aligned to a dash in the final site. Note that in this case the other sites (before this last site) define a pairwise alignment of \( X_{i-1} \) and \( Y_{j} \).
- Case 3: \( y_j \) is aligned to a dash in the final site. Note that in this case the other sites (before this last site) define a pairwise alignment of \( X_{i} \) and \( Y_{j-1} \).

In the first of these cases, the pairwise alignment of \( X_i \) and \( Y_j \) either involves a match (when \( x_i = y_j \)) or a mismatch (when \( x_i \neq y_j \)). The second and third cases each involve an indel (so the second case involves a deletion of \( x_i \) and the third case involves an insertion of \( y_j \)). The costs of these events are as follows:

- Case 1: The cost implied by the last site is 0 if \( x_i = y_j \) and otherwise the cost is 1. Hence, the total cost is \( \text{Cost}(i-1, j-1) + \text{Hamming}(x_i, y_j) \), where \( \text{Hamming}(x_i, y_j) = 1 \) if \( x_i \neq y_j \) and otherwise \( \text{Hamming}(x_i, y_j) = 0 \).
- Case 2: The cost of the last site is 1. Hence, \( \text{Cost}(i, j) = \text{Cost}(i-1, j) + 1 \).
- Case 3: The cost of the last site is 1. Hence, \( \text{Cost}(i, j) = \text{Cost}(i, j-1) + 1 \).

Now, although we don’t know the optimal pairwise alignment, we do know that it takes one of these forms. Hence, if we have already computed \( \text{Cost}(i, j-1), \text{Cost}(i-1, j) \) and \( \text{Cost}(i-1, j-1) \), we can set \( \text{Cost}(i, j) \) to be the minimum of the three possible costs (using the above analysis). In other words, we set

\[
\text{Cost}(i, j) = \min\{\text{Cost}(i-1, j-1) + \text{Hamming}(x_i, y_j), \text{Cost}(i-1, j) + 1, \text{Cost}(i, j-1) + 1\}
\]
9.4 Edit distances and how to compute them

Filling in the DP matrix. We need to compute \( \text{Cost}(i, j) \) for all \( 0 \leq i \leq n \) and \( 0 \leq j \leq m \). We can compute these entries in any way we like, as long as we don’t try to compute \( \text{Cost}(i, j) \) before we compute the values on which it depends. Hence, we can fill in the matrix row-by-row, column-by-column, or even in a diagonal way. We’ll do this (for simplicity’s sake) row by row. Thus:

- For all \( 1 \leq i \leq n \) and \( 1 \leq j \leq m \), \( \text{Cost}(0, j) = j \) and \( \text{Cost}(i, 0) = i \).
- For \( i = 1 \) to \( n \) DO
  - For \( j = 1 \) to \( m \) DO
    - \( \text{Cost}(i, j) = \min\{\text{Cost}(i - 1, j - 1) + \text{Hamming}(x_i, y_j), \text{Cost}(i - 1, j) + 1, \text{Cost}(i, j - 1) + 1\} \)
  - Return \( \text{Cost}(n, m) \)

Note that how \( \text{Cost}(i, j) \) is defined depends on whether \( i \) or \( j \) is 0, since these require special treatment. In general, a dynamic programming solution will have boundary cases that need special treatment. Note also that the final answer is located in \( \text{Cost}(n, m) \) – and you need to return that value.

Finding the optimal alignment from the DP matrix. This algorithm computes the edit distance between two strings, under the assumption that all events (indels and substitutions) have unit cost. It does not compute the actual edit transformation, however. How would you modify the algorithm to give you that information? Or, how would you use the results of the algorithm to figure out the edit transformation?

The answer is pretty easy. To set \( \text{Cost}(i, j) \), you are finding the minimum of three values. Whichever entry (or entries, if there are more than one) gives you the smallest value, put an arrow from the box for \( \text{Cost}(i, j) \) to the box that gave you the smallest value. For example, if \( \text{Cost}(i, j) \) was set to be \( \text{Cost}(i - 1, j - 1) + 1 \), then put an arrow from \( \text{Cost}(i, j) \) to \( \text{Cost}(i - 1, j - 1) \). Then, at the end of the computation, there will be a path from \( \text{Cost}(n, m) \) all the way back to \( \text{Cost}(0, 0) \), defining a minimum cost edit transformation - and its pairwise alignment.

Modifying the DP algorithm to handle arbitrary cost matrices. Finally, how would you modify the algorithm to handle the case where there is a substitution cost matrix, so that not all substitutions have the same cost? This is not hard to do either – you just need to modify how you define \( \text{Cost}(i, j) \) to account for different costs for mismatches in the final site.

However, if the cost of a gap of length \( L \) is no longer just \( cL \) for some constant \( c \), then the algorithm will need to be modified more extensively. For example, many gap cost functions are “affine”, which means they take the form of \( \text{gapcost}(L) = c_0 + c_1L \), where \( c_0 \) denotes the cost of opening a gap, and \( c_1 \) denotes the cost of extending the gap. Other more elaborate gap cost functions have also been described. All of these can be handled, but require modifications to the DP algorithm, and increase the running time.
Multiple sequence alignment

Running time. It is easy to see that the algorithm takes $O(1)$ time to compute $Cost(i, j)$ for each $i, j$, given that you compute the values in an appropriate order. Thus, the running time is $O(nm)$.

Maximizing similarity instead of minimizing distance. A variant of this problem is obtained by defining the similarity between two strings, and then seeks the pairwise alignment yielding the maximum pairwise similarity score. It is not hard to modify the Needleman-Wunsch algorithm appropriately so that it is described in those terms; however, note that instead of penalizing for mismatches, the algorithm must explicitly favor matches.

6.4.2 Smith-Waterman

The next problem we consider is where we want to find a close match of a short string $S$ to a longer string $S'$. The application for this is where we are searching a database $D$ of “full-length” strings for a match to a short string $S$; hence, we only want to focus on the substrings within long strings that give good matches to $S$. Thus, given a full-length string $S'$ in $D$, we would search for a pair of indices $i, j$ so that the substring of $S'$ between indices $i$ and $j$ gave a very good match to $S$. This is the local pairwise alignment problem. The Needleman-Wunsch algorithm doesn’t solve this problem, since it finds a global alignment rather than a local alignment. However, Smith and Waterman (1981) showed how to modify the dynamic programming approach so that the optimal local alignment could be found.

9.5 Optimization problems for multiple sequence alignment

Now that you have seen how to compute an optimal pairwise alignment using edit distances, we can consider multiple sequence alignment. The first question is how to define the cost of a multiple sequence alignment. There are two very natural ways to define the cost of a multiple sequence alignment that can be considered, both of which are extensions of the cost of a pairwise alignment.

So suppose we have a function $c$ that defines the cost of a pairwise alignment of two sequences. Typically, $c$ will be based on some cost for substitutions (which may depend on the particular pair of letters involved) and an affine gap penalty to account for indels. Given this cost function for pairwise alignments, how do we extend it to a cost for a multiple sequence alignment? Here we describe two commonly considered extensions: the “Sum-of-Pairs” cost and the “Tree Alignment” cost.

9.5.1 Sum-of-Pairs Alignment

The first optimization problem we present is called “Sum-of-Pairs” (SOP), and is defined as follows. Given an alignment on two sequences, we define the cost of the alignment using the formulation described above. Now, if there are more than two sequences, then given a
Table 9.5 A multiple sequence alignment on three sequences with SOP score 8, where each indel or substitution has unit cost.

<table>
<thead>
<tr>
<th></th>
<th>s1</th>
<th>-</th>
<th>-</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>s2</td>
<td>A</td>
<td>T</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>s3</td>
<td>C</td>
<td>A</td>
<td>-</td>
<td>G</td>
</tr>
</tbody>
</table>

Table 9.6 A second multiple sequence alignment on the same three sequences from Table 9.5.

<table>
<thead>
<tr>
<th></th>
<th>s1</th>
<th>-</th>
<th>-</th>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>s2</td>
<td>-</td>
<td>A</td>
<td>T</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>s3</td>
<td>C</td>
<td>A</td>
<td>-</td>
<td>-</td>
<td>G</td>
</tr>
</tbody>
</table>

As an example, consider three sequences $s_1, s_2, s_3$ in a multiple sequence alignment given in Table 9.5. Suppose that every insertion, deletion, and substitution has unit cost. Then, examining the multiple sequence alignment in Table 9.5, we compute the total sum-of-pairs (SOP) cost as follows. The pairwise alignment between $s_1$ and $s_2$ has cost 2, the pairwise alignment between $s_1$ and $s_3$ has cost 3, and the pairwise alignment between $s_2$ and $s_3$ has cost 3. Hence, the multiple sequence alignment has cost 8. In contrast, the alternative multiple sequence alignment on the same three sequences, given in Table 9.4 has a larger total sum-of-pairs (SOP) cost. However, the multiple sequence alignment in Table 9.3 is not optimal with respect to SOP score, because an alignment with SOP score 7 can also be obtained.

The SOP optimization problem asks us to find a multiple sequence alignment with minimum SOP cost. Finding the best multiple sequence alignment for a set of $n$ sequences, based on a given cost function $c$, is NP-hard (Wang and Jiang, 1994). On the other hand, the optimal multiple sequence alignment (under this criterion) can be found in time that is exponential in the number $n$ of sequences (using an extension of the dynamic programming algorithm described above), and so is achievable for very small numbers of sequences. However, in general, optimal multiple sequence alignments under the SOP criterion are not generally attempted.

### 9.5.2 Tree Alignment

In the tree alignment problem, the input is a tree $T$ with leaves labelled by the set $S$ of $n$ sequences, $S = s_1, s_2, \ldots, s_n$, and the cost function $c$. Our objective is to find sequences
Multiple sequence alignment

for the internal nodes of $T$ so that the total cost, defined to be the sum of the costs on the edges of the tree, is minimized. Note that the cost of an edge in a tree is the cost of the minimum transformation between the sequences labelling the endpoints of the edge. Furthermore, as noted above, once we have a minimum transformation on every edge, we also have a pairwise alignment for every edge, and hence we can define the multiple sequence alignment associated with the minimum transformations on the edges.

**Example 9.3** Suppose the set of sequences is $S = \{s_1, s_2, s_3\}$, where $s_1 = \text{ATA}, s_2 = \text{AAT}$, and $s_3 = \text{CAA}$, and suppose indels have a very large cost (say 100) while substitutions have unit cost. Now let $T$ be the tree with the sequences in $S$ at the leaves. What is the best label of the internal node you can find, and what is the cost for this tree? You should be able to find that the sequence AAA gives a cost of only 3, and is the best possible sequence that could be obtained for this tree. However, if we had been constrained to pick the label only from the sequences at the leaves, then the cost would have been larger. Thus, when the internal nodes can be labelled differently from the sequences at the leaves, finding an optimal set of sequences at internal nodes allows lower cost trees.

**Example 9.4** As a second example, suppose the input is $s_1 = \text{AC}, s_2 = \text{ATAC}$, and $s_3 = \text{CAG}$, and suppose we are given a tree $T$ with an internal node $X$, and $s_1, s_2$, and $s_3$ at the leaves. Suppose insertions, deletions, and substitutions each have unit cost. If we set $X = s_1$, the cost of the tree would be the sum of the costs of the optimal pairwise alignments between $s_1$ and the other two sequences. Note that the cost of the optimal alignment between $s_1$ and $s_3$ is 2, and the cost of the optimal alignment between $s_1$ and $s_3$ is 2; hence, the total cost would be $2 + 2 = 4$. What would the cost have been if we used $X = s_2$? It would have been the cost of the pairwise alignment between $s_1$ and $s_2$, plus the cost of the pairwise alignment between $s_2$ and $s_3$, and so $2 + 3 = 5$. Finally, if we used $X = s_3$, then the cost would have been the cost of the pairwise alignment between $s_1$ and $s_3$, plus the cost of the pairwise alignment between $s_2$ and $s_3$, or $2 + 3 = 5$. Hence, if we restricted $X$ to be one of the input sequences, then the best result would have cost 4, and would be obtained by setting $X = s_1$. On the other hand, if we are not constrained to selecting from among the input sequences, we could let $X$ be some other sequence. Can you find a better solution for this problem than $X = s_1$?

The Tree Alignment problem (Sankoff, 1975b; Sankoff and Cedergren, 1993) allows internal nodes to be labelled by arbitrary strings, and is NP-hard even for simple gap penalty functions (Wang and Jiang, 1994; Wareham, 1995). In contrast, the maximum parsimony problem on a fixed tree can be solved exactly is polynomial time using dynamic programming (see Chapter 4.3). Thus the Tree Alignment problem is harder than the maximum parsimony problem. However, approximate solutions can be found (Wang et al., 1996; Wang and Gusfield, 1997; Wang et al., 2000), which have bounded error. For example, Wang and Gusfield (1997) showed that a simple “lifted-alignment” approach would be a 2-approximation to the optimal alignment, which means that the method would return an alignment whose total treelength was no more than twice that of the optimal solution.
9.5 Optimization problems for multiple sequence alignment

9.5.3 Generalized Tree Alignment

Now suppose that the tree is not given in the input, so that the input would be a set \( S \) of unaligned sequences, and the output would be a tree \( T \) with internal nodes labelled by sequences over the same alphabet of minimum total cost. This is called the Generalized Tree Alignment problem. Finding an approximate solution to the Generalized Tree Alignment problem is also easy – in polynomial time!

Minimum Spanning Trees. Here’s the simple idea that produces a tree \( T \) with leaves labelled by the sequences in \( S \), and sequences labelling the internal nodes, so that the multiple sequence alignment defined by the optimal pairwise alignments on the edges is guaranteed to have a total tree alignment cost that is at most twice the optimal tree alignment cost over all trees on \( S \):

- Step 1: Compute the pairwise edit distances between every pair of sequences in \( S \).
- Step 2: Construct the graph \( G \) with vertices defined by the elements of \( S \) and the weight of edge \((s, s')\) the edit distance between \( s \) and \( s' \).
- Step 3: Find a minimum spanning tree \( T \) for \( G \).
- Step 4: For every vertex \( v \) in \( T \) that is not a leaf, add a vertex \( v' \) that is adjacent to \( v \) and give it the same sequence label. Call this tree \( T' \).

Note that \( T' \) has all nodes labelled by sequences, and so we can define the edit distance on every edge of \( T' \), which we’ll use to weight the edges of \( T' \). We define \( w(T') \) to be the total of the edge weights of \( T' \). Similarly, we can define \( w(T) \) to be the sum of the edge weights of \( T \) for any tree with sequences labelling its nodes.

**Lemma 9.5** Let \( S \) be any non-empty set of sequences, and \( c \) the edit distance function. Let \( T' \) be a tree constructed by the Minimum Spanning Tree algorithm and let \( T^{opt} \) be an optimal solution to the Generalized Tree Alignment problem. Then \( w(T') \leq 2w(T^{opt}) \).

**Proof** Note that \( w(T') = w(T) \), so if we show \( w(T) \leq 2w(T^{opt}) \) then we are done.

Given a graph \( G \) with every node labelled by a sequence, we denote by \( w(G) \) the total weight of all the edges in graph \( G \), where the weight of an edge is the edit distance between the sequences at the endpoints of the edge. Consider the graph \( G' \) obtained by doubling every edge in \( T^{opt} \); note that \( G' \) is Eulerian since every node has even degree, and hence there is a cycle \( C \) that covers every edge twice. Then \( w(C) = w(G) = 2w(T^{opt}) \).

Break the cycle \( C \) by removing one edge (arbitrarily) to obtain path \( P \). Therefore, \( w(P) < w(C) \leq 2w(T^{opt}) \). Since \( P \) will generally have multiple appearances of a given vertex, we will obtain a second path \( P' \) that contains exactly one copy of each vertex by only including the initial occurrence of each vertex, and skipping over the remaining vertices. For example, if \( P = v_1, v_2, v_3, v_1, v_4, v_3, v_2, v_4 \), then \( P' \) would be \( v_1, v_2, v_3, v_4 \). Since edit distances satisfy the triangle inequality, it follows that \( w(P') \leq w(P) \). Hence, \( w(P') \leq w(P) < w(C) \).

Now, \( P' \) is a spanning tree for \( G \), and hence \( w(T) \leq w(P') \) since \( T \) is a minimum spanning tree for \( T \). Hence, \( w(T') \leq 2w(T^{opt}) \), and the lemma is proved.
Profile Hidden Markov Models

Profile Hidden Markov models are widely used in molecular sequence analysis, and also have wide use in multiple sequence alignment methods. Given the focus on multiple sequence alignment, we will describe the basic concepts and techniques used in Profile HMMs.

9.6.1 Profiles

We begin with how multiple sequence alignments define profiles. Given a multiple sequence alignment (MSA), we can compute the frequency of each letter (nucleotide or amino acid) in a given position within the MSA. For example, for a DNA sequence alignment, for each position in the alignment we can associate a 4-tuple giving the proportion of the times each of the four nucleotide appears. Thus, a nucleotide alignment with \( k \) sites (none of which is entirely gapped) defines a profile of length \( k \), with the \( i \)th position occupied by the 4-tuple for the distribution of nucleotides in the \( i \)th site. Similarly, an amino acid alignment with \( k \) sites defines a profile of length \( k \) where the \( i \)th position is occupied by a 20-tuple. As an example, see Table 9.7, which presents a single MSA with five sequences and its associated profile.

A profile can be seen as a simple representation of the frequency of the different letters in each position, but it can also be used as a generative model. For example, using the profile given in Table 9.7, we would generate DNA strings of length four by picking a nucleotide for the first position with probabilities equal to the observed relative frequencies, then picking a nucleotide for the second position, etc. As a result, the two most probable sequences for that profile would be \( \text{ATAC} \) and \( \text{ATGC} \), each of which has probability \( 0.6 \times 0.5 \times 0.5 \times 0.6 = 0.09 \). It should also be obvious that any sequence with a \( \text{G} \) in the second position cannot be generated by this profile, and has probability 0.

<table>
<thead>
<tr>
<th>Alignment</th>
<th>A</th>
<th>-</th>
<th>-</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_1 )</td>
<td>A</td>
<td>T</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>( s_2 )</td>
<td>C</td>
<td>C</td>
<td>A</td>
<td>G</td>
</tr>
<tr>
<td>( s_3 )</td>
<td>A</td>
<td>T</td>
<td>G</td>
<td>C</td>
</tr>
<tr>
<td>( s_5 )</td>
<td>G</td>
<td>A</td>
<td>G</td>
<td>G</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Profile</th>
<th>A</th>
<th>C</th>
<th>T</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_1 )</td>
<td>0.6</td>
<td>0.25</td>
<td>0.5</td>
<td>0.0</td>
</tr>
<tr>
<td>( s_2 )</td>
<td>0.2</td>
<td>0.25</td>
<td>0.0</td>
<td>0.6</td>
</tr>
<tr>
<td>( s_3 )</td>
<td>0.0</td>
<td>0.50</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>( s_4 )</td>
<td>0.0</td>
<td>0.50</td>
<td>0.0</td>
<td>0.4</td>
</tr>
</tbody>
</table>

Table 9.7 A multiple sequence alignment on five sequences, and its associated profile.
9.6.2 Gap-free Profile Hidden Markov Models

Although a profile can be seen as a generative model, it’s limited to generating sequences of the same length as the profile. If we include gaps in a profile, so that the probability vector associated with the \(i^{th}\) site has an extra entry for a gap (i.e., a vector of length 5 for nucleotide sequences, or a vector of length 20 for amino acids), then we can generate sequences that are shorter than the profile length, but we still cannot generate sequences that are longer. A more general model than profiles is needed to enable sequence length heterogeneity.

Profile Hidden Markov Models are graphical models that were developed to address this issue. As a gentle introduction to profile Hidden Markov models, we’ll begin by reformulating a profile as a graphical model.

Given a profile \(P\) associated with an alignment, we can construct a probabilistic graphical model to generate sequences with the same distribution as \(P\). There is a start state (also called a “begin” state, and sometimes marked as “B”) and an end state (often marked as “E”). In between, there is a single state for each position in the alignment. There is a transition edge from the start state to the state for position 1, then a transition edge from the state for position 1 to the state for position 2, etc. Finally, there is a transition edge from the state for position 4 to the end state. Note that the graphical model is a directed graph, and that all edges move from left to right; thus, this is a directed acyclic graph (DAG). Each transition edge from state \(A\) to state \(B\) is annotated with the probability of moving from \(A\) to \(B\) – which in this case is 1. Note that all states other than the start and end states are associated with positions in the alignment, and so are associated with a distribution on the nucleotides at that position.

Consider now a path through this graphical model. By design, every path starts at the start state, visits every state associated to a position in the alignment in turn, and then ends at the end state. The states other than the start and end states are called “match” states.

For each match state, we have a vector of emission probabilities for the different nucleotides, using the probability distributions we obtained from the profile. What this means is that when we visit one of these states, we will select a nucleotide using the distribution for the state, and write it down. In this way, following a path through the graphical model produces a DNA sequence of length four.

This is a gap-free profile Hidden Markov Model. Note that given any sequence generated by the model, if you know the number of states in the model, then you know exactly the path through the model. On the other hand, as we will see, profile Hidden Markov Models that have insertion or deletion states can generate sequences of different lengths, and so are much more powerful.

9.6.3 Profile Hidden Markov Models with insertion and deletion states

Consider an alignment now that has gaps – representing insertions and deletions. How do we represent this? Suppose for example that the alignment has 100 sequences, 90 of which
Figure 9.3 (Based on figure from Durbin et al. (1998)) **Profile Hidden Markov Model.** A profile Hidden Markov Model (HMM) is a probabilistic graphical model that represents a family of sequences, and can be built from an input multiple sequence alignment. Each profile HMM has a begin state and an end state, and then three other kinds of nodes: match states (indicated by squares), insertion states (indicated by diamonds), or deletion states (indicated by circles). Nodes representing insertion states can have self-loops, but no other nodes can; other than these self-loops, the underlying directed graph is acyclic. The arcs on the profile HMM have probabilities, with the sum of the probabilities on the outgoing edges of any node being 1. Every match state and every insertion state emits a single letter based on a probability distribution on the alphabet (four letters for nucleotides, 20 letters for amino acids, etc.).

Moreover, suppose that all 10 that are of length 7 lack nucleotides for positions 2 through 4. We can also represent this as a graphical model with a start and end state, and a match state for every position in the alignment. As before, we can compute the probability distribution of nucleotides in each position, using whichever nucleotides actually appear in that position. However, we allow for a single indel of length 3 by having a transition edge from position 1 to position 5; we will refer to this transition edge as a “deletion” edge. Now, consider a path through the graphical model that uses the transition edge representing the deletion; then, the path visits match states.
for positions 1, 5, 6, 7, 8, 9, 10, and so produces a sequence of length 7. Similarly, a path through the graphical model that does not use the deletion edge produces a sequence of length ten. As before, we need to assign probabilities to each transition edge. Most states have only one outgoing transition edge, and so those transition edges have probability 1. However, match state 1 has two outgoing transition edges - one going to match state 2, and one going to match state 5. We can note that 10 out of the 100 sequences had a gap in positions 2 through 4, so we set the probability of the transition edge to state 5 to be 0.1, and the probability of the transition edge to state 2 to be 0.9.

This is a very simple case of a sequence alignment with a single gap (here, of length three), and a profile HMM to represent the alignment.

Note that the construction of the profile HMM assumed we had a match state for every position in the alignment, and interpreted the sequences of length 7 as involving a deletion. What if those sequences resulted from an insertion instead of a deletion? Equivalently, suppose the match states were only for positions 1, 5, 6, 7, 8, 9, and 10, and all sequences of length 10 involved an insertion between positions 1 and 5? In that case, we could use an “insertion state” generate sequences of length 3 (from a distribution), or we could have three insertion states (one for each position within the three-letter insert). However, in general, because the majority of the sequences lack the gap event, we would prefer to interpret the gap as a deletion event instead of an insertion event.

More generally, though, we could have arbitrary length gaps, and insertions as well as insertions. To keep the graphical model small (so that it has a linear number of transition edges and states), we have a single insertion state between every two match states, and a single deletion state between every two match states. We model long insertions by having a self-loop at the insertion states, and we model long deletions by having transition edges from each deletion state to the next deletion state. We can also have an insertion state and a deletion state between the begin state and the first match state, and an insertion state between the last match state and the end state.

Figure 9.3 from Durbin et al. (1998) shows an example of the underlying directed graph for a profile HMM in which there are three match states (names beginning with M), a begin state, an end state, three deletion states (names beginning with D) and four insertion states (names beginning with I). Every path through the graph would start at Begin and end at End. Note that the insertion states have self-loops, so that insertions of any length can be produced.

This figure shows the graphical structure but does not show the numeric parameters of the model, which are the transition probabilities on the edges and the emission probabilities for the match and insertion states. Recall that the transition probability on an edge is the probability that the edge will be taken, and so for every node v, the transition probabilities on the outgoing edges from v must sum to 1. Similarly, the emission probabilities of the different letters in the underlying alphabet must sum to 1 for any match or insertion state. Note that the alphabet depends on the type of data; thus, the alphabet would be \{A,C,T,G\} for DNA sequence, or \{A,C,U,G\} for RNA sequences, or the alphabet of 20 amino acids for AA sequences.
Every path through the model defines a set of strings over the alphabet that can be generated by the path. The empty string (i.e., the string of length 0, so the string that has no letters) can only be generated by the path that avoids all the match and insertion states, but can use the deletion states; for example, in the model in Figure 9.3., the path \( \text{Begin} \to D1 \to D2 \to D3 \to \text{End} \) will produce the empty string, but all other paths will produce non-empty strings. Not all paths can generate the same set of strings; for example, using the model in Figure 9.3, the path \( \text{Begin} \to M1 \to M2 \to M3 \to \text{End} \) generates strings of length 3, but cannot generate strings that are shorter or longer than 3. Furthermore, depending on the emission probabilities, some strings may not be able to be generated by any path a given model. For example, if the match and insertion states set emission probability for nucleotide A to 0, then no DNA string with any As can be generated by the model.

Note that some strings can be generated by more than one path through one of these models. For example, consider the model produced by using the graphical structure in Figure 9.3, and set the probability of emitting each nucleotide to be non-zero for every insertion or match state, and set the transition probabilities from the match state to its neighboring insertion state strictly between 0 and 1. In this case, every non-empty sequence can be generated by multiple paths through the model. For example, the string AAA can be generated by the path \( \text{Begin} \to M1 \to M2 \to M3 \to \text{End} \) and also by \( \text{Begin} \to M1 \to I1 \to D2 \to M3 \to \text{End} \). Thus, one of the interesting consequences of this quite general model is that even if we know the entire probabilistic graphical model (i.e., the underlying directed graph, the emission probabilities for the match and insertion states, and the transition probabilities on all the directed edges), we will typically not be able to infer the sequence of states that were taken to generate a given sequence. Although we’ve already noted some cases (e.g., the empty string) where it is possible to know the path that was taken through the model to generate the string, the general case is that many strings can be generated by two or more paths through one of these models. Whenever a model has the property that some strings can be generated by two or more paths through the model, we will say that the model has “hidden states”, and we will refer to these models as hidden Markov models. Furthermore, because these models are based on profiles (with match states corresponding to sites in a multiple sequence alignment), these are called “profile Hidden Markov Models”, or profile HMMs. Finally, the “Markov” in the name is because the emission probabilities and transition probabilities for each state depend only on the state, and not on the path taken to reach the state.

Although it is not possible to know the true path in a profile HMM that generated a given string \( s \), the probabilistic framework allows us to answer several questions that will be useful to us, such as:

- **Question 1**: Given a profile HMM \( M \) and a string \( s \) that can be generated by \( M \), what path through \( M \) is most likely to have produced \( s \)?
- **Question 2**: Given a profile HMM \( M \) and a string \( s \) that can be generated by \( M \), what is the probability that \( M \) generated \( s \)?
Question 3: Given two profile HMMs $M$ and $M'$ and a string $s$, which profile HMM is more likely to have generated $s$?

As we will show, each of these questions can be addressed using dynamic programming algorithms. However, building a profile HMM from a multiple sequence alignment is more difficult than answering these probability questions.

### 9.6.4 Probability calculations on profile HMMs

Given a profile HMM (and hence, knowing the probability of each transition between states), it is very easy to compute the probability of any given path through the model. The path defines a sequence of transitions between states, and so the probability of the path is just the product of the transition probabilities. Note that a path through the profile HMM also specifies the length of the sequence that is generated.

Similarly, given a path through the profile HMM, it is easy to compute the probability of any sequence that can be generated by the path, since the path defines for each position in the sequence the specific state (insertion or match state) that generated the letter in that position. Hence, given the path, and given the emission probabilities associated to the match and insertion states, we can compute the probability of generating a given sequence by multiplying the probabilities of generating each letter in the sequence (as defined by the emission probabilities for the state that generated the letter).

Note that the probability of a given sequence thus depends on the path through the profile HMM that generated the sequence. Furthermore, many different paths could have generated a given sequence. Thus, to compute the total probability of generating a sequence, we would need to sum the probabilities over all the paths that could generate the sequence. Note that the paths are themselves not equally probable, and so these calculations also depend on the probability of following each path.

Question 1 (finding the path in the model that is most likely to have generated a given string) can be answered in polynomial time, using a dynamic programming algorithm called the Viterbi Algorithm, after Andrew Viterbi who developed it in the context of coding theory. However, the phrase now more or less refers to any dynamic programming algorithm that involves maximizing probabilities.

Question 2 (computing the total probability of all paths in a given model that could have generated a given string) can also be solved in polynomial time using two dynamic programming algorithms called the Forward and Backward Algorithms (Durbin et al., 1998; Ewens and Grant, 2001). Note that Question 3 (comparing two models and selecting the one most likely to have generated a given string) is easily addressed using the answers to Question 2 for the two models.

Building a profile HMM from an input multiple sequence alignment is a complicated process. The graphical model, without the numeric parameters, is generally not difficult, though the number of match states can vary (some methods for building the graphical model will only have a match state if the site is ungapped, or has very few gaps). Once
Multiple sequence alignment

the graphical model is constructed, the numeric parameters must be estimated, and this
turns out to be much more challenging. Because no methods have been able to provably
find optimal numeric parameters, heuristics are used instead to find acceptable numeric
parameters. One of the popular approaches is the Baum-Welch method, presented in Baum
(1972).

9.7 Algorithmic techniques

9.7.1 Progressive alignment

Nearly all the popular multiple sequence alignment methods rely on progressive align-
ment, to lesser or greater extents. Progressive alignment techniques, introduced in Feng
and Doolittle (1987), begin by computing a rooted binary tree (called a merge tree or a
guide tree) from the input set of sequences; then, the multiple sequence alignment is built
from the bottom-up, starting at the internal nodes that have only leaves as children and then
moving towards the root. When visiting a node \( v \) with children \( v_1 \) and \( v_2 \), the assumption is
that the sequences below \( v_1 \) are in a multiple sequence alignment \( A_1 \) and the sequences be-
low \( v_2 \) are in a multiple sequence alignment \( A_2 \). The two multiple sequence alignments, \( A_1 \)
and \( A_2 \), are then merged into an alignment that contains all the sequences below \( v \). These
pairwise mergers are performed in a variety of ways, but do not change \( A_1 \) or \( A_2 \). When the
root is reached, the resultant alignment contains all the sequences at the leaves of the tree,
and so is a multiple sequence alignment of the entire dataset.

There are many techniques that have been developed to merge pairs of alignments. A
simple technique is to represent each alignment by a profile and then align the two profiles.
Since the positions in the profiles correspond to positions in the alignment, the alignment
of two profiles produces an alignment of the two alignments, and hence defines a way
of merging the two alignments into an alignment on the entire dataset. Similarly, each
alignment could be represented by a Hidden Markov Model (HMM), and then the match
states in the two HMMs can be aligned. Since the match states in the HMMs correspond
to positions within the alignments, alignments of match states produce alignments of the
two alignments. Mergers like these use polynomial time, and do not change the alignments
they operate on.

**Example 9.6** We show how a profile-profile alignment technique can be used to merge
two multiple sequence alignments on disjoint sequences into an alignment on the full
dataset. Suppose alignments \( \Phi_1 \) and \( \Phi_2 \) are given as input; see Table 9.8.

We compute profiles \( P_1 \) and \( P_2 \) for \( \Phi_1 \) and \( \Phi_2 \), respectively (see Table 9.6). We will
use very simple profiles, where the \( i^{th} \) position is just the frequency vector of the four nu-
cleotides \( A,C,G,T \), for the \( i^{th} \) site in the alignment. For example, for \( \Phi_1 \), the first position is
100% C’s, so its frequency vector will be \([0,1,0,0]^T\). We will denote the \( i^{th} \) vector in
\( P_1 \) by \( a_i \), and the \( i^{th} \) vector in \( P_2 \) by \( b_i \). Note that each profile has the same length as its
associated alignment; thus, \( P_1 \) has the length 5 and \( P_2 \) has length 6.

We now align these two profiles. If we write \( P_1 = (a_1,a_2,a_3,a_4,a_5) \), where \( a_i \) is the
9.7 Algorithmic techniques

Table 9.8 Two multiple sequence alignments on disjoint sets of sequences; Tables 9.6 through 9.6 will show the steps performed to merge these two alignments together using profile-profile alignment techniques.

<table>
<thead>
<tr>
<th></th>
<th>(s_1)</th>
<th>(-)</th>
<th>(-)</th>
<th>(T)</th>
<th>(A)</th>
<th>(C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(s_2)</td>
<td>(-)</td>
<td>(A)</td>
<td>(T)</td>
<td>(A)</td>
<td>(C)</td>
<td></td>
</tr>
</tbody>
</table>

\(A_1: \begin{align*} s_3 &= \text{C} \quad \text{A} \quad \text{-} \quad \text{-} \quad \text{G} \\ s_4 &= \text{C} \quad \text{A} \quad \text{A} \quad \text{T} \quad \text{G} \\ s_5 &= \text{C} \quad \text{-} \quad \text{T} \quad \text{-} \quad \text{G} \end{align*} \)

<table>
<thead>
<tr>
<th></th>
<th>(s_6)</th>
<th>(C)</th>
<th>(T)</th>
<th>(-)</th>
<th>(-)</th>
<th>(A)</th>
<th>(C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(s_7)</td>
<td>(C)</td>
<td>(-)</td>
<td>(A)</td>
<td>(T)</td>
<td>(A)</td>
<td>(C)</td>
<td></td>
</tr>
<tr>
<td>(s_8)</td>
<td>(G)</td>
<td>(-)</td>
<td>(A)</td>
<td>(-)</td>
<td>(A)</td>
<td>(T)</td>
<td></td>
</tr>
</tbody>
</table>

Table 9.9 Profiles \(P_1\) and \(P_2\) for the alignments \(A_1\) and \(A_2\), respectively, from Table 9.6.

\[
P_1: \begin{array}{cccccc}
a_1 & a_2 & a_3 & a_4 & a_5 \\
A & 0 & 1 & \frac{2}{3} & \frac{1}{3} & 0 \\
C & 1 & 0 & 0 & 0 & \frac{2}{3} \\
G & 0 & 0 & 0 & 0 & \frac{1}{3} \\
T & 0 & 0 & \frac{2}{3} & \frac{1}{3} & 0 \\
\end{array}
\]

\[
P_2: \begin{array}{cccccc}
b_1 & b_2 & b_3 & b_4 & b_5 & b_6 \\
A & 0 & 0 & 1 & 0 & 1 & 0 \\
C & \frac{2}{3} & 0 & 0 & 0 & 0 & \frac{2}{3} \\
G & \frac{1}{3} & 0 & 0 & 0 & 0 & 0 \\
T & 0 & 1 & 0 & 1 & 0 & \frac{1}{3} \\
\end{array}
\]

Table 9.10 An optimal alignment of profiles \(P_1\) and \(P_2\) from Table 9.6, where \(a_i\) and \(b_i\) represent the frequency vectors for the \(i^{th}\) positions in \(P_1\) and \(P_2\), respectively.
Multiple sequence alignment

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>T</th>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>s₁</td>
<td>-</td>
<td>-</td>
<td>T</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>s₂</td>
<td>-</td>
<td>-</td>
<td>A</td>
<td>T</td>
<td>A</td>
</tr>
<tr>
<td>s₃</td>
<td>C</td>
<td>-</td>
<td>A</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>s₄</td>
<td>C</td>
<td>-</td>
<td>A</td>
<td>A</td>
<td>T</td>
</tr>
<tr>
<td>s₅</td>
<td>C</td>
<td>-</td>
<td>-</td>
<td>T</td>
<td>-</td>
</tr>
<tr>
<td>s₆</td>
<td>C</td>
<td>T</td>
<td>-</td>
<td>-</td>
<td>A</td>
</tr>
<tr>
<td>s₇</td>
<td>C</td>
<td>-</td>
<td>A</td>
<td>T</td>
<td>A</td>
</tr>
<tr>
<td>s₈</td>
<td>G</td>
<td>-</td>
<td>A</td>
<td>-</td>
<td>A</td>
</tr>
</tbody>
</table>

Table 9.11 The final multiple sequence alignment of A₁ and A₂ from Table 9.8, obtained by computing their profiles P₁ and P₂ (see Table 9.6) and then aligning them (see Table 9.6).

ments A₁ and A₂; this is an aspect of a progressive alignment strategy that makes it vulnerable to early mistakes in the alignment computation.

9.7.2 Template-based methods

Some alignment methods use a very different type of algorithmic design, called “template-based” (Kemena and Notredame, 2009). These methods use statistical models for the sequence family, and align each sequence in the input dataset to the model in order to produce the final multiple sequence alignment. As discussed in Chapter 9.10, these template-based methods can be quite elaborate and rely on substantial external knowledge about the sequences (e.g., structural and biochemical properties), especially in the context of amino acid alignment. Here we describe a simple version of this type of approach, which does not require any external information, and can be used on any kind of molecular sequence data (DNA, RNA, or amino acids). We will refer to this subclass of template-based methods as “seed alignment” methods, since they will operate by computing an alignment on a subset of the sequences, and then using that alignment (which we will refer to as a seed alignment) to align the remaining sequences.

Seed alignment methods. The input is a set S of unaligned sequences, which can be of any type. A seed alignment method has three steps: (1) a seed alignment $\mathcal{A}$ is computed for a subset $S₀ \subset S$, (2) a statistical model $\mathcal{M}$ is built for $\mathcal{A}$, and (3) the remaining sequences are added to $\mathcal{A}$ using $\mathcal{M}$.

In the first step, a small set $S₀$ of sequences from the family is selected and then aligned using a preferred multiple sequence alignment method. This alignment is referred to as a “seed alignment”. Note that how $S₀$ is selected and then aligned will impact the resultant alignment.

In the second step, a statistical model is constructed for the seed alignment. The simplest version of this approach uses a sequence profile, as described in Chapter 9.6. Slightly more
elaborate versions use Hidden Markov Models, and some versions have used Ensembles of Hidden Markov Models (Mirarab et al., 2012; Nguyen et al., 2014, 2015b). However, other statistical models such as Markov Random Fields (Ma et al., 2014) can also be used.

In the third step, the statistical model is used to align each sequence to the seed alignment (which does not change during the process), and then inserted into the alignment.

One of the main advantages of this kind of method is its tremendous scalability to large datasets. If the seed alignment is small and the statistical model does not rely on any external knowledge, then the first two steps can be completed very quickly. The third step, where the remaining sequences are added to the seed alignment using the statistical model, is generally very fast, since each sequence can be aligned independently; this also enables parallelism, and hence scalability to ultra-large datasets.

The main challenges of this kind of approach is accuracy, as each step affects the final alignment. For example, how the seed sequences are selected is important; random selection may be sufficient in many cases, but if the input set has substantial heterogeneity, then some other strategy may be needed. When the number of seed sequences is small, then sophisticated but computationally intensive multiple sequence alignment methods can be used, but alignment estimation often benefits from dense sampling, and so larger seed datasets may be preferable. The statistical model that is used to represent the seed alignment also has an impact, but the more complicated the model is, the more computationally expensive it can be to build and then to use later, in aligning the remaining sequences.

9.8 Co-estimation of alignments and trees

9.9 Current challenges

9.10 Further reading

Potential pitfalls in progressive alignments. Progressive alignment strategies are widely used, but have several vulnerabilities. First, errors in the early alignment mergers persist, since subsequent mergers never change the alignments they merge together. One strategy to deal with this is to attempt to correct errors in the resultant alignment; examples of such techniques include polishing (Berger and Munson, 1991; Hirosawa et al., 1995), which break the set of sequences into subsets and then re-align the induced sub-alignments. Other strategies attempt to avoid making errors in early alignments, using estimates of the consistency of the signal in the other alignments suggesting that a given pairs of sites should be aligned (Notredame et al., 2000; Do et al., 2005).

The impact of the guide tree on progressive alignment. Many studies have shown that the choice of the guide tree also has an impact on the resultant alignment, and hence on phylogenies computed on these alignments (Penn et al., 2010; Nelesen et al., 2008; Capella-Gutiérrez and Galbadón, 2013; Liu et al., 2009a; Toth et al., 2013). For example, Nelesen et al. (2008) studied the impact of the guide tree on alignment methods, and showed that improved phylogenetic accuracy can be obtained by first estimating a maximum likelihood
Multiple sequence alignment

tree on a MAFFT (Katoh et al., 2005) alignment. Nelesen et al. (2008) noted particular benefits in using Probcons (Do et al., 2005) with this guide tree, and called the resultant method “Probtree”. Capella-Gutiérrez and Galbadón (2013) showed that Prank (Loytynoja and Goldman, 2005) is very sensitive to guide trees, and Liu et al. (2009a); Toth et al. (2013) showed that improved results can be obtained by the use of carefully computed guide trees (maximum likelihood on good alignments) instead of Prank’s default guide tree.

Based on these observations, Capella-Gutiérrez and Galbadón (2013) recommended that alignment estimation methods should use the true tree (if possible), or else use an iterative co-estimation method that infers both the tree and the alignment.

Using the gaps in MSAs to compute phylogeny. Several studies have explored the use of gaps in alignments to help compute phylogenies. However, Capella-Gutiérrez and Galbadón (2013) found that the placement of gaps in an alignment results from the choice of the guide tree, and hence the gaps are not phylogenetically informative.

Template-based methods. Template-based methods are another category of alignment method that are most often used in the context of alignment of homologous protein sequences where some of the sequences have solved three-dimensional structures (Kemena and Notredame, 2009). Those sequences with known structures can be more accurately aligned, and a structurally-defined model of their common structural features can be built; this model is called a “template”. Once the template is computed, it can be used to add new sequences to the alignment.

There are several examples of methods that use this approach (Preusse et al., 2007; DeSantis et al., 2006; Neuwald, 2009; Gardner et al., 2012; Nawrocki et al., 2009; Nawrocki, 2009; Lytynoja et al., 2012; Nguyen et al., 2015b) (see pages 526-529 in (Morrison, 2006)). Some of these methods use curated seed alignments based on structure and function of well-characterized proteins or rRNAs; for example, the protein alignment method in Neuwald (2009) and the rRNA sequence alignment method in et al. (Gardner et al., 2012) use curated alignments. Constraint-based methods, such as COBALT (Papadopoulos and Agarwala, 2007), 3DCoffee (O’Sullivan et al., 2004) and PROMALS (Pei and Grishin, 2007), similarly use external information like structure and function, but then use progressive alignment techniques (or other such methods) to produce the final alignment. Clustal-Omega also has a version, called “External Profile Alignment”, that uses external information (in the form of alignments) to improve the alignment step.

Several studies (Pei and Grishin, 2007; Gardner et al., 2012; Neuwald, 2009; Deng and Cheng, 2011; Zhou and Zhou, 2005; Ortuno et al., 2013; Sievers et al., 2013) have shown that alignment methods that use high quality external knowledge can surpass the accuracy of some of the best purely sequence-based alignment methods. However, to date, none of these multiple sequence alignment methods (whether or not based upon external biological knowledge) have been tested for their impact on phylogenetic estimation; instead, they
have only been tested with respect to standard alignment criteria (e.g., SP-score), identification of functional or structural residues, or membership in a gene family. Thus, we do not know whether the improvements obtained with respect to traditional alignment accuracy metrics will translate to improvements in phylogeny estimation.

Pagan: a “phylogeny-aware” method. PAGAN (Lytynoja et al., 2012) is another member of this class of methods; however, it has some specific methodological differences to the others. First, unlike several of the others, it does not use external biological information (about structure, function, etc.) to define its seed alignment. Second, while the others tend to use either HMMs, profiles, or templates as a model to define the alignment of the remaining sequences, PAGAN estimates a tree on its seed alignment, and estimates sequences for the internal nodes. These sequences are then used to define the incorporation of the remaining sequences to the seed alignment. This technique is very similar to the technique used in PaPaRa (Berger and Stamatakis, 2011), which was developed for the phylogenetic placement problem. Thus, PAGAN is one of the “phylogeny-aware” alignment methods, a technique that is atypical of these seed-based methods, but shared by progressive aligners. PAGAN was compared to an HMM-based method (using HMMER on the reference alignment to build an HMM, and then using HMMALIGN to align the sequences to the HMM) on several datasets (Lytynoja et al., 2012). The comparison showed that PAGAN had very good accuracy, better than HMMALIGN, under low rates of evolution, and that both methods had reduced accuracy under high rates of evolution. They also noted that PAGAN failed to align some sequences under model conditions with high rates of evolution, while HMMER aligned all sequences; however, the sequences that both HMMER and PAGAN aligned were aligned more accurately using PAGAN.

Divide-and-conquer methods. Some alignment methods use a divide-and-conquer strategy in which the taxon set is divided into subsets (rather than the sites) in order to estimate the alignment; these include the mega-phylogeny method developed in Smith et al. (2009a), SATé (Liu et al., 2009a, 2011) and its successor PASTA (Mirarab et al., 2015), SATCHMO-JS (Hagopian et al., 2010), PROMALS (Pei and Grishin, 2007), and the method in (Neuwald, 2009). (The SATé and SATCHMO-JS methods co-estimate alignments and trees, and so are not strictly speaking just alignment methods.) Neuwald’s method is a bit of an outlier in this set, because the user provides the dataset decomposition, but we include it here for comparative purposes.

While the methods differ in some details, they use similar strategies to estimate alignments. Most estimate an initial tree, and then use the tree to divide the dataset into subsets. The method to compute the initial trees differs, with SATCHMO-JS using a neighbor joining (Saitou and Nei, 1987) (NJ) tree on a MAFFT alignment, SATé using a maximum likelihood tree on a MAFFT alignment, PROMALS using a UPGMA tree on k-mer distances, and mega-phylogeny using a reference tree and estimated alignment. (See the description of mega-phylogeny provided by in Roquet et al. (2013) for more details.)
The subsequent division into subsets is performed in two ways. In the case of mega-phylogeny, SATCHMO-JS, and PROMALS, the division into subsets is performed by breaking the starting tree into clades so as to limit the maximum dissimilarity between pairs of sequences in each set. In contrast, SATé-2 (Liu et al., 2011) and PASTA (Mirarab et al., 2015) removes centroid edges from the unrooted tree, recursively, until each subset is small enough (below 200 sequences). Thus, the sets produced by the SATé-2 decomposition do not form clades in the tree, unlike the other decompositions. Furthermore, the sets produced by the SATé-2 and PASTA decompositions are guaranteed to be small (the maximum size is controlled by the user, and by default is 200) but are not constrained to have low pairwise dissimilarities between sequences.

Alignments are then produced on each subset, with PROMALS, SATé, SATé-2, PASTA, and mega-phylogeny estimating alignments on each subset, and SATCHMO-JS using the alignment induced on the subset by the initial MAFFT alignment. These alignments are then merged together into an alignment on the full set, but the methods use different techniques. PROMALS and mega-phylogeny use seed-based alignment methods to merge the alignments together, SATCHMO-JS, SATé, SATé-2 use progressive alignment techniques, and PASTA uses a combination of progressive alignment and transitive closure. PROMALS also uses external knowledge about protein structure to help determine how to merge the subset alignments together.

The method in Neuwald (2009) shares many features with these four methods, but has some unique features that are worth pointing out. First, like SATCHMO-JS and PROMALS, Neuwald’s method can only be used on proteins (mega-phylogeny, SATé, SATé-2, and PASTA can be used on both nucleotides and protein sequences). Neuwald’s method requires the user to provide a dataset decomposition and also a manually curated seed alignment reflecting structural and functional features of the protein family. The algorithm operates by estimating alignments on the subsets using simple methods, and then uses the seed alignment to merge the subset-alignments together.

Running times differ between these methods, with the progressive alignment methods being the most computationally intensive, and the seed-alignment methods the fastest. Thus, since SATCHMO-JS, SATé, and SATé-2 rely upon progressive alignment, their running times are longer; SATé-2 is the most scalable of these, but the largest dataset it has been able to successfully analyze has only 50,000 sequences. The seed-alignment methods, such as Neuwald’s method, PROMALS, and mega-phylogeny, are very scalable once the template for their seed alignments is computed; however, if the template construction depends on expert curation, this first step can be very labor-intensive. Mega-phylogeny has been used to analyze a dataset with more than 50,000 nucleotide sequences (Smith et al., 2011), and Neuwald’s method has been used to analyze a dataset with more than 400,000 protein sequences. UPP is another seed-alignment method that has been shown to be highly accurate on very large datasets, including one with 1,000,000 sequences (Nguyen et al., 2015b).

Finally, PASTA’s strategy for merging alignments is a mixture of progressive alignment and transitivity, and as a result it is much more scalable than the purely progressive align-
ment strategies. As shown in Nguyen et al. (2015b), PASTA was able to produce highly accurate alignments on very large datasets, including one with 1,000,000 sequences. We will return to the description of PASTA’s algorithm design in Chapter 12.

Neuwald’s method, SATCHMO-JS, and PROMALS, have been assessed using protein alignment benchmarks, and shown to give excellent accuracy compared to standard methods. Ortuno et al. (2013) explored the conditions in which PROMALS gave improvements over the other methods, and showed that the conditions in which the improvements were substantial were when the sequences were close to the ‘twilight zone’ (i.e., almost random with respect to each other), which is where sequence homology is difficult to detect, and information about structure is the most helpful.

PASTA is generally more accurate than SATé and SATé-II (Mirarab et al., 2015), indicating that the combined use of transitivity and progressive alignment to merge two alignments provides better accuracy than a purely progressive alignment technique. PASTA’s accuracy is maintained to large datasets, even with 1,000,000 sequences (Nguyen et al., 2015b).

Combining information from different alignments. A relatively new approach in multiple sequence alignment uses a collection of alignments, each computed using potentially different techniques, for the same set \( S \) of sequences, and then build a new alignment on \( S \) by examining the support for the different possible homologies. The best known method of this type is T-Coffee (Notredame et al., 2000), which uses a library of MSAs to produce a final MSA.

9.11 Review questions

9.12 Homework problems

1. Let \( s \) and \( s' \) be two sequences, where \( s = GGATT \) and \( s' \) is obtained from \( s \) by substituting the \( A \) in \( s \) by a \( C \), and then appending \( AT \) to the end of \( s \).
   - What is \( s' \)?
   - What is the true pairwise alignment of \( s \) and \( s' \)?

2. Let \( s = AACT \) and \( s' = CTGG \) be two sequences, and consider the pairwise alignment of \( s \) and \( s' \) given in Table 9.4. Describe an evolutionary history relating \( s \) and \( s' \) for which the given alignment would be the true pairwise alignment.

3. Let an indel and a substitution each have unit cost, and let \( s = AATTAAG \) and \( s' = TTAAGC \). Use the dynamic programming algorithm to compute the minimum edit distance between these two sequences (i.e., the entries of the matrix should always be non-negative, and should represent the least cost of any transformation of the associated prefixes). Show all entries in the matrix you compute using this dynamic programming algorithm, and the optimal pairwise alignment that you obtain.

4. Give a dynamic programming algorithm (in pseudo code!) to compute the longest common subsequence between two strings \( A \) and \( B \). What are your subproblems? How do
you set the boundary cases? How do you solve subproblems given other subproblems? What is the order in which you fill in your DP matrix? Analyze the running time.

5. Let $S$ and $S'$ be two DNA sequences, with $S$ of length $L$ and $S'$ of length $L'$. Give a polynomial time dynamic programming algorithm to determine the length of the longest common subsequence of $S$ and $S'$. (Note that a common subsequence is not the same thing as a common substring; for example, AAA is a common subsequence of $S=$ATTGATA and $S'=$TAGGATCA, but AAA is not a substring of either $S$ or $S'$.)

6. Let $s_1 = AC, s_2 = ATAC,$ and $s_3 = CAG.$ Suppose that insertions, deletions, and substitutions each have unit cost, and let $T$ be a tree with $s_1, s_2,$ and $s_3$ at the leaves, and an internal node $X$.
   - Prove or disprove: there is only one optimal solution to tree alignment for this input, and it has $X = s_1$.
   - Draw the multiple sequence alignment implied by setting $X = s_1$. What is the SOP cost of this alignment? What is its tree alignment cost?

7. Let $s_1 = ATA, s_2 = AAT,$ and $s_3 = CAA.$ Suppose all insertions, deletions, and substitutions have unit cost. Find an optimal solution to the Tree Alignment problem on the tree $T$ with one internal node and three leaves (i.e., find the best sequence to label the internal node). Is your solution unique? If so prove it, or else show another sequence with as good a score.

8. Let $S$ be an arbitrary set of sequences and assume that insertions, deletions, and substitutions have unit cost. Let $T$ be a tree with one internal node and all the sequences in $S$ at the leaves. Let $M$ be the tree alignment on $S$ obtained by assigning a sequence to the internal node of $T$. Prove or disprove: the SOP cost of $M$ is at least the Tree Alignment cost of $M$.

9. Suppose you have two sequences $x$ and $y$ of the same length, and the alignment of $x$ and $y$ in which there are no gaps. Let $P_{x,y}$ denote the associated profile for the gap-free alignment of $x$ and $y$, and let $S_{x,y}$ denote the set of all sequences that could be generated by $P_{x,y}$ (i.e., the set of all sequences that have non-zero probability of being generated by $P_{x,y}$).
   - Prove that $\{x, y\} \subseteq S_{x,y}$ (i.e., that $x$ and $y$ always have non-zero probability of being generated by $P_{x,y}$ when the profile is based on a gap-free alignment).
   - Given an example of $x$ and $y$ where $S_{x,y} = \{x, y\}$ (i.e., every other sequence has zero probability of being generated by $P_{x,y}$).
   - Given an example of $x$ and $y$ where $|S_{x,y}| > 2$.

10. Consider the gap-free alignment given for sequence dataset $s_1 = AACTAAG, s_2 = AATATAG, s_3 = ATAAAAG, s_4 = TTATAG,$ and $s_5 = TATATAG.$
    - Write down the profile Hidden Markov Model that represents this multiple sequence alignment, and that doesn’t include any insertion or deletion states. (Do not include any correction for unseen nucleotides in any position.)
    - What are the most likely sequences to be generated by this model? (If there is only one, say so - and otherwise give them all.)
9.12 Homework problems

- What is the probability of generating sequence AACTAAG?
- What is the probability of generating sequence CTAAAAA?

11. Suppose you have a profile HMM $M$ where the match states only emit letter $A$ and the insertion states only emit letter $T$.
- Suppose $M$ has no deletion states, and the insertion states have no self-loops. Is $M$ a Hidden Markov Model? More precisely, if you know $M$ (the graph and its associated numerical parameters) and you are given a sequence $s$, can you determine the path taken through $M$ to generate $s$? Does the answer to your question depend on $s$ or the details about $M$?
- Same question as above, but now suppose the insertion states have self-loops.
- Same question as above, but now suppose that the insertion states have self loops and that the model has deletion states.

12. In the text, we showed that the string AAA could be generated by two paths through the model in Figure 9.3 where for all match and insertion states the probability of emitting each nucleotide is non-zero, and transition probabilities from the match state to the adjacent insertion state are in $(0, 1)$. Give another two paths through the model that can generate this string.
Constructing species trees under the multi-species coalescent model

10.1 Introduction

One of the fascinating challenges in estimating the evolutionary history of a set of species is that different regions within the genomes can evolve differently, due to various biological phenomena. One of the most obvious causes for this difference is horizontal gene transfer, whereby DNA is transferred from the genome of one species into that of another, so that the evolutionary history of the set of species is no longer tree-like but rather requires a more general graphical model called a “phylogenetic network”. Another biological process that requires a phylogenetic network representation is hybrid speciation, whereby two different species have viable offspring, which are called hybrids. However, there are biological processes that cause different parts of the genomes to evolve differently, where the species history is still correctly modelled as a tree; for example, gene duplication and loss and incomplete lineage sorting. In these cases, the genes are seen to evolve within the branches of a species tree, so that the genes can have their own histories, which can differ from that of the species tree.

In this chapter, we will examine the question of how to infer a species tree in the presence of gene tree incongruence due to incomplete lineage sorting (ILS). We discuss three types of methods:

• Summary methods. In this category, the input is a set of gene trees, and then a species tree is estimated from the gene trees. For some methods, the gene trees must be rooted, but some methods are designed for unrooted gene trees. Summary methods are the most popular because they tend to be reasonably fast.

• Co-estimation methods. In this case, the input is a set of sequence alignments for each of a number of loci, and then the gene trees and species tree are estimated together. The advantage of co-estimation methods is potential improvement in accuracy, but at the expense of running time.

• Site-based methods. The input for these methods - as for co-estimation methods - is a set of sequence alignments. However, the species tree is estimated directly from the site patterns, and the gene trees are not estimated. The advantage of these methods is that they do not depend on the ability to reconstruct accurate gene trees, either directly (as in summary methods) or indirectly (as in co-estimation methods).
In the remainder of this chapter, we discuss the basic theory of coalescent-based species tree estimation, beginning with the multi-species coalescent model. Then we present some of the major algorithmic approaches to species tree estimation.

10.2 Theoretical foundations

10.2.1 The multi-species coalescent model

The multi-species coalescent model treats each species as a population of individuals, with each individual having a pair of alleles for each gene. This basic perspective – of treating a species at a given point in time as a population of individuals - is an important and powerful perspective that leads to substantial insights into the process of evolution, and that also introduces new computational problems.

Over time, different alleles assort into different populations, and this means it becomes possible for speciation events to lead to different species having different sets of alleles among its individuals. When this happens, it becomes possible for the trees computed on the basis of a single allele from each selected individual to be different from each other, and from the species tree. When this happens, the forward process is called “incomplete lineage sorting”.

The multi-species coalescent model was developed to provide a mathematically tractable way to model incomplete lineage sorting. In this process, one or more individuals are selected from each species at the leaves of a species tree, and one allele from one gene within the genomes of these individuals is selected. Then, a gene tree is created by moving from the individuals (here, leaves of the gene tree) to their parents – but only to the specific parent for each individual from whom they inherited that allele. Since all organisms have a common ancestor, this “backwards” process eventually creates a tree, which we call a “gene tree”. Note that the gene tree topology depends on the choice of individuals within each species. Furthermore, if more than one individual is selected within each species, then automatically the gene tree will not be identical to the species tree since it will have extra leaves. However, even if only one individual is selected for each species, the gene tree may not be identical to the species tree!

In the multi-species coalescent model, each branch $e$ of the species tree is labelled with elapsed time $t(e)$ and the population size $\text{pop}(e)$. Under the assumptions of random mating and constant population size within each branch, the chances of two lineages coalescing on a branch decreases with the population size (larger populations make it less likely that two lineages will have the same common ancestor in the previous generation). Furthermore, the number of generations that took place within that branch of the species tree also impacts the chances of two lineages coalescing – fewer generations decrease the probability, and more generations increase the probability. The ratio between the elapsed time and the population size on edge $e$ is called the “coalescent branch length” (or “branch length in coalescent units”), and we will refer to this as $l(e)$. Thus, as $l(e) \to \infty$, the probability of two lineages coalescing on the edge approaches 1, while as $l(e) \to 0$ the probability of two
Constructing species trees under the multi-species coalescent model

lineages coalescing approaches 0. (While the estimation of the specific parameters \(t(e)\) and \(\text{pop}(e)\) is interesting for many applications, interpreting branch lengths \(l(e)\) in terms of the population size and elapsed time generally requires additional information, such as fossil data.) Under this model, if \(k > 1\) lineages enter an edge then any two of the lineages in the set have equal probability of coalescing first on the edge.

Under the multi-species coalescent model, we fully describe the process by the pair \((T, \theta)\), where \(T\) is a rooted binary tree with leaves labelled by a set of species, and branch lengths on the edges of the tree given in coalescent units. However, for small enough species trees, we can equally well describe this by a pair \((T, \theta')\), where the parameters in \(\theta'\) are the probabilities of coalescence on each edge.

Note that the multi-species coalescent model defines the probability for each gene tree contained within the species tree, and so defines a probability distribution on the gene trees. However, can we use this distribution to estimate the species tree?

10.2.2 Anomalous gene trees

One of the surprising challenges in estimating the species tree from a set of gene trees is that the most probable gene tree is not necessarily identical to the species tree! When this happens, the most probable gene tree is called an “anomalous gene tree”, and the model species tree is said to be in the “anomaly zone”. Here we present a very simple analysis showing that there are no anomalous rooted 3-leaf gene trees or unrooted 4-leaf gene trees, but that there are anomalous rooted 4-leaf gene trees.

Suppose we have the rooted binary model species tree \(T\), and the probability distribution on the set of gene trees defined by the coalescence probabilities on the edges of \(T\). We consider the simplest case first – where \(T\) has only three leaves \(a, b,\) and \(c,\) and \(a\) and \(b\) are siblings. Let \(e\) be the edge from the root of \(T\) to the parent of \(a\) and \(b\). We begin by asking what the probability is of the gene tree \(t\) with the same topology as \(T\), under the assumption that we pick one individual from each species.

Consider the gene tree as it grows from the leaves of the species tree to the root. If the lineages coming from \(a\) and \(b\) coalesce on edge \(e\), then the gene tree \(t\) is topologically identical to the species tree \(T\). To obtain a gene tree with a different topology from the species tree, therefore, there must be no coalescent event on the edge \(e\). Hence, the three lineages (one from \(a\), one from \(b\), and one from \(c\)) will all “enter” the edge above the root, at which point any two of them will have equal probability of coalescing first. If the first pair to coalesce comes from \(a\) and \(b\), then we still obtain a gene tree with topology equal to that of \(T\). Furthermore, under this model, the probabilities of the two gene trees with different topologies from \(T\) are equal. Putting this together, if \(p_0\) is the probability of coalescing on edge \(e\), then letting \(t\) denote the gene tree, we obtain the probabilities of each gene tree topology as follows:

- \(\Pr(t = (a, (b, c))) = \frac{1-p_0}{3}\)
- \(\Pr(t = ((a, c), b)) = \frac{1-p_0}{3}\), and
• $\Pr(t = ((a,b),c)) = p_0 + \frac{1 - p_0}{3}$.

Note that the probability of each gene tree topology other than the species tree is strictly less than $\frac{1}{3}$, and hence the probability of the gene tree topology matching the species tree is strictly greater than $\frac{1}{3}$. In other words, the probability of generating a gene tree with exactly the same tree topology as the species tree is strictly greater than the probability of producing either of the other two gene trees, and the other two gene trees have equal probability of being generated. Another way of putting this is that the most probable gene tree topology is the species tree topology.

Now consider a four-leaf rooted species tree $T$. Will it still be the case that the rooted gene tree topology with the highest probability will be identical to the species tree topology? In this case, the answer is no. Specifically, for the species tree $(a,(b,(c,d))))$, it is possible to set the branch coalescence probabilities to very small values so that the most probable rooted gene tree will be topologically different from the species tree! This is an example of the anomaly zone (Degnan and Rosenberg, 2006; Rosenberg, 2002; Degnan, 2013; Rosenberg, 2013). On the other hand, it is not too hard to show that the most probable unrooted gene tree is topologically identical to the unrooted species tree, no matter what the four-leaf species tree and branch coalescence probabilities are!

This nice observation about unrooted four-leaf gene trees and species trees does not extend to unrooted five-leaf gene trees and species trees, though, since for five leaves there are model species trees for which the most probable five-leaf unrooted gene tree is topologically different from the unrooted species tree. In other words, the anomaly zone exists for unrooted gene trees with five or more leaves, for rooted gene trees with four leaves, but not for unrooted gene trees with four leaves or for rooted gene trees with three leaves.

We summarize this discussion with the following theorem:

**Theorem 10.1** For all rooted 3-leaf species trees with branch lengths in coalescent units, the most probable rooted gene tree is topologically identical to the rooted species tree. For all rooted 4-leaf species trees with branch lengths in coalescent units, the most probable unrooted gene tree is topologically identical to the unrooted 4-leaf species tree.

We will show how to use these results in producing highly accurate species tree estimation methods that can handle gene tree incongruence due to incomplete lineage sorting.

### 10.2.3 Concatenation under the multi-species coalescent model

Concatenation is a standard method for estimating trees from multiple loci, and operates in the following manner. The input is a set of multiple sequence alignments, with one alignment for every locus. The first step operates by concatenating all the alignments into one large “superalignment” (also called a supermatrix). In the second step, a tree is estimated on the superalignment using the preferred phylogeny estimation method, such as maximum likelihood heuristics. Here we describe concatenation analyses using maximum likelihood.

The simplest version of the concatenation analysis using maximum likelihood (CA-ML)
Constructing species trees under the multi-species coalescent model seeks the maximum likelihood model tree \((T, \Theta)\), where \(T\) is a binary tree and \(\Theta\) is the set of numeric model parameters. For example, under the Jukes-Cantor model, \(\Theta\) contains the branch lengths for every edge in \(T\). This is called an unpartitioned maximum likelihood analysis. Note that under this analysis, all loci are assumed to evolve down a single model tree.

Under a more general approach, the different loci are allowed to evolve down different model trees, but the assumption is that all the model trees share the same topology, and hence only differ in their numeric parameters. Concatenation analyses under this assumption can therefore require the estimation of numeric parameters for each locus, and are therefore more computationally intensive. These analyses are called fully partitioned maximum likelihood analyses.

Roch and Steel (2015) proved that unpartitioned maximum likelihood is statistically inconsistent (and even positively misleading) under the multi-species coalescent model, thus establishing that this common way of estimating phylogenies is not a statistically rigorous way of estimating species trees from multi-locus data.

### 10.3 Summary methods

Recall that summary methods estimate the species tree by combining gene trees. There are many different types of summary methods, but at a top-level they can be distinguished by whether they do or do not require rooted gene trees.

#### 10.3.1 Summary methods that require rooted gene trees

We begin with the case of estimating the rooted species tree from rooted gene trees. If \(T\) is a rooted binary species tree with three leaves, to estimate \(T\) from gene trees is easy: we would simply count the number of times each of the three possible gene trees occurs, and return whichever one appears the most frequently. By Theorem 10.1, as the number of genes increases, with probability approaching 1, the gene tree that appears the most frequently will be the true species tree. Thus, we have a statistically consistent method for estimating the species tree.

Now, suppose \(T\) is a rooted binary tree on leafset \(S = \{s_1, s_2, \ldots, s_n\}\), and let \(\mathcal{T} = \{t_1, t_2, \ldots, t_k\}\) be the gene trees we observe. How can we estimate \(T\) given \(\mathcal{T}\)? Since the most frequent rooted gene tree may be different from \(T\), we cannot just follow the same process as we did for the case with 3 leaves, but we can do something else. For every set \(A\) of three leaves \(\{s_i, s_j, s_k\} \subset S\), we will constrain each of the gene trees in \(\mathcal{T}\) to \(A\). This defines a set of rooted gene trees, each defined only on the set \(A\) of species, which we denote \(\mathcal{T}_A\). By Theorem 10.1, the most probable gene tree on \(A\) will be topologically identical to the true species tree on \(A\), which is the subtree of \(T\) induced by \(A\). Hence, we can estimate the species tree on \(A\) by just using the most frequent gene tree in \(\mathcal{T}_A\). As the number \(k\) of gene trees goes to infinity, the probability that the most frequent gene tree is the true species tree on \(A\) approaches 1.
We do this for every set $A$ of three species, and thus assemble a set of rooted three-leaf trees (one tree for every three species), which should be equal to the species tree on their species subset with probability approaching 1. If all these rooted three-leaf trees are correct (i.e., equal to the true rooted species tree on the three species), then we can construct the true species tree $T$ using the ASSU algorithm of Aho et al. (1978), described in Chapter 3. Therefore, we apply one of these algorithms to the set of rooted three-leaf trees. If the algorithm returns a tree, this is our estimate of the species tree. Else, the algorithm rejects the dataset, saying the three-leaf trees are not compatible. While rejection is always possible, as the number of genes goes to infinity the probability of rejection goes to 0, and the probability of returning the true species tree goes to 1.

In other words, we have described a very simple algorithm (SRSTE, for a “simple rooted species tree estimation”) for inferring the rooted species tree from rooted gene trees, which we now summarize.

**SRSTE: a simple algorithm to construct rooted species trees under the multi-species coalescent model.** The input is a set of rooted gene trees, each on the same set $S$ of $n > 3$ species, and the output is either an estimated tree $T$ or “Fail”.

- **Step 1:** For all three leaves $a, b, c$, determine the most frequent induced gene tree on $a, b, c$, and save it in a set $\mathcal{T}$.
- **Step 2:** Apply the ASSU algorithm (Aho et al., 1978) to the set $\mathcal{T}$ of rooted three-leaf gene trees. Note that this algorithm only returns a tree if the set $\mathcal{T}$ is compatible; else it returns Fail.
  - If $\mathcal{T}$ is compatible, then the previous step outputs a tree $T$, which we return.
  - Else $\mathcal{T}$ is not compatible, and the previous step does not return a tree; in this case, we output “Fail”.

**Theorem 10.2** Let $T$ be a model rooted species tree on $n > 3$ leaves, and suppose we are given a set of randomly sampled gene trees. Then the probability that SRSTE estimates $T$ correctly converges to 1 as the number of gene trees increases. In other words, SRSTE is a statistically consistent method for estimating the rooted species tree from rooted gene trees under the multi-species coalescent model.

The SRSTE method is a very simple statistically consistent method for estimating species trees from rooted gene trees under the multi-species coalescent model. However, by its design it is likely to fail to return a tree under many conditions; hence, this is a theoretical construct more than a useful tool.

However, many coalescent-based summary methods have been developed for constructing species trees from rooted gene trees. For example, MP-EST (maximum pseudo-likelihood estimation) (Liu et al., 2010), STEM (Kubatko et al., 2009), STAR (Liu et al., 2009b), and GLASS (Mossel and Roch, 2011), are all summary methods that estimate species trees from rooted gene trees.
For many biological datasets, the input set of gene trees contains some unrooted trees, and hence methods such as SRSTE cannot be applied. Yet statistically consistent methods for estimating the unrooted form of the species tree is possible under the multi-species coalescent. Here we describe some of the methods and theory for this problem.

We begin with a very simple method to estimate species trees by combining unrooted gene trees. This method, which we call SUSTE (for simple unrooted species tree estimation), is the unrooted equivalent of SRSTE.

**SUSTE: a simple quartet-based method to estimate species trees under the multi-species coalescent model.** Recall Theorem 10.1, which said that there are no anomalous four-leaf unrooted gene trees. What this means is that under the multi-species coalescent, for all rooted four-leaf model species trees (tree topologies and coalescent unit branch lengths), the most probable unrooted four-species gene tree is identical to the unrooted species tree. Hence, a simple method suffices to estimate the unrooted species tree. As with SRSTE, we do the following:

The input is a set of unrooted gene trees, each on the same set $S$ of $n > 4$ species, and the output is either an estimated tree $T$ or “Fail”.

- **Step 1:** For all sets $A$ of four leaves, determine the most frequent induced gene tree on $A$ and store it in a set $\mathcal{T}$.
- **Step 2:** Apply the All Quartets Method from Chapter 3.4.2 to the set $X$.
  - If the set $\mathcal{T}$ of four-leaf trees is compatible, then the All Quartets Method will return a tree $T$, which we return.
  - Else the set $\mathcal{T}$ of four-leaf trees is not compatible, and the All Quartets Method will not return a tree. In this case, we return “Fail”.

The SUSTE method is statistically consistent for constructing unrooted species trees from unrooted gene trees under the multi-species coalescent model. We summarize this with the following theorem:

**Theorem 10.3** Let $T$ be a model rooted species tree on $n \geq 4$ leaves, and suppose we are given a set of randomly sampled unrooted gene trees. Given a large enough number of gene trees, then with high probability the set of dominant quartet trees will all be topologically identical to the true species tree on their leafset, and so the SUSTE algorithm will return the true unrooted species tree. Hence, the probability that SUSTE returns the unrooted version of $T$ converges to 1 as the number of gene trees increases, and so SUSTE is a statistically consistent method for estimating the unrooted species tree from unrooted gene trees under the multi-species coalescent model.

**Proof** For any model rooted species tree $T$ and for any four leaves $a, b, c, d$, the most probable unrooted gene tree on $a, b, c, d$ is topologically identical to the unrooted species tree on $a, b, c, d$. Therefore, for any $\varepsilon > 0$, there is a positive number $K$ so that if we are
given \( k \geq K \) gene trees then for all four species, the most frequently observed unrooted gene tree on those four species is the same as the unrooted species tree on those four species, with probability at least \( 1 - \varepsilon \). The theorem follows.

As we noted for SRSTE, the SUSTE method is best seen as a mathematical construct rather than as a method to use in practice, since it is likely to return Fail, instead of returning a tree, on many inputs. Yet, there are several statistically consistent summary methods that have been developed to species trees from unrooted gene trees under the multi-species coalescent model. Some of these, such as ASTRAL (Mirarab et al., 2014a; Mirarab and Warnow, 2015) and BUCKy-pop (the population tree computed by BUCKy) (Larget et al., 2010), are also quartet-based, and use the same basic reasoning as in SUSTE to establish statistical consistency.

The Quartet Median Tree Problem
Suppose the input set of unrooted gene trees is \( \mathcal{T} = \{ t_1, t_2, \ldots, t_k \} \), and each gene tree is on taxon set \( S \). Let \( d(\mathcal{T}, t_i) \) be the number of four-taxon subsets of \( S \) on which \( \mathcal{T} \) and \( t_i \) differ, and let \( d(\mathcal{T}, \mathcal{T}) = \sum_i d(\mathcal{T}, t_i) \). Then the quartet-median tree of \( \mathcal{T} \) is the tree \( T \) that has the smallest total quartet distance to \( \mathcal{T} \); i.e.,

\[
T_{\text{median}} = \arg \min_T \{ d(T, \mathcal{T}) \}. 
\]

So consider the optimization problem that takes as input the set \( \mathcal{T} \) and returns a quartet-median tree \( T \) for the set. Suppose we could solve this problem exactly. What theoretical properties would an algorithm providing an exact solution to this problem have?

**Theorem 10.4** Let \( \mathcal{T} \) be a set of \( k \) true gene trees and let \( T^* \) be the true species tree. Then for any \( \varepsilon > 0 \) there is a \( K > 0 \) so that if \( k > K \) then the probability that the true species tree is the unique quartet-median tree is at least \( 1 - \varepsilon \).

**Proof** Note that the quartet-median tree \( T \) of \( \mathcal{T} \) maximizes \( \sum_{t \in \mathcal{T}} \text{Sim}_q(T, t) \), where \( \text{Sim}_q(T, t) \) is the number of four-taxon subsets of \( S \) on which \( T \) and \( t \) agree. Note then that \( \sum_{t \in \mathcal{T}} \text{Sim}_q(T, t) = \sum_X N(T, X) \), where \( X \) is a four-taxon subset of \( S \) and \( N(T, X) \) is the number of trees in \( \mathcal{T} \) that agree with \( T \) on set \( X \). Finally, for any \( \varepsilon > 0 \), there is a \( K > 0 \) so that given \( k > K \) genes then with probability at least \( 1 - \varepsilon \) the dominant gene tree on any set \( X \) of four taxa will be equal to the species tree on \( X \). Hence, for such datasets, for every subset \( X \) of four species, \( N(T, X) \) attains its maximum at \( T = T^* \). Equivalently, for such datasets, \( \text{Sim}_q(T, t) \) attains its maximum at \( T = T^* \). Hence, in the limit (as the number of genes increases), the true species tree \( T^* \) is the quartet-median tree of \( \mathcal{T} \) with probability converging to 1.

Hence,

**Corollary** An exact solution to the quartet-median tree problem is a statistically consistent method for estimating the species tree under the multi-species coalescent.

The quartet-median tree optimization problem has not been shown to be NP-hard, but similar problems are NP-hard. For example, the Maximum Quartet Consistency problem
is NP-hard, even when the set of quartet trees contain a tree on every four leaves (Jiang et al., 2001). However, no polynomial time method for this problem have been found, and we conjecture that the quartet-median problem is NP-hard. However, as shown in Mirarab et al. (2014a); Mirarab and Warnow (2015); Bryant and Steel (2001), a constrained version of this problem can be solved in polynomial time.

The Constrained Quartet-Median Tree Problem. Recall that $C(T)$ denotes the set of bipartitions of a tree $T$. The constrained quartet-median tree problem seeks the best solution to the quartet-median tree optimization problem, but restricts the set of allowed trees to draw their bipartitions from a set $X$. Thus, if $X$ is very small, the search space is quite limited, but if $X$ is all possible bipartitions then there is no constraint on the search. We now formally state this approach.

The input is the set $\mathcal{T}$ on taxon set $S$ and a set $X$ of bipartitions on set $S$, and the objective is a tree $T$ on $S$ such that $T$ minimizes the total quartet distance to $\mathcal{T}$, subject to $C(T) \subseteq X$.

Note that if we set $X$ to be all possible bipartitions on $S$, then this problem is identical to the Quartet-Median Tree problem, and also that $|X|$ is exponential in $S$ in this case. However, for smaller sets, $|X|$ may be only polynomial in the number of gene trees and number of taxa. For example, if $X = \cup_{t \in \mathcal{T}} C(t)$ then $|X|$ is $O(nk)$. Furthermore, as shown in Mirarab et al. (2014a); Mirarab and Warnow (2015); Bryant and Steel (2001), the optimal constrained quartet-median tree can be found in time that is polynomial in $n = |S|, k = |\mathcal{T}|$, and $|X|$, if we require that the solution be a fully resolved tree (i.e., binary).

It is easy to see that if $X$ is all possible bipartitions, then the optimal constrained quartet-median tree is a statistically consistent way of estimating the species tree. Interestingly, the following is also true:

Theorem 10.5 (From Mirarab and Warnow (2015)) : Let $T$ be a model species tree, and let $\mathcal{T}$ be a set of gene trees generated at random by $T$. If $C(t) \subseteq X$ for every $t \in \mathcal{T}$, then an optimal constrained quartet-median tree is a statistically consistent estimate of $T$ under the multi-species coalescent model.

Proof We know that the unconstrained quartet-median tree problem is statistically consistent under the multi-species coalescent model, and the proof is based on the fact that when the set of gene trees is large enough then $T$ will have an optimal score. All we need to do to prove that the constrained quartet-median tree problem remains statistically consistent, even when $X$ is the set of bipartitions from the input gene trees, is prove that the bipartitions of $T$ are contained in the set $X$. Note that for every branch of $T$ there is probability of coalescence on the edge that falls strictly between 0 and 1. Hence, the probability that $T$ is the set of gene trees is strictly greater than 0, and so with high probability, given a large enough number of gene trees, $T$ will be in the set of gene trees. Another argument for this would point out that for any given edge $e$ in $T$, with probability greater than 0 all lineages coalesce on $e$, and when this happens the bipartition defined by $e$ appears in some gene tree.

Hence, given $\epsilon > 0$, from a large enough number of gene trees every edge in $T$ will
appear in at least one gene tree with probability at least $1 - \epsilon$. When this occurs, $T$ is a feasible solution to the quartet-median tree problem. Also, when the set of genes is large enough, then for all sets $A$ of four taxa the dominant quartet tree on $A$ will be the true species tree on $A$, with probability at least $1 - \epsilon$. Therefore, when the set of genes is large enough, then with high probability the true species tree will be an optimal solution to the constrained quartet-median tree problem, in which $X$ is the set of bipartitions from the input gene trees.

\[ \square \]

Distance-based estimation. Another type of summary method for estimating species trees from unrooted gene trees produces a dissimilarity matrix from the input set of unrooted gene trees, and then compute a species tree from the dissimilarity matrix. There are many methods that use this approach, including NJst (Liu and Yu, 2011).

Given a gene tree $t$ on taxon set $S$, we define the topological distance between two leaves $x, y$ to be the number of edges in the path between $x$ and $y$ in $t$. Then, given an input set of $k$ unrooted gene trees, the matrix $M$ of average topological distances between every pair of leaves (i.e., the “internode” distance matrix) is computed.

**Theorem 10.6** (From Allman et al. (2016):) *Given a set of $k$ true gene trees sampled randomly from the distribution defined by the model species tree $T$, the matrix $M$ of average inter-leaf topological distances converges in probability, as $k \to \infty$, to an additive matrix $A$ that corresponds to $(T, w)$, for some edge-weighting function $w$.*

In other words, if we can sample enough true gene trees, then the matrix $M$ will converge to a matrix which is additive for the true species tree. Therefore, if we apply distance-based tree estimation methods to the matrix $M$, we can estimate the species tree in a statistically consistent manner. The NJst method (Liu and Yu, 2011) was the first method to use such an approach, and was based on the neighbor joining (Saitou and Nei, 1987) method described earlier. Since neighbor joining has a positive safety radius (see Chapter 5.7), NJst is statistically consistent under the multi-species coalescent model.

However, other methods can also be used to estimate the species tree from the matrix of internode distances, and Vachaspati and Warnow (2015) showed that using FastME (Desper and Gascuel, 2002) improved the topological accuracy of the species tree. Furthermore, when the matrix of internode distances has missing data, using the BioNJ* from Criscuolo and Gascuel (2008) also results in improved accuracy. Thus, ASTRID (Vachaspati and Warnow, 2015), which uses FastME when the distance matrix has no missing entries and otherwise uses BioNJ*, is an improvement of NJst that is statistically consistent under the MSC. Furthermore, as shown in Vachaspati and Warnow (2015), ASTRID is comparable to ASTRAL-2 on many datasets in terms of accuracy, and is generally much faster.

So far, the proofs of statistical consistency for the algorithms we have presented assume that the input is a set of true gene trees. Yet, estimated gene trees typically have some estimation error. Hence, in biological data analysis, the input to the coalescent-based species tree method will be a set of estimated gene trees, and many of these will have some estima-
tion error. The question we ask, then, is what can we say about species trees estimated using coalescent-based methods, using estimated gene trees that have some estimation error?

The answer, unfortunately, is essentially this – we don’t know. We have no theory yet that addresses this question, at least not for any of the standard coalescent-based methods! See Roch and Warnow (2015) for a discussion of this issue.

10.4 Site-based methods

Another type of coalescent-based method uses individual sites within different unlinked loci, and estimates the species tree from the distribution it obtains on site patterns. Examples of this type of approach are SNAPP (Bryant et al., 2012), METAL (Dasarathy et al., 2015), SMRT (super-matrix rooted triples) (DeGiorgio and Degnan, 2010), and SVDquartets (Chifman and Kubatko, 2014). This type of approach also has an advantage over summary methods in that they do not depend on having accurate gene trees. However, much less is known about the accuracy of these methods compared to summary methods.

We discuss SVDquartets here. The SVDquartets method is actually a technique for estimating quartet trees from the concatenated multiple sequence alignment; once these quartet trees are computed, it relies on methods that combine quartet trees in order to produce a species tree. The theoretical foundation of SVDquartets assumes that the gene sequences all evolve under a strict molecular clock, and uses this assumption to estimate the quartet trees on every set of four species. Hence,

**Theorem 10.7** Let $T$ be a model species tree, let $\mathcal{T}$ be a set of $k$ gene trees that evolve within $T$ under the multi-species coalescent model, and let $S_1, S_2, \ldots, S_k$ be sequence alignments for each of the $k$ gene trees, where sequence evolution is under the strict molecular clock. Assume that $\Phi$ is a method for combining quartet trees into a species tree that is guaranteed to return $t$ given $Q(t)$ for all trees $t$. Then, SVDquartets followed by $\Phi$ is statistically consistent under the multi-species coalescent model.

10.5 Co-estimation of gene trees and species trees

Another type of coalescent-based method operates by co-estimating the gene trees and the species tree from the sequence alignments for the different loci; this approach has the benefit of not depending on an accurate gene tree for each gene. The most computationally efficient of this type of method is *BEAST (Heled and Drummond, 2010). As shown in Bayzid and Warnow (2013), gene trees computed by *BEAST can be more accurate than maximum likelihood trees estimated on individual gene sequence alignments. While *BEAST can be highly accurate, its running time can be excessively large, so that *BEAST analyses are typically limited to at most 20 or so species and perhaps 50 loci.
10.6 Improving scalability of coalescent-based methods

Some of the most popular coalescent-based methods are computationally intensive; as an example, *BEAST is limited to small numbers of loci and species because it uses an MCMC analysis to co-estimate gene trees and species trees. Other coalescent-based methods, such as MP-EST, are also computationally intensive, because they use heuristics to seek optimal trees with respect to maximum likelihood or maximum pseudo-likelihood. Thus, several techniques have been developed to improve the scalability of computationally intensive coalescent-based methods. Here we present two of these techniques, each using a divide-and-conquer strategy.

Improving *BEAST’s scalability to larger numbers of loci: As noted in Bayzid and Warnow (2013), one of the major reasons that *BEAST is more accurate than summary methods is that it is able to produce estimated gene trees that are more accurate than maximum likelihood analyses of individual gene sequence alignments. The BBCA method (Zimmermann et al., 2014) is a simple divide-and-conquer technique that take advantage of this observation. BBCA was developed to improve the scalability of *BEAST to larger numbers of loci, and operates by randomly dividing the loci into smaller bins (e.g., of 25 genes per bin), running *BEAST on each bin, and then taking the gene trees estimated by *BEAST and combining them using a summary method. As shown in Zimmermann et al. (2014), using this approach with the MP-EST summary method reduced the running time needed for *BEAST to converge to the stationary distribution (since it converges more quickly on small numbers of genes than on larger), and did not seem to reduce accuracy compared to a full *BEAST analysis. However, BBCA only improves the scalability with respect to the number of loci, and so does not address the restriction of *BEAST to small numbers of taxa.

Improving MP-EST’s scalability to larger numbers of loci: MP-EST uses a heuristic search strategy to seek optimal pseudo-likelihood trees; hence, its running time increases quickly with the number of taxa, as observed in Mirarab and Warnow (2015). An iterative divide-and-conquer strategy based on DACTAL (Nelesen et al., 2012) (and hence referred to as “DACTAL-boosting”) was developed to improve the scalability of MP-EST to large numbers of taxa in Bayzid et al. (2014). Each iteration begins with the tree computed in the previous step, and then computes a new tree. In a given iteration, the taxon set is divided into smaller, overlapping subsets, using the tree from the previous step. The division into subsets is accomplished using the DACTAL (Nelesen et al., 2012) decomposition, which uses recursion and the topology of the current tree to produce subsets of the desired size, and where each taxon subset is a set of species occupying a local part of the species tree. Then, for each taxon subset, it restricts the gene trees to the taxa in that set, and computes a species tree using the specified summary method. Finally, it uses SuperFine+MRL (Swenson et al., 2012; Nguyen et al., 2012), a supertree method, to combine the smaller estimated species trees together. After iterating a few times, the quartet support is computed.
for the species trees computed in each iteration, and the species tree with the highest quartet support is returned. As shown in Bayzid et al. (2014), “DACTAL-boosting” improved the scalability of MP-EST by reducing the running time needed to analyze datasets. Interestingly, it also improved the topological accuracy of the resultant tree. DACTAL is an example of a disk-covering method (DCM), and is described further in Chapter 11.5.2.

10.7 Further reading

10.8 Review questions

10.9 Homework problems

1. Let \( T \) be a rooted species tree with topology \((a, (b, (c, d)))\), and assume that the probability of coalescence on each edge is \( \varepsilon > 0 \).
   - Compute the probability of the unrooted gene tree \( t_1 = (a, (b, (c, d))) \) under the multi-species coalescent (this will be a function of \( \varepsilon \)). For what values of \( \varepsilon \) is this greater than 0.5?
   - Compute the probability of the unrooted gene tree \( t_2 = (a, (c, (b, d))) \) under the multi-species coalescent (this will be a function of \( \varepsilon \)). For what values of \( \varepsilon \) is this greater than 0.5?
   - Compute the probability of the rooted gene tree \( t_3 = ((a, d), (c, b)) \) under the multi-species coalescent (this will be a function of \( \varepsilon \)). For what values of \( \varepsilon \) is this greater than 0.5?

2. Consider the model species tree from the previous problem. Prove or disprove: \( \exists \varepsilon > 0 \) such that the most probable unrooted gene tree is not the unrooted species tree.

3. Consider the rooted model species tree \(((a, b), (c, d))\), and assume every edge has coalescence probability \( \varepsilon > 0 \). Prove or disprove: \( \exists \varepsilon > 0 \) such that the most probable rooted gene tree is not the rooted species tree.

4. Apply the SRSTE algorithm to each of the following input set of rooted gene trees:
   - \( T = \{ (a, (b, (c, d))), (a, (c, (b, d))), (b, (a, (c, d))) \} \).

5. Apply the SRSTE algorithm to the following input set of rooted gene trees:
   - \( T = \{ (a, (b, (c, d))), (a, (c, (b, d))), (b, (a, (c, d))), (b, (c, (a, d))) \} \).

6. Consider the inputs to SRSTE given in the previous two problems. For each of those inputs, interpret the gene trees as unrooted gene trees, and apply the SUSTE algorithm. What do you obtain?

7. Let \( T \) be an arbitrary model species tree, and consider a set \( \mathcal{T} \) of rooted gene trees generated by \( T \) under the multi-species coalescent model. For the sake of this problem, assume that every gene tree is correctly computed. Suppose you were to compute the strict consensus tree for \( \mathcal{T} \) (i.e., treating each gene tree as an unrooted tree). What would you expect to obtain, in the limit, as the number of gene trees in \( \mathcal{T} \) increases?
8. Suppose you have a rooted species tree $T$, with branch lengths in coalescent units. Recall that every such species tree defines a distribution on rooted gene trees under the multi-species coalescent model. We define $p_{\text{sib}}(x, y) = Pr(x, y \text{ are siblings})$ to be the probability that taxa $x$ and $y$ are siblings in a rooted gene tree that is sampled at random from the distribution. Is it the case that the pair $x, y$ that maximizes $p_{\text{sib}}(x, y)$ are siblings?

9. Suppose you want to use ASTRAL to find a species tree for a set $S$ of gene trees. ASTRAL requires a set $X$ of allowed bipartitions, and so you set $X = C(T)$, where $T$ is a single MRP tree for the set of gene trees. What is optimal solution to the constrained quartet-median tree optimization problem?

10. Suppose you want to use ASTRAL to find a species tree for a set $S$ of gene trees. ASTRAL requires a set $X$ of allowed bipartitions, and so you set $X = C(T)$, where $T$ is the greedy consensus tree of the gene trees. What is optimal solution to the constrained quartet-median tree optimization problem?

11. Suppose you want to use ASTRAL to find a species tree for a set $S$ of gene trees, and you also have the sequence alignments for the genes. ASTRAL requires a set $X$ of allowed bipartitions, and so you set $X = C(T)$, where $T$ is the maximum likelihood tree produced using the concatenation of the alignments of these genes. What is optimal solution to the constrained quartet-median tree optimization problem?

12. Imagine you are a graduate student working in a bioinformatics laboratory, and your PI asks you to construct some trees for four different species, using different loci. For the sake of simplicity, we’ll call the species $H$, $C$, $G$, and $R$; you can think of them as being human ($H$), chimp ($C$), gorilla ($G$), and rhesus monkey ($R$), but they could be any four species. You are fortunate that all the genomes have been assembled and aligned, and you have several thousand loci you can compare. You select five loci at random from these genomes, and use the best method you can to construct a tree for each of the five loci. Suppose four of these trees have topology $((H, C), (G, R))$, but in the fifth you get $((H, G), (C, R))$. You report your results to the PI, and he says “You must have made a mistake - you shouldn’t get $((H, G), (C, R))$, because I’m sure $H$ and $C$ are siblings in the true tree.” How would you answer the PI?
Designing methods for large-scale phylogeny estimation

11.1 Introduction
In this chapter, we investigate techniques for constructing phylogenetic trees for large sequence datasets. We explore algorithm design, including standard heuristics used in many software packages, and also the divide-and-conquer techniques used to scale computationally intensive methods to large datasets.

11.2 Standard approaches
Many phylogeny estimation methods fall into the following categories: distance-based methods, subtree assembly-based methods, heuristics for NP-hard optimization methods, or Bayesian methods. Understanding each of these types of methods is helpful in developing methods for large datasets.

Distance-based methods: Generally the fastest type phylogeny estimation methods operate by computing a matrix of pairwise “distances” between every pair of taxa, and then use that distance matrix to compute a tree. These distance-based methods (described in Chapter 5) inherently require $\Omega(n^2)$ time, where $n$ is the number of species, just to compute the distance matrix. Most techniques to compute the tree from the distance matrix use $O(n^3)$ time, but some variants run in $O(n^2)$ time. Distance-based methods are therefore polynomial, and fast enough to be used on most datasets.

Subtree assembly-based methods: Some methods operate by computing trees on a collection of small subsets of the taxon set, and then combine the subset trees together into a tree. We call these “subtree assembly-based methods” since they operate by estimating subtrees and then assembling them into a larger tree. Many subtree assembly-based methods compute quartet trees, and so are called “quartet-based phylogeny estimation methods”. Most typically the quartet trees are computed using maximum likelihood, and then the quartet trees are combined using different quartet amalgamation methods such as Quartet Puzzling (Strimmer and von Haeseler, 1996), Weight optimization (Ranwez and Gascuel, 2001), Quartet Joining (Xin et al., 2007), Quartets MaxCut (Snir and Rao, 2010), and Quartet
If trees are computed for every possible quartet, then this kind of approach inherently requires $\Omega(n^4)$ time. However, quartet amalgamation methods tend to be computationally intensive, and may require more than $\Theta(n^4)$ time. Even with fast quartet amalgamation methods, quartet-based tree estimation is generally not that feasible for large $n$ (i.e., $n \geq 1000$). Restricting the set of quartet trees to a proper subset of the possible quartet trees can enable this type of method to become computationally feasible, but can reduce accuracy compared to analyses of the full set of quartet trees (Swenson et al., 2010). While some quartet-based methods have been shown to be as accurate as neighbor joining (Xin et al., 2007; Snir et al., 2008), to date, quartet-based estimation (and more generally subtree assembly-based methods) have not been shown to provide comparable accuracy to maximum likelihood heuristics or Bayesian methods. This disappointing performance of quartet-based methods have led some researchers to assume that quartet-based methods, where quartet trees are estimated independently, may have inherent limitations (Ranwez and Gascuel, 2001).

Heuristics for NP-hard optimization problems: Heuristics for maximum likelihood or other NP-hard optimization problems (such as maximum parsimony) tend to be based on local search heuristics that operate in the following manner:

- An initial tree is produced, and its score (under the desired criterion) is computed.
- Then the search for a better tree (i.e., a local optimum) begins by exploring trees that are within some neighborhood of the current best tree. The neighborhood is typically defined by a single change to the tree obtained usually by cutting the tree into two parts and then re-attaching the two parts. For example, NNI (nearest neighbor interchanges), SPR (subtree prune-and-regraft), and TBR (tree bisection and reconnection) are common modifications (Felsenstein, 2003). Other strategies, including p-ECR (Ganapathy et al., 2004, 2003), where $p$ edges are contracted and then the tree is refined optimally, have also been explored. The score of each of these trees is computed, and the best scoring tree is maintained.
- If no neighboring tree has a better score, then either the best scoring tree (or set of trees, if there is more than one best tree) is returned, or the search begins at a new starting tree. Typically this new starting tree is obtained using a strategy that employs randomness. For example, the parsimony ratchet (Nixon, 1999) operates by (1) producing a bootstrap alignment (i.e., a modification of the original multiple sequence alignment produced by sampling with replacement from the sites of the alignment), (2) running the heuristic search strategy anew on this bootstrap alignment until a local optimum is obtained, and then (3) continuing the search from the new tree, but now based on scores computed using the original alignment, until a local optimum is obtained. This alternation between searches based on the original alignment and the bootstrap replicate alignment can be repeated several times.
- This process repeats until a desired stopping criterion is met, which is sometimes just how long the search lasts (in time or number of iterations) or when no change in the best
found score has been found after some number of algorithmic steps. At the end, the set of all best trees is returned.

These heuristic searches are generally run from several starting trees to increase the chances that good trees will be found. These starting trees are computed using several strategies, including random trees, or using fast methods (such as neighbor joining) applied to the input set. A common strategy is to use a random sequence addition order, and then build a tree greedily by adding each sequence in turn, optimizing its placement into the tree on the previous sequences.

For maximum parsimony, the score of a tree is computable in polynomial time using dynamic programming (see Chapter 4), and some other criteria can also be solved exactly on fixed tree topologies. However, the score of a tree topology under maximum likelihood involves finding the best numeric parameters for the tree (such as the General Time Reversible (GTR) substitution matrix, the shape parameter for rates across sites, and the branch lengths), and this estimation problem is not known to be solvable in polynomial time. Instead, local search strategies are used for these estimations of numeric parameters on a fixed tree, and are not known to produce optimal parameter values.

Note that these heuristic searches can be described by (a) the starting trees, (b) the strategy used to explore search space (i.e., the definition of the neighborhood of a given tree), (c) how they compute the score of a tree, and (d) the stopping criterion. All of these choices affect the outcome. Whether attempting to solve maximum likelihood, maximum parsimony, or some other optimization problem, all these search strategies are heuristics that have no guarantees in terms of finding optimal trees; indeed, since the optimization criteria are NP-hard, unless P=NP, local search strategies are unlikely to provably find global optima for all inputs. Furthermore, these heuristic search strategies can take a long time (and use large amounts of memory) on large datasets, meaning days, weeks, or even more, of CPU time; as an example, a maximum likelihood phylogenomic analysis of an avian biological dataset with about 50 species and 14,000 concatenated markers used more than 200 CPU years and 1Tb of memory (Jarvis et al., 2014).

**Bayesian MCMC methods:** Bayesian analyses are based on the same models as in maximum likelihood searches, but are not attempts to find the single “best” tree. Instead, they seek to sample from the distribution on model trees according to the probability of the data under each model tree. This objective typically cannot be solved analytically, and so instead it is often based on MCMC (Markov Chain Monte Carlo) techniques, which in essence are random walks through model tree space. After the walk has progressed long enough (i.e., the “burn-in” period), the model trees that are visited can be used to build the estimate of the distribution. Typically, therefore, techniques are used to determine the probability that the MCMC walk has reached the stationary distribution, the burn-in portion of the walk is discarded, and then a random set of trees (e.g., just one out of every 100 trees) is sampled. This sample is then used to produce a probability distribution on the parameters of the model tree, for example, estimating the probability that the model tree
11.2 Standard approaches

has a particular bipartition by the fraction of the sampled trees that have that bipartition. Alternatively, the sample can be used to produce a point estimate of the model tree; for example, the greedy consensus of the sampled trees can be used.

From a computational viewpoint, MCMC methods are very expensive, because they can take a very long time in order to converge. This convergence time grows with the number \( n \) of taxa, as the number of possible tree topologies grows exponentially in \( n \). However, MCMC methods also have running times that grow with the number of characters. For example, MCMC methods that estimate gene trees from DNA sequence data under the GTR model of DNA sequence evolution will generally take longer for longer sequences than for shorter sequences. Similarly, MCMC methods that co-estimate species trees and gene trees from multi-locus sequence data under the multi-species coalescent (MSC) model of gene tree evolution will take longer to converge when the number of loci is large rather than small (Zimmermann et al., 2014). Thus, both dimensions of the dataset - the number of taxa (whether species or individuals) and the number of characters (whether sites or loci) - impact the time needed to reach convergence for Bayesian MCMC methods.

Comparisons between methods: All these methods are in current usage. Some biologists use distance-based methods, especially on large datasets, because of their speed, others prefer maximum parsimony heuristics, or maximum likelihood, or Bayesian methods. Indeed, the different methods each have their own proponents.

Simulation studies have generally supported the use of likelihood-based methods (e.g., maximum likelihood or Bayesian MCMC method) over maximum parsimony, but this is to be expected since the models used in the simulations generally match the models used by the estimation methods. More generally, statistically consistent methods (such as distance-based methods, maximum likelihood, and Bayesian MCMC) are generally expected to have better accuracy when the data evolve under the models they assume than inconsistent methods, and this expected behavior tends to be observed in many simulation studies.

Even so, some interesting deviations from this expected dominance of statistically consistent methods over parsimony appear, especially for datasets with large numbers of taxa but limited numbers of characters (i.e., where the improvement implied by statistical consistency would not be applicable). For example, some simulations have shown that distance-based methods, such as neighbor joining, can be less accurate than maximum parsimony under model conditions with high rates of evolution and large numbers of taxa, unless the number of sites is very large (Moret et al., 2002; Nakhleh et al., 2002).

Although there are only a few studies examining the relative accuracy of phylogeny estimation methods on large datasets, maximum likelihood and Bayesian MCMC methods are generally considered to be at least as accurate as the alternative approaches, and - because of their statistical consistency guarantees - more dependable. However, Bayesian MCMC methods are not used on very large datasets nearly as often as maximum likelihood heuristics, largely because of running time.

Many software packages exist with effective heuristics for maximum likelihood under standard sequence evolution models (such as GTR). Examples of these methods include
Designing methods for large-scale phylogeny estimation

RAxML (Stamatakis, 2006), FastTree-2 (Price et al., 2010), PhyML (Guindon and Gascuel, 2003), and IQTree (Nguyen et al., 2015a). Comparisons between methods on small to moderate-sized datasets sometimes favor one of IQTree, RAxML, or PhyML, but only FastTree-2 has been shown to be efficient enough to run on datasets with very large numbers of sequences (Liu et al., 2012). However, FastTree-2 is not designed for very long sequences, and some other software packages (e.g., ExaML, a variant of RAxML) are more efficient under those conditions. Software packages also differ in terms of the sequence evolution models they allow. For example, there are many different protein sequence evolution models, and RAxML allows more protein models than FastTree-2 does. For DNA sequence evolution models, IQTree enables a wide variety of models, including the PoMo (De Maio et al., 2015) model that allows for polymorphism, and nhPhyml (Bousau and Guoy, 2006) enables non-homogeneous models. Similar differences exist between Bayesian MCMC methods. Thus, the choice between maximum likelihood or Bayesian MCMC software packages depends on a variety of factors, including their ability to efficiently analyze large datasets, the models they enable, and their relative accuracy as reported in the literature.

11.3 Introduction to Disk-Covering Methods (DCMs)

11.3.1 Objectives of a DCM: boosting a base method

As we have seen, standard phylogeny estimation methods typically have good performance (in terms of accuracy and speed) for small enough datasets that evolve under slow enough rates of evolution. However, depending on the method, datasets that are large and/or have high evolutionary diameters can become problematic for the phylogeny estimation method. This pattern suggests that divide-and-conquer strategies could be helpful in improving the accuracy and/or scalability of standard phylogeny estimation methods. We describe algorithmic strategies that can be used in the development of a divide-and-conquer strategy to improve phylogeny estimation method. These methods, which we refer to as “disk-covering methods”, or DCMs, have been used in a number of contexts with a variety of types of phylogeny estimation methods, which we will refer to as “base methods”.

11.3.2 The three phases of a DCM

Each DCM is fundamentally based upon a divide-and-conquer strategy, which has the following three phase structure:

- **Phase I**: Compute a decomposition of the dataset into overlapping subsets, and construct trees on the subsets using the base method.
- **Phase II**: Use a supertree method to merge the trees on the subsets into a tree on the full dataset.
- **Phase III**: If the tree obtained in Phase II is not fully resolved (i.e. if the tree is not a binary tree), we refine it into a binary tree.
As we will show, for some algorithmic designs, this strategy can iterate, using the tree obtained in the previous iteration to improve the decomposition. Also, the design of a DCM depends in part on one’s objectives, and whether they are theoretical (e.g., establishing a bound on the sequence length required for accuracy with high probability) or empirical (e.g., finding a good solution to an NP-hard optimization problem, such as maximum parsimony or maximum likelihood, or improving the topological accuracy).

As we will show, the decomposition part of the first phase of the DCM is obtained by computing a triangulated graph (i.e., a graph that has no induced simple cycles of size four or more) and then using the triangulated graph to divide the dataset into overlapping subsets. Furthermore, the second phase of the DCM, where the smaller trees are combined into a tree on the full dataset, also uses the triangulated graph to ensure that the supertree that is obtained has good theoretical properties. Thus, the design of a DCM and the analysis of its algorithmic properties depends on understanding triangulated graphs. Therefore, we now discuss triangulated graphs, and then show how we can use this theory to develop DCMs with strong algorithmic properties.

### 11.4 Triangulated graphs

Here we provide the definitions and theorems about triangulated graphs that are relevant to phylogeny estimation and DCM design in particular, but without proofs. Triangulated graphs, which are also called rigid circuit graphs, were introduced in Dirac (1974), and their theoretical properties were established in a sequence of papers, including Berge (1967); Rose (1970); Buneman (1974a). For an overview of triangulated graphs, see Golumbic (2004), which provides proofs and a much richer set of results about triangulated graphs.

#### 11.4.1 Definitions and basic properties of triangulated graphs

**Definition 11.1** A graph which has no induced simple cycles of length greater than three is a **triangulated graph**.

Here we present an equivalent definition, which will be useful for establishing theorems about the dataset decompositions we employ. Let $T$ be a tree with vertex set $V$. Thus, $T$ is just a connected acyclic graph, and there is no constraint on the degrees of the vertices in $V$; also $T$ is not rooted. A **subtree** of $T$ is just a connected subgraph of $T$, and so is defined by a subset of the vertices of $V$. Two subtrees are said to intersect if they share any vertices. Thus, given an arbitrary set $X$ of subtrees of an arbitrary tree $T$, the intersection graph of $X$ has node set $X$ and edges $(t_1, t_2)$ where $t_1$ and $t_2$ share at least one node in common. It is not hard to see that any intersection graph of a set of subtrees of a tree $T$ is triangulated, but it is also true that every triangulated graph can be written as such an intersection graph!

We summarize this with the following theorem:

**Theorem 11.2** (From Buneman (1974a)) A graph $G$ is triangulated if and only if $G$ is the intersection graph of a set of subtrees of a tree.
Definition 11.3 A perfect elimination scheme for a graph $G = (V, E)$ is an ordering of the vertices $v_1, v_2, \ldots, v_n$, so that for each $i = 1, 2, \ldots, n - 1$, $X_i = \Gamma(v_i) \cap \{v_{i+1}, v_{i+2}, \ldots, v_n\}$ is a clique (here $\Gamma(v_i)$ indicates the neighbor set of $v_i$).

A basic theorem about triangulated graphs is that every triangulated graph has a perfect elimination scheme Rose (1970), and given a triangulated graph, one such perfect elimination scheme can be found in polynomial time.

11.4.2 Decompositions of triangulated graphs

Theorem 11.4 Every triangulated graph $G = (V, E)$ has at most $n = |V|$ maximal cliques, and these can be found in $O(n^2)$ time.

Theorem 11.5 For every triangulated graph $G = (V, E)$, $\exists X \subseteq V$ such that $X$ is a clique and $G \setminus X$ is the disjoint union of components $C_1, C_2, \ldots, C_k$. Furthermore, we can find such an $X$ that minimizes $\max_i |C_i \cup X|$ in $O(n^3)$ time, where $n = |V|$. This set $X$ is called a clique-separator.

These theorems will be useful for showing how a connected triangulated graph can be decomposed into overlapping subsets. For example, if a connected triangulated graph $G$ is decomposed into its set of maximal cliques, these cliques will be overlapping. On the other hand, if we compute a clique separator $X$ for $G$, then the sets $C \cup X$, where each $C$ is a component of $G \setminus X$, are overlapping subsets. In other words, a connected triangulated graph can be decomposed into overlapping subsets using each of the following two decompositions:

- **Max-clique decomposition**: Given a triangulated graph $G$, return the set of maximal cliques of $G$.
- **Separator-Component decomposition**: Given $G$, find a clique separator $X$ and compute all the components of $G \setminus X$. Then return the sets of the form $X \cup C$, where $C$ is one of the components of $G \setminus X$.

To use the decompositions, we have to show how to compute a triangulated graph from a set of taxa; here we describe two such ways: threshold graphs and short subtree graphs.

11.4.3 Threshold graphs

Let $S = \{s_1, s_2, \ldots, s_n\}$ be a set of taxa, and let $[d_{ij}]$ be a distance matrix on $S$. Let $q \geq 0$ be any non-negative real number, which we will refer to as the “threshold”. Then the threshold graph for $d$ and $q$ denoted by $TG(d, q)$, is defined as follows:

Definition 11.6 The threshold graph $TG(d, q)$ has vertex set $S$ and edges $(s_i, s_j)$ such that $d_{ij} \leq q$. 

*Designing methods for large-scale phylogeny estimation*
Note that \( TG(d, q) \) depends on both the distance matrix \( d \) and the threshold \( q \), and that \( TG(d, q_1) \) is a subgraph of \( TG(d, q_2) \) if \( q_1 \leq q_2 \). It is easy to see that if \( q \geq \max d_{ij} \) then \( TG(d, q) \) is a clique, while if \( q < d_{ij} \) for all \( i, j \) with \( i \neq j \), then \( TG(d, q) \) will be the empty graph (i.e., a graph without any edges). Intermediate values of \( q \) will contain edges, but may not be connected until \( q \) is large enough. Hence, the choice of \( q \) will impact the decomposition, and only a subset of the values (ones that are large enough to create a connected graph, and not so large that they create a complete graph) will produce decompositions that could be useful.

The connection between threshold graphs and phylogeny estimation follows from the following theorem, which shows that threshold graphs defined for additive distance matrices are always triangulated.

**Theorem 11.7** (From Roshan et al. (2004)) Let \( d \) be an additive matrix, and let \( q \) be a real number. Then \( TG(d, q) \) is triangulated.

Since estimated distances computed on taxa converge, as the amount of data increases, to an additive matrix, this theorem implies that given enough data, the threshold graph computed on the matrix of estimated distances will be triangulated. For example,

**Corollary** Let \((T, \theta)\) be a Jukes-Cantor model tree with \( n \) leaves, and let \( S \) be a set of \( n \) sequences that evolve down \((T, \theta)\). Let \( d \) be the matrix of Jukes-Cantor distances computed on \( S \), and let \( q \) be any real number. Then, as the sequence length for \( S \) increases, the threshold graph \( TG(d, q) \) converges in probability to a triangulated graph.

Note that this corollary does not depend on the model tree being Jukes-Cantor, but rather that the dissimilarity matrix of estimated distances be properly corrected for the model of evolution, as described in Chapter 5. Furthermore, the corollary also works if the input is a set of gene trees that differ from each other due to incomplete lineage sorting and the distances between species are computed in a way that converges to an additive matrix defining the species tree. For example, Allman et al. (2016) proved that the average internode distances (the average number of edges in the paths between two taxa), as used in the coalescent-based species tree method, in NJst (Liu and Yu, 2011) and ASTRID (Vachaspati and Warnow, 2015), converges to an additive matrix for the species tree.

**Computing a triangulated threshold graph:** Even though the threshold graph converges to a triangulated graph, in practice, the amount of data (whether sequence length or number of gene trees) may not be sufficient for \( TG(d, q) \) to be triangulated. Therefore, to use these decompositions, we may need to modify the threshold graph so that it becomes triangulated. One way to do this (and which has been done in practice) is by the addition of edges. However, the choice of edges can impact the accuracy and theoretical guarantees of the resultant method, and adding edges with lower weights (corresponding to edges between taxa with smaller distances according to the estimated distance matrix) improves the theoretical properties (Huson et al., 1999a; Warnow et al., 2001). Optimizing the choice of edges to minimize the weight of the heaviest added weight is NP-hard, but Lagergren
(2002) developed a polynomial time algorithm that produces a triangulation with good theoretical properties. In practice, a greedy technique to triangulate the graph can produce good results.

Once the triangulated threshold graph is obtained, it can be decomposed into overlapping subsets using either the max-clique decomposition or the separator-component decomposition. The max-clique decomposition produces smaller taxon subsets, and was used in DCM-Buneman (Huson et al., 1999a) and in DCM1-NJ (Warnow et al., 2001).

### 11.4.4 Short subtree graphs

We now describe another kind of triangulated graph, which we call the short subtree graph, and show how to compute it from a set $S$ of taxa. Given a set $S$ of taxa, let $T$ be any tree on $S$ with edges given with positive weights; as an example, $T$ could be a maximum likelihood tree or neighbor-joining tree, and the edge weights could be the lengths of the edges in terms of the expected number of substitutions. Given these edge weights, we can define the distance of a leaf $x$ to an edge $e$ to be the sum of the weights of the edges in the path from $x$ to the nearest endpoint of $e$. Note that the computation of the tree $T$ could be achieved in polynomial time (e.g., if we compute the neighbor joining tree) or it might be computationally intensive (e.g., if we compute a maximum likelihood tree).

**Definition 11.8** Let $e$ be an edge of an edge-weighted binary tree $T$. Let $t_1, t_2, t_3,$ and $t_4$ be the four subtrees around the edge $e$ (i.e., $t_1$ through $t_4$ are the components of $G \setminus \{x, y\}$, where $e = (x, y)$). Let $x_i$ denote those leaves in $t_i$ that are closest to the edge $e$ (using the path lengths defined by the edge-weighting on $T$). Then the **short subtree around** $e$ is $x_1 \cup x_2 \cup x_3 \cup x_4$.

**Definition 11.9** Let $T$ be a tree with leaf set $S$ and edge weighting $w : E(T) \to \mathbb{R}^+$. Let $G$ be the graph with vertex set $S$ and edge set $E$ defined by $(s_i, s_j) \in E$ if and only if $e \in E(T)$ such that $s_i$ and $s_j$ are both in the short subtree around $e$. This is the **short subtree graph** of $(T, w)$, denoted by $SSG(T, w)$.

In Roshan et al. (2004) we showed that every short subtree graph is triangulated:

**Theorem 11.10** (From Roshan et al. (2004)) Let $T$ be any tree with positive edge-weighting $w$. Then the short subtree graph $SSG(T, w)$ is triangulated.

A variant of the short subtree graph that has been explored is the **padded short subtree graph**, where instead of using just the single closest leaf in each subtree around an edge, the $p$ closest leaves in each subtree are selected, where $p$ is a small integer. This padding has the impact of ensuring that the subsets have higher overlap, but also increases the size of the subsets. The vertices of the padded short subtree graph with padding parameter $p$ defined by a tree $T$ with edge weighting $w$ are the leaves of $T$, and the edges of the padded short subtree graph are those $(u, v)$ such that there is an edge $e$ in $T$ where both $u$ and $v$ are among the $p$ nearest leaves to $e$ in some subtree off $e$. Note that in this definition, $u$ and $v$
11.5 Designing DCMs

11.5.1 Introduction

The basic DCM design has three phases. Phase 1 performs the decomposition of the taxon set into overlapping subsets and then computes trees on each subset. Phase 2 merges the subset trees together into a tree on the full taxon set using a preferred supertree method, and Phase 3 refines the tree obtained in Phase 2 into a binary tree, if it is not fully resolved.

In the DCMs we have developed, we have used SuperFine with a variety of supertree methods (most notably with MRL) to perform Phase 2, and (generally) random refinement to refine any non-binary tree produced in Phase 2. However, the choice of how to perform Phase 1 is more complicated. In particular, the choice of whether to use a threshold graph or a short subtree graph, and whether to use a max-clique decomposition or a separator-component decomposition, depends very much on the properties of the base method (i.e., its scalability and accuracy issues) and whether the objectives are theory or practice.

For example, the time needed to compute the decomposition may matter, but the number of subsets and their sizes also impact the overall running time, and may dominate the running time when the base method is computationally intensive. The design choices also impact accuracy, since some phylogeny estimation methods degrade in accuracy on large datasets, or on datasets with high evolutionary diameters. Hence, design choices that produce small datasets that have low evolutionary diameters tend to produce the most accurate subset trees, for most base methods, and also tend to be fast. However, the overlap patterns between the subsets that are produced also impacts accuracy (and running time, for that matter). In particular, if there is insufficient overlap between subset trees that are being merged together in the Strict Consensus Merger (SCM) phase of SuperFine, then the SCM tree may be highly unresolved, and the resultant SuperFine tree may be reduced in accuracy compared to what it might have been if the subset trees had greater overlap. So far this discussion has been largely about empirical performance. However, theoretical guarantees also depend on the design choices, as we will see.

Thus, the choices made in the design of a DCM depend on the properties of the base phylogenetic estimation method (that will be used to produce trees on each of the subsets), and will impact the empirical and the theoretical properties of the resultant DCM-boosted method.

Furthermore, as we will see, some DCMs have used elaborate designs to obtain addi-
Designing methods for large-scale phylogeny estimation

11.5.2 DACTAL: a DCM using iteration and recursion

DACTAL (Nelesen et al., 2012) was originally developed to enable the estimation of a tree from unaligned without needing to align the full dataset. In this context, the input is a set of unaligned sequences. In the first step, the sequence set is decomposed into small overlapping subsets where each subset contains sequences that are highly similar to each other. Then, multiple sequence alignments and trees are computed on each subset using the preferred alignment and tree estimation methods, thus producing a collection of overlapping subset trees. Finally, the subset trees are combined together using a preferred supertree method. Note that this protocol produces a tree on the full set of sequences, but does not produce a multiple sequence alignment. Once a tree is obtained on the full set of sequences, it can be used to produce a new decomposition of the sequence set into overlapping subsets, and the process can iterate. Hence, DACTAL, or “Divide-and-conquer trees (almost) without alignments”, is a technique that combines iteration with divide-and-conquer, and it enables trees to be computed without requiring the estimation of a multiple sequence alignment of the full dataset.

The original implementation of DACTAL (Nelesen, 2009) performed the initial decomposition of the sequence dataset into overlapping subsets with at most 200 sequences using BLAST (Altschul et al., 1990), then computed maximum likelihood trees on each subset using RAxML applied to MAFFT alignments, and merged the subset trees together using SuperFine+MRP. Subsequent iterations of DACTAL obtained the decomposition of the taxon set using a “padded-recursive DCM3” decomposition, which in essence is an enhanced version of a short subtree graph where instead of using just the nearest leaf in each subtree, it used up to \( p \) nearest leaves (where \( p \) is a parameter set by the user).

Note that if the dataset is not too large, then the decomposition can be initiated using a quick-and-dirty two-phase tree estimation method, such as computing an alignment using the PartTree command within MAFFT (Katoh and Toh, 2007), followed by a fast maximum likelihood method such as FastTree-2 (Price et al., 2010).

Although DACTAL was initially described for use in constructing phylogenies where multiple sequence alignment is challenging due to dataset size, it has subsequently been used for other tree estimation problems (Bayzid et al., 2014). Figure 11.5.2 shows how the same framework can be used as a generic technique to enable computationally intensive methods to be applied to large datasets.

The input to this more general use of DACTAL is a set of taxa along with some kind of associated data, which could be molecular sequences, other types of character data, whole genomes, or even gene trees. In the first step, the taxon dataset is decomposed into overlapping small subsets of a desired size. Then trees are computed on each taxon subset, and these subset trees are combined together using a supertree method. If desired, the cycle can then begin again, using the current tree. Each subsequent iteration begins with the...
Scaling methods to large datasets using DACTAL-like design. The input is an arbitrary set of taxa with associated data (e.g., sequences). In the first step, the dataset is decomposed into overlapping small subsets of a desired size. Two ways of decomposing a sequence dataset into subsets have been developed: one just uses the sequences, but the other begins by computing a tree (through a quick-and-dirty two-phase method). Then trees are computed on each subset, and the subset trees are combined together using a supertree method. If desired, the cycle can then begin again, using the current tree. Each subsequent iteration begins with the current tree, divides the dataset into subsets using the tree, computes trees on subsets, and combines the subset trees using the supertree method. Note that the objective is for the subset trees to be estimates of the underlying species tree, so that the subset trees - if correctly estimated - should be compatible with each other.

11.5.3 More elaborate DCM designs

The basic DCM algorithm design is essentially a divide-and-conquer strategy that does its steps using algorithms designed for triangulated graphs. DACTAL elaborates on this basic design in several ways. First, it uses a padded short subtree graph, and decomposes the graph using a separator-component decomposition recursively until each subset is small enough for the application. But DACTAL also uses iteration, so that each iteration begins...
with the tree computed in the previous iteration. Finally, depending on the application, the tree returned by DACTAL may either be the last tree (as in Nelesen et al. (2012), where it was used to compute a tree without computing an alignment) or the tree with the best quartet support score (as in Bayzid et al. (2014), where it was used to improve MP-EST, a coalescent-based species tree estimation method). Other DCMs that have used similar combinations of recursion, iteration, and decompositions based on triangulated graphs, include DCM2-boosting (Huson et al., 1999b) and Rec-I-DCM3 (Roshan et al., 2004) (the use of both recursion and iteration with the DCM3 strategy), each of which was designed to improve heuristic searches for maximum parsimony.

Some other DCMs have an algorithmic structure where they compute a set of trees, each based on a particular way of using the basic three-phase DCM design, and then either select a single tree from the set or compute a consensus of these trees. DCM-Buneman Huson et al. (1999a), one of the first disk-covering methods, and DCM1-NJ Warnow et al. (2001) are two examples of disk-covering methods that use this strategy in order to produce methods with strong theoretical guarantees in terms of the sequence lengths that suffice for accuracy with high probability.

**DCM-Buneman.** The first DCM was designed for use with the Buneman Tree distance-based method (Huson et al., 1999a). Given the input distance matrix \( d \), it computed the threshold graph \( TG(d, q) \) for every entry \( q \) in the distance matrix. For those threshold graphs that are connected, it then computes a supertree, using the following technique. First, it uses the max-clique decomposition, and computes subset trees using the Buneman Tree method on every maximal clique. Then, it combines the subset trees using the strict consensus merger tree. This process produces a set \( A \) of \( O(n^2) \) trees, each associated with a particular threshold value. In Huson et al. (1999a), two ways of producing a tree from the set \( A \) was suggested. The first of these, which was specifically designed for the Buneman Tree, returned the tree with the largest number of internal edges (i.e., that is the most resolved), noting that if there is more than one such tree, then the tree associated with the largest threshold is returned. The second way of processing the set \( A \), which was recommended for other base phylogeny estimation methods (such as neighbor joining), was to return an asymmetric median tree (Phillips and Warnow, 1996) (see Chapter 6.2.5). As shown in Huson et al. (1999a), both ways of computing a final tree from the set \( T \) of trees had lower tree error than the unboosted method, for the Buneman Tree method, the neighbor joining method, and also for the 3-approximation algorithm for the \( L_\infty \)-nearest tree described in Agarwala et al. (1996).

**DCM1-NJ**

### 11.5.4 Observations about DCM design strategies

All of our DCMs use the same three phase structure (although some also use recursion and/or iteration), with the main difference between the DCMs being the decomposition
All current DCMs first construct a triangulated graph and then apply either the max-clique or a separator-component decomposition to the graph to obtain subproblems. The combination of base method, choice of triangulated graph, and decomposition technique on that triangulated graph, impact the behavior of the resultant “DCM-boosted” method. For example, methods which will take a long time on big datasets will finish faster on the max-clique decomposition. A more subtle point is the impact of error on subsets – since the technique we use in merging subtrees contracts edges whenever two subtrees disagree, there is a potential for a greater loss of resolution in the max-clique decomposition than in a separator-component decomposition, especially when the separator is small. The difference between using threshold graphs and short subtree graphs is also interesting, but depends as much on the dataset as on the method. Thus, the design of a DCM reflects the particular properties of the base method and of the particular dataset, and only by studying the actual performance of the resultant DCM-boosted methods can we tell which design strategy will be the most beneficial.

We begin this section with a description of how we obtain triangulated graphs from molecular datasets.

Threshold graph decompositions: Threshold graph decompositions can be used on any dataset for which a distance can be defined between each pair of taxa. In molecular sequence datasets, these distances can be Hamming distances, or distances obtained under some statistical estimation procedure selected to match the model of evolution underlying the dataset.

In a threshold graph decomposition, we are given a set $S$ of sequences and a matrix $[d_{ij}]$ of distances on the set $S$ of taxa. To compute a threshold graph we must first select the threshold (the value $q$). We then construct the threshold graph, $TG(d,q)$ (see Definition 11.6). If $TG(d,q)$ is triangulated, we can compute either the max-clique decomposition, or a separator-component decomposition; however, if $TG(d,q)$ is not triangulated, then we must first triangulate it, by adding edges to $TG(d,q)$ so that the graph is triangulated. However, when we add edges to $TG(d,q)$ we affect the decompositions (either max-clique or separator-component) that we can obtain from the triangulated graph.

In some of our DCMs our objective is to minimize the maximum evolutionary distance within any subproblem, so that in a max-clique decomposition we would wish the cliques to have the smallest maximum distance. This suggests the following objective in the triangulation process: add edges to $TG(d,q)$ so as to minimize the “weight” of the heaviest edge added. This optimization problem is NP-hard, however, and so in our experiments, we have used a greedy triangulation scheme that works reasonably well: compute for each vertex $v$ in the graph the value $W(v) = \max\{d_{ij} : \{i, j\} \subseteq \Gamma(v)\}$, where $\Gamma(v)$ denotes the neighbors of the vertex $v$. Select the vertex $v$ that minimizes $W(v)$ and make it simplicial (i.e., make the neighbors of $v$ into a clique). Recurse on $G - \{v\}$. This approach produces a triangulation of $G$ but may not minimally triangulate the graph $G$; however, in our experiments this worked quite well. (See also Lagergren (2002) for a polynomial time technique that creates a triangulation with good theoretical properties.)
Thus, threshold graph decompositions have the following structure. They begin with a distance matrix \( d \), and operate as follows:

1. Pick a threshold \( q \in \{d_{ij}\} \)
2. Construct \( TG(d, q) \)
3. Add edges to \( TG(d, q) \) to triangulate it, producing graph \( G \)
4. Compute either the max-clique or a separator-component decomposition from \( G \)

**Guide tree decompositions:** We now describe how we can obtain guide trees, and from them the triangulated short subtree graph.

The most typical technique for obtaining a guide tree is to use some phylogeny reconstruction method (such as a heuristic for maximum parsimony or maximum likelihood, or perhaps a fast method such as neighbor joining) to obtain an estimation \( T \) of the true tree. Given \( T \), we can then use one of many techniques to assign edge lengths. For example, if the set of taxa are biomolecular sequences in a multiple alignment, then we can assign edge lengths to \( T \) by using the Fitch-Hartigan dynamic programming fixed-tree maximum parsimony algorithm of Fitch (1971); Hartigan (1973) presented in Chapter 4.3 to assign sequences to internal nodes, and then use Hamming distances to define edge lengths. We can also use maximum likelihood estimation of edge lengths, which may be more accurate but will take more time than maximum parsimony. In general, however, if \( T \) was obtained using a phylogeny reconstruction method, it will typically already have edge lengths (such is the case with the three techniques we mentioned earlier - heuristic MP, heuristic ML, or neighbor joining).

Given the guide tree \( T \) with its edge lengths, we then compute the short subtree graph. This is easily done in polynomial time. Once the short subtree graph is obtained, we can compute either the max-clique or a separator-component decompositions on it (since it is already triangulated). As noted before, finding an optimal separator-decomposition - although polynomial time - can be more expensive than desired; consequently faster decompositions based upon clique-separators can also be used. In Roshan (2004) we showed how we could compute a decomposition we call the “heuristic centroid-edge” decomposition in linear time (this fast running time was accomplished without explicitly constructing the short subtree graph). This decomposition worked very well in practice, as we showed in Roshan (2004); Roshan et al. (2004).

**Considerations in design strategies:** We have described ways we can obtain triangulated graphs, and ways we can decompose a triangulated graph. How do these choices interact with base methods?

Some methods - in particular, distance-based methods like neighbor joining Saitou and Nei (1987), have poor topological accuracy on datasets with large evolutionary diameters, although they are quite fast. These methods would therefore seem to benefit from decompositions that produce the smallest diameter subproblems. Other methods, in particular exhaustive searches for optimal trees under hard optimization criteria, can only realistically
handle quite small datasets – maximum parsimony is limited to perhaps 20 or 25 taxa, and maximum likelihood limited to much smaller datasets; these methods would require subproblems to be as small as possible. In a third class are local-search heuristics (like the hill-climbing heuristics used in maximum parsimony searches), which seem not to be impacted by large diameters so much, but are still impacted by dataset size. Understanding the best design strategy for these local-search heuristics is more complicated.

The particular technique used to obtain a triangulated graph also has an impact on the resultant DCM. If we use a guide tree, there is only one triangulated graph that we can obtain, but different guide trees will produce potentially different decompositions. This makes guide tree decompositions useful for heuristic searches for optimal trees under criteria such as maximum parsimony, because as the search finds better solutions, it can become a new guide tree, and a new decomposition can be obtained.

The issues involved in selecting threshold graph decompositions are more complicated. Although the distance matrix $d$ is fixed, the threshold can change. If the threshold is too small, the threshold graph will not even be connected, and so the tree on the full dataset cannot be reconstructed from subtrees, even if they are correctly computed. If the threshold is too big, the subproblems become essentially as difficult (almost as large and with almost the same evolutionary diameter) as the full dataset, although correct subtrees on the subproblems would then be likely to define the full tree. Thus, finding the “correct” threshold to use is a difficult problem.

The choice of how to obtain the decomposition impacts the subset size in interesting ways, however. For example, threshold graphs obtained for biomolecular sequence datasets often have very large cliques, so that the largest subsets can have nearly all the taxa (Roshan, 2004; Roshan et al., 2004). Decompositions like these are not very helpful, since the base method has to be applied to very large subsets. On the other hand, if an estimated tree can be obtained for the sequence dataset (perhaps using a quick distance-based method, or a fast heuristic search for maximum parsimony), then the short subtree graph decomposition based on the estimated tree can produce much smaller subsets (Roshan, 2004; Roshan et al., 2004). Decompositions producing small subsets are generally much more helpful in terms of empirical performance if the base method is computationally intensive, since it need only be applied to small subsets. In particular, decompositions based upon the short subtree graph are more suited for boosting heuristic searches for maximum parsimony or maximum likelihood.

Other experiments (Moret et al., 2002; Nakhleh et al., 2002) showed that neighbor joining became less accurate as the evolutionary diameter increased in the dataset (i.e., as the maximum number of expected changes across any leaf-to-leaf path increased). This observation is consistent with the theory regarding the convergence rates (i.e., the sequence lengths that suffice for accuracy with high probability) of neighbor joining (Atteson, 1999). In order to develop methods with good theoretical bounds on the convergence rate, we needed to provably bound the evolutionary diameter of each subset, and this was easiest to do using threshold graphs.

But because we did not know a priori the best evolutionary diameter to use, we examined
all possible diameters (i.e., one for every entry in the distance matrix). Then, we computed all possible threshold graphs (one for each threshold that creates a connected graph), and for each threshold graph we computed a tree; hence, if we have $n$ taxa, then we will have computed $\Theta(n^2)$ trees. Finally, we select one of the trees in the set of trees based on an optimization criterion that has good theoretical properties. The result of this approach is a method that is provably “absolute fast converging” (see Chapter 13), which means it has excellent theoretical guarantees in terms of the sequence length that suffice for accuracy with high probability. A simplification of this method to improve the running time was also explored, and shown to have excellent empirical performance. We will return to this particular use of DCMs in Chapter 13, when we discuss absolute fast converging (afc) methods.

DCMs have also been designed for reconstructing species trees from multi-locus datasets. The first use of DCMs for species tree estimation from multi-locus datasets occurred in Moret et al. (2005), where a DCM based on a threshold decomposition was used with the GRAPPA software suite for computing species trees using gene order data. More recently, a DCM based on the short subtree graph has been used with MP-EST (Liu et al., 2010), a method for estimating species trees from gene trees under the multi-species coalescent model (MSC), and shown to improve both the accuracy and the scalability of MP-EST (Bayzid et al., 2014).

11.6 Further reading

The design of statistical estimation methods, especially Bayesian MCMC, is discussed in depth in Yang (2009).

Other divide-and-conquer approaches to large-scale tree estimation have also been proposed, including the Mega-phylogeny approach from Smith et al. (2009b).

11.7 Review questions

11.8 Homework problems
Designing methods for large-scale multiple sequence alignment estimation
In this chapter we present some advanced topics in phylogenetics, including “fast-converging” methods that have been proven to return the true tree with high probability from sequences lengths that are only polynomial in the number of leaves for the model tree (under some assumptions), phylogenetic networks, genome rearrangements, genome-scale alignment, and computing forests rather than trees.

13.1 “Fast-converging methods”

13.1.1 Sequence length requirements

13.1.2 Short Quartets Methods

Recall that the Naive Quartets Method (NQM) is guaranteed to reconstruct the tree $T$ given an $n \times n$ dissimilarity matrix $d$ that is close enough to an additive matrix $D$ associated with $(T, w)$, where $w$ is a mapping of the branches of $T$ to positive real numbers. The reason that this algorithm works is that the set of quartet trees $Q(T)$ associated to $T$ uniquely defines $T$. Hence, when all the quartet trees of $T$ can be computed correctly, then the All Quartets Method can reconstruct $T$. Therefore, when $L_\infty(d, D) < f/2$, here $f$ is the minimum length of any branch of $T$, then $NQM(d) = T$. However, when $L_\infty(d, D) \geq f/2$, this algorithm can fail.

Thus, while the Naive Quartets Method is statistically consistent, it is only correct when every quartet tree can be computed correctly. The analysis shown above shows that quartet trees can be computed correctly for those quartets of taxa for which all the estimates of pairwise distances are sufficiently close to their true (model) distances, but that the calculated quartet trees can be incorrect when the error in the estimated pairwise distances are too large.

This analysis suggests the idea of selecting only a subset of the quartet trees to compute, and then using the Dyadic Closure to infer the remaining quartet trees. If this initial set of quartet trees is correctly computed and contains all the short quartets of the tree, then this approach will result in $Q(T)$, the set of all quartet trees of $T$, and we can then compute $T$ using the All Quartets Method. This is a natural approach, but the challenge is to figure out which quartet trees to start with. As it turns out, rather than making this decision up front,
we can explore several different selections of sets of quartet trees. We will prove that this approach either returns a single tree $T$, or it returns $\text{Fail}$. Furthermore, we will prove that this approach will return the true tree $T$ from polynomial length sequences, and is therefore an absolute fast converging (afc) method.

Recall Definition 3.5 given in Chapter 3.4.3, where we defined the concept of a “short quartet” in an edge-weighted tree. In brief, a short quartet is any quartet of leaves that can be formed by selecting an edge in the tree, and then taking one nearest leaf in each of the four subtrees around the edge. Any given edge can have multiple short quartets around it, and the set of all short quartets in a tree, denoted by $Q_{\text{short}}(T)$, is the union of all those sets of short quartets around the different edges. Finally, the set of quartet trees on these short quartets is denoted by $Q^*_{\text{short}}(T)$.

Recall Theorem 3.4.4, which showed for any pair of binary unrooted trees $T$ and $T'$, if there is an edge-weighting of $T$ so that $Q^*_{\text{short}}(T) \subseteq Q(T')$, then $T = T'$. This theorem was proven in Erdos et al. (1999b), and is the basis for several algorithms, jointly called “short quartets methods”, that are used to construct trees from dissimilarity matrices. The two earliest short quartets methods include the dyadic closure method (Erdos et al., 1999b) and the witness-antiwitness method (Erdos et al., 1999c), both of which are polynomial time statistically consistent distance-based methods.

The benefit of the short quartets methods over the Naive Quartets Method is that they can be accurate under conditions where the Naive Quartets Method will fail to recover a tree. Thus, the short quartets methods have both theoretical and empirical advantages over the Naive Quartet Method.

The benefit of the short quartets methods over the more usual distance-based methods, such as Neighbor Joining (discussed in Chapter 5.7) is largely theoretical rather than empirical: from a theoretical perspective, the short quartets methods will be correct with high probability from asymptotically shorter sequences than what Neighbor Joining needs for accuracy with high probability. Here we show how to develop methods that will recover the true tree with high probability from polynomial length sequences!

In Chapter 3, we described the dyadic closure of a set of quartet trees, and Theorem 3.7, which established that if $Q$ is a set of quartet trees that contains all the short quartets in some tree $T$ and has no incorrect quartet tree (i.e., $Q^*_{\text{short}}(T) \subseteq Q \subseteq Q(T)$), then the dyadic closure of $Q$, denoted $cl_2(Q)$, produces $Q(T)$. As a result, the application of the All Quartets Method to $cl_2(Q)$ produces $T$. In other words, we can compute the tree $T$ in polynomial time from any set $Q$ of quartet trees that satisfies $Q^*_{\text{short}}(T) \subseteq Q \subseteq Q(T)$. Note that this analysis also reveals the following:

**Observation 13.1** If $cl_2(Q)$ contains two or more trees on any set of four taxa, then it is not possible for $Q$ to satisfy $Q^*_{\text{short}}(T) \subseteq Q \subseteq Q(T)$.

Based on Theorem 3.7 and Observation 13.1, an algorithm for constructing a tree from a dissimilarity matrix $d$ was developed in Erdos et al. (1999b). Here we present a simple version of the algorithm to illustrate the basic ideas.
Dyadic Closure Method. We describe a simple version of the dyadic closure method presented in Erdos et al. (1997).

- Input: $n \times n$ dissimilarity matrix $d$
- Output: a tree $T$ or else $fail$.

- For each entry $x$ in $d$, DO
  - Compute the set $Q_x$ of quartets of taxa where all six pairwise distances are at most $x$.
  - Compute the set of quartet trees for every quartet in $Q_x$, using the Four Point Method, and call it $Q^*_x$.
  - Compute the dyadic closure $cl_2(Q^*_x)$ and call it $Trees_x$.
  - If $Trees_x$ has exactly one quartet on every four leaves, then apply the All Quartets Method to $Trees_x$. This analysis is guaranteed to produce a tree $T$ satisfying $Q(T) = Trees_x$; return $T$.

- If no entry $x$ in $d$ produced a set $Trees_x$ that resulted in a tree, then return $Fail$.

As shown in Erdos et al. (1999b), while there can be more than one $x$ for which $Trees_x = Q(T)$ for some $T$, we can never have $x, y$ such that $Trees_x = Q(T)$ and $Trees_y = Q(T')$ where $T \neq T'$. Hence, if this technique returns a tree, then there is only one possible tree that it can return. As we showed in Erdos et al. (1999b), the Dyadic Closure Method runs in $O(n^5 \log n)$ time, where the input matrix is $n \times n$.

13.1.3 Other fast-converging methods

13.2 Phylogenetic networks

Many methods have been developed to analyze datasets where reticulate evolution is believed to have occurred, and typically these methods produce graphical outputs that are then used by biologists to understand the evolutionary history of their datasets. Some of these methods are designed to produce explicit graphical representations of the evolutionary history for the dataset (Jin et al., 2007; Nakhleh et al., 2004, 2005; Yu et al., 2011, 2012), but most of the methods are best suited to exploratory data analysis (EDA). This distinction between the two types of “networks” is important.

The biologist David Morrison elaborates on the importance of the distinction between these two types of networks in Morrison (2011), where he refers to networks that are best suited for EDA of phylogenetic data as “data-display networks, and networks that are graphical representations of a reticulate evolutionary history as “evolutionary networks.

As Morrison says (page 47 in Morrison (2011)):

The basic issue, of course, is the simple fact that data-display networks and evolutionary networks can look the same. That is, they both contain reticulations even if they represent different things... Many people seem to have confused the two types of network, usually by trying to interpret a data-display network as an evolutionary network... The distinction between the two types of network has frequently been noted in the literature, so it is hardly an original point for me to make here. Interestingly, a number of authors have explicitly noted the role of display networks in exploratory
data analysis and then proceeded to treat them as genealogies anyway. It is perhaps not surprising, then, that non-experts repeatedly make the same mistake.

See Gusfield (2014); Huson et al. (2010); Morrison (2011) for some textbooks about phylogenetic networks.

13.3 Genome rearrangements

13.4 Phylogenetic forests
PART THREE
LINGUISTIC PHYLOGENETICS
14
Constructing phylogenetic trees for languages

14.1 Introduction
Languages, like species, evolve over time, and through the accumulation of changes, they become unintelligible. The reconstruction of the evolutionary process (modelled by a tree) underlying the collection of languages is one of the goals of historical linguistics researchers. We describe some of the work in this area in this chapter.

14.2 Linguistic character data
The main types of linguistic characters are lexical, phonological, and morphological. Lexical characters are the words for a given meaning, but then analyzed with respect to cognate class membership. That is, if two words for the same meaning in two different languages were derived from a common ancestral word via sound changes, then the two words are considered to be cognates. Note that words that mean the same thing and sound the same may not be cognate! For example, the words ‘mucho’ in Spanish and ‘much’ in English are not cognate – they are derived from different words in their common ancestor. Similarly, the words for television in English and Japanese are not cognate, although clearly the word in Japanese was derived from the word in English. The reason they are not cognate is that the common ancestor of these two languages (if there is such a language) did not have televisions. Instead, the word for television in Japanese is a “loan word”, and loan words are not cognate to the words they are similar (or even identical to) in the lender language. Because there can be any number of cognate classes, lexical characters can have an unbounded number of possible states. More generally, each lexical character partitions the languages into cognate classes. We can code these cognate classes using any discrete set, since all that matters is the partition defined by the character states.

Phonological characters indicate the presence or absence of a sound change, where saying that a sound change is “present” means that somewhere in the evolutionary history of the language, the sound change occurred. Thus, phonological characters have two states – presence/absence, or 0/1.

The final character type is morphology – and most importantly inflectional morphology. Like lexical characters, morphological characters are defined by cognate judgments, which
means that they have evolved from a common ancestor. Thus, if two languages share the same state for a morphological character the assumption is that they inherited the state from their common ancestor. Note that morphological characters can have many different states.

Thus, by assumption, linguistic characters should be compatible on the true evolutionary tree for the languages. However, when two linguistic communities are in contact, linguistic characters can be transmitted laterally. In other words, the correct model of linguistic evolution is not a tree, but rather a tree with contact edges (indicating lateral transfer between languages resulting from contact between the linguistic communities) on top of the tree. Thus, this is a kind of phylogenetic network.

Some types of linguistic characters are easily transferred between linguistic communities, but others are more resistant to transfer. However, lateral transfer is very common for lexical characters, as is evidenced by the many loan words in many languages. Typically, however, loan words can be detected as loans - because of patterns of regular sound changes that the loan words may not exhibit. Since the coding of lexical characters does not assign the same state to the donor and recipient languages, when loan words are detected, the lexical character for the meaning is compatible with the underlying language tree.

Phonological characters can also be transmitted laterally, but this is less likely for complex phonological characters defined by a sequence of simpler phonological characters (e.g., Grimm’s Law).

It is debated whether morphological characters can be laterally transferred, and some linguists would argue that morphology either cannot be transferred horizontally or that such transfers are extremely unlikely. What is clear, however, is that the resistance to borrowing is strongest for inflectional morphology, somewhat weaker for complex phonological characters, but relatively low for lexical characters.

As described, the assumption is that lexical characters can evolve with lateral transfer but these laterally transferred states can often be detected. This is an idealized state that is not perfectly true, however, as there is some evidence that at least some character evolution also involves back mutation and/or parallel evolution. In other words, linguistic evolution also involves homoplasy.

Note that some linguistic characters can be directed, because there is a clear directionality of the evolutionary process. For example, presence/absence characters based upon sound changes (phonological characters) typically have an ancestral state (the absence of the sound change) and a derived state (the presence of the sound change).

Sometimes new states arise without replacement of the current state, so that a taxon exhibits two states (or more) at once. This is called polymorphism. Polymorphism in linguistic data occurs quite frequently – for example, when there are two or more words for the same basic meaning (examples include ‘big’ and ‘large’, or ‘rock’ and ‘stone’). Longterm polymorphism for linguistic characters does not seem to be tolerated well, so that over time, there are losses of character states, reducing the total amount of polymorphism in any language.
14.3 Models of linguistic character evolution

Several parametric models of linguistic character evolution have been proposed. Here we discuss the model proposed in Warnow et al. (2006), which allows for borrowing and homoplasy.

14.4 Inferring properties about proto-languages

In linguistics, ancestral taxa are typically called proto-languages, and various approaches have been used to estimate the properties of these proto-languages. Here we describe a technique for inferring the state of the ancestral proto-language, under the assumption that the linguistic character evolves in a treelike fashion (i.e., there is no borrowing) and that the character evolves without any homoplasy. Now suppose we know that a character state is shared by two or more languages, and that the most recent common ancestor of the languages that have that state is the root of the tree. In this case, we can infer that the proto-language also has that state. More generally, under the assumption of tree-like and homoplasy-free evolution, we can infer that all the ancestral languages represented by nodes that are on the path between any two languages sharing a character state will also have that character state. This inference is rigorous, but presumes fairly strong properties about the linguistic character homoplasy-free and tree-like evolution that may not hold for a given linguistic character. Thus, understanding the linguistic character in question is essential to being able to infer the properties of ancestral languages.

Knowing the tree, even if only partially, is essential to these inferences. Here we illustrate this with some examples from Indo-European, perhaps the best studied (and most well understood) language family. Suppose we assume that PIE (Proto-Indo-European) split into two groups, with the Anatolian languages in one group and all the other IE languages in the other group. Now suppose that \( c \) is a linguistic character for which we are confident that there is no homoplasy nor borrowing, and we know that Hittite and Greek both exhibit this state. Then, even if we are unsure about any other aspects of the phylogeny for Indo-European, we can be confident that PIE also has character state \( \alpha \) for character \( c \), because the node representing PIE (i.e., the root) is on the path between Hittite and Greek! On the other hand, suppose we have another character with the same strong properties (no homoplasy nor borrowing), and every non-Anatolian IE language has state \( \alpha \), and all the Anatolian languages have state \( \beta \). What can we infer about PIE? In fact, we cannot infer anything at all about the ancestral state - it might be \( \alpha \), it might be \( \beta \), or it might be some other state that did not survive in the daughter languages.

14.5 Computing linguistic phylogenies

Many methods have been developed to estimate linguistic phylogenies, and nearly all of them have corresponding methods for biological phylogeny estimation. Here we describe
the main methods that have been used, and present some discussion about the assumptions implicit in the methods about linguistic character evolution.

### 14.5.1 Glottochronology

One of the original methods used to estimate phylogenetic trees for languages was based on lexical characters, and computed the distance between two languages by counting the number of lexical characters in which the two languages were not cognates (i.e., had different cognate classes). Then, UPGMA is used to construct a tree on the languages.

This method, called “glottochronology”, is clearly dependent on the assumption that cognate loss is a clock-like process, meaning that the expected number of lexical character state changes from the root to any two languages sampled at the same time (e.g., modern languages) is the same. But, just as the strict molecular clock has been discredited, the strong lexical clock has also been discredited. For this reason, glottochronology is no longer in use for linguistic phylogeny estimation.

### 14.5.2 Maximum parsimony and maximum compatibility

Maximum parsimony and maximum compatibility, and their weighted versions, can also be used to estimate language trees. Both are motivated by the assumption that many linguistic characters will evolve in a treelike and homoplasy-free manner. Hence, a tree on which most characters are compatible will be more likely than a tree on which many characters are incompatible. However, as we have noted, some types of linguistic characters are considered more resistant to borrowing and homoplasy than others (e.g., morphological characters are generally more resistant to borrowing and homoplasy than lexical characters). Therefore, a tree on which a rigorously coded morphological character is not compatible is less credible than a tree on which it is compatible.

This reasoning leads to the use of character weighting, where the character weight indicates the confidence with which the linguist believes the character should be resistant to borrowing and homoplasy. Then, given a set of characters with their weights, the support for the tree is the total weight of all the characters that are compatible on the tree, and the objective is a tree that has the largest support. Formally, this is the weighted maximum compatibility problem, which is the natural extension of the maximum compatibility problem described in Chapter 4.

### Maximum weighted compatibility

**Input:** Matrix $M_{ij}$ where the rows represent the $n$ given languages, the columns represent the character set $C$ (i.e., $C = \{1, 2, \ldots, m\}$, and $M_{ij}$ is the state of language $i$ for character $j$. We are also $w_1, w_2, \ldots, w_m$, where $w_i$ is a non-negative real number that denotes the weight of character $j$. 

14.5 Computing linguistic phylogenies

Output: Tree $T$ on the set of languages so as $\text{score}(T) = \sum_{j \in \text{Compat}(T,C)} w_j$ is maximized, where $\text{Compat}(T,C)$ is the set of characters in $C$ that are compatible on $T$.

Note that the weights on the characters will impact the tree that is returned, and that the assessment of the relative probability of homoplasy involves a great deal of linguistic expertise.

Finding a solution to maximum compatibility (whether weighted or unweighted) is hard, because the problem is NP-hard. Thus, solutions that are guaranteed to solve the problem optimally use techniques like branch-and-bound or exhaustive search. Unfortunately, no software exists for solving this problem in an automated fashion. Instead, solutions to this problem have been obtained by first finding solutions to maximum parsimony (discussed below), and then scoring each of the trees with respect to the maximum compatibility criterion. This approach works reasonably because the two problems are very similar, so that optimal solutions to one problem are often near-optimal solutions to the other. Furthermore, while effective software for maximum compatibility does not really exist, there are many very effective software packages for maximum parsimony, due to its frequency of use in biological phylogenetics.

14.5.3 Gray and Atkinson’s approach

A very different type of approach to phylogeny estimation for languages was presented in Gray and Atkinson (2003). They used the multi-state lexical characters, and represented each multi-state lexical character as a set of binary characters – as described below. They then assumed that the binary characters evolve $i.i.d.$ down the model tree, and used a Bayesian analysis to estimate the tree. We now describe the technique used by Gray and Atkinson to produce a binary encoding of multi-state characters.

Suppose you have a character that exhibits $r$ states on a set $S$ of taxa. You replace that single $r$-state character by $r$ binary characters, one for each state. Then, the character for the state $i$ will indicate whether the language has that state or not. For example, consider a three-state character $C$ defined on set $\{L_1, L_2, \ldots, L_6\}$, so that $\{L_1, L_2\}$ have state 1, $\{L_3, L_4\}$ have state 2, and $\{L_5, L_6\}$ have state 3. The binary encoding of this three state character would produce three binary characters. The character for state 1 would split the taxa into two sets: those having state 1 (i.e., $\{L_1, L_2\}$) and those not having state 1 (i.e., $\{L_3, L_4, L_5, L_6\}$). Note that the evolution of character $C$ might have very different properties than the evolution of the binary characters derived from $C$. For example, this character $C$ is compatible on the tree $((L_1, L_2), (L_3, (L_4, (L_5, L_6))))$, but not all its derived characters are. Also, $C$ will change state on some edges of the tree but not all its derived characters will.
Constructing phylogenetic trees for languages

14.5.4 Geoff Nichols’ approach: Bags of words

14.5.5 Performance on data

14.6 Controversies

14.7 Further reading

14.8 Review questions

14.9 Homework problems
15

Constructing phylogenetic networks for languages

15.1 Introduction
In the previous chapter, we discussed linguistic character evolution, and the assumption that linguistic characters should be *compatible* on the true evolutionary tree for the languages. However, we also noted that when two linguistic communities are in contact, linguistic characters can be transmitted laterally, and this is particularly evident in the presence of loan words in different languages. Furthermore, while borrowing can often be detected, there are cases where even careful scholarship fails to detect borrowing. As a result, phylogenetic networks can be better models of language evolution than trees. This chapter discusses the models and methods that have been proposed to address this kind of reticulate phenomenon in language evolution.

15.2 Perfect phylogenetic networks

15.3 Further reading

15.4 Review questions

15.5 Homework problems
16
The future of linguistic phylogenetics

16.1 Introduction
16.2 Further reading
This chapter will provide a basic introduction to biological data and evolution.
Appendix B
Algorithm design and analysis

We assume some background in algorithm design and analysis, and in proving algorithms correct. Thus, you should know how to calculate the running time of an algorithm, as well as the standard “big-oh” notation. You will also need to know what it means for a problem to be \(NP\)-hard or \(NP\)-complete, and for a problem to be polynomial time. Much of the material involves probabilistic analysis of algorithms under stochastic models of evolution, so some very rudimentary probability theory is helpful.

The material in this chapter is not meant as a substitute for undergraduate courses in discrete mathematics and algorithm design and analysis, but rather to provide some of the material that the reader needs to know. We recommend that readers without sufficient background in this area consult other textbooks for background material.

B.1 Discrete mathematics

B.1.1 Graph theory

A graph \(G\) consists of a set \(V\) of vertices and a set \(E\) of edges, where the edges are unordered pairs of vertices; we often write graphs using notation \(G = (V, E)\). The edges of a simple graph are always distinct (thus, there are no self-loops, which are edges between a vertex and itself), and no parallel edges (two edges with the same endpoints). In this text, we will only discuss simple graphs.

Two vertices that are connected by an edge are said to be adjacent and the edge that connects them is said to be incident to its vertex endpoints. If a vertex \(a\) is adjacent to vertex \(b\), it is said to be a neighbor of \(b\). The degree of a vertex is the number of edges that are incident with the vertex, which is the same as the number of vertices that are neighbors of the vertex.

A path in a graph is a sequence of vertices \(v_1, v_2, \ldots, v_k\) so that \(v_i\) is adjacent to \(v_{i+1}\) for each \(i = 1, 2, \ldots, k - 1\). A graph for which every two vertices are connected by a path is said to be connected. The maximal connected subgraphs of a graph are called the components of the graph. (Saying that “\(X\) is a maximal connected subgraph” means that there is no subgraph \(Y\) of \(G\) that strictly contains \(X\) and is also connected. Thus, “maximal” is not the same as “maximum”.)
A cycle in a graph is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that $v_i$ is adjacent to $v_{i+1}$ for each $i = 1, 2, \ldots, k-1$, and $v_k$ is adjacent to $v_1$. A graph that has no cycles is said to be acyclic.

A tree is a graph that is connected and acyclic. Note that this definition of a tree may differ from what you are used to seeing; in particular, this definition of a tree does not provide a “root” for the tree, and makes no constraints on the degrees of the nodes in a tree. Later, we will distinguish between “rooted” and “unrooted” trees, but graph-theoretically the definition of a tree is quite simple (it’s just an acyclic connected graph).

Some concepts you should know include:

- **Clique**: a clique $X$ in a graph $G = (V, E)$ is a subset of the vertex set $V$, such that all pairs of vertices in $X$ are adjacent.
- **Independent Set**: an independent set $X$ in a graph $G = (V, E)$ is a subset of the vertex set $V$, such that no pair of vertices in $X$ are adjacent.
- **Hamiltonian Graph**: A graph $G$ is Hamiltonian if there is a cycle in the graph that covers every vertex exactly once.
- **Eulerian Graph**: A graph $G$ is Eulerian if there is a cycle in the graph that covers every edge exactly once.

Finally, what we have described so far is the usual kind of graph, in which edges are simply pairs of vertices. Sometimes, it is useful to direct the edges from one vertex to another, and graphs in which all the edges are directed are called directed graphs or digraphs. We use $\langle a, b \rangle$ or $a \rightarrow b$ to indicate the directed edge (also called an arc) from $a$ to $b$ in a digraph.

### B.1.2 Binary relations

Binary relations are used to represent many real world situations. Mathematically, a binary relation on a set $S$ is a set $R$ of ordered pairs of elements of $S$; thus, $R \subseteq S \times S$. For example, consider the binary relation $R$ on integers where $\langle a, b \rangle \in R$ means that $a$ divides $b$ without remainder. Hence, $\langle 3, 6 \rangle \in R$ but $\langle 3, 5 \rangle \notin R$, and $\langle 6, 3 \rangle \notin R$. However, binary relations do not have to mean anything in particular; you could for example take an arbitrary set $X$ of ordered pairs and consider it a binary relation. Two types of binary relations are frequently used – partial orders and equivalence relations. We will use both in this text, and so discuss these further, below.

### B.1.3 Hasse Diagrams and partially ordered sets

A partial order is a binary relation $R$ on a set $S$ satisfying

- $\langle A, B \rangle \in R$ and $\langle B, C \rangle \in R$ implies that $\langle A, C \rangle \in R$.
- $\langle A, A \rangle \in R$ for all $A \in S$
- $\langle A, B \rangle \in R$ and $\langle B, A \rangle \in R$ implies that $A = B$
We say that two elements \( A \) and \( B \) are comparable if \(< A, B > \in R \) or \(< B, A > \in R \), and when all pairs of elements are comparable then the partial order is called a “total order”. However, for most partial orders, not all pairs of elements of the set \( S \) are comparable. Suppose we define the partial order \( R_{\text{div}} \) on positive integers so that \(< A, B > \in R_{\text{div}} \) if and only if \( A \) divides \( B \) evenly (without remainder). It is easy to see that \( R_{\text{div}} \) is a partial order, but not a total order, since 3 and 5 are not comparable.

A partially ordered set (or \textit{poset}) is a set \( S \) with a partial order. Thus, the positive integers under \( R_{\text{div}} \) constitutes a partially ordered set. Another example of a partially ordered set is the set of all subsets of the integers, with partial order \( R_{\text{subset}} \) defined by \(< A, B > \in R_{\text{subset}} \) if and only if \( A \subseteq B \).

A Hasse Diagram is a drawing of the transitive reduction of a partially ordered set in the plane\(^1\). To construct the Hasse Diagram, create one vertex for each element in \( S \) and a directed edge \( x \rightarrow y \) if \(< x, y > \in R \) and \( x \neq y \). Order them from bottom to top on your page so that all the directed edges go upward. Then, repeatedly remove directed edges \( x \rightarrow y \) if there is a third vertex \( z \) such that \(< x, z > \in R \) and \(< z, y > \in R \).

### B.1.4 Equivalence relations

A binary relation \( R \) on the set \( S \) is said to be an \textit{equivalence relation} if it satisfies the following properties:

- \(< a, a > \in R \) for all \( a \in S \); this is called the \textit{reflexive property}.
- If \(< a, b > \in R \), then \(< b, a > \in R \); this is called the \textit{symmetric property}.
- If \(< a, b > \in R \) and \(< b, c > \in R \), then also \(< a, c > \in R \); this is called the \textit{transitivity property}.

When two elements \( x \) and \( y \) in the set \( S \) are in the relation (i.e., \(< x, y > \in R \)), then we say that \( x \) and \( y \) are equivalent. The equivalence relation \( R \) thus partitions the set \( S \) into \textit{equivalence classes}.

For example, you can define an equivalence relation \( R \) on a set \( S \) of people by saying \( x \) and \( y \) are equivalent if they earn the same salary (in dollars, not counting the amount past the decimal point). You can also define an equivalence relation \( R \) on the set \( \mathbb{Z}^+ \) of positive integers by saying \(< x, y > \in R \) if they have the same set of distinct prime factors. For example, under this relation, \(< 6, 18 > \) are in relation, since their prime factors are 2 and 3.

### B.1.5 Transitive Closure

Given a binary relation \( R \) on a set \( S \), the \textit{transitive closure} of \( R \) is the binary relation \( R^+ \) that is obtained by repeatedly adding \(< A, B > \) to \( R \) whenever \( R \) contains both \(< A, C > \)

---

1 Hasse Diagrams are named after Helmut Hasse, a mathematician who used Hasse diagrams in his research in algebraic number theory, but were introduced earlier by another mathematician Vogt (1895)
and \(< C, B >\) for some \(C \in S\), until no additional ordered pairs can be added. Equivalently, the transitive closure of \(R\) is the smallest binary relation containing \(R\) that is transitive.

### B.1.6 Counting

The running time analysis of algorithms depends on being able to count the number of operations the algorithm executes as a function of its input size. This analysis then depends on a kind of discrete mathematics called “combinatorial counting” (or “counting”, for short).

Suppose \(S\) is a set of \(n\) distinct objects, \(s_1, s_2, \ldots, s_n\). Consider the following questions:

1. How many possible subsets of \(S\) are there, including the empty set and the set \(S\)?
2. How many non-empty subsets are there?
3. How many subsets are there that contain \(s_1\)?
4. How many subsets are there that do not contain \(s_1\)?
5. How many subsets are there that contain exactly one of \(s_1\) and \(s_2\)?
6. How many ways can you partition this set into two non-empty sets?
7. How many functions are there from \(\{1, 2, \ldots, k\}\) to \(S\)?
8. How many functions are there from \(S\) to \(\{1, 2, \ldots, k\}\)?
9. How many ways can you order the elements of \(S\)?

Techniques for combinatorial counting vary from very easy (enumerate all the objects algorithmically) to somewhat complicated (use Inclusion-Exclusion).

### B.2 Proof techniques

You will often need to prove theoretical results, and different techniques can be used to prove these results. Here we describe a few basic techniques.

#### B.2.1 Proof by induction

Suppose that the sequence \(a_1, a_2, \ldots\), is defined recursively by \(a_1 = 2\) and \(a_i = 3 \times a_{i-1}\) for \(i \geq 2\). Thus, \(a_2 = 6, a_3 = 18, a_4 = 54\), etc. We will prove that \(a_i = 2 \times 3^{i-1}\) for all \(i \geq 1\) by induction on \(i\).

In a proof by induction, you have to establish that a statement is true for the smallest value of some parameter (here, \(i\)); this is called “proving the base case”. The next step is the “inductive hypothesis”: you assume it is true for arbitrary setting of the parameter. If you can then show that it will be true for the next value of the parameter, then it will be true for all settings of the parameter, starting with the base case.

**Theorem B.1** Let \(a_1 = 2\) and \(a_i\) be defined to be \(3 \times a_{i-1}\) for integers \(i \geq 2\). Then \(a_i = 2 \times 3^{i-1}\) for all integers \(i \geq 1\).
Proof We will prove this statement by induction on \( i \). Note that the statement depends on \( i \); hence, we can think of this as proving that the statement \( S(i) = "a_i = 2 \times 3^{i-1}" \) is a true statement for \( i \geq 1 \). We begin by showing that the statement is true for the base case (the smallest value for \( i \)). We then assume the statement is true for some arbitrary value of \( i \), and infer from this that it is true for the next value of \( i \). Equivalently, we assume that the statement \( S(I) \) is true, and we use that to infer that \( S(I+1) \) is also true, where \( I \) is an arbitrarily chosen value for \( i \).

The base case is \( i = 1 \). We know that \( a_1 = 2 \), by definition. We then check that \( 2 \times 3^{i-1} = 2 \) when \( i = 1 \). We note that \( S(1) \) is true, and so the base case holds.

We then assume that \( S(I) \) is true; hence, \( a_I = 2 \times 3^{I-1} \). This is called the Inductive Hypothesis. Note that the Inductive Hypothesis is a statement about what happens when \( i = I \), and not about any other value for \( i \). Now, by definition, \( a_{I+1} = 3 \times a_I \). By the inductive hypothesis, \( a_I = 2 \times 3^{I-1} \). Hence, \( a_{I+1} = 3 \times (2 \times 3^{I-1}) = 2 \times 3^I \). This is what we wanted to prove, so we are done.

B.2.2 Proof by contradiction

In a proof by contradiction, to prove that a statement is true, you assume it is not and then derive a contradiction. For example, here’s a simple proof of a relatively obvious fact.

Theorem B.2 There are an infinite number of prime numbers.

Proof To prove this, we assume there is a finite number of prime numbers, and try to derive a contradiction. So let \( \{p_1, p_2, \ldots, p_k\} \) be the set of all primes, and let \( Y = 1 + \prod_{i=1}^{k} p_i \). Note that 1 is not a prime number, and so \( p_i \geq 2 \) for \( i = 1, 2, \ldots, k \). Also note that \( Y > p_i \) for each \( i \). Every number can be written as a product of its prime divisors, and so \( Y \) must have a prime factor \( p_i \) for some \( i, 1 \leq i \leq k \). Then, consider the result of dividing \( Y \) by \( p_i \); since \( p_i \) is a prime factor, the result must be an integer. But the result is not an integer, since \( Y = 1 + p_i Z \), where \( Z = \prod_{j \neq i} p_j \). This is a contradiction.

B.3 Running time analysis

B.3.1 Pseudo-code

Before we analyze the running time of a method, we need to be able to describe what the method does. For the purpose of describing methods, pseudo-code is better than using a real programming language. The objective is to make it as easy as possible for your reader to understand the algorithm, so the description should be simple, provide all the information necessary to understand it, and not require knowledge of any particular programming language. Your pseudo-code can certainly include English, but when you use English make sure that you aren’t omitting necessary information.

You will need to have symbols that express assignment of values to variables (I use :=, but others use ←). You will need to have symbols that express comparisons of variables
B.3 Running time analysis

(Use a different symbol to test equality than to do assignment, so “X=Y” should not mean that X and Y have the same value and also be used to assign value Y to variable X). Otherwise, you can use all the usual things (arithmetic operations, logical expressions such as IF/THEN/ELSE, and WHILE and FOR loops). You can use subscripts to refer to variables (i.e., $a_i$) or use elements in an array (i.e., $a[i]$), as you prefer.

Now, to analyze the running time of the method, we need to count the number of operations the algorithm uses on an input of size $n$; note that this means we need to be able to quantify, in some way, what we mean by the “size” of an input. For graphs, the size depends on the representation, but depends on the number $n$ of vertices and number $m$ of edges. We can represent a graph $G$ on $n$ vertices with an $n \times n$ adjacency matrix $M$ where $M_{ij} = 1$ if and only if the edge $(v_i, v_j) \in E$, or with an adjacency list $L = (L_1, L_2, \ldots, L_n)$, where $L_i$ is the list of indices that are adjacent to $v_i$. If we don’t worry about the space needed to represent the indices $1, 2, \ldots, n$, the first representation uses $\Theta(n^2)$ space, and the second representation uses $\Theta(m+n)$ space. When the graph is sparse (i.e., has relatively few edges), then an adjacency list can require less space than an adjacency matrix, but the space usage for the two data structures are about the same for dense graphs.

Despite the fact that adjacency lists are more efficient representations of graphs, some algorithms are more efficiently implemented using adjacency matrices instead of adjacency graphs. However, if the only issue is whether the running time is polynomial or not, then either representation can be used. To provide a proper running time analysis, however, you will need to specify the data structures you use to represent your input. However, what is your algorithm is attempting to determine if an integer input number $K$ is prime? A simple algorithm would look at every integer $i$ between 2 and $K-1$ and see if $i$ divides $K$. This would take $O(K)$ time, if every division operation takes $O(1)$ time. However, is this polynomial in the input size? The question comes down to how we can efficiently represent integers. If we represent $K$ using base 10 (the usual representation), this will use $O(\log_{10}(K))$ digits. If we switch to a binary representation, this will use $O(\log_2(K))$ bits. These two representations differ only by a constant factor. Therefore, for representations of integers, we say that the “size of $n$” is $\log n$ (where log can be any base greater than 1).

We also have to say which operations are allowed and how much they cost. Running time analyses normally just consider every operation to have the same (unit) cost, and allow standard arithmetic operations, I/O operations, and logical operations. Thus, assigning values to variables, adding or multiplying numbers, comparing two numbers, and IF/THEN/ELSE operations all have the same cost.

B.3.2 Big-Oh analysis

Computational methods that are designed to solve problems should be highly accurate (preferably completely accurate) and also fast. In this section, we discuss how to characterize the asymptotic running time of a method.

We are normally concerned with obtaining an upper bound on the running time, and when we talk about “big-oh” running times, we are providing an upper bound on the run-
ning time. This upper bound essentially hides all the constants that are involved. Thus, if we say that an algorithm has $O(n^4)$ (“big-oh of $n$ to the fourth”) running time, then we are saying that the running time on inputs of size $n$ will never be larger than $Cn^4$, for some constant $C > 0$ and large enough values of $n$.

Note that we can use big-oh analysis to compare two functions, not just to characterize the running time of a method. For example, the following statements are all true:

1. $5n^4$ is $O(n^4)$
2. $5n^4 + 500n^3 + 300,000$ is $O(n^4)$
3. $5n^2 + \log n$ is $O(n^2)$
4. $500n^2 - 3n$ is $O(n^2)$
5. $5n^2$ is $O(n^3)$

To verify these statements are true, you’d need to be able to find the constants $C$ that make the statements true for large enough $n$. For example, statement (1) is easily seen as true, by letting $C = 5$, since then the statement becomes $5n^4 \leq 5n^4$, which is always true. For the second statement, if you set $C = 5$ the statement will not be true – because $5n^4 + 500n^3 + 300,000$ is greater than $5n^4$ when $n > 0$. However, if you set $C = 6$, then you can find $N_0$ so that $5n^4 + 500n^3 + 300,000 \leq 6n^4$ for $n > N_0$. (For example, see if $N_0 = 500$ makes the statement true.) Hence, the second statement can be proven true as well. Similarly, every one of these statements can be proven.

B.4 Different types of problems: decision, optimization, and construction problems

Decision problems. Decision problems simply ask for the answer to a Yes/No question. A simple example of this would be “Does this array have the value 5 in it?” The answer to the problem is just Yes or No. Determining the answer to this particular problem is easy (just scan the array, looking for value 5, and return “Yes” if you find it, and otherwise return “No”). Other decision problems aren’t always as easy to answer. For example, you might want to know if you can assign three colors (red, yellow, and blue) to the vertices of a graph so that no two adjacent vertices have the same color. Such a vertex coloring is called a “proper 3-coloring”, where “proper” means that there are no adjacent vertices having the same color. How efficiently can we determine if a graph has a proper 3-coloring?

Exhaustive search techniques evaluate all possible solutions, and so can solve problems exactly when the number of possible solutions is finite. Since the number of possible 3-colorings of a graph with $n$ vertices is finite (i.e., only $3^n$), the exhaustive search solution will certainly work; however, checking all $3^n$ different colorings is not efficient (and certainly not polynomial time). More efficient strategies exist, but none of them have been shown to have polynomial worst case running times.

On the other hand, determining whether a graph has a proper 2-coloring (i.e., whether you can assign just two colors to the graph so that no vertices with the same color are adjacent, then this problem is easy to solve. Yes, you could still use exhaustive search (which would require checking $2^n$ different colorings), but there are more efficient ways to
solves the problem. In fact, a greedy algorithm will correctly solve this problem: Color one vertex red, then color its neighbors blue, then color vertices adjacent to those blue vertices red, and continue until all reachable vertices have been colored. If there are any vertices not yet colored, then start again with some uncolored vertex. At the end, all vertices will be colored, and you just need to check that no two vertices with the same color are adjacent. Thus, some decision problems seem to be easy to solve, and others seem harder.

Optimization problems. Optimization problems are another type of problem, where instead of finding the answer to a Yes/No question, you want to find the score of the best possible solution to some problem. For example, you might want to find the largest value \( k \) so that a graph has a vertex \( v \) of degree \( k \) (i.e., a vertex that has \( k \) neighbors). Or you might want to find the largest \( k \) so that the graph has a clique of size \( k \) (i.e., the Maximum Clique problem). Or you might want to find the smallest value \( k \) so that the graph can be properly vertex-colored using \( k \) colors (the Minimum Vertex Coloring problem).

Again, exhaustive search will provide correct solutions to optimization problems, but usually these approaches have exponential running times. However, some optimization problems can be solved in polynomial time, even when exhaustive search is exponential. For example, suppose we are given an array of integers, and we want to find an ordering of the integers \( x[1], x[2], \ldots, x[n] \) so that we minimize \( \sum_{i=1}^{n-1} |x[i] - x[i+1]| \). If we use exhaustive search, we can evaluate the result of using every possible ordering, but there are \( n! \) orderings. However, it is not hard to see that the best ordering is obtained by sorting the elements, which can be solved in polynomial time.

Nevertheless, many optimization problems are hard to solve efficiently, in that despite many efforts, no polynomial time algorithms have been found for them. Examples of such problems include the Minimum Vertex Coloring and Maximum Clique problems, defined above.

Construction problems. Finally, construction problems are ones where you want to find an object (if it exists). For example, you might want to find a maximum sized clique in a graph, or a proper vertex coloring of a graph using a minimum number of colors; these would be the construction problems for a given optimization problem. However, you might also define the construction problem version of a decision problem; hence, instead of asking “Does the graph have a proper 3-coloring?” you might say “If the graph has a proper 3-coloring, then find one”.

Thus, decision problems, optimization problems, and construction problems are different types of problems. However, there are connections between the different types of problems. For example, it is obvious that if you can solve the construction problem version of the decision problem “Does the graph have a proper 3-coloring”, then you can also answer the decision problem. What is less obvious is that if you can solve the decision problem “Does the graph have a proper 3-coloring”, then you can use it to construct a proper 3-coloring!

Here’s an algorithm that shows how you can do this. Assume that you have an algorithm
A that takes as input a graph $G$ and returns Yes if the graph $G$ can be properly 3-colored, and else returns No. Suppose you want to construct a proper 3-coloring for graph $G_0$, if it exists. You first apply algorithm $A$ to $G_0$. If the answer is that there is no proper 3-coloring, you immediately give up (no point in trying to construct something that doesn’t exist). However, if algorithm $A$ says there is a proper 3-coloring, you continue, as follows.

- If $G_0$ has three or fewer vertices, then give a different color to every vertex, and you are done.
- Else, make a list $L$ of all pairs of vertices that are not adjacent to each other. Maintain an equivalence relation on the vertices of $G_0$ in which initially all vertices are in their own equivalence class. Let $G'$ be a copy of $G_0$. Then, for every pair $v, w$ in the list $L$, DO:
  - Make a new graph $G_1$ in which $v$ and $w$ are collapsed into a new node $x$, which is adjacent to all nodes that $v$ and $w$ were adjacent to in $G'$. Note that $G_1$ has one fewer vertex, and the number of edges in $G_1$ is at most the number of edges in $G_0$.
  - Use $A$ to find out if $G_1$ has a proper 3-coloring. If the answer is Yes, then replace $G'$ by $G_1$, merge the equivalence classes for $v$ and $w$, and rename the equivalence class $vw$.

After processing the list, the graph $G'$ will have at most three vertices, and each vertex will represent an equivalence class of vertices in the original graph $G_0$. Assign the three colors arbitrarily to the (at most three) equivalence classes; all the vertices in the same equivalence class get the same color. This will be a proper 3-coloring of the graph $G_0$.

### B.5 The classes $P$ and $NP$ of decision problems

Some decision problems can be solved exactly in polynomial time; examples of these include the 2-colorability problem, which asks “Does the graph have a proper 2-coloring?”. Other decision problems can also be solved in polynomial time, although the polynomial time algorithms that solve them may be less obvious. For example, a graph $G$ is said to be Eulerian if there is a cycle in the graph that covers every edge exactly once. The decision “Eulerian Graph” problem, which asks “Is this graph Eulerian?”, can also be solved in polynomial time using a very simple algorithm, but proving the algorithm correct is somewhat more challenging.

The class $P$ is the set of decision problems that can be solved exactly in polynomial time. Thus, the 2-colorability and the Eulerian Graph problems are both in $P$. Determining whether a given decision problem is in $P$ is equivalent to asking whether it can be solved in polynomial time.

The class $NP$ is also a set of decision problems, but is defined differently. Instead of asking whether the problem can be solved in polynomial time, we are interested in whether we can prove that the answer is Yes in polynomial time. The trick here, however, is that we don’t consider the time to come up with the proof. For example, suppose that the problem is 3-colorability, so we want to know if the vertices of the input graph can be assigned three colors, so that no two adjacent vertices have the same color. Some graphs can be 3-colored,
and some cannot. We are not concerned with any graph that cannot be 3-colored, because these are not ones for which the answer is Yes. However, suppose that the graph $G$ can be 3-colored. How can we prove this, and make the proof only take polynomial time?

Don’t consider the time it takes to figure out that the graph can be 3-colored (e.g., we might go ahead and examine every possible 3-coloring, to find one that is proper, but we won’t count the time to do this). Once we have found the proper 3-coloring, how can we use this to prove that the graph can be 3-colored?

The answer is obvious: write down the 3-coloring, and then verify, edge by edge, that no two adjacent vertices have the same color. The running time is easily seen to be polynomial: writing down the 3-coloring takes $O(n)$ time, and verifying that no edge connects vertices of the same color takes $O(n^2)$ time (since there are fewer than $n^2$ edges in a graph with $n$ vertices).

Thus, while we do not know how to solve 3-colorability in polynomial time, when a graph $G$ can be 3-colored then we can prove that $G$ has a 3-coloring in polynomial time. This means that 3-colorability is in the class $NP$.

B.6 The $NP$-complete problems

B.6.1 Introduction

Informally, the $NP$-Complete problems are the hardest problems in $NP$. As a result, $NP$-complete problems are decision problems, since they are problems in $NP$, and $NP$ is a set of decision problems. But what do we mean by “the hardest problems in $NP$”?

Saying that a problem $X$ is at least as hard as any problem in $NP$ means that if $X$ could be solved in polynomial time, then every problem in $NP$ could be solved in polynomial time. For example, 3-colorability (i.e., the decision problem that asks whether the vertices of the input graph can be properly colored using three colors) has been proven to be one of the $NP$-complete problems. What this means is that if anyone ever manages to develop a polynomial time algorithm for 3-colorability, then every other problem in $NP$ could be solved in polynomial time.

To prove that a problem $X$ is $NP$-complete you therefore need to prove two things: (1) that it is in $NP$ (this is generally the easy part), and (2) that $X$ is at least as hard as any other problem in $NP$ (this is the hard part). However, some problems have already been proven to be $NP$-Complete (for example, 3-colorability). Therefore, all we need to do is to find some $NP$-complete problem $Y$, and show that if we can solve $X$ in polynomial time, then we can solve $Y$ in polynomial time. Once we do this, the result follows – because if $X$ can be solved in polynomial time, then so can $Y$, and hence so can every other problem in $NP$ (because $Y$ is $NP$-complete).

We will show how to do this with a very simple example. We will use the fact that 3-colorability is $NP$-complete to prove that 4-colorability is $NP$-complete. What we will do is give a technique that will take any input $I$ to 3-colorability and turn it into an input $f(I)$ to 4-colorability.
B.6.2 Karp reductions

The technique we describe is an example of a Karp Reduction (named after Richard Karp, who developed this technique for establishing NP-completeness). Karp reductions go from problem $Y$ (already established to be NP-Complete) to problem $X$ (the problem you wish to prove NP-complete). A Karp reduction has the following properties:

- The technique $f$ maps inputs (instances) to problem $Y$ to inputs to problem $X$, and the time $f(I)$ takes is polynomial in the size of $I$.
- For inputs $I$ to problem $Y$, the size of $f(I)$ is bounded by a polynomial in the size of $I$.
- $I$ is a yes-instance for $Y$ if and only if $f(I)$ is a yes-instance for problem $X$.

The size of an input: Note that the size of an input $I$ is an important quantity that determines whether $f$ is a Karp reduction or not. In essence, the size is what is needed to represent $I$. For example, to represent an integer $x$ in binary, we use $\log_2(x) + 1$ space, and to represent it in base 10, the way we write it, uses $\log_{10}(x) + 1$ space. Although $\log_{10}(x) < \log_2(x)$, they are related a constant $C$ (i.e., $\log_2(x) = C \log_{10}(x)$). Hence, a polynomial in $\log_{10}(x)$ is a polynomial in $\log_2(x)$. Therefore, for the purposes of defining a Karp Reduction, the specific choice of base does not matter.

As another example, how much space do we need to represent a simple graph with $n$ vertices, $v_1, v_2, \ldots, v_n$? Recall that we can represent a graph $G = (V, E)$ with an adjacency matrix or with an adjacency list, and both use a polynomial in $n$ amount of space (but different polynomials). Hence, although the space needed to represent a graph using an adjacency list can be less than the space needed to represent it using an adjacency matrix, both are equivalent in terms of the implications for Karp Reductions.

Suppose 4-colorability can be solved in polynomial time, and that $A$ is a polynomial time algorithm that solves 4-colorability exactly. Now suppose also that we have a Karp reduction $f$ from 3-colorability to 4-colorability. We will show how we will use $f$ and $A$ to solve 3-colorability in polynomial time. Given an input $G$ to 3-colorability (note that $G$ is a graph), we apply the transformation $f$ to obtain a graph $G'$. Because $f$ is a Karp reduction, the calculation of $G'$ uses polynomial time, the graph $G'$ has at most $\text{poly}(n)$ vertices, where $n$ is the number of vertices in $G$ and $\text{poly}(n)$ is a polynomial in $n$. Furthermore, $G$ can be properly 3-colored if and only if $G'$ can be properly 4-colored. We apply algorithm $A$ on $G'$; the output is Yes or No, depending on whether $G'$ can be 4-colored. If the answer is Yes, then we know $G'$ has a proper 4-coloring, and hence $G$ has a proper 3-coloring; similarly if the answer is No, then we know $G$ does not have a proper 3-coloring. Running algorithm $A$ on $G'$ uses polynomial time in the number of vertices in $G'$, and so bounded by the composition of two polynomials, which is itself polynomial. Therefore, the entire process (computing $G'$ given $G$, and running the algorithm $A$ on $G'$) completes in time that is polynomial in the size of $G$, and so is a polynomial time algorithm for 3-colorability.

Hence, if we can find a Karp Reduction from 3-colorability to 4-colorability, then if 3-colorability can be solved in polynomial time then so can 3-colorability. This analysis had nothing to do with the details of the two problems – all we needed was the Karp Reduction.
Hence, if $X$ and $Y$ are both problems in $NP$, and $X$ has a Karp Reduction to $Y$ and $X$ is NP-complete, then $Y$ is NP-hard. And, since 3-colorability is already established to be $NP$-complete, then if 4-colorability can be solved in polynomial time, then every problem in $NP$ can be solved in polynomial time.

To prove that 4-colorability is NP-Complete, we need to show that it is in $NP$, and then present the Karp reduction from some $NP$-complete problem $Y$ to 4-colorability. It is easy to see that 4-colorability is in $NP$ (just present a proper 4-coloring, and verify that all vertices are colored one of four colors, and no edge connects vertices of different colors). Now we need to come up with the Karp reduction, i.e., a polynomial time transformation $f$ that maps instances to 3-colorability to instances to 4-colorability, so that $G$ can be properly 3-colored if and only if $f(G)$ can be properly 4-colored, and so that $G'$ has size bounded by a polynomial in the size of $G$.

A Karp reduction from 3-colorability to 4-colorability. The Karp reduction is the function $f$, which maps graphs to graphs, as we now describe. Given graph $G = (V, E)$, the graph $G' = f(G)$ is formed by adding one vertex $v^*$ to $G$, and making $v^*$ adjacent to every vertex in $G$. We need to show that $f$ is a Karp reduction.

We begin by analyzing the size of $G'$, and showing that it is bounded by a polynomial in the size of $G$. Note that $G'$ has $n + 1$ vertices (where $n = |V|$) and $m + n$ edges (where $m = |E|$). Thus, the size of $G'$ is bounded by a polynomial in the size of $G$. We then note that $f$ takes polynomial time to compute. The next part is to show that $f$ maps yes-instances to yes-instances, and no-instances to no-instances. Now, suppose $G$ has a proper 3-coloring; if we use the same coloring on $G'$, and then add a new color for vertex $v^*$, we have a proper 4-coloring on $G'$. Conversely, suppose $G'$ has a proper 4-coloring, using colors red, blue, yellow, and green. Without loss of generality, suppose $v^*$ is colored red. Since $v^*$ is adjacent to every vertex in $G$, then the proper 4-coloring does not assign red to any other vertex in $G'$, and hence defines a proper 3-coloring to $G$. Hence, $G$ has a proper 3-coloring if and only if $G'$ has a proper 4-coloring. Thus, we have proven that the transformation is a Karp reduction, and hence that 4-colorability is $NP$-complete.

B.7 The $NP$-hard problems

The difference between $NP$-hard and $NP$-complete is only that a problem that is $NP$-complete must be in the class $NP$. Thus, $NP$-hard problems are computational problems (maybe not decision problems, however) that are at least as hard as any problem in $NP$. Hence, to say that a problem $X$ is $NP$-hard means that if it could be solved in polynomial time, then every problem in $NP$ could be solved in polynomial time.

As we said earlier, algorithms for decision problems can be used to solve optimization and construction problems, and vice-versa. Therefore, suppose we want to prove that Minimum Vertex Coloring is $NP$-hard; this means that we need to show that if we can solve Minimum Vertex Coloring in polynomial time, then we can solve any problem in $NP$ in
Algorithm design and analysis

polynomial time. We will do this by showing that if we can solve Minimum Vertex Coloring in polynomial time, then we can solve 3-colorability in polynomial time. So suppose we have a polynomial time algorithm $A$ for Minimum Vertex Coloring. Let graph $G$ be an input to 3-colorability. We will use the algorithm $A$ to answer whether $G$ can be properly 3-colored. We run algorithm $A$ on $G$. If the answer is at most 3, then we know that $G$ can be properly 3-colored, and else we know $G$ cannot be properly 3-colored. Hence, if Minimum Vertex Coloring can be solved in polynomial time, then so can 3-colorability. Since 3-colorability is $NP$-hard, this means that Minimum Vertex Coloring is $NP$-hard.

B.8 General algorithm design techniques

B.8.1 Introduction

Algorithms researchers design methods that solve problems, whether decision, optimization, or construction problems. The usual objective is an algorithm that is guaranteed to solve the problem exactly, and that does so efficiently. Thus, we would like not only a polynomial time algorithm, but one that is as fast as possible. Thus, we distinguish between methods that run in $O(n^3)$ time and $O(n^2)$ time, and prefer the quadratic running times to the cubic running times. (Similarly, we would prefer an $O(n)$ algorithm to an $O(n^2)$ algorithm.)

As we have seen, some problems are NP-hard, and the only exact solutions found for NP-hard problems have required more than polynomial time. However, for problems that can be solved in polynomial time, algorithm design techniques can make a difference between an exponential time and a polynomial time algorithm, or between an $O(n^3)$ algorithm and an $O(n^2)$ algorithm. Here we describe a few of these techniques.

B.8.2 Dynamic programming

In a dynamic programming algorithm, the idea is to decide in advance all the subproblems you need to solve, the order in which you’ll solve them, and how solving all the subproblems allows you to solve the entire problem. As long as the number of subproblems is small enough (preferably polynomial in the input size) and solving a subproblem is polynomial time once the earlier subproblems are solved, the entire approach uses polynomial time. An important point about dynamic programming is that when you solve the subproblem, you have to save its solution in memory so that you can easily access it later (without having to recalculate it).

A very simple example of a dynamic programming algorithm is one that computes the $n^{th}$ Fibonacci number. Recall that the Fibonacci numbers are formed by having the first two numbers equal to 1, and then each successive number is the sum of the previous two numbers. Thus, $F(1) = F(2) = 1$ and $F(i) := F(i - 1) + F(i - 2)$; in other words, the Fibonacci numbers are defined recursively. To compute the $n^{th}$ Fibonacci number, we design a dynamic programming (DP) algorithm, as follows.
The input to the problem is \( n \), and the output is the \( n \)th Fibonacci number \( F(n) \). We let the \( j \)th subproblem be computing \( F(j) \) and we note that we need to compute \( F(j) \) for all \( j \) between 3 and \( n \) (because \( F(1) \) and \( F(2) \) are already known to be 1). The algorithm computes these values in increasing order for \( j \), and stores the results in an array \( \text{Fib}[1...n] \). We first compute \( \text{Fib}[3] \), then \( \text{Fib}[4] \), etc., until we obtain \( \text{Fib}[n] \). To compute \( \text{Fib}[i] \) given all the previously computed values, we set \( \text{Fib}[i] \) to the sum of the previous two values. Thus, computing \( \text{Fib}[i] \) takes constant time, as long as we calculate \( \text{Fib}[i-1] \) and \( \text{Fib}[i-2] \) before we compute \( \text{Fib}[i] \). In other words, the algorithm has the following form:

**Computing \( F(n) \), the \( n \)th Fibonacci number:** Comment: We will compute values and store them in an array, from the smallest index to the largest. We then return the value in the last element of the array.

If \( n \) is not a positive integer, return Null.

\[
\text{Fib}[1]:=1, \text{Fib}[2]:=1
\]

For \( j=3 \) up to \( n \) DO:

\[
\text{Fib}[j]:= \text{Fib}[j-1] + \text{Fib}[j-2]
\]

End(For)
Return \( \text{Fib}[n] \)

A running time analysis shows that the initialization (before the loop is entered) uses 3 operations (one to check that \( n \) is a positive integer, and the other two to set the values of \( \text{Fib}[1] \) and \( \text{Fib}[2] \)). Then, every time the loop is entered, we use four operations (two to look at the values of \( \text{Fib}[j-1] \) and \( \text{Fib}[j-2] \), one operation to add those values, and then one operation to set the value of \( \text{Fib}[j] \)). The loop is entered \( n - 2 \) times. Hence, the total time is \( O(n) \), where \( n \) is the input.

### B.8.3 Recursive algorithms

Recursive algorithms can look a lot like dynamic programming algorithms, because the solution to a problem depends on solving smaller subproblems. However, unlike the dynamic programming approach, the number of possible subproblems need not be small – since typically only a proper subset of the possible subproblems will be solved during the course of the algorithm.

Analyzing the running time for a recursive algorithm typically amounts to analyzing a recursively defined function for the running time \( t(n) \), where \( n \) is the input (or the size of the input). For example, consider the simple sorting algorithm that scans an array, finds the largest value, and swaps that largest value with the last element of the array, and then recurses on the first \( n - 1 \) elements. In this case, the running time \( t(n) \) for arrays of size \( n \) satisfies

\[
t(n) = Cn + t(n - 1)
\]
Algorithm design and analysis

and

\[ t(1) = C', \]

where \( C \) and \( C' \) are two positive constants. This recursively defined function can be solved exactly, and yields \( t(n) \leq Cn^2 + C' \). Hence, \( t(n) \) is \( O(n^2) \), and the recursive algorithm runs in \( O(n^2) \) time.

Note that using dynamic programming to sort the array would not have been so pleasing; if we decided to sort all subarrays, the number of possible subarrays would have been exponential, and the running time would have been exponential as well. Thus, recursion trumps dynamic programming in this case, because even though there are an exponential number of possible subproblems, the recursive algorithm only explores a linear number of subproblems, and each one can be solved in polynomial time.

On the other hand, sometimes recursion is less efficient than dynamic programming. For example, we saw above that we could use dynamic programming to compute the \( n^{th} \) Fibonacci number \( F(n) \). If we try to use recursion to compute \( F(n) \), the algorithm would look like this:

1. If \( n \) is not a positive integer, return Null
2. Else, if \( n \leq 2 \) return 1
3. Else, return \( F(n-1) + F(n-2) \)

Note that in this pseudo-code, when \( n \geq 3 \), we are calling the algorithm recursively on \( n-1 \) and \( n-2 \); therefore, the running time \( t(n) \) for the algorithm on input \( n \) satisfies \( t(n) = t(n-1) + t(n-2) + C \) (for some constant \( C \)) and \( t(1) = t(2) = C' \). Finding a closed form for this function is not easy, but bounding the running time is possible. Unfortunately, it is not hard to show that \( t(n) \sim g^n \) for some constant \( g > 1 \). (See homework!)

The problem with using recursion here is that although the number of subproblems is polynomial, unlike with dynamic programming, we may compute each subproblem more than once. For example, if \( n = 15 \), then when we compute \( F(15) \) we recursively compute \( F(14) \) and \( F(13) \). But when we compute \( F(14) \) we also recursively compute \( F(13) \). So we computed \( F(13) \) twice during this analysis. It is not hard to see that \( F(12) \) is computed even more often. (See homework!)

B.8.4 Divide-and-conquer

Divide-and-conquer algorithms are a type of recursive algorithm, but typically have a more elaborate design. For example, Merge Sort is a divide-and-conquer approach to sorting an array of \( n \) integers: the array is divided into two approximately equal sets, each subset is recursively sorted, and then the two sorted arrays are merged together. Since the merger technique takes only \( O(n) \) time, the running time \( t(n) \) can be shown to be \( O(n \log n) \).
B.9 Designing algorithms for \( NP \)-hard problems

If you learn that a problem is \( NP \)-hard, what does this mean in terms of practice? As we have discussed, by definition, if any \( NP \)-hard problem can be solved in polynomial time, then they can all be solved in polynomial time. Equivalently, this is the same as saying that if some \( NP \)-hard problem is solvable in polynomial time, then \( P = NP \). Whether “\( P = NP \)” or not is of the most fundamental questions in computer science, and while it is still open (i.e., unsolved), most researchers assume that \( P \neq NP \). In other words, most computer scientists assume that no \( NP \)-hard problem can be solved in polynomial time.

Therefore, if you know a problem is \( NP \)-hard, then don’t try to solve it exactly, unless you are willing to use more than polynomial time on some inputs. In other words, you need to sacrifice something - either running time (take a lot of time on some inputs) or guarantees of accuracy. Similarly, if you know a problem is \( NP \)-hard and you have a dataset that is too large for an exact solution, do not be too confident in the result you obtain by using some software package to analyze your dataset. No fast method is currently guaranteed to find the correct solution to \( NP \)-hard problems on large inputs!

B.10 Method evaluation

When we evaluate methods - such as tree estimation methods, or multiple sequence alignment methods - we want to be able to quantify their error and compare them to other methods. To do this, we use a reference tree or reference alignment, and quantify the error (or accuracy) with respect to the reference. Fundamentally, this kind of evaluation comes down to evaluating the method in terms of false positives, false negatives, true positives, and true negatives.

Given this, we can relate these error and/or accuracy evaluations to basic statistical concepts, such as sensitivity (also called the true positive rate, or recall rate), specificity (also called the true negative rate), and precision.

Each of these concepts has a mathematical definition in terms of false positives (FP), true positives (TP), false negatives (FN), and true negatives (TN), that may help you remember them. First, however, we have to define what we mean by these terms – FP, TP, FN, TN.

A binary classifier is a function that maps objects to “Positive” (has a trait) or “Negative” (does not have the trait). Some of these classifications are correct and some are not correct. The classifications that are correct are either “True Positives” or “True Negatives”, depending on whether the objects were classified as Positive or Negative. Then, the classifications that are incorrect are either “False Positives” or “False Negatives”. For example, a “False Negative” is something the classifier thought was negative but really was positive.

To make this concrete, consider the case where you are trying to determine who has the flu, using a test. Everyone you test is either characterized as having the flu or not having the flu - so this is a binary classifier. The people you think have the flu who do have the flu are the true positives. The people you think have the flu but don’t have the flu are the false positives. The people you think don’t have the flu but do have the flu are the false negatives.
Algorithm design and analysis

negatives. Finally, the people you think don’t have the flu and who actually don’t have the flu are the true negatives.

The **precision** of a binary classifier is the fraction of the “positives” that are true positives: i.e., it is the ratio of TP (true positives) to TP+FP (i.e., all the ones you classified as positives). So when you say that a binary classifier has 80% precision, this means that 80% of the objects classified as having the trait actually do have the trait. Precision is also referred to as **positive predictive value** (PPV).

The **recall** of a binary classifier is the fraction of the objects that truly do have the trait that you correctly detect as having the trait. In other words, it is the ratio of TP to TP+FN. **Sensitivity** is another term for “recall”.

The **false discovery rate** (FDR) is the fraction of the objects you identify as having the trait that do not have the trait; this is the same as 1 − PPV. Think of this as the fraction of the people you think have the flu but actually don’t have the flu.

- Precision/Positive Predictive Value (PPV) = \( \frac{TP}{TP+FP} \)
- Sensitivity/Recall = \( \frac{TP}{TP+FN} \)
- Specificity = \( \frac{TN}{FP+TN} \)

**B.11 Homework problems**

For all these problems, assume that the graph has no self-loops or parallel edges.

1. Prove that the number of vertices with odd degree in a simple graph must be even.
2. Consider the binary relation \( R \) containing exactly those ordered pairs \(<x, y>\) for which \( x \) divides \( y \). Thus, \(<3, 15> \in R \) but \(<3, 8> \notin R \). Draw the Hasse Diagram for the partially ordered set defined by \( R \) on the set of integers \([1, 15]\) (i.e., \{1, 2, 3, 4, ..., 15\}).
3. Do all the counting problems in Section B.1.6. (Explain your answers.)
4. Consider the binary relation \( R \) defined for an input graph \( G \) with \( n \) vertices: \(<x, y> \in R \) if there is a path from \( x \) to \( y \) with at most 2 edges.
   1. Give examples of a graph \( G_1 \) and \( G_2 \) for which this binary relation is an equivalence relation or not an equivalence relation, respectively.
   2. What properties does this relation always hold, independent of the graph \( G \)?
   3. What happens if you allow \( G \) to be a directed graph, instead of an undirected graph?
5. Consider the binary relation \( R \) defined for an input consisting of a graph \( G \) with \( n > 1 \) vertices, and \( v_0 \) one of the vertices in \( G \): \(<x, y> \in R \) if the distance from \( v_0 \) to \( x \) is at most the distance from \( v_0 \) to \( y \). (Recall that the distance from \( a \) to \( b \) is the number of edges in the shortest path from \( a \) to \( b \).) What properties does this relation always hold, independent of the graph \( G \)?
6. Give a real world example of a partial order that is not a total order.
7. Let \( S \) be the set \{a, b, c, d\}. Give an example of a relation \( R \) on \( S \) whose transitive closure is \( R \).
8. Let $S$ be the set $\{a, b, c, d\}$. Give an example of a relation $R$ on $S$ whose transitive closure is $S \times S$.

9. Consider a simple graph $G = (V,E)$ and the binary relation $R$ on $V$ containing those pairs $<x,y>$ such that there is a path in $G$ from $x$ to $y$.
   1. Prove that $R$ is an equivalence relation.
   2. Given an example of a graph $G$ for which this equivalence relation has three equivalence classes.

10. Consider a recursively defined function $t(n)$ defined by $t(1) = 1$ and $t(n) = 3 + t(n-1)$ when $n \geq 2$. Find a closed form solution for $t(n)$ and prove it correct using induction.

11. Consider a recursively defined function $t(n)$ defined by $t(1) = t(2) = 1$ and $t(n) = t(n-1) + t(n-2)$. Prove, using induction, that $t(n)$ is $O(2^n)$.

12. Prove by contradiction that the square root of 2 is not rational.

13. Prove by contradiction that the square root of 3 is not rational.

14. Let $(0, 1)$ denote the open interval between 0 and 1 (i.e., the set $\{x : 0 < x < 1\}$). Prove by contradiction that $(0, 1)$ does not contain a smallest element.

15. Prove by contradiction that the number of functions from the set of positive integers to the set $\{0, 1\}$ is infinite.

16. Consider a simple undirected graph $G = (V,E)$ in which every vertex has even degree, and assume that $G$ is connected. Prove by induction on the number of edges in $G$ that the edges in $G$ can be ordered $e_1, e_2, \ldots, e_m$ so that $e_i$ and $e_{i+1}$ share an endpoint for $i = 1, 2, \ldots, m-1$, and $e_1$ and $e_m$ also share an endpoint. (In other words, there is a cycle that covers every edge exactly once.)

17. Prove the five statements about big-oh running times in Section B.3.2, by finding the positive constants $C, C'$.

18. Suppose you have an oracle that correctly answers Yes/No questions of the form “Does this graph have a clique of size 5?” (where you can specify the graph). Show how to use the oracle to find a maximum clique in an input graph $G$ on $n$ vertices, without calling the oracle more than $O(n)$ times.

19. Suppose you have an oracle that correctly answers Yes/No questions of the form “Does graph $G$ have a clique of size $k$?” (where you can specify graph $G$ and the value for $k$). Show how to use the oracle to find a maximum clique in an input graph $G$ on $n$ vertices, without calling the oracle more than $O(n+k)$ times.

20. Use the fact that maximum clique is NP-hard to prove that maximum independent set is NP-hard.

21. Consider the function $f$ that maps inputs for 2-colorability to inputs for 3-colorability, that operates as follows: $f(G)$ is the graph $G'$ formed by adding a vertex $v^*$ to $G$ and making it adjacent to every other vertex in $G$.
   - Prove that this function is a Karp Reduction.
   - Since 3-colorability is NP-complete, does this mean that 2-colorability is NP-complete?
   - Since 2-colorability can be solved in polynomial time, does this mean that 3-colorability can be solved in polynomial time?
22. Design a DP algorithm, and provide a running time analysis, that computes the longest increasing subsequence in an input array of $n$ integers.

23. Design a DP algorithm, and provide its running time analysis, that computes the longest common subsequence given two arrays of integers.

24. Consider the recursively defined algorithm for computing the $n^{th}$ Fibonacci number $F(n)$. Find and prove (using induction) lower and upper bounds on its running time. (Full points where both bounds are exponential.)

25. Consider the recursively defined algorithm for computing the $n^{th}$ Fibonacci number $F(n)$. Let $n = 7$. How many times is $F(4)$ computed? How many times is $F(3)$ computed?

26. Find and prove (using induction) upper and lower bounds for $t(n)$ (as a function of $n$), for $t(n)$ defined as follows:
   - $t(1) = t(2) = 2$
   - $t(n) = t(n - 1) + t(n - 2) + 1$ for $n > 2$

27. Find and prove (using induction) a closed form formula for $t(n)$ (as a function of $n$), for $t(n)$ defined as follows:
   - $t(1) = 5$
   - $t(n) = t(n - 1) + 3$ for $n > 1$

28. Find and prove (using induction) a closed form formula for $t(n)$ (as a function of $n$), for $t(n)$ defined as follows:
   - $t(1) = 2$
   - $t(n) = 5t(n - 1)$ for $n > 1$

29. Find and prove (using induction) lower and upper bounds for $t(n)$ (as a function of $n$), for $t(n)$ defined as follows:
   - $t(1) = 2$
   - $t(n) = 5t(n - 1) + n$ for $n > 1$

30. Design a DP algorithm, and provide its running time analysis, for the following problem. The input is a graph $G$ with vertices $v_1, v_2, \ldots, v_n$, and with positive weights on the edges. The output is a symmetric $n \times n$ matrix $M$ where $M[i,j]$ is the length of the shortest path from vertex $v_i$ to $v_j$. (Hint: consider solving subproblems that give the length of the shortest path from $v_i$ to $v_j$, using at most $k$ edges. Alternatively, consider solving subproblems that give the length of the shortest path from $v_i$ to $v_j$ in which the only additional vertices that are permitted are $v_1, v_2, \ldots, v_k$. Both of these approaches yield polynomial time algorithms, but have different running times.)

31. Design an $O(n)$ DP algorithm and provide the running time analysis for the following problem. The input is a rooted binary tree $T$ with $n$ leaves, $s_1, s_2, \ldots, s_n$, and with internal nodes labelled $s_{n+1}, s_{n+2}, \ldots, s_{2n-1}$. The output is an array $w[1 \ldots 2n - 1]$ where the $i^{th}$ element $w[i]$ is the number of leaves in the subtree of $T$ rooted at $s_i$. (Note that $w[i] = 1$ for $1 \leq i \leq n$.)
32. Design an $O(n^2)$ DP algorithm and provide the running time analysis for the following problem. The input is a rooted binary tree with $n$ leaves $s_1, s_2, \ldots, s_n$, and with internal nodes labelled by $s_{n+1}, s_{n+2}, \ldots, s_{2n-1}$. The output is an $n \times n$ matrix $LCA$ where $LCA(i, j) = k$ means that the least common ancestor of $s_i$ and $s_j$ is $s_k$.

33. Design an $O(n)$ DP algorithm and provide the running time analysis for the following problem. The input is a rooted binary tree $T$ with $n$ leaves $s_1, s_2, \ldots, s_n$, and with every internal node also labelled by $s_{n+1}, s_{n+2}, \ldots, s_{2n-1}$. The edges of the rooted binary tree have positive lengths. The output is an array $Longest[1 \ldots 2n - 1]$, where $Longest[i]$ is the length of the longest path from $s_i$ to a leaf in the subtree of $T$ rooted at $s_i$. Thus, $Longest[i] = 0$ for $1 \leq i \leq n$, but $Longest[i] > 0$ for all $i$ such that $n + 1 \leq i \leq 2n - 1$.

34. Consider the following two-player game. At the start of the game there are two piles of stones, and at least one pile has at least one stone. Thus, you can consider the starting point to be a pair $p, q$, where $p$ is the number of stones in the first pile, $q$ is the number of stones in the second pile, and you assume $p + q \geq 1$ and both are non-negative. The first player starts, and then they take turns, until the game ends. In each turn, the player must take a stone off of at least one pile, but cannot take more than one stone off of any pile; thus, the choice is between taking a stone off of each pile or a stone off of one pile. The game ends when the last stone is removed, and the player who took the last stone wins the game. For this problem, do the following:

1. Write a DP algorithm to determine which player wins, given input values for $p, q$. Have your DP algorithm output a $(p + 1) \times (q + 1)$ matrix $Winner$, where $Winner[i, j]$ is $T$ if the first player wins on input $i, j$, and is $F$ if the first player does not win on input $i, j$. You should assume both players play optimally.
2. Give the matrix for $Winner$ where $p = q = 5$.

35. Consider the following two-player game. At the start of the game there are three piles of stones and at least one pile has at least one stone. A player must take off at least one stone, and can take as many as two, but can only take off one stone from any one pile. Again, you need to determine who wins. The input is the number of stones on each pile, $p, q, r$, non-negative values, where at least one of $p, q, r$ is positive.

1. Write a DP algorithm to determine which player wins, given input values for $p, q, r$. Have your DP algorithm output a $(p + 1) \times (q + 1) \times (r + 1)$ matrix $Winner$, where $Winner[i, j]$ is $T$ if the first player wins on input $i, j$, and is $F$ if the first player does not win on input $i, j$. You should assume both players play optimally.
2. Give the matrix for $Winner$ where $p = q = r = 3$.

36. Design an exact algorithm for maximum clique and analyze its running time.
37. Design an exact algorithm for maximum independent set and analyze its running time.
38. Design an exact algorithm for three-colorability and analyze its running time.
39. Design an exact algorithm for Hamiltonian Path and analyze its running time.
40. We know that the optimization problem Max Clique of finding the size of the largest clique in a graph is NP-hard. Suppose that your uncle is a software developer, and has
created an algorithm for Max Clique that runs in polynomial time and claims to solve the problem optimally. What do you think is going on?

41. Suppose that you have designed a test for a disease that comes up either positive (indicating that the person has the disease) or negative (indicating that the person does not have the disease). Suppose in your population there are 1000 people, ten of them have the disease, and 990 people do not have the disease. You use your test on these 1000 people.

- Suppose all the tests come back positive. What is your true positive rate? What is your true negative rate? What is your PPV, sensitivity, and specificity?
- Suppose all tests come back negative. What is your true positive rate? What is your true negative rate? What is your PPV, sensitivity, and specificity?
- Suppose 9 of the 10 people with the disease come back positive, 10 people without the disease come back positive, and the rest come back negative. What is your true positive rate? What is your true negative rate? What is your PPV, sensitivity, and specificity?
- Suppose that 9 of the 10 people with the disease come back positive, 100 other people come back positive, and every one else comes back negative. Now suppose that you learn that Sarah has a positive test, but you don’t know whether she has the disease. What is the probability that she has the disease, based upon the information you have?
- Suppose that 9 of the 10 people with the disease come back positive, 500 other people come back positive, and every one else comes back negative. Now suppose that you learn that Sarah has a positive test, but you don’t know whether she has the disease. What is the probability that she has the disease, based upon the information you have?
Appendix C
Guidelines for writing papers about computational methods

We provide some very simple guidelines for evaluating computational methods, expressed as advice if you are doing the performance study (especially if you are presenting a new method). However, the guidelines are also relevant if you are reading a paper with a performance study, since they will indicate issues you should think about as you read the paper.

The advice here aims to get at one or more of the following basic objectives:

- **clarity of exposition**, so that both you and the reader understand what you’ve done and can draw correct inferences from the data,
- **reproducibility**, so that the experiment can be performed by someone else, using the exact same methods and data,
- **rigour**, so that what you infer makes sense, and
- **scientific relevance**, so that what you generate is relevant to some real data analysis.

The best advice is, of course, to read the scientific literature as much as you can, and note what you like and don’t like about each paper, and why. Extensive reading helps in terms of developing a good writing style, and also developing skills in designing and doing experiments that are convincing and appropriate. It’s also very good practice to read the supplementary materials of all the papers you like, because often it’s only in the supplement that you will find out the details that are the most important. For that matter, some journals make it quite difficult to provide sufficient detail, due to space limitations, and may not even make it feasible to provide supplementary materials on the journal’s website. So, as you read, develop a sense of how the different journals enable or discourage reproducibility. It may be that this will end up informing your thoughts on where you wish to publish.

The advice is given in the form of “rules”, along the lines of the PLOS (Public Library of Science) journals, which have a series of articles each with ten simple rules.

**Rule 1:** Clearly describe your new method, and make the code available so that it can be run by others. It’s important to do both - be clear about the method so that the user understands what it is doing, but also to provide the code; one without the other is better than nothing, but doing both enables the user to both understand the work and redo your experiments (and so potentially confirm your results). In particular, avoid using terms that have
multiple meanings to describe your method, and if your method relies on other software, then specify exactly how you use the other software. To fully explain your new method, you may need to provide the details in a supplementary document (especially since some journals have page limits).

**Rule 2: Evaluate your method on appropriate datasets.** The datasets you pick to evaluate your method are essential, but you need to justify the choices. If there are established benchmarks, use them (or explain why you don’t use them). If you are using simulations, make sure the simulations produce datasets with properties that are biologically relevant. You should think about what your objective is: speed, memory usage, accuracy, or something else. If your concern is speed, then make sure you include datasets that are challenging for running time, and similarly for memory usage, accuracy, or whatever criterion you are focusing on.

Try to ensure that your datasets match the empirical properties of the real datasets that are of current interest (or future interest). For example, if the objective is to enable highly accurate analyses of datasets of a certain size, make sure that your simulated datasets have this size. Also, it doesn’t make a lot of sense to focus on datasets that are either too easy (so all methods can be extremely accurate) or too hard (so no methods do well); at a minimum, you will want to make sure that your collection of datasets includes some where the best current methods have accuracy levels that are worth improving, but are not terrible.

One way to decide what level of accuracy is reasonable for the simulation is to think about what happens on real data - if the typical error rates on real data seem to be within some range (say, 5% to 20%), then reducing error from 1% to 0.5% may not be exciting to the practitioner (because the datasets are too easy), and similarly reducing error from 90% to 80% may not be exciting (because the datasets are too difficult). Always think about whether your datasets match the real datasets in ways that are convincing to the practitioner. Remember that relative performance under one model condition may not hold under another model condition!

In general, it’s best to explore many model conditions - but the trick is to not do so many model conditions that the results cannot be comprehended. Also, to fully understand the impact of the model conditions, a good practice is to divide the study into a few experiments, each changing one variable at a time; that way, you will better understand how each variable impacts accuracy.

**Rule 3: Consider statistical significance.** In a simulation study, you can generate enough datasets that you can test for the statistical significance of a difference in performance between two methods. This is important, since sometimes differences are really due to random fluctuations in performance, and you won’t know if an improvement you see for your method compared to another isn’t just the result of randomness. But, make sure you correct for multiple tests, so that you reduce the false discovery rate.
Rule 4: Don’t make a big deal out of a small difference in performance. Avoid getting excited about small differences, whether they are in favor of your method or against your method. For example, if you reduce error from 0.01% to 0.008%, it may not matter to anyone. And don’t confuse statistical significance with importance – even if the result is statistically significant, it may not matter in terms of practice.

Rule 5: Don’t avoid the cases where your method doesn’t perform well. Most methods will have some weaknesses, and being able to find out what those weaknesses are is important. Explore many datasets, varying the model conditions for your simulations or the empirical properties of your real datasets, to find out where your method performs well and where it doesn’t. Then, report all of this. You may not be comfortable doing this (no one likes to reveal weaknesses), but there are many benefits to doing this. First, you will learn essential things about your method that you won’t otherwise. Second, you will earn the respect and trust of your reviewers and readers, because (unlike some other authors), you will gain a reputation of not over-hyping your results. Third, it’s really much better if you criticize your own method, rather than having a follow-up study by your competition criticizing your method!

Rule 6: Don’t test on your training data. Many methods have algorithmic design parameters that can be modified, for example to suit different datasets. If you use datasets to set these parameters, don’t then report results based on the same datasets. Instead, use a sample of the data to set the parameters for the algorithm, and then test on other datasets!

Rule 7: Compare your methods to the best alternative methods. Comparing your method to other methods is important, but which other methods you select is critical. If your comparison is to methods that are no longer consider leading methods, then the comparison is not helpful. Again, consider your objective (speed, memory usage, or accuracy), and pick methods that are best for your objective. If you cannot use a leading method for some reason, then state why you didn’t use it, and modify your conclusions appropriately. Also, make sure you are using the current version for the method, and the best way of running the method (and, of course, provide full details for how you ran the method).

Rule 8: Make all your data available. Reproducibility is an important objective, and so making all the data you use available is key to this. Since simulated datasets can be large, many authors may prefer to simply provide commands for regenerating the data; however, software can change, and simulated datasets are not always exactly reproduced, even using the same commands. If random number generators are used, then make sure you provide the seeds you used, since otherwise the same datasets will not be generated. If you can, put your data into some public depository (with a DOI), rather than hosting them on your own machines, as too often datasets you think you have stored securely end up moved to another location, or deleted, and you won’t be able to find them. In other words, do what
you can to ensure that your data will be available in a semi-permanent way, and make the data easy to find.

Related to this, make sure that how you generated your data is completely described in your paper. This may be best accomplished by providing the commands and the software you used to generate the data.

*Rule 9: Show your results visually in a way that is most helpful for understanding trends.* How you display your results is also important. Sometimes, tables can make small differences seem big, so consider using figures instead. Show error bars, since overlapping error bars can suggest that differences are not statistically significant. Make your figures easy to understand (with informative x-axes and y-axes, and enough detail in the caption that the reader quickly understands the trends). Be careful with how you set the ranges for your x-axes and y-axes of different figures for the same questions, so that they can be compared to each other.

*Rule 10: Compare to other studies.* If you are working in an area where there is other literature, make sure you discuss the most important related papers. That comparison may include early work, but should also include the recent work on the topic. If you observe the same trends, say so; but if you find differently, then indicate this, and try to understand why there are differences. Sometimes the differences are due to different datasets with different properties, sometimes due to the choice of method, or sometimes due to how the method was run. It’s also possible that the other study made a big deal out of something small, and so it’s not that your data suggest something different from their data, but perhaps only that your conclusions are different! So, don’t just read the conclusions in the other papers – look at their experiments carefully, and decide if you agree with their conclusions. Learning to be a careful reader is important, and essential to being a good researcher.

*Discussion.* These rules are very basic, but point to the difficulties in doing rigorous work in method evaluation that is also relevant to a real application, and which can be understood. Other rules might be even more important than these, so please don’t think of these as exhaustive or more important than others you might think of.
Appendix D
Projects

Introduction

There are three types of projects in this collection: short projects, long projects, and projects that involve the development of novel methods. Each project requires data analysis, either on real or simulated data, and also writing. Therefore, even the short projects will require about a week for completion.

The main purpose of the short projects is to familiarize the student with the process of computing and interpreting alignments and trees on datasets. Because the data analysis part of these projects should be fast to complete, they are focused on relatively small nucleotide datasets. If the student has access to sufficient computational resources, then analyses of larger datasets or amino acid datasets is possible. Each short project also asks the student to explore the impact of method choice (i.e., alignment method and tree estimation method) or dataset on the resultant tree, typically using visualization tools.

The long projects build on the short projects, but do more exploration of the impact of method choice (for alignment estimation or tree estimation) or dataset on phylogeny estimation. Some of these projects examine scalability of methods to large datasets, and so will require substantial computational resources. As the student will learn, the degree to which the method selection impacts the final phylogeny can depend on the properties of the data, such as number of sequences, number of sites (i.e., sequence length), rate of evolution, percentage of missing data, etc. The use of both biological and simulated data will help the students evaluate the impact of the different factors on the final outcomes.

The projects aimed at novel method development are likely to be the most difficult, and success in these projects will probably require substantial effort beyond the period of the course. However, a student who wishes to do a novel method development project is usually best served by starting with a long project to identify the competing methods and select datasets that are best able to differentiate between methods.

Final projects for the course are typically long projects rather than novel method development projects, and are focused on comparisons of leading computational methods on simulated or biological datasets, with an eye towards assessing the relative performance of these methods, and gaining insight into the conditions that impact each method. Studies
that provide such insights can be published in bioinformatics conferences and journals, as well as in biology journals focusing on phylogenetics and systematics.

Each of these projects, including the short projects, requires the use of external software for computing alignments, computing trees, and visualizing alignments and/or trees. The long projects also require external software for bootstrapping, computing error rates of estimated trees and alignments, and comparing trees to each other. These external tools are under rapid development, and the projects should be based on the current best methods for each part of the analysis. Therefore, this list of projects does not suggest specific software to try to “beat”. Instead, the choice of method for each step should be based on the current research in the field.

Short projects

Read Appendix C, guidelines for evaluating computational methods, before starting your project. It is likely that not all the recommendations in the appendix will be specifically relevant to your study, but much of the advice is generally relevant to any scientific study concerned with methods and their performance on data.

1. Find or create a nucleotide sequence dataset for a single gene, with at most one sequence per species. Compute a multiple sequence alignment and phylogenetic tree on the dataset using any standard method. Visualize the tree. If your dataset has an outgroup, you can root the tree at that outgroup (though this may not be an accurate way of rooting the tree in some conditions); otherwise be careful not to interpret the tree as rooted. If the tree has numbers on the branches, what do they mean?

2. Find or create a small (at most 20 sequences) nucleotide sequence dataset for a single gene. Compute a multiple sequence alignment and phylogenetic tree on the dataset using two different standard methods. Visualize the trees. What differences, if any, do you see?

3. Find or create a small (at most 20 sequence) nucleotide sequence dataset for a single gene. Compute a multiple sequence alignment on the dataset using any standard method. Now compute a UPGMA tree, a maximum parsimony tree, and a maximum likelihood tree, on the alignment. (You can use any standard software packages for these calculations.) Visualize the trees. What differences, if any, do you see?

4. Find or create a small (at most 20 sequence) nucleotide sequence dataset for a single gene. Compute a multiple sequence alignment on the tree using any standard method. Now compute maximum likelihood trees on the alignment under two different models: Jukes-Cantor and GTRGAMMA. You can use any standard software package for these calculations. Visualize the trees. What differences, if any, do you see?
Projects

Long projects

Read Appendix C, guidelines for evaluating computational methods, before starting your project. It is likely that not all the advice will be specifically relevant to your study, but much of the advice is generally relevant to any scientific study concerned with methods and their performance on data.

Each of these longer projects has a stated purpose that explores the impact of method choice or dataset property on the final alignment and/or tree. For each of these longer projects, you should write a paper describing what you did, what you learned, etc. Your paper should provide enough detail to be reproducible (e.g., software version numbers and commands, access to datasets), and should have some interesting discussion about what you observed. If the project suggests hypotheses, describe them, and consider what you might do to settle the hypotheses.

1. **The purpose of this project is to explore the choice of method on gene tree estimation.** Compare gene trees computed on a biological dataset with at least 50 unaligned sequences using at least two different techniques. You can use your own dataset or find a published dataset.
   - If you wish, you can use an “alignment-free” method (of your choice), in addition to a method that either co-estimates alignments and trees or a two-phase method (i.e., one that first estimates an alignment and then computes a tree on the alignment). Read the literature to identify the most promising methods.
   - Get bootstrap support on the branches of the tree you compute.
   - Compare the gene trees, taking bootstrap support into account. Where are they different? Are these differences interesting or important? What is your interpretation of these differences? If one method did particularly poorly, was there something about the data that was difficult for the method? What did you learn about the methods you used?

2. **The objective of this project is to evaluate the impact of species tree estimation method on the estimated species tree for multi-locus datasets with gene tree heterogeneity.** Compare species trees computed on a biological dataset with at least 10 genes and between 10 and 100 species. It would be most interesting if you pick a dataset where gene tree heterogeneity has been observed or where it is expected. You can use your own dataset or find a published dataset.
   - Compute gene sequence alignments and gene trees using reasonable methods. (If you are using a dataset from a published study, these may already be computed for you!)
   - Compute species trees using at least two coalescent-based methods and one concatenation analysis. Unless you have access to substantial computational resources, try to select reasonably fast methods so that they each complete on your dataset with 24 hours and do not have high memory requirements.
   - Compare the species trees that you obtain using different species tree estimation methods. Where are they different? Are these differences interesting or important?
What is your interpretation of these differences? What does this tell you about the methods you used?

- Modify your input somehow (delete rogue taxa if any, delete gene trees that have poor bootstrap support, collapse low support branches in gene trees) and then re-estimate the species trees. What differences do you see? Are these differences interesting or important? What is your interpretation of these differences?

3. **The purpose of this project is to explore the impact of missing data on gene tree estimation.** Find or create a small nucleotide sequence dataset for a single gene; call this $M$. Compute a multiple sequence alignment on $M$ using any standard method, and call this alignment $A$. Pick an arbitrary sequence $x$ in the dataset, and delete the first 50% of the nucleotides in $x$. Now re-compute the alignment on this new dataset using the same method you used to produce $A$, and call this alignment $A'$. Compute trees using any preferred method. Are the trees the same? If you remove the leaves for $x$ from each tree, are the trees the same? If not, how are they different? Vary the experiment to explore the impact of method choice in the presence of missing data, for example:

- Vary the alignment estimation method
- Vary the tree estimation method
- Vary the experiment by removing more nucleotides (varying from 50% to 90%) from $x$
- Vary the experiment by modifying more sequences (up to 50% of the original dataset)

4. **The objective of this study is to evaluate the impact of random sequences in a dataset.** Find a small (at most 20 sequences) nucleotide sequence dataset for a single gene; call this dataset $M$. Pick a sequence $x$ in $M$, and replace the DNA sequence by a random nucleotide sequence of the same length; call this new dataset $M'$. Construct trees on $M$ and $M'$ using the same protocol (e.g., if you compute a Muscle alignment on $M$ and then run RAxML to estimate the ML tree, both in default mode, then do exactly the same for $M'$). Are the trees the same? How do they differ? Remove $x$ from both trees. Do the trees (without $x$) look the same?

5. **The objective of this study is to evaluate techniques for distance-correction when datasets are “saturated”.** When analyzing sequence datasets using distance-based methods, such as Neighbor Joining, a corrected distance-matrix must be computed. Recall that the calculation for Jukes-Cantor distances implicitly assumes that all pairs of sequences $s_i$ and $s_j$ have normalized Hamming distances that is strictly less than 75%. Yet, for fast evolving sequences, or for sequences spanning large evolutionary distances, this may not be the case. In fact, a dataset is said to be *saturated* when at least one pair of sequences has normalized Hamming distance that matches or exceeds the expected value for a random pair of sequences (which is 75% for DNA sequences). The question here is how to handle such datasets, and in particular how to correct distances so that phylogenies computed on these corrected distance matrices are as accurate as they can be. To do this project, you should find out how this situation is treated in general, and then think about whether you can handle it better. Evaluate multiple ways of correcting the
6. The objective of this study is to evaluate the impact of “masking” sites within multiple sequence alignments on phylogeny estimation. Several techniques have been developed to identify sites that are noisy, and perhaps have substantial error, within multiple sequence alignments. After these sites are identified, they can be deleted from the alignment, thus producing a new alignment that contains a subset of the sites from the original alignment; this is called “masking”. Early studies suggested that masking alignments would lead to improved phylogeny estimation, but were limited to very long alignments and small numbers of sequences. Evaluate the impact of masking multiple sequence alignments on phylogeny estimation, using larger numbers of taxa and/or single gene datasets.

7. The objective of this study is to evaluate the impact of substitution model on phylogeny estimation. Likelihood-based phylogeny estimation is performed under a substitution model that must be estimated from the data; yet how much this model impacts the resultant phylogeny in terms of topology, branch lengths, etc., is unknown. Evaluate the impact of choice of model on the accuracy of these parameters using a combination of simulated and biological data. Use both nucleotide and amino acid datasets, since the problems are subtly different.

8. The objective of this study is to compare heuristics for maximum likelihood estimation. Maximum likelihood (ML) phylogeny estimation is an NP-hard problem for all the standard models of sequence evolution, such as Jukes-Cantor and GTR. The methods that are used to estimate ML trees use heuristics, largely based on a combination of hill-climbing and randomization, to find locally optimal solutions. How well the various heuristics for maximum likelihood solve the optimization problem can impact the accuracy of the parameters they are estimating (e.g., gene tree topology, branch lengths, and substitution matrix) but also impacts the running time. While there has been some benchmarking of ML heuristics (e.g., see Liu et al. (2012)), little is known about their performance (accuracy, running time, and peak memory usage) when the datasets are allowed to be quite large (i.e., most comparisons have been restricted to single gene datasets with at most a few hundred sequences). In addition, relatively little is known about how the properties of the multiple sequence alignment (such as percentage of the alignment that is occupied by gaps) impacts the performance of the different methods. Finally, some methods can be run in various ways, for example, using TBR moves or only using NNI moves, using multiple starting points or only using one starting point. Using both simulated and biological datasets, explore these questions on the current leading ML methods for large datasets.

9. The objective of this study is to compare Bayesian methods to maximum likelihood methods for phylogeny estimation. Bayesian and maximum likelihood phylogeny estimation methods are both likelihood based and highly popular, yet little is known about the relative performance of these methods with respect to accuracy and computational requirements. Evaluate this on a collection of biological and simulated datasets. Note
that since Bayesian methods produce a distribution on tree space rather than a point estimate of the tree, to use Bayesian methods to produce a point estimate you would need to summarize the distribution in some way.

10. **The objective of this study is to explore the impact of model misspecification on gene tree estimation.** When using likelihood-based methods to estimate a gene tree, the usual assumption is that the entire tree is under one set of model parameters (e.g., there is one GTR substitution matrix that governs all the branches of the model tree). While some methods can estimate trees under models in which this assumption is violated, most cannot. Explore the impact of violations of this assumption on phylogeny estimation using a combination of simulated and biological datasets. If you have time, see if you can develop an alternative technique to handle such datasets.

11. **The objective of this study is to explore the impact of using inputs from Bayesian gene tree methods instead of maximum likelihood on coalescent-based species tree methods that combine gene trees (i.e., summary methods).** Many methods have been developed to estimate the species tree from collections of gene trees; these are called “summary methods”. Examples of summary methods that are statistically consistent in the presence of gene tree estimation error due to incomplete lineage sorting (ILS) include ASTRAL, ASTRID, BUCKy-pop (the population tree produced by BUCKy), MP-EST, STAR, and STEM. BUCKy-pop was designed to work with inputs computed using Bayesian gene tree estimators, so that each gene was represented by a collection of gene trees produced by a Bayesian MCMC analysis; the other methods have only been studied when used with a single gene tree, typically computed using maximum likelihood heuristics. Examine the impact of using a distribution of gene trees computed by a Bayesian MCMC analysis instead of a single ML tree on the point estimate produced by these summary methods for species tree estimation. Also consider how to use the distribution of gene trees for each gene in order to produce branch support values.

12. **The objective of this study is to explore statistical coalescent-based species tree estimation methods on datasets with very few loci.** Statistical methods for species tree estimation from multiple loci can have good accuracy when there is a large number of loci, but little is known about how well they perform under conditions where the number of loci is small and there is substantial gene tree heterogeneity. Explore this problem using a combination of simulated and biological datasets.

13. **The objective of this study is to explore statistical coalescent-based species tree estimation methods on datasets where each locus has very few sites.** Most studies of coalescent-based methods have been performed on datasets where each locus has a sufficient number of sites that the gene trees have reasonable, even if imperfect, accuracy. Yet some researchers have argued that analyses of coalescent-based methods should be based on very short genomic regions, in order to avoid intra-locus recombination, which violates the assumptions of the MSC (multi-species coalescent) model. Explore the impact of short loci on species tree estimation produced using different coalescent-based methods. See Chou et al. (2015) for an example of an early study of this issue.
14. **The objective of this study is to evaluate the impact of restricting coalescent-based species tree estimation to small numbers of loci.** Some coalescent-based species tree methods are very computationally intensive and so in practice are limited to 50 loci, and even analyses with 25 loci can take days for convergence statistics to be acceptable. Some studies have suggested a protocol where a random small subset of the loci is selected, and then the tree is estimated based on that subset. Evaluate the impact of this strategy on a collection of biological and simulated datasets. Does the impact depend on the ILS level? How much variance is there between analyses of different random subsets? Are there principled ways of selecting the subset of loci to analyze that improve accuracy?

**Projects involving novel method development**

Read Appendix C, guidelines for evaluating computational methods, before starting your project. It is likely that not all the recommendations there will be specifically relevant to your study, but much of the advice is generally relevant to any scientific study concerned with methods and their performance on data. The work you do for this project could lead to a publication; document everything you do so that it is reproducible, and save your data so that you can enable others to verify your results.

Good luck!

**Multiple sequence alignment and/or gene tree estimation**

1. **The objective of this study is to design a new heuristic for maximum likelihood gene tree estimation so it can run more effectively than current methods on datasets with thousands of sequences.** The estimation of maximum likelihood gene trees from multiple sequence alignments containing thousands of sequences is computationally very intensive. Develop a new heuristic, and compare it to the current heuristics methods for maximum likelihood. Note the computational effort (e.g., running time and peak memory usage) and likelihood scores. If you use simulated data, then also record the topological accuracy of the trees each method produces.

2. **The objective of this study is to design divide-and-conquer methods to scale Bayesian co-estimation of multiple sequence alignment and gene trees to larger datasets.** Bayesian methods for co-estimation of multiple sequence alignments and gene trees are computationally intensive, and limited to relatively small numbers of sequences. Select one such method (e.g., BAli-Phy), and evaluate the impact of dataset size (number of sequences and also the average or maximum sequence length) on the running time, peak memory usage, and accuracy of the alignments and trees that are computed. Then design a method, perhaps using divide-and-conquer, that enables the selected method to scale to larger datasets.

3. **The objective of this study is to design methods to detect non-homologous sequences**
Projects

in a dataset. Multiple sequence alignment and phylogeny estimation methods assume that all the sequences in the input are homologous, which means that they share a common ancestor. Little is understood about the impact of the inclusion of non-homologous sequences in phylogenetic datasets. Evaluate the consequence of including non-homologous sequences in input datasets on the resultant phylogenies, and also develop a method for detecting the non-homologs so that they can be removed from the dataset.

Species tree estimation

1. The objective of this study is to design better methods for quartet tree amalgamation. Many methods (e.g., BUCKy-pop, ASTRAL, and SVDquartets) for coalescent-based species tree estimation operate by estimating quartet trees (perhaps with weights on each quartet tree) and then combine the estimated quartet trees together. A standard optimization problem for this purpose is to find a tree whose total (weighted) quartet support is maximal. Develop a heuristic for this optimization problem and test it within a coalescent-based species tree estimation pipeline.

2. The objective of this study is to design divide-and-conquer methods to scale Bayesian co-estimation of gene trees and species trees to larger datasets. Bayesian methods for co-estimation of gene trees and species trees are computationally intensive; for example, the coalescent-based method *BEAST Heled and Drummond (2010) is typically limited to at most 20 species and 50 genes. Other methods (e.g., MP-EST) can scale to large numbers of loci but become computationally intensive as the number of taxa increases. Design methods using divide-and-conquer that enable these methods to scale to larger datasets.
Appendix E
Glossary

**GTR:** General Time Reversible model.

**JC69:** Jukes-Cantor model

**K2P:** Kimura 2-parameter model

**MRCA:** Most recent common ancestor

**MSC:** Multi-species Coalescent model

**NP:** The class of decision problems (yes/no) for which the yes-instances can be verified in polynomial time

**NP-hard:** A problem that is at least as difficult as the hardest problems in the class NP

**NP-complete:** A decision problem that is NP-hard and also in the class NP

**polytomy:** A node in an unrooted tree of degree greater than three, or a node in a rooted tree with more than two children
Appendix F
Notation

- $ab|cd$: quartet tree on leaf set $a, b, c, d$ with one internal edge separating $a, b$ from $c, d$
- $(a, (b, c))$: rooted tree on three leaves $a, b, c$ in which $b$ and $c$ are siblings
- Clades($T$): the set of clades of a rooted tree $T$
- $C(T)$: the set of bipartitions on the leaf set induced by edge deletions in a tree $T$
- $C_I(T)$: the set of non-trivial bipartitions on the leaf set induced by deletions of internal edges in a tree $T$
- $L(T)$: the set of leaves of a tree $T$
- $MP(T, M)$: The maximum parsimony score of a tree $T$ given the character matrix $M$
- $Q(T)$: the set of unrooted quartet trees induced on the leaf set of a tree $T$
- $Q_r(T)$: the set of unrooted binary quartet trees in $Q(T)$
- $|S|$: the number of elements in the set $S$
- $S\setminus S'$: the set $\{x : x \in S$ and $x \notin S'\}$ (i.e., the elements of $S$ that are not in $S'$)
- $T|X$: the subtree of $T$ induced on leaf set $X$, with nodes of degree two suppressed
- $T_u$: when $T$ is rooted, this is the unrooted tree obtained by ignoring the location of the root for $T$
References


Bastkowski, Sarah, Moulton, Vincent, Spillner, Andreas, and Wu, Taoyang. 2016. The


References


References


References


Steel, M.A. 1994b. The maximum likelihood point for a phylogenetic tree is not unique. Systematic Biology, 43(4), 560–564.


References


absolute fast converging methods, 26, 102, 221
additive matrix, 16, 17, 19, 89, 90, 148, 212
topology invariant neighborhood, 96
Agarwala et al. method, 147, 148
agreement subtrees
  maximum agreement subtree, 114
maximum compatibility subtree, 114, 115
Aho, Sagiv, Szymanski, and Ullman algorithm, 133
alignment-free phylogeny estimation, 215
All Quartets Method, 56, 57
ancestral sequence estimation, 156
Bayesian MCMC tree estimation, 151, 207, 208
  computational challenges, 22, 151, 152, 207, 208
  statistical consistency, 152
binary encoding of multi-state characters, 233
binary tree, 11, 25
bipartition compatibility, 43
bipartition encoding of a tree, 25
  non-trivial bipartitions, 113
branch length estimation, 156
branch support calculation, 3, 155
  Bayesian support values, 156
  bootstrapping, 3, 155
Cavender-Farris-Neyman distance correction, 19
character data, 11
Chris Tuffley, 153
clad compatibility, 37, 38
consensus trees, 4, 108, 109
  asymmetric median tree, 109, 111, 114, 217
  compatibility tree, 109–111
  extended majority consensus, 110
  greedy consensus tree, 109, 110, 114
  majority consensus tree, 109, 114
  strict consensus tree, 46, 109, 114
DACTAL, 27, 202, 203, 215, 216
DCM1-NJ, 25
disk-covering methods (DCMs), 26, 209–214, 216, 217, 219–221
  DCM-Buneman, 26, 217
  DCM1-NJ, 26, 217
short subtree graph, 211–213
  threshold graph, 211–213, 217, 218, 220
dissimilarity matrix, 17
distance matrix, 17
distance-based tree estimation, 86, 87, 95, 96, 205,
  208
  absolute fast converging methods, 102
  Agarwala et al. 3-approximation algorithm, 100,
  217
branch length estimation, 97
Buneman Tree, 217
cornergence rate, 102
DCM-Buneman, 217
DCM1-NJ, 26
error tolerance, 92
FastME, 21, 101
Four Point Condition, 17
Four Point Method, 17, 18, 91, 92
  how to prove statistical consistency, 96, 97
  impact of missing data, 102
minimum evolution, 98
Naive Quartet Method, 18, 21, 23, 24, 27, 92, 93,
  99, 148, 152, 154
neighbor joining, 21, 23, 24, 27, 94, 95, 101, 219
  optimization problems, 97, 100
safety radius, 93, 99
  statistical consistency, 102
UPGMA, 14, 21, 24, 27, 88
Dobzhansky, 1
dyadic closure method, 57
estimating dates, 158
evolutionary diameter, 26, 219, 220
false negative error rate, 23
false negative rate, 46
false positive error rate, 23
false positive rate, 46
FastME, 101
Felsenstein zone tree, 152
Fitch-Hartigan algorithm, 68, 69
Four Point Condition, 17, 90
Four Point Method, 17, 18, 91
gene order phylogeny
software, 221

glottochronology, 232

Gray and Atkinson method for phylogeny estimation

binary encoding, 233

Hasse Diagram, 36–38, 44

Hasse Diagram algorithm, 37, 38, 43

historical linguistics, 1

homoplasy, 157, 230–232

incomplete lineage sorting, 129, 191, 212

Indo-European, 231

ingroup taxa, 40

insertions and deletions (indels), 101, 163

Joe Felsenstein, 152

lexical clock, 232

Markov Chain Monte Carlo techniques, 151

matrix representation with distances, 134

maximum compatibility, 232

maximum compatibility tree estimation, 152

compatibility informative sites, 153

positively misleading, 153

maximum likelihood

NP-hard, 150

maximum likelihood tree estimation, 149, 150, 208, 219, 220

heuristics, 151, 207

NP-hardness, 21, 151

software, 133, 208, 215

tree score calculation, 207

maximum parsimony, 232

maximum parsimony tree estimation, 21, 24, 27, 124, 152, 208, 219, 220

Fitch-Hartigan algorithm, 219

heuristics, 21, 207

NP-hardness, 21

parsimony informative sites, 153

positively misleading, 152, 153

statistical inconsistency under standard models, 21

tree score calculation, 207

Maximum Triplet Support problem, 133

Mike Seel, 153

minimum spanning tree, 175

missing data, 163

molecular sequences, 3, 20, 65

multi-species coalescent model, 191–201

multiple sequence alignment, 1, 3, 27, 163, 219

alignment accuracy measurement, 167

alignment error measurement, 165, 167

impact on tree estimation, 4

optimization problems, 172

positional homology, 164

software, 215

Sum-of-Pairs Alignment, 172

Tree Alignment, 172, 174, 175

Naive Quartet Method, 92, 93, 148, 152, 154

neighbor joining, 94, 95, 101

NP-hard problems, 3

heuristic search strategies, 3, 4, 206, 207

outgroup taxa, 39, 40, 157

pairwise sequence alignment, 165, 168

edit distances, 167

Needleman-Wunsch, 168

Smith-Waterman, 172

PASTA, 27

Peter Buneman, 90

phylogenetic networks, 1, 4, 191

phylogenies, 1, 32

phylogenies of languages, 6

phylogenomic analyses, 207

polytomy, 129, 132

profile Hidden Markov Model, 178

profile Hidden Markov Models, 176–180

quartet trees, 56

All Quartets Method, 57, 92

quartet amalgamation methods, 125, 133, 205

quartet compatibility, 57

quartet tree compatibility problem, 59

Robinson-Foulds distance, 23, 46

Robinson-Foulds tree error rate, 23, 25, 26, 28, 46, 47

rooted tree compatibility problem, 60

rooted triplet trees, 55

Aho, Sagiv, Szymanski, and Ullman algorithm, 55, 56

rooting trees, 157

Sankoff algorithm for fixed-tree maximum parsimony, 70–72

Sankoff algorithm for fixed-tree weighted maximum
parsimony, 70

SAFe, 27

sequence evolution models, 143

i.i.d. assumption, 20, 149

amino acid models, 145

Cavender-Farris-Neyman (CFN), 5, 11–13, 16–18,
20, 21, 95, 125, 143, 148–150

computing the probability of a sequence dataset, 149

General Markov Model, 145, 163

Generalized Time Reversible (GTR) Model, 22, 145, 149, 150, 163, 207

heterotachy, 153

Jukes-Cantor Model, 22, 144, 145, 149, 150, 163

Kimura 2-parameter (K2P) model, 145

No Common Mechanism Model, 153, 154

rates-across-sites, 20

Thorne-Kishino-Felsenstein 1991 (TKF91), 101

time-reversible, 149, 150

short- quartets, 57

simulation study, 2, 22–25, 208

false negative branches/edges, 22, 25

false positive branches/edges, 22, 25

species tree estimation
Index

anomaly zone, 193, 194
coalescent-based methods, 6, 27, 129, 135, 191, 195, 196, 200, 208, 212, 221
canonical analysis, 3, 194, 195
gene duplication and loss, 135
gene tree discordance, 3
SRSTE method, 196
Split-Constrained Quartet Support Problem, 126, 127, 129
star tree, 47, 114
statistical consistency, 5, 19, 20, 22, 27, 146, 154
statistical identifiability, 146
strict molecular clock, 13–15, 19, 88, 232
SuperFine, 131–133
supertree methods, 109, 119, 209, 215
compatibility supertree, 59, 60, 120
Matrix Representation with Likelihood (MRL), 125
missing data treatment, 125
Matrix Representation with Parsimony (MRP), 123, 124
missing data treatment, 124
MRP matrix, 123
Maximum Likelihood Supertree, 135
MinCut Supertree, 120, 133
MinFlip Supertree, 120
Modified MinCut Supertree, 120
MRF Supertree, 120
profile of trees, 120
Quartets MaxCut, 133
Robinson-Foulds Supertrees, 121, 135
statistical aspects of, 129, 134
Strict Consensus Merger (SCM), 129, 131–133
SuperFine, 131–133
use within DCMs, 209
tree compatibility, 110
tree error measurement, 25
tree estimation error, 22
trees, 32
binary vs. multifurcating, 32, 33, 43
bipartitions of unrooted trees, 41, 42
clade representations of rooted trees, 35, 36
comparing two unrooted trees using bipartitions, 43
constructing a rooted tree from its clades, 36, 37
constructing an unrooted tree from its bipartition set, 43, 44
determining if a set of bipartitions is compatible, 44
different graphical ways of representing trees, 33
edges, 32
fully resolved trees, 43
Newick notation for rooted trees, 33–35
Newick notation for unrooted multifurcating trees, 42
Newick notation for unrooted trees, 40, 41
non-trivial bipartitions of unrooted trees, 42
polytomy, 33
rooted vs. unrooted, 32
tree refinement, 43, 110, 111
vertices, 32
triangle inequality, 17, 19, 87
triangulated graphs, 26, 210–213, 218–220
decompositions of, 211
definition, 210
perfect elimination scheme, 211
short subtree graph, 211–213
threshold graph, 211, 212, 217, 218, 220
triplet trees
Aho, Sagiv, Szymanski, and Ullman algorithm, 133
triplet tree compatibility problem, 60
two-phase phylogeny estimation, 164, 215, 216
ultrametric matrix, 13, 88, 89
Unrooted Tree Compatibility Problem, 131
Unrooted Tree Compatibility problem, 60
UPGMA, 21, 88, 89, 232
statistical inconsistency of, 89
weighted maximum compatibility, 232