Computational Phylogenetics

Tandy Warnow, The University of Illinois at Urbana-Champaign
## Contents

**Preface**  
9

### 1 Brief introduction to phylogenetic estimation  
13

1.1 The Cavender-Farris-Neyman model  
13

1.2 An analogy: determining whether a coin is biased towards heads or tails  
13

1.3 Estimating the Cavender-Farris-Neyman tree  
14

1.3.1 Estimating the CFN tree when evolution is clock-like  
14

1.3.2 Estimating the unrooted CFN tree when evolution is not clock-like  
16

1.3.3 Computing unrooted CFN trees from dissimilarity matrices  
17

1.4 The Naive Quartet Method  
18

1.5 Some comments about the CFN model  
19

1.6 Phylogeny estimation methods used in practice  
19

1.7 Measuring error rates on simulated datasets  
20

1.8 Performance of methods on data  
21

1.9 Summary  
23

1.10 Review questions  
24

1.11 Homework problems  
25

### 2 Trees  
27

2.1 Rooted trees  
27

2.1.1 Newick notation for rooted trees  
27

2.1.2 The clade representation of a rooted tree  
30

2.1.3 Constructing a rooted tree from its set of clades  
30

2.1.4 Compatible sets of clades  
31

2.1.5 Difficulties in rooting trees  
32

2.2 Unrooted trees  
32

2.2.1 Newick formats for unrooted trees  
34

2.2.2 The bipartitions of an unrooted tree  
34

2.2.3 Representing non-binary trees  
36

2.2.4 Comparing trees using their bipartitions  
36

2.2.5 Constructing $T$ from $C(T)$  
36

2.2.6 Testing compatibility of a set of bipartitions  
38

2.3 Quantifying error in estimated trees  
39

2.4 The number of binary trees on $n$ leaves  
39

2.5 Rogue taxa  
40

2.6 Induced subtrees  
40

2.7 Some special trees  
41

2.8 Review questions  
41

2.9 Homework problems  
41
### 6.2.3 Greedy consensus ................................. 84
### 6.2.4 Compatibility trees ............................... 85

### 6.3 Supertrees ........................................... 88
   6.3.1 Why supertree construction is important .......... 88
   6.3.2 Matrix Representation with Parsimony (MRP) .... 89
   6.3.3 Matrix Representation with Likelihood (MRL) ...... 89
   6.3.4 Robinson-Foulds Supertrees ....................... 90
   6.3.5 Maximum Quartet Support Supertrees ............... 90
   6.3.6 SuperFine ...................................... 90
   6.3.7 Popular supertree methods ....................... 91

### 6.4 Agreement subtrees .................................. 91
   6.4.1 Maximum Agreement Subtree ....................... 91
   6.4.2 Maximum Compatibility Subtree .................... 91

### 6.5 Exploring differences among trees ................. 92

### 6.6 Further reading ..................................... 92

### 6.7 Review questions ................................... 92

### 6.8 Homework problems .................................. 93

### 7 Statistical gene tree estimation methods .......... 95
   7.1 Introduction to statistical estimation in phylogenetics 95
   7.2 Models of site evolution .............................. 95
   7.3 Statistical identifiability and consistency .......... 97
   7.4 Calculating the probability of a set of sequences ... 97
   7.5 Maximum Likelihood ................................ 98
   7.6 Bayesian methods ................................... 99
   7.7 Statistical properties of Maximum Parsimony and Maximum Compatibility 99
   7.8 The No Common Mechanism Model ..................... 100
   7.9 Beyond statistical consistency: performance on data 100
   7.10 Estimating branch support ........................... 101
   7.11 Other gene tree estimation problems ............... 102
   7.12 Review questions ................................... 103
   7.13 Homework problems .................................. 104

### 8 Multiple sequence alignment ......................... 109
   8.1 Evolutionary history and sequence alignment ....... 109
   8.2 Edit distances and how to compute them ........... 111
      8.2.1 Needleman-Wunsch ............................. 112
      8.2.2 Smith-Waterman ............................... 115
   8.3 Optimization problems for multiple sequence alignment 115
      8.3.1 Tree Alignment ................................ 116
      8.3.2 Generalized Tree Alignment ..................... 116
   8.4 Profile Hidden Markov Models ....................... 117
      8.4.1 Profiles .................................... 117
      8.4.2 Gap-free Hidden Markov Models ................ 117
      8.4.3 Profile Hidden Markov Models with insertion and deletion states 118
      8.4.4 Probability calculations on profile HMMs ....... 119
   8.5 Popular methods for multiple sequence alignment 119
   8.6 Quantifying error in estimated multiple sequence alignments .............................................. 119
   8.7 Divide-and-conquer alignment strategies .......... 121
   8.8 The impact of multiple sequence alignment on tree estimation ............................................ 121
   8.9 Co-estimation of alignments and trees ............ 121
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 microorganisms, 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” (55).

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.

Linguistics is another discipline where methods are used to address the understanding of how languages evolved, and which has multiple applications (e.g., understanding the properties of ancestral languages, human migrations, etc.). Because languages evolve under different processes than those that operate on biomolecular sequences, methods 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 computer science, applied mathematics, or statistics) to be able to contribute new methods for phylogeny estimation, and in particular for large heterogeneous datasets that are characteristic of the types of inputs that are increasingly of interest in practice. Thus, this text emphasizes high-level algorithmic design (especially, but not exclusively, divide-and-conquer strategies) that enable 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 textbook 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 methods 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 performance 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 theoretical 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, discrete 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 linguistics) 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 (98; 99; 117), and that errors in estimated gene trees results in errors in estimated species trees (14; 13; 73; 103). Furthermore, research has also shown that scientific questions, such as the detection of positive selection, can be misled by errors in alignments (67).

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 6 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 techniques for exploring sets of trees for various purposes, including supertree estimation and determining support for different phylogenetic hypotheses.

Chapters 7 through 12 introduce the statistical foundations of phylogenetic estimation, and its application to both biological and linguistic data. Chapter 7 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 8.

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 9.

Since nearly all good approaches to phylogeny estimation are based on attempts to solve NP-hard problems, heuristics are used instead of exact algorithms. Chapter 10 presents some standard heuristics, as well as some new heuristic approaches, and discusses the impact of these design choices on accuracy and running time.

Chapter 11 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 Chapter 12.

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 three appendices. The first 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 second appendix provides guidelines about how to evaluate computational methods in general (i.e., not just for phylogenetic estimation methods), which can be useful for method developers who wish to test their methods. The third 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 (e.g., (166; 45; 39; 12)) for my Computational Phylogenetics courses have grown into journal publications that have influenced the computational phylogenetics research community and provided new methods for phylogenetic analysis.
Chapter 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 are 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 \(T\). 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”.
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.

\[ a \quad b \quad c \quad d \quad e \]

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.

**Definition 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 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, to reconstruct a tree with \(n\) leaves, once you determine that two species \(A\) and \(B\) are siblings, then remove \(A\) and reconstruct the tree on what remains; finally, add \(A\) into the tree on the \(n - 1\) species by making it a sibling to \(B\). This approach, which is a variant of the UPGMA (138) method, is easily seen to converge to the true tree 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 \(C\) is 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, UPGMA would converge to the wrong tree as the sequence length increases. This is clearly an undesirable property of a phylogeny estimation method!

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.
Figure 1.2: Constructing evolutionary trees when evolution is not clocklike. We show a cartoon of a model tree on five leaves, with branch lengths drawn proportionally to the expected number of changes of a random site. Note that leaves b and c are not siblings, but have the smallest evolutionary distance (measured in terms of expected number of changes). Hence, methods that make the most similar sequences siblings will likely fail to reconstruct the model tree, and the probability of failure will increase to 1 as the number of sites increases.

– 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 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 $\lambda_{ij}$ of path distances in the tree. Without loss of generality, assume the tree $T$ has an internal edge $e_I$ that separates $s_1$ and $s_2$ from $s_3$ and $s_4$. Note that $\lambda(e_I) > 0$, under the CFN model. Now, consider the three following pairwise sums:

- $\lambda_{1,2} + \lambda_{3,4}$
- $\lambda_{1,3} + \lambda_{2,4}$
- $\lambda_{1,4} + \lambda_{2,3}$

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 $T$. 

17
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

We call this approach to constructing a tree from a matrix of pairwise distances the Naive Quartet Method, and note that it has the following structure:

- Compute a tree on every four leaves using the Four Point Method
- Construct a tree that agrees with all the quartet trees if it exists, and otherwise return fail.

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 \( \lambda \) 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 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 nucleotide sequences (which are over a four-letter alphabet) or amino acid sequences (which are over a 20-letter alphabet). Statistical models of nucleotide and amino acid sequence evolution (discussed in Chapter 7) 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”.

### 1.6 Phylogeny estimation methods used in practice

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 (68) 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 (62).

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 neighbor joining (131), maximum likelihood (115;
63), and Bayesian estimations of phylogenetic trees (82). The proofs of statistical consistency for these methods tend
to be mathematically technical, and are omitted.

However, as we discussed earlier, 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. There are in fact many statistically consistent polynomial
time methods, of which Neighbor Joining (131) is perhaps the most well known and frequently used.

### 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.

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 (88) and Generalized Time Reversible (127) model, 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. Figure 2.3 gives a description of this process.

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 branches (or false negatives, for short), and the bipartitions that are present in the estimated tree but not the model tree are referred to as false positive branches (or false positives). See Figure 2.4 for an example of this comparison. See Appendix B for more about how to evaluate methods well.

1.8 Performance of methods on data

Our discussion has introduced the basic theoretical framework for phylogeny estimation, including statistical models of sequence evolution and some simple methods for estimating trees under these models. We have also noted that the Naive Quartet Method and Neighbor Joining are statistically consistent methods for estimating the true unrooted tree under standard stochastic models of evolution. In contrast, we showed that UPGMA and maximum parsimony are not statistically consistent under the same stochastic models, and will even 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 (83; 80). 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
Quantifying Error

Figure 1.4: **How tree estimation error is calculated.** Note that the true tree is rooted and the inferred tree is unrooted; the error calculation is based on bipartitions induced by the 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 the sum of the number of internal edges in the two trees, is the Robinson-Foulds (RF) error rate. When both trees are binary, the FN, FP, and RF rates are identical.
5. 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 even more realistic 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 (109). 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 K2P+Gamma model trees, and distances were corrected under the K2P model. Note that maximum parsimony (MP) is more accurate than neighbor joining (NJ) on these data.

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 computed using Maximum Parsimony heuristics are substantially more accurate than trees computed using Neighbor Joining. Figure 2.5 presents a reproduction of Figure 5(b) from (109), which shows model conditions where maximum parsimony analyses can produce substantially more accurate trees than Neighbor Joining.

In this figure, four methods are compared on simulated datasets with 400 sequences, where the simulation under the K2P+Gamma model of DNA sequence evolution. The four methods include Neighbor Joining (NJ), a heuristic for Maximum Parsimony (MP), and two other methods (Weighbor (20) and DCM-NJ+MP (110), both of which will be described in Chapter 11). The y-axis is tree error, computed using the Robinson-Foulds rate (i.e., the proportion of the true bipartitions in the model tree that do not appear in the estimated tree). The x-axis is the “diameter”, and refers to the expected number of changes across the model tree. Note that Neighbor joining is less accurate than Maximum Parsimony under many of the model conditions shown here, but especially when the model tree diameter is large.

1.9 Summary

The discussion here has focused on just a few 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 7, 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 8 addresses phylogeny estimation and multiple sequence alignment when sequences evolve also with insertions and deletions. Chapter 9 discusses species tree estimation, under genome-scale evolution models in which gene trees evolve within species trees. Chapter 12 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 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 11), 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 method based just on theory. In other words, the performance of phylogenetic estimation methods is a good example of a more general phenomenon where in theory, there is no difference between theory and practice, but in practice there is. The gap between theory and practice is one of the major themes in this text, and which is one of the reasons that phylogenetic method development and evaluation is such an interesting research area.

The challenge to the algorithm developer is to develop methods that have outstanding performance on data and that also have the desirable theoretical guarantees of being statistically consistent and not requiring excessive amounts of data to return the true tree with high probability. Developing the theoretical framework to design methods with strong guarantees, the empirical framework to evaluate methods on data and determine the conditions in which they perform well or poorly, and algorithm design strategies (including divide-and-conquer) that can enable highly accurate methods to scale to large datasets, are the goals of the remaining chapters of this text.

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?

---

¹The source of this quote is unknown; it may be Yogi Berra, Jan van de Snepscheut, Walter Savitch, or perhaps others.
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 $\lambda(e)$.

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 $p(X, A) = p(X, B) = 0.4$.

   (a) Compute the values for $\lambda(e)$ for every edge $e$, and draw the CFN tree with these branch lengths.
   (b) 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?
   (c) Write down the matrix $M$ of leaf-to-leaf CFN distances for this tree.
   (d) What is the longest leaf-to-leaf path in this tree?
   (e) What is the smallest value in the $M$?
   (f) Are the two leaves with this smallest distance siblings in the tree?
   (g) Write down the three pairwise sums. Which one is the smallest?
   (h) 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 $p(Y, D) = p(X, B) = 0.4$.

   (a) Compute the values for $\lambda(e)$ for every edge $e$, and draw the CFN tree with these branch lengths.
   (b) 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?
   (c) Compute the matrix $M$ of leaf-to-leaf CFN distances.
   (d) What is the longest leaf-to-leaf path in this tree?
(e) What is the smallest value in the matrix $M$? Are the two leaves with this smallest distance siblings in the tree?

(f) Write down the three pairwise sums. Which one is the smallest?

(g) Is the matrix additive?

8. Consider how $\lambda(e)$ is defined by $p(e)$.

(a) Compute $\lim_{p(e) \to 0} \lambda(e)$

(b) Compute $\lim_{p(e) \to 0.5} \lambda(e)$

(c) 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.
Chapter 2

Trees

Trees are graphs, with vertices (also known as “nodes”) and edges. The trees we will be working with in phylogenetics represent the evolution of a set of species from a common ancestor. Therefore, species trees have leaves representing the taxa of interest (generally these are extant species), and internal nodes represent ancestral species. Because they represent evolutionary histories, they may be called “phylogenies”, “phylogenetic trees”, or “evolutionary trees”. Sometimes phylogenetic trees are drawn rooted, although (as we shall see) most methods for estimating evolutionary trees produce unrooted trees. 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. This will turn out to be important in understanding how trees are constructed from character data.

2.1 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 3 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 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 4 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.

The representation of a polytomy can vary between different graphical representations. In Figure 3.1, 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.

2.1.1 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.
Figure 2.1: Two ways of drawing the same tree

Figure 2.2: Tree ((a,b),(c,d))

Figure 2.3: Tree (((a,b),c),(d,e))
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 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 tree given in Figure 3.2:

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

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 trees given in Figures 2.3 through 2.5 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 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.
2.1.2 The clade representation of a rooted tree

We begin with some basic terminology.

Definition 5 Let $T$ be a rooted tree in which every leaf is labelled by a distinct element from a set $S$. Thus, $L(T) = S$, and is called the leafset of $T$. Any subset $A$ of $L(T)$ 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 $L(T_v)$. 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 $L(T)$ and also the internal nodes).

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

Definition 6 Let $T$ be a rooted tree on leaf-set $S$. We define the set $\text{Clades}(T) = \{L(T_v) : v \in V(T)\}$. 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 1 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, $\text{Clades}(T) = \{\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{a, b\}, \{a, b, c\}, \{d, e\}, \{c, d, e\}, \{a, b, c, d, e\}\}$.

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 3.3, 3.4, and 3.5, you’ll see that they all have the same set of clades. Thus, they are all identical.

2.1.3 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 $x \subset y$. Since containment is transitive, if $x \subset y$ and $y \subset z$, then $x \subset z$. Hence, if we have directed edges
from $x$ to $y$, and from $y$ to $z$, then we know that $x \subset z$, and so can remove the directed edge from $x$ to $z$ 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** Consider the following input:

- $A = \{\{a\}, \{a, b, c, d\}, \{a, d, e, f\}, \{a, b, c, d, e, f\}\}$.

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 1** 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 $\text{Clades}(T) = \cup_i \text{Clades}(t_i) \cup \mathcal{L}(T)$. 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 $T$. \hfill $\square$

### 2.1.4 Compatible sets of clades

In the previous material, 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 $X$ 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 7** A set $X$ of subsets 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 $X \subseteq \text{Clades}(T)$.

Here we consider a related question: given a set $X$ of subsets of a set $S$ of taxa, is there a tree $T$ so that $X \subseteq \text{Clades}(T)$?

To answer this, see what happens when you construct the Hasse Diagram for the set $X$.

**Example 3** We begin with a simple example, $X_1 = \{\{a, b, c\}, \{d, e, f\}, \{a, b\}\}$. Note that $X_1$ contains three subsets and the set $S$ contains six elements. Thus, $X_1$ does not contain the singleton sets, nor the full set of leaves, and so it is not possible for $X_1$ 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 $X_1$ and obtain $X_1'$. 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 $X_1 \subseteq \text{Clades}(T)$. However, there are other trees, such as $T'$ denoted by $(((a,b),c),(d,(e,f)))$, that also satisfy $X_1 \subseteq \text{Clades}(T')$. Note that $T$ can be derived from $T'$ by contracting an edge in $T'$.
Example 4 As a second example, consider the set of subsets $X_2 = \{ \{a,b\}, \{b,e\}, \{c,d\} \}$. Note that $X_2$ contains three sets and the set $S$ contains five elements. Also, as we saw for $X_1$, $X_2$ does not contain the singleton sets nor the full set of leaves, and so $X_2$ is not the set of clades of any tree. We add all the trivial clades to $X_2$ to obtain $X'_2$, and construct the Hasse diagram for $X'_2$. Note that $\{b\} \subset \{a,b\}$ and $\{b\} \subset \{b,e\}$. Hence, the Hasse Diagram for $X'_2$ has a node with outdegree two – which is inconsistent with $X'_2$ being the subset of $\text{Clades}(T)$ for some tree $T$.

These two examples suggest an algorithm that you can apply to a subset $X$ to determine if it is compatible.

Algorithm to determine compatibility of a set $X$ of clades.

- Step 1: Compute the set $S$ of taxa, and add all the singleton sets and $S$ to $X$, to form set $X'$.
- Step 2: Construct the Hasse Diagram for $X'$.
- Step 3: The set $X$ is compatible if and only if the Hasse Diagram is a tree (with arcs directed towards the root).

Lemma 1 A set $A$ of subsets is compatible if and only if for any two elements $X$ and $Y$ in $A$, either $X$ and $Y$ are disjoint or one contains the other.

Proof: If a set $A$ 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 $A$ is the set of leaves below some vertex in $T$. Let $X$ and $Y$ be two elements in $A$, and let $x$ be the vertex of $T$ associated to $X$ and $y$ be the vertex associated to $Y$. If $x$ is an ancestor of $y$, then $X$ contains $Y$, and similarly if $y$ is an ancestor of $x$ then $Y$ contains $X$. 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 $A = \text{Clades}(T)$. □

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

Corollary 1 A set $A$ of subsets of $S$ is compatible if and only if every pair of elements in $A$ are compatible.

2.1.5 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 2.6, 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 2.7, 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.

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.

2.2 Unrooted trees

We begin with writing down rooted versions of unrooted trees, and then writing down unrooted versions of rooted trees.
Figure 2.6: Tree on some mammals with fly as the outgroup

Figure 2.7: Two trees which differ only in the placement of the outgroup
2.2.1 Newick formats 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 2.8, 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 8 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.

Two or more nodes with a common neighbor are siblings.

Two of the rooted trees consistent with the unrooted tree given in Figure 2.8 are provided in Figures 3.9 and 3.10.

2.2.2 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, we cannot write down clades. Instead, we write down “bipartitions”.

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
Figure 2.9: One rooted version of ((a,b), (c,d)).

Figure 2.10: Another rooted version of ((a,b), (c,d)).
We summarize this discussion with the following definition:

**Definition 9** 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 \( C_I(T) \).

### 2.2.3 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 \( (a, b)cde/f \). Similarly, we may represent it 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.)

### 2.2.4 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 \( T' \) 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 10** 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 11** 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.)

### 2.2.5 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 12** 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 \( A \subseteq C(T) \).

We define the **canonical tree** for a compatible set \( A \) of bipartitions of the set \( S \). 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 the following other sets:

36
Figure 2.11: Unrooted tree on \{1...9\}, obtained by running the algorithm on the set of compatible bipartitions from Example 5.

- \(S\),
- \(\{x\}\), for each \(x \in S\) (the singleton sets) and
- \(S \setminus \{r\}\).

The set of clades that is produced in this fashion is then used to construct the rooted tree, using the technique given above. This algorithm will produce a rooted tree \(T\) that you can then unroot to obtain the unrooted tree containing all the bipartitions.

**Example 5** Consider the set \(A\) of bipartitions defined by

\[
A = \{(123|456789), (12345|6789), (12|3456789), (89|1234567)\}.
\]

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 other sets, i.e., the full set \(S\), all the singleton sets, and the set \{2, 3, 4, 5, 6, 7, 8, 9\}. This gives us a larger set of sets (which we display without the singletons): 

\[
\{\{4, 5, 6, 7, 8, 9\}, \{6, 7, 8, 9\}, \{3, 4, 5, 6, 7, 8, 9\}, \{8, 9\}, \{2, 3, 4, 5, 6, 7, 8, 9\}, \{1, 2, 3, 4, 5, 6, 7, 8, 9\}\}.
\]

We then 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 1 is not actually the root of this tree, but instead is a leaf. Thus, 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 a root. However, the parent of 1 is the root of the tree, by construction. We then unroot this tree, to obtain the tree given in Figure 2.11.
2.2.6 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 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 6 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 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 $Y_2 \subseteq X_2$, 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 1. Hence, the theorem is proven. 

□
2.3 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 $2n - 6$ (see below for a discussion about this).

**Comments:** A few comments are worth making here. First, most typically, 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.

Finally, the following can be established:

**Observation 1** 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, and how they compare to the error rates of the trees on which they are based.

2.4 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 $n = 1, 2, \text{ or } 3$, 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, \( t_n = (2n-5)!! \).

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 \( T^u \) (obtained by ignoring the root of \( T \)), and that every unrooted binary tree \( T^u \) corresponds to \( 2n - 3 \) rooted binary trees formed by rooting the tree \( T^u \) on one of its edges. Hence, the number of rooted binary trees on \( n \) leaves is \( (2n-3)!! \).

2.5 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 set of trees 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 a strict consensus or a majority consensus tree) will then be largely unresolved.

Scientists can disagree as to what constitutes a rogue taxon. Furthermore, defining what a rogue taxon is depends on the technique used to estimate the tree on the taxa – and so the rogue taxa could be different depending on the method.

We will return to the problem of detecting rogue taxa later in the textbook, after we discuss phylogeny estimation methods.

2.6 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 you have a tree \( T \) (rooted or unrooted), and a subset of the leaf set that is of particular interest to you, and you wish to know what \( T \) tells you about that subset. 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 do the following: delete the other leaves and their incident edges, and then suppress nodes of degree two. If \( A \) is the subset of interest, then we denote by \( T | A \), the subtree of \( T \) induced by the set \( A \). See Figure 2.12 for a tree and one of its induced subtrees.
2.7 Some special trees

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

**Definition 13** The caterpillar tree on \( n \) leaves \( s_1, s_2, \ldots, s_n \) is given by \((s_1, (s_2, (s_3, (s_4, \ldots))))\). 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 14** 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.

2.8 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?

2.9 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 posets defined for the following sets of clades, and 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?
   - (a) $(a, (b, (c, ((d, e), (f, g))))))$
   - (b) $(((a, b), c), ((d, e), (f, g)))$

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\{cde\}, abc\{def\}, ab\{cde\})\}, with root “b”. Then do this again using root c. Are the unrooted trees you get different or the same?
• \{(ab\{cde\}, abc\{def\}, ab\{cde\})\}, with root “d”.
• \{(abcd\{ef\}, abc\{def\}, ab\{cde\})\}, 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:
• \((a, (b, (c, d))) \text{ and } ((a, b), (d, c))\)
• \((a, (b, (c, d))) \text{ and } (c, (d, (b, a)))\)

Your algorithm should return “NO” for
• \((a, (b, (c, d))) \text{ and } ((b, d), (a, c))\)

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).

   (a) What is the Robinson-Foulds (RF) rate of \(T_1\) with respect to the true tree?
   (b) For what trees \(T_2\) will \(T_1\) have a better RF rate than \(T_2\)?
   (c) 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 13). Now let $T$ 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 $T$?

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 $T_n$ of unrooted binary trees on leafset $S = \{s_1, s_2, \ldots, s_n\}$. If you pick a tree uniformly at random from $T_n$, what is the probability that $s_1$ and $s_2$ are siblings in $T$?

29. Consider a caterpillar tree $T$ on a set $S$ of $n$ taxa. Suppose there is a very “rogue” taxon, $x$. Consider the set $T$ that contains all the trees formed by adding $x$ into $T$?

   (a) What is $|T|$?
   (b) What is the strict consensus of all the trees in $T$? (Give its bipartition set.)
   (c) Draw the majority consensus when $n = 4$.
   (d) Draw the majority consensus when $n = 5$.
   (e) Draw the majority consensus when $n = 6$.
   (f) Can you generalize the observations seen here to provide a rule for the majority consensus trees, as a function of $n$?

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 $(f, (a, (c, (g, (d, e)))))))$ (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 $\{d, e, f\}$. 

44
Chapter 3

Constructing trees from subtrees

3.1 Constructing rooted trees from rooted triples

Here we 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, Sagiv, Szymanski, and Ullman (3), and is widely known in phylogenetics. However, it was developed for a problem in relational databases!

The input to this problem will be 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, by putting two leaves $a$ and $b$ in the same set if there is a rooted triple that puts them together. Compute the transitive closure of this relationship.
- If this produces one equivalence class, reject - no tree is possible. Otherwise, let $C_1, C_2, \ldots, C_k$ ($k \geq 2$) be the equivalence classes. For each equivalence class $C_i$,
  - Let $Trip_i$ be the set of triplets in $Trip$ that have all their leaves in $C_i$.
  - Recurse on $(C_i, Trip_i)$, and let $t_i$ be the compatible rooted tree, if it exists.

Make the roots of the $t_1, t_2, \ldots, t_k$ all children of a root, and return the final tree.

This surprisingly simple algorithm is provably correct, and runs in polynomial time. Proving that it works is also not that difficult.

3.2 Constructing unrooted binary trees from their quartet subtrees

We begin with some notation.

**Definition 15** Given an unrooted binary tree $T$ on $n$ distinctly labelled leaves, we denote by $Q(T)$ the set of all quartet trees on leaves in $T$. We denote the quartet tree on $a, b, c, d$ that splits $a, b$ from $c, d$ by $ab|cd$, $(ab, cd)$, $(a, b|c, d)$, or $((a, b), (c, d))$. 

45
Note that there are many ways to denote a quartet tree; we present them all because each appears in the scientific literature.

It is easy to see that every unrooted binary tree $T$ is defined by $Q(T)$, but proper subsets of $Q(T)$ can also uniquely define a tree $T$. For example, the tree given by $(1, (2, (3, (4, 5))))$ is defined by the set of quartet trees

\[(12|34), (12|35), (12|45), (13|45), (23|45)\]

We begin by presenting an algorithm that will construct $T$ from the set $Q(T)$ of its 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 $T$, and we will prove that the All Quartets Method is guaranteed to construct $T$ when the input is $Q(T)$. However, the All Quartets Method has no guarantees when the input is a subset of $Q(T)$! Before we present the algorithm, we define a general decision problem – determining if a set of quartet trees is compatible:

**Quartet Compatibility:**

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

The Quartet Compatibility problem is NP-complete (142), but some special cases of the problem can be solved in polynomial time, as we will show. In particular, the case where $X$ has at least one tree on every four leaves in a set $S$ can be solved in polynomial time, using the All Quartets Method, described below.

**All Quartets Method:** The input to this method is a set $Q$ of quartet trees, and we assume that there is exactly one tree on every four leaves from $S = \{s_1, s_2, \ldots, s_n\}$. We will assume that $|S| \geq 4$, since otherwise there are no quartets. We will also assume that every tree in $Q$ is binary (fully resolved).

The algorithm we describe here addresses the case where it is possible for the quartet trees to be incompatible, so that no tree exists on which all the quartet trees agree.

- 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 $i$.
- 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 7** We show how to apply this algorithm on the input given above. 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 tree on that set is (23|45). We then reintroduce the leaf for 1 as sibling to 2, and obtain the tree given by $(1, (2, (3, (4, 5))))$.

Note that the All Quartets Method suggests an algorithm for constructing trees: compute the tree on every quartet, and then combine them together. If all quartets are correctly computed, the resultant tree will be correct. However, in practice, not all quartet trees are likely to be correctly computed, and so methods for constructing trees from quartet subtrees need to be able to handle some errors in the input trees.

### 3.3 Compatibility supertrees

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 $X$. 46
Example 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 13) $(a, (b, (c, (d, (e, (f, g))))))$ is a compatibility supertree.

Because the quartet tree compatibility problem is NP-complete (142), determining if all the subtrees are compatible (and finding the compatibility supertree if it exists) is also NP-complete.

In the previous sections, we described a method for constructing a rooted tree from a set of rooted triples (i.e., rooted 3-leaf trees), and another method for constructing an unrooted tree from a set of unrooted quartet trees (i.e., unrooted four-leaf trees). Here we discuss the related question of testing whether a given set of rooted triples or unrooted quartet trees is compatible.

### 3.3.1 Testing compatibility of a set of rooted trees

We begin with the decision problem where the input is 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 algorithm of Aho, Sagiv, Szymanski, and Ullman (described above) on the resultant set of rooted triplet trees. If the result 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 one of the recursive calls, only one equivalence class is computed); in that case, the rooted triplet trees are not compatible, and hence also the set $X$ is incompatible. In other words, it is easy to see that testing a set of rooted trees for compatibility is a polynomial time problem. We summarize this as follows:

**Theorem 3** Given a set $X$ of $k$ leaf-labelled rooted trees, each leaf-labelled by a subset of a set $S$ with $|S| = n$, we can test whether $X$ is compatible in time that is polynomial in $n$ and $k$.

### 3.3.2 Testing compatibility of a set of unrooted trees

Now consider the decision problem where the input is a set $X$ of unrooted leaf-labelled trees (leaves drawn from set $S$) and we want to know if there is an unrooted tree $T$ on leafset $S$ that induces each of the trees in $X$.

As with the analysis in the previous section, we can encode each tree in $X$ with a set of quartet trees, and then ask whether the union of all the sets of quartet trees (taking all the quartets from all the trees in $X$ together) is compatible. If the set we obtain contains at least one tree on every four-leaf subset of $S$, then the All Quartets Method solves this problem: if All Quartets returns a tree, then it is the compatibility supertree, and otherwise the set $X$ is not compatible.

The problem with this approach is that in general it will not be the case that every four-leaf subset of $S$ is in some tree in $X$ (which is what we need for this approach to work). Furthermore, the problem of whether an arbitrary set of four-leaf trees is compatible is NP-hard!

**Theorem 4** The Unrooted Tree Compatibility Problem – determining if a set $X$ of unrooted trees, each leaf-labelled by elements from $S$, is compatible (i.e., has a supertree that induces all the trees in $X$) – is NP-complete.

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.

### 3.4 Optimization problems

The previous material assumed that the input is a set of trees that is compatible, and the objective is to construct a tree (or set of trees) that agrees with all the input trees. However, in the context of phylogeny estimation, source trees will be estimated and so will almost always have some error, and this is particularly true when rooted source trees are
estimated. Hence, instead of finding trees that agree with all the source trees, it is necessary to consider optimization problems – finding trees that are close to the set of source trees.

### 3.4.1 Maximum Triplet Support

We begin with optimization problems when the input is a set of rooted trees, which we then encode as a set of rooted triples. A natural optimization problem is **Maximum Triplet Support**, where the objective is a rooted binary tree that agrees with a maximum number of rooted triples. Equivalently, this can be formulated as finding a subset of the triplets that are setwise compatible.

The Maximum Triplet Support problem, however, is NP-hard (22), and so heuristics have been developed to find good solutions to the problem. The most well known of these heuristics is MinCutSupertree (134), 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 ASSU algorithm, 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 (135), and found to produce more accurate trees.

### 3.4.2 Maximum Quartet Support

Optimization problems when the input is a set of quartet trees have also been posed. Some of the methods that have been developed for constructing supertrees use heuristics to attempt to produce trees that satisfy as many of the input quartets as possible; examples of such methods include Quartet Puzzling algorithm (145), Weight Optimization (120), Short Quartet Puzzling (137), Quartets Max Cut (136), and QFM (121). However, because quartet tree compatibility is NP-complete (142), finding trees that satisfy a maximum number of the input quartet trees is NP-hard. However, approximation algorithms can be developed, and when the set contains a tree on every quartet, then a PTAS is also possible (89).

One of the interesting approaches to estimating trees from unrooted quartet trees allows the input quartet trees to be equipped with arbitrary positive weights (i.e., \( w(q) \in \mathbb{R}^+ \)) 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, this problem is also NP-hard. Heuristics for the problem have been proposed, including Weighted Quartets Max Cut (8).

### 3.4.3 Split-Constrained Quartet Support

A variant of the Maximum Quartet Support problem is the **Split-Constrained Quartet Support Problem**, which we now define:

- **Input**: Set \( Q \) of unrooted binary quartet trees with weights, \( w : Q \to \mathbb{R}^+ \), and set \( X \) of bipartitions of set \( S \) of taxa
- **Output**: Unrooted binary tree \( T \) such that \( Q(T) \) has maximum total weight among all unrooted binary trees \( t \) that satisfy \( C(t) \subseteq X \)

To understand what this means, imagine that \( X \) is the set of all possible bipartitions on \( S \); then 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.

First, note that 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 set \( T \) of quartet trees, possibly with weights on the quartet trees. The objective is a rooted binary tree \( T \) such that \( Clades(T) \subseteq C \) 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

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

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 tree, thus producing an unrooted binary tree with the best quartet support.

To see how this might work, let $T$ 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 $Y$, we take the set $X$ of bipartitions, and we include every half of every bipartition in $X$. In other words, if $A|B$ is a bipartition in $X$, then $Y$ will contain sets $A$ and $B$. We also include the full set $S$ and every singleton leaf in $Y$.

Now let $T \in R_{C,S}$ be arbitrary. If $R_{C,S} = \emptyset$, 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 Fail. Suppose $t \in T$ is a quartet tree that supports $T$; thus, $t$ is induced by $T_u$, the unrooted version of $T$. Let $v$ be the (unique) lowest node in $T$ (i.e., the node that is furthest from the root of $T$) where 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 $T \in R_{C,S}$, 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 $S \setminus A$ of leaves that are not below $v$ is also in $C$. In other words, the node $v$ defines a tripartition of the leafset $S$ into three sets of allowed clades, $(A_1, A_2, S \setminus A)$, where $A = S$ is possible.

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

1. $ab|cd$ 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 $ab|cd$ 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 $Support(T) = \sum_{v \in V(T)} QS(U, V, W)$, 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 $A_3 = S \setminus A$. 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 $U \cup V \cup W = S$. 

49
• 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 (23).

A variant of this constrained search problem has an input set $T$ 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. This variant of the problem is discussed in greater detail in Chapter 9, due to its relevance to constructing species trees from gene trees.

### 3.5 Further reading

Other quartet-based methods have also been developed that have a different kind of objective, such as producing trees whose bipartitions do not contradict any quartet trees (e.g., the $Q^*$ method (17), also called the Buneman Tree) or trees that contradict only a restricted number of quartet trees (18; 15; 16; 51; 59).

### 3.6 Review questions

1. What is a “rooted triple”?

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. What is the Split-Constrained Quartet Support problem? Does it run in polynomial time?

7. 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 $Q$?

8. 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 method of Aho, Sagiv, Szymanski, and Ullman to test for compatibility of the set $R$?
3.7 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. (a) Write down the set $X$ of rooted triples for the caterpillar tree given by $(1, (2, (3, (4, 5))))$.

   (b) Apply the Aho, Sagiv, Szymanski, and Ullman 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 Aho, Sagiv, Szymanski, and Ullman 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 Aho, Sagiv, Szymanski, and Ullman algorithm correctly solves the problem of determining if a set $Trip$ of rooted, fully resolved, three-leaf trees is compatible. (Hint: use induction.)

7. 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 $Trip$ of triplet trees that is allowed to have these hard polytomies, and show that the Aho, Sagiv, Szymanski, and Ullman (ASSU) algorithm will not correctly solve the compatibility problem on $Trip$. Thus, either $Trip$ should be compatible but the ASSU algorithm should say it is not, or vice versa.

8. Consider input sets $Trip$ of rooted triplet trees, and suppose some of them can be hard polytomies.

   (a) Modify the Aho, Sagiv, Szymanski, and Ullman algorithm so that it can correctly handle input sets $Trip$ that have triplet trees with hard polytomies.

   (b) Prove your algorithm correct.

9. In the text, we stated that the algorithm of Aho, Sagiv, Szymanski, and Ullman is polynomial time. Prove a running time analysis, where the input is a set of $k$ rooted triplet trees drawing their leaves from set $S$ of $n$ taxa.

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

11. 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?

12. Construct a tree on leaf set \{a, b, c, d, e, f\} that induces each of the following quartet trees:

   - $(ab|cd)$,
   - $(ab|ce)$,
   - $(ac|de)$,
   - $(bc|de)$,
   - $(ab|de)$,
   - $(ab|ef)$,
   - $(ab|df)$,
   - $(ab|ef)$,
   - $(ac|df)$,
13. 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). Give an example of a set $A$ that is compatible, and so $A \subset Q(T)$ for some binary tree $T$, but where the All Quartets Method can fail to construct $T$ from $A$.

14. 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.)

15. 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 $T$.

16. 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 $(a, b, c, d)$).

17. 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.

18. 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.

19. 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?

20. 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?

21. 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?

22. 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 $n$).

23. 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 $n$.

24. 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:
   (a) Are these unrooted trees compatible? Justify your answer.
   (b) Root all the three trees at leaf 1, and draw the rooted versions of these trees. Are these rooted trees compatible? Justify your answer.
Chapter 4  

Constructing trees from qualitative characters

4.1 Introduction  

We now begin the discussion of how to estimate trees from data. 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 each of these cases, characters divide the dataset into different pieces, and the taxa (species or languages) within each piece are equivalent with respect to that character – they share the same state.

One way to think of a character is as an equivalence relation on the set $S$ of species, and the different equivalence classes are the states of the character. However, characters are also described by functions from the species set $S$ to the set of states.

Mathematically, most models for the evolution of characters down trees assume that character state changes occur due to substitution. 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 homoplasmy-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.

Finally, 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).

Molecular characters in biology are derived from alignments of nucleotide or amino acid sequences, and thus have a maximum number of possible “states” (four for DNA or RNA, and 20 for amino acids). When only two states are exhibited, the character is said to be binary. Some characters (for example, presence/absence characters) are explicitly always binary.

Methods for estimating trees  

Methods for reconstructing trees from characters come in several variants. Maximum parsimony is a frequently used method, especially for morphological data. However, methods based upon statistical models of evolution, and hence involving calculations of the likelihood, are also favored by some researchers. Finally, methods that first transform the character data input into distance matrices are also popular; these are called “distance-based” methods. We cover maximum parsimony (and related approaches) in this chapter, and distance-based methods and likelihood-based methods (both Maximum Likelihood and Bayesian methods) in Chapters 5 and 7, respectively. Finally, these methods all produce trees, but reticulations (borrowing or creolization) can also occur. We will therefore include a chapter on estimating reticulate evolutionary histories.

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 matrix format, 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.

### 4.2 Constructing rooted trees from directed binary characters

Constructing trees from characters can be very simple or very complicated. We begin with the very simple situation: binary characters that evolve without any homoplasy. To make it even simpler, we’ll assume that the characters are given with an orientation, so that the ancestral state is known. We call these **directed binary characters**, and note that they have an **ancestral** state and a **derived state**. Under the assumptions that the characters have no homoplasy and you know the ancestral state, tree estimation from directed binary characters reduces to *given a set of clades in a tree, construct the tree*. This is a problem we solved in the previous section! Note that this produces a rooted tree, with the root having the ancestral state for all the characters.

### 4.3 Constructing unrooted trees from compatible binary characters

The problem becomes slightly harder when we are given binary characters without information on the ancestral state. However, this case also has an easy solution, as we will show. We treat one taxon as the root, and let its state for each character be the ancestral state of that character. This makes the problem equivalent to constructing a rooted tree (on the remaining taxa) from clades. Once that rooted tree is constructed, we add the taxon that represented the root to the tree, and then *unroot* the tree.

In the case where the characters evolve without homoplasy (and so are compatible on a tree), whether the characters are directed or not, the **minimal tree** that fits the character evolution assumptions is **unique**, and can be computed in polynomial time. Here, by minimal, we mean the **minimally resolved tree**. Thus, we seek a tree in which no edge can be contracted while still having the property that all the characters are compatible. This minimal tree may not be binary, however, since the tree that is computed will only have edges on which the binary characters change. More generally, what this means is that if you use only a subset of the available characters, the tree you obtain may not be fully resolved. Importantly, this means that the interpretation of polytomies (nodes of high degree) in trees estimated using this technique is that they are likely due to incomplete information (not all the characters are used, or perhaps are not available). Note that the algorithms can be used in two ways: to construct a tree for which the characters evolve without homoplasy, or to determine that no such tree exists! Finally, note that these algorithms require that all taxa exhibit states for all the characters – that is, it is not possible to apply the algorithms when some character data are missing. Therefore, when the state of some characters for some taxa is unknown, you cannot use these algorithms.

We now turn to some examples.

**Example 9** Suppose that the input is given by

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

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. The tree has only two edges: \( e_1 \) defining the bipartition \( ABC \mid DEF \) and the edge \( e_2 \) defining the bipartition
\( ABCD \mid EF \). If we collapse \( e_1 \), the first character becomes incompatible. If we collapse \( e_2 \), the second character
becomes incompatible. Hence, the tree is minimally resolved, subject to all the characters being compatible on the
tree. Finally, because all the characters are compatible on the tree, it is an optimal solution to MC and also to MP.

Comments: In the example we just did, the input dataset was a set of six taxa defined by five compatible binary
characters. The application of the algorithm produced a tree on which all the characters are compatible (i.e., could
evolve in a homoplasy-free manner), but the tree was not fully resolved; in fact, it had a node of degree four when
written as an unrooted tree. Furthermore, this is a minimally resolved tree on which all the characters are compatible.
Only those edges of the true tree on which changes occur will be reconstructed. In this case, the proper interpretation
of the polytomies is that you lack information sufficient to resolve the tree.

Finally, although it may not be obvious, there is no other minimally resolved tree on which all the characters are
compatible. When the input is a set of compatible binary characters, then the algorithm for constructing trees from
compatible bipartitions (which are the same as compatible binary characters) will produce the unique such minimum
compatibility tree.

This example used binary characters, so that each character has only two states (here we used 0 and 1, but the same
outcome would have happened using any two other states). If we had an input with multi-state characters, which are
much more the rule in biology, we could not use the algorithms presented so far to determine whether the characters
are compatible and construct a tree when they are. Instead, more complicated algorithms would have been necessary.

4.4 General issues in constructing trees from characters

Until this point, our discussion has assumed that all taxa exhibit states for all characters in the input matrix, and that
all the characters are compatible and binary (exhibit two states). Under these assumptions, it is easy (polynomial time,
and easy to do by hand) to construct trees: we use the algorithm for constructing trees from compatible bipartitions,
and use the tree that results. However, can we apply the simple algorithm when these assumptions do not hold? That
is, when the input consists of characters that are binary (for example, presence/absence), but we are missing some
information? Or when the input has non-binary characters? Or when the input is incompatible?

4.4.1 Missing data issues

We begin with the complication when not all taxa exhibit states for all the characters. A natural approach is to find out
if it is possible to assign values for the missing entries in the character matrix so as to make the input compatible.

Example 10 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 11** 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-hard, even when only two states otherwise appear. The computational method for solving this problem involves a mathematical transformation of the input matrix so that there are no missing entries. Instead, every question mark is replaced with a new state that does not appear in the dataset for any other language. Thus, the initial data matrix might only have two states (presence/absence, or 0/1), but the transformed data matrix could have many more states. For example, if we apply this technique to the input given above, we obtain:

• $A = (0, 0, 2)$
• $B = (0, 1, 0)$
• $C = (1, 0, 0)$
• $D = (1, 2, 1)$
• $E = (2, 1, 1)$

Now, if we begin with an input $M$ with missing entries, and do this transformation, we obtain a new input $M'$. Note that a perfect phylogeny exists for $M$ if and only if a perfect phylogeny exists for $M'$. Unfortunately, while determining if a perfect phylogeny exists for binary characters is easy (and can be constructed in polynomial time), determining if a perfect phylogeny exists for multi-state characters is computationally harder: no longer polynomial time, and not easy to do by hand.
4.4.2 Constructing trees from compatible multi-state characters

The previous section was all about binary characters, typically based upon presence/absence of some feature. We also primarily focused on characters that evolve without homoplasy (back-mutation or parallel evolution). But what about other types of characters?

We begin with the assumption that the characters evolve without homoplasy. In this case, algorithms to find the trees on which all the characters evolve without any homoplasy do exist, but they are computationally more expensive – no longer polynomial, as in the case of binary characters. Also, when there are three or more states, it is no longer the case that there is a unique minimal tree that is consistent with the input; this complicates the problem of determining whether a set of multi-state characters is compatible.

Before we go into how to construct trees from multi-state characters, we address the “simpler” issue of testing whether a multi-state character is “compatible” on a tree (meaning, it could have evolved without any homoplasy on the tree).

4.4.3 Testing compatibility of a character on a tree

To do this, we wish to set 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. When this is the case, the character is said to be compatible with the tree. Testing whether a character is compatible on a tree is straightforward, and can be done by eye.

For a given internal node \( v \) in the tree, if it 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. Otherwise, homoplasy-free evolution is not possible on this tree.

4.5 Maximum Compatibility

Algorithms for constructing trees under the assumption that all the characters are compatible will fail when any character evolves with homoplasy.

We begin with the definition of the maximum compatibility problem. Recall that a character \( c \) is said to be compatible on a tree \( T \) if it is possible to define the character states at the internal nodes so that for all states of \( c \), the set of nodes exhibiting that state is connected. An equivalent definition is that if \( c \) exhibits \( r \) states on the tree \( T \), then there are exactly \( r-1 \) edges of the tree \( T \) on which \( c \) changes state.

The maximum compatibility problem is then as follows:

**Maximum Compatibility**

**Input:** Character 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 leaf set \( S = \{s_1, s_2, \ldots, s_n\} \) on which the number of characters in \( C = \{c_1, c_2, \ldots, c_k\} \) that are compatible is maximized.

Related to this search problem is the problem of determining the number of characters that are compatible on a given tree (i.e., scoring a tree with respect to compatibility).

**Computing the compatibility score of a tree**

**Input:** Character 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 \( T \).
This problem is polynomial, since (as we showed in the previous section), determining if a character is compatible on a tree can be done quite simply. Hence, scoring a given tree under compatibility is polynomial.

On the other hand, finding the tree with the largest compatibility score is more computationally challenging. If we use an exhaustive search technique, scoring each of the possible solutions in turn, this would take time $O(t(n)nk)$, where $n$ is the number of taxa, $k$ is the number of characters, and $t(n)$ is the number of binary trees on $n$ leaves (i.e., $t(n) = (2n - 5)x(2n − 7) · x3$). The reason we only need to examine binary trees, is that optimal solutions to maximum compatibility are obtained at the binary trees (i.e., if a non-binary tree could be any optimal solution, each of its refinements will also be an optimal solution).

We now look at computing solutions to maximum compatibility.

**Example 12** Consider the following set of taxa, represented by three characters.

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

One tree on which these characters are all compatible is given by $(A, (B, (E, (F, (C, D))))))$.

**Example 13** The next example is more interesting:

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

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. One of those trees is given by $((A, B), (C, D))$, and the other is $((A, D), (B, C))$. The third possible unrooted tree on these taxa is $((A, C), (B, D))$, which is incompatible with both these characters.

### 4.6 Maximum Parsimony

Maximum parsimony 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 on a tree.

For those characters that evolve without any homoplasy, it is easy to assign states on the tree so that the character changes state the minimum number of times. And, in fact, if the character exhibits $r$ states on the dataset, then it will change state exactly $r − 1$ times if it evolves without homoplasy (and otherwise it will change state more than $r − 1$ times). Determining the minimum number of times the character must change state is a polynomial time problem, but not an easy one to do by hand. We will return to this another time! However, on small enough trees, it can be done by eye if you are careful.

Recall the discussion of this issue given in the introduction. First, if a character is defined for all nodes in a tree, then this means that every node of the tree is given a state for that character. In this case, the number of state changes for that character on the tree is simply the number of edges on which the character changes state, and is easily
computed. However, if the character is only defined on the leaves, we will want to compute the best state assignment to the internal nodes so as to minimize the total number of state changes for the character. This problem is easily done by inspection for small trees, and can even be done efficiently (meaning in polynomial time) on large trees — although the technique is then best done using software rather than by eye. Thus, when the tree $T$ and character matrix $M$ are given, it is possible to compute the number of character state changes on $T$ for the matrix $M$ in polynomial time. This minimum total number of changes of a character matrix $M$ on a tree $T$ is called the length of the tree, and also the parsimony score. Thus, maximizing parsimony means producing the minimum parsimony score. Somewhat confusing terminology, eh? Finding the best tree $T$ for a given character matrix $M$ is the maximum parsimony problem, i.e.:

### 4.6.1 Maximum parsimony problem formulation

**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 leaf set $\{s_1, s_2, \ldots, s_n\}$ with the smallest total number of changes for character set $\{c_1, c_2, \ldots, c_k\}$.

While maximum parsimony is polynomial time if the tree is given, the problem is NP-hard when the tree is not known and instead must be found (68). 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). These heuristics are not guaranteed to solve the optimization problem exactly, but seem to produce trees that are close in score and topology to the optimal solution, in reasonable time frames (i.e., hours rather than months).

Like maximum compatibility, maximum parsimony is optimized on binary trees, and heuristics for solving maximum parsimony need only examine completely resolved trees. Even so, these heuristics are computationally expensive, taking (in some cases) many days of analysis to come to what can only be guaranteed to be local optima.

### 4.6.2 Dynamic programming algorithm to compute parsimony score on a tree

We begin with the problem of computing the unweighted parsimony score of a fixed tree. In this problem, the tree is given and all substitutions have equal weight. We will show a very simple dynamic programming algorithm for this problem that allows you to compute the parsimony score in polynomial time. The algorithm also allows you to compute an assignment of states for each character to each node in the tree, in such a way that you produce the smallest number of changes.

The simplest form of the algorithm operates as follows (here we assume the input tree is unrooted and binary; modifying the algorithm for non-binary trees is slightly more complicated): The algorithm is applied to each character independently.

- Root the tree on an edge, thus producing a rooted binary tree.
- If $x$ is a leaf, let $A(x)$ denote the state at the leaf $x$ for the given character.
- 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 $A(v) = A(w) \cup A(x)$.
- When you reach the root, $r$, pick an arbitrary state in $A(r)$ to be its state. Then traverse the tree downwards towards the leaves, picking states for each node, as follows:
  - If the parent of node $y$ has been assigned a state that is within $A(y)$, 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-phase process (one up the tree, and one down), you will have assigned states to each node in the tree. Note that in the upwards phase, some nodes will be assigned definite states, but others may be given more than one possible state. When \( A(v) \) has only one element in it, then \( v' \) assigned under maximum parsimony is uniquely determined. This will be relevant to issues involving estimation of the properties of ancestral taxa, using maximum parsimony.

### 4.6.3 Dynamic programming algorithm to compute weighted parsimony score on a tree

The problem is somewhat more complicated if the substitution costs can depend on the particular pair of letters. 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 it turns out, this is not really any harder than unweighted maximum parsimony. 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:

\[
\text{Cost}(v, x) = \min\{\text{Cost}(w, y) + M[x, y] : y \in \Sigma\} + \min\{\text{Cost}(w', y) + M[x, y] : y \in \Sigma\}
\]

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 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 the 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.

### 4.7 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.

### 4.8 Informative and uninformative characters

In terms of solving maximum parsimony, or analyzing the properties of the maximum parsimony as a method, it is helpful to evaluate when a character has an impact on the tree that will be returned. In other words, if your input matrix $M$ (where $M_{i,j}$ is the state of the taxon $s_i$ for character $j$) is given as input, you would like to know whether removing some specific character (say character $x$) has any impact on the tree that is returned. 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 MP($M-x$) returns the same optimal tree (or set of optimal trees) as MP($M$)?" A character that has no impact on tree estimation using maximum parsimony methods (when solving MP exactly) is called “parsimony uninformative”, and is formally defined as follows:

**Definition 16** 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 $x$.

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 2** 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 14 Consider the following set of four DNA strings, \( u, v, w, x \):

- \( u = AAAAAAG \)
- \( v = ACTTTCG \)
- \( w = TTTTTTG \)
- \( x = TACTGGG \)

Numbering the eight sites from left to right, we can see that the first, second, and third sites are parsimony informative (each has two big states), but the remaining sites are not parsimony informative. That is, each of the remaining sites either has only one big state, or has no big states. Hence, in an MP analysis, only the first three sites will impact the MP scores, and hence only these three sites need to be considered in solving MP.

We now try to solve MP on this dataset. There are three possible trees on \( u, v, w, x \), given by \( T_1 = uv|wx, T_2 = ux|vw, \) and \( T_3 = uw|vx \). The first site has cost 1 on \( T_1 \) and cost 2 on the other two trees. The second site also has cost 1 on \( T_1 \) and cost 2 on the other two trees. The third site has cost 1 on \( T_2 \) and cost 2 on the other two trees. Hence, the least cost is obtained on \( T_1 \), and so \( T_1 \) is the single best MP tree for this dataset.

If we were to try to solve MC on the dataset, the same analysis would work, and so \( T_1 \) would be the single best MC tree for the dataset.

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.

   \[
   \begin{align*}
   a &= (1, 1, 0, 0, 1, 0) \\
   b &= (1, 0, 1, 0, 1, 0) \\
   c &= (0, 0, 0, 1, 0, 0) \\
   d &= (0, 0, 0, 1, 1, 1)
   \end{align*}
   \]

2. Take the data matrix from the previous problem, and add in the root sequence, \( r \), given by \( r = (0, 0, 0, 0, 0) \). 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.)

   \[
   \begin{align*}
   a &= (0, 0, 1, 1) \\
   b &= (1, 0, 0, 1) \\
   c &= (1, 1, 0, 1) \\
   d &= (1, 0, 1, 0)
   \end{align*}
   \]

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:

   \[
   \begin{align*}
   a &= (0, 0, 0, 1, 1) \\
   b &= (0, 1, 1, 0, 0) \\
   c &= (1, 0, 0, 1, 1)
   \end{align*}
   \]
5. For the following input, show how to set the entries given with “?” so as to produce a compatible matrix:

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

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 leaf set, 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.

(a) Is it possible for a solution to the “binary tree maximum parsimony” problem to not be optimal for the standard maximum parsimony problem?

(b) 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 leaf set, and \( T' \) refines \( T \). Prove that the parsimony score of \( T' \) is at most that of \( T \).

12. Suppose the rooted tree \( T \) is given as \( (A, (B, (C, (D, E)))) \). Suppose we have four characters describing these taxa, so that each taxon is described by their 4-tuple of character states:

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

First, determine which characters are compatible on the tree. For these character(s), determine which nodes of the tree have uniquely determined states for the character. Finally, for the character(s) that are not compatible on the tree, which nodes have uniquely determined states for these characters? To answer this, apply the maximum parsimony algorithm, and determine the character state assignments that optimize the parsimony score.

13. Suppose the tree is given by \( (A, (B, (C, (D, E)))) \), and that we have three homoplasy-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.

14. 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?

- \( \Theta(nk) \)
- \( \Theta(2^n k) \)
- \( \Theta(2^k n) \)
- \( \Theta(n^2 k) \)

15. 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.

16. 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.

17. 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.
(a) 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?)

(b) If you use normalized Hamming distances instead of Hamming distances, does your answer to the previous question change?

18. 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 optimal solutions to maximum parsimony on this input, and explain how you obtain your answer. Do not solve this by looking at all possible trees on $\{a, b, c, d, e, f\}$.

19. 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.
Chapter 5

Distance-based methods

5.1 Overview

We now discuss a different way of computing phylogenies that are called “distance-based methods”. 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: represent the set of $n$ taxa by an $n \times n$ matrix $d$ (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 17** An $n \times n$ matrix 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, we should not refer to them as distance matrices, and instead will refer to them as “dissimilarity” matrices. With that said, nearly all the literature in phylogeny estimation abuses the term, and refers to these 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.

5.2 How to compute “distances”

The first step in a distance-based method is producing the pairwise distance matrix. Assume that the input is a set of $n$ taxa that are described by a set of $k$ characters, each taking one of $r$ possible states. Most typically these taxa would be described by molecular sequences in an alignment, so that $k$ is the sequence length, and $r = 4$ for nucleotide sequences or $r = 20$ for amino acid sequences. We refer to the $i^{th}$ position in a sequence $x$ by $x_i$. There are several techniques for calculating distances between two sequences $x$ and $y$, each of the same length, of which the most popular is the Hamming distance (the number of sites in which two sequences are different).

**Definition 18** The **Hamming distance** between two sequences $x$ and $y$, each of the same length $k$, is the number of positions $i$ for which $x_i \neq y_i$. This distance can be normalized by the sequence length $k$ to produce a value between 0 and 1, in which case it is referred to either as the **normalized Hamming distance** or the **p-distance**.

Note that the Hamming distance between two sequences is a lower bound on the number of changes that occurred in the history relating the two sequences, but may not be identical to that number when a position changes more than
once. For example, if ACA changes to TCA and then again to GCA, then there are two changes that have occurred in the history between ACA and GCA, but the Hamming distance between these two sequences is only 1. One approach to phylogeny estimation uses what is known (or assumed) about the stochastic process that generated the data, in order to estimate a distance that is as close as possible to the actual number of changes that occurred between every pair of sequences. If these actual numbers of changes can be computed, then the tree that relates the sequences can be computed exactly, and in polynomial time (as we will show).

Recall the discussion in Chapter 1, where we introduced the Cavender-Farris-Neyman model and showed how to compute Cavender-Farris-Neyman distances based on the model parameters, and also from sequences. Specifically, we defined the Cavender-Farris-Neyman distance between sequences \( s_i \) and \( s_j \) to be

\[ d_{ij} = -\frac{1}{2} \ln(1 - 2 \frac{H_{ij}}{k}), \]

where \( k \) is the sequence length and \( H_{ij} \) is the Hamming distance between sequences \( s_i \) and \( s_j \). Thus, the Cavender-Farris-Neyman distances are based on normalized Hamming distances (p-distances).

These Cavender-Farris-Neyman distances are called “corrected” distances, and are therefore different from these “uncorrected” distances. This is an example of a more general procedure in statistical phylogenetics, where these p-distances are corrected with respect to an assumed stochastic model of evolution, in order to account for hidden changes. Thus, for DNA sequences, if you observe A in position \( i \) for sequence \( s \) and C in position \( i \) for sequence \( s' \), it is possible that more than one substitution occurred in position \( i \) in the evolutionary history relating sequences \( s \) and \( s' \).

Whether the distance matrix is corrected or not, however, the output of this first step is a matrix of pairwise distances. This matrix will be 0 on the diagonal and symmetric, but depending on the details of how the distances are computed, it may not satisfy the triangle inequality (that is, it may be that for some triple of taxa, \( s_i, s_j \) and \( s_k \), we have \( d(s_i, s_j) + d(s_j, s_k) < d(s_i, s_k) \)).

These comments also hold for the models of DNA sequence evolution, such as the Jukes-Cantor model, the General Time Reversible model, and the General Markov model (141). Thus, all these models come equipped with techniques for computing distances, and the distances “correct” for unobserved changes, in order to come close to an additive matrix defining the model tree. For example, under the General Markov Model (and hence for all of its submodels), the “logdet” distance is a statistically consistent method for computing an additive distance matrix that defines the tree topology of the General Markov Model tree (141).

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.

### 5.3 UPGMA

One of the original ways of computing trees is UPGMA, which stands for “Unweighted Pair Grouping Method of Agglomeration”. UPGMA computes a tree from an input distance matrix, in an iterative fashion. In each iteration, it finds a pair of taxa that have the smallest distance, and makes them siblings. One of them is then removed (or else both are removed and replaced by a new taxon), and the process is repeated. This technique produces a rooted tree, but the root can be ignored.

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 strong 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 leaves obtained by adding up the lengths of each edge. Note that the pair that minimizes the
Figure 5.1: Ultrametric matrix.

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

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 wrong tree.

Some comments are worth making about UPGMA. As we have described UPGMA here, we have only shown what it does to compute the first sibling pair. If the dataset has only three leaves, you can determine the rooted tree that UPGMA will produce based on this information. If the dataset has four leaves, then you can determine the unrooted tree it will produce (since that’s determined by the first pair of siblings it computes), but you won’t be able to determine the rooted tree it will produce without knowing more precisely what it does when it recurses. Furthermore, this description does not allow you to know precisely what it will output for larger datasets (with five or more leaves). Therefore, proving that UPGMA will produce the correct tree is challenging in general. However, proving that UPGMA produces the wrong tree is generally easier – since if the first sibling pair it produces isn’t a true sibling pair, the result will be incorrect, no matter what.

5.4 Additive Matrices

Consider the case where we have a tree $T$ with edge weights (or lengths), and we compute a matrix of path distances between all pairs of leaves. Thus, the distance between leaves $x$ and $y$ is the sum of the weights of the edges on the path between $x$ and $y$. For now, consider the special case where all branches must be non-negative, so that we will allow branches to have length 0. When a distance matrix $M_{i,j}$ fits an edge-weighted tree $T$ exactly, in that the distance $M_{x,y}$ equals the path distance (sum of edge weights) in the tree $T$ exactly, the matrix 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.

The reason we are concerned with whether the internal branch lengths are strictly positive is this. If a tree $T$ has an internal edge of length 0, then collapsing that edge results in a tree $T'$ that also fits the matrix perfectly, and which has a node of degree at least 4. This tree $T'$ can then be refined into a binary tree $T'' \neq T$ that also fits the matrix perfectly. In other words, when internal branch lengths can be 0, then there is no unique binary tree that fits the matrix perfectly. On the other hand, if we collapse all the zero-length branches in a tree $T$, we obtain a minimally refined tree $T_0$ that fits the matrix perfectly, and where the weights on the edges of $T_0$ are all positive.

Additive matrices are defined in the literature (27) as matrices corresponding to trees with non-negative edge lengths. For this reason, we continue this usage, but note that additive matrices used in phylogenetic estimation are generally defined by model trees in which all branch lengths are strictly positive. Some of the theory we will develop will assume and exploit the fact that the branch lengths are strictly positive.

Peter Buneman (among others) observed the following property about additive matrices (27):

**Theorem 5 The Four Point Condition:** A $n \times n$ matrix $A$ 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:

- $A_{i,j} + A_{k,l}$
Given four taxa, $A, B, C, D$.

**Step 1:** Compare the three pairwise sums $M_{A,B} + M_{C,D}, M_{A,C} + M_{B,D}$, and $M_{A,D} + M_{B,C}$, 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 $M_{A,B} + M_{C,D}$ has the smallest value, and return the tree $((A, B), (C, D))$.

**Theorem 6** Let $M$ be a $4 \times 4$ additive matrix corresponding to a tree $T$ on four leaves with strictly positive branch lengths. Then the FPM applied to $M$ returns tree $T$.

### Nearly additive matrices.
Suppose we have two $n \times n$ distance matrices $d$ (which may not be additive) and $M$, which is additive. Let $\epsilon = L_\infty(d, M) = \max_{i,j} |d_{i,j} - M_{i,j}|$. Then we will say that $d$ is $\delta_\epsilon$-additive, which expresses the fact that there is an additive matrix $M$ that is within $\epsilon$ of $d$ under the $L_\infty$-metric.

Recall that when $M$ is an additive matrix, then by definition there exists a binary tree $T$ with non-negative edge weights $w : E(T) \to \mathbb{R}^+$ so that $M_{i,j}$ is the path distance between leaves $s_i$ and $s_j$ in the tree $T$.

**Definition 19** 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 for trees with positive branch lengths (i.e., $w(e) > 0$ for all edges $e$), it follows that $f > 0$. However, additive matrices only require that $w(e) \geq 0$, and so $f = 0$ is possible for trees defined by additive matrices. Therefore,
Lemma 3 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 $ij|kl$ if and only if $d_{ij} + d_{kl}$ is strictly less than the other two pairwise sums formed using these four indices (and 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 $d_{ij} + d_{kl}$ is the smallest of the three pairwise sums, under the assumption that $M_{ij} + M_{kl}$ is the smallest of its three pairwise sums.

Note that $M_{ij} + M_{kl} = M_{ik} + M_{jl} - 2F = M_{il} + M_{jk} - 2F$, 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 $d_{ij} + d_{kl}$ and the other two pairwise sums can only be reduced by $4\delta$ (adding $\delta$ to each of $M_{ij}$ and $M_{kl}$ 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 $\delta < f/2$ 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 $L_\infty(d, M) < f/2$.

In other words, the Four Point Method will correctly reconstruct the tree topology $T$ on any four leaves, whenever every estimated pairwise distance is less than $f/2$ from the true pairwise distance (as defined by the edge-weighted tree $T$), where $f > 0$ is the length of the shortest internal edge in $T$.

5.6 The Naive Quartet Method

We now describe a very simple algorithm we call the Naive Quartet Method for estimating trees with $n$ leaves. The input is an $n \times n$ dissimilarity matrix $M$, and we assume $n \geq 5$.

- Step 1: For every four indices $i, j, k, l$, use the Four Point Method on matrix $M$ restricted to the rows and columns for $i, j, k, l$ to compute the tree on $i, j, k, l$ (denoted $t_{ijkl}$). If any quartet of indices has more than one pairwise sum achieving the minimum value, report failure.

- Step 2: Apply the All Quartets Method (from Chapter 3.2) 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 report failure.

We now describe the result of applying the Naive Quartet Method to matrix $M$.

- Case 1: $M$ is defined by an edge-weighted tree $T$ with strictly positive lengths on internal edges. In this case, the Four Point Method will be correct on every four taxa, and so the quartet trees that are produced will be correct. The All Quartets Method applied to the quartet trees computed using the Four Point Method will return $T$. In other words, the Naive Quartet Method will return the tree $T$ defined by $M$ in this case.

- Case 2: $M$ is additive but some of the internal branch lengths for its tree $T$ are zero. Consider a quartet defined by an internal edge (i.e., that have two of their leaves on one side and the other two on the other side of the edge) of length 0. For such a quartet and edge, the three pairwise sums will be identical. In this case, the Four Point Method will return a star, and the Naive Quartet Method will return failure.

- Case 3: $M$ is not additive, but there is an $n \times n$ matrix $D$ corresponding to a tree $T$ with positive branch lengths, and minimum internal branch length $f > 0$, such that $L_\infty(M, D) < f/2$. In this case, the Naive Quartet Method will return the tree $T$, because the Four Point Method will compute quartet trees that are true for $T$, and then the All Quartets Method will combine them correctly into the tree $T$.

- Case 4: All other cases. It is not possibly to definitely state the outcome for the Naive Quartet Method on any input that does not fall into one of the first three cases. However, in general the Naive Quartet Method is
most likely to return failure when it is not very close to a matrix defined by an edge-weighted tree with positive edge lengths. For example, if the set of quartet trees computed by the Four Point Method is incompatible, then the All Quartets Method will return failure. Since set-wise incompatibility of quartet trees is less likely than compatibility, the Naive Quartet Method is highly likely to fail whenever the input matrix $M$ for this case.

However, for the purposes of phylogeny estimation, we will be most interested in the case where the input matrix $M$ is close to an additive matrix corresponding to a model tree whose branch lengths are strictly positive. Thus, for this case, we have the following theorem:

**Theorem 7** Let $M$ be the input $n \times n$ dissimilarity matrix and $D$ be the matrix corresponding to a tree $T$ with positive edge lengths. Assume $L_\infty(d, M) < f/2$, where $f > 0$ is the minimum length of any internal edge of $T$. Then the Naive Quartet Method applied to $M$ will return $T$.

**Proof:** By Lemma 3, 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.

What we have shown is that the Naive Quartet Method will be accurate on a neighborhood of every additive matrix that corresponds to a tree with positive branch lengths. The neighborhood is defined by the $L_\infty$ distance between matrices, and the 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:

**Definition 20** Let $(T, w)$ be an edge-weighted tree defining additive matrix $A$, and let $\Phi$ be a distance-based phylogeny estimation method. Then the 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 $A'$ with the same tree topology as $T$.

We have described the Naive Quartet Method, and proved that its safety radius is $f/2$, where $f$ is the minimum length of any internal edge in the model tree; hence, the Naive Quartet Method is statistically consistent under the CFN model. In fact, the Naive Quartet Method is statistically consistent under a very large set of models. However, the Naive Quartet Method is not used in practice, because most often it just returns failure, except on very small datasets. The value of the Naive Quartet Method is that it illustrates how to prove theorems about methods, and also the limits of these theorems.

### 5.7 Neighbor joining

Neighbor joining (131) is an iterative method, which operates as follows. At the beginning of an iteration, the taxa are partitioned into subsets, and there is a rooted tree for each subset. Initially all taxa are in their own subset, and the taxon is the tree for its subset within the partition. The effect of a single iteration is to select a pair of trees to be merged into a larger rooted tree by making them 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.

First, given the input $n \times n$ dissimilarity matrix $d$, it computes a different matrix $Q$, defined by

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

Then, in each iteration, it finds the pair $i,j$ minimizing $Q_{i,j}$. A new node is created, and the roots of the trees associated to $i$ and $j$ are made children of the new node; this creates a new rooted tree, which replaces the two subtrees that were merged. The partition from the previous iteration is modified by taking the union of the taxa in the subsets corresponding to $i$ and $j$. The $d$ matrix is then updated to show distances between the smaller number of subtrees, and the $Q$ matrix is recomputed based on the new $d$ matrix. Then the next iteration can take place.
We now show how to compute the new $d$ matrix. We will refer to the taxon representing the newly formed subtree (created by making the roots of the trees associated to $i$ and $j$ siblings) as $k$. Then, for all $u \notin \{i, j\}$, we set

$$d_{u,k} = \frac{1}{2} [d_{i,k} + d_{j,k} - d_{i,j}]$$

We then remove the rows and columns associated to $i$ or $j$ from the matrix $d$, and add in the row and column associated to $k$. This results in decreasing the number of rows and columns in $d$ by 1. After the matrix $d$ is modified, the new $Q$ matrix is calculated.

**Theorem 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 distance matrix $d$ with $L_{\infty}(d, D) < f/2$, where $f = \min\{w(e) : e \in E_I(T)\}$ (where $E_I(T)$ 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 (131), but the guarantee that neighbor joining will return the true tree given a dissimilarity matrix sufficiently close to an additive matrix for the true tree comes from (7). 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 going to 1. □

### 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 21** 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 $M$, there is some $L_{\infty}$-neighborhood of $M$ on which the distance-based method $\Phi$ is continuous. That is, for all additive matrices $M$ and any $\delta > 0$, there is an $\epsilon > 0$ such that for all dissimilarity matrices $d$ satisfying $L_{\infty}(d, M) < \epsilon$ it follows that $L_{\infty}(\Phi(d), \Phi(M)) < \delta$.

One of the key techniques used to analyze a distance-based method exploits the fact that if two $n \times n$ additive matrices $A$ and $B$ are close enough (see Theorem 9), then they define the same tree topology. Then, when a distance-based method is continuous on neighborhoods of additive matrices, we can establish conditions under which the method is guaranteed to return the true tree topology.

**Theorem 9** 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 $L_{\infty}(D, D') < f/2$. Then $D'$ corresponds to the same tree $T$ with some other edge weighting $w'$.

**Proof:** $D$ and $D'$ induce the same set of quartet trees and hence they define trees with branch lengths that are topologically identical. □
Theorem 10 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 \( M \) be the additive matrix corresponding to the model tree, and let \( f > 0 \) be the minimum length of any edge in the model tree \( T \). Let \( \delta = f/2 \). Since \( \Phi \) is continuous on some neighborhood of every additive matrix, there is some \( \epsilon > 0 \) so that whenever matrix \( d \) satisfies \( L_{\infty}(d, M) < \epsilon \) then \( L_{\infty}(\Phi(d), \Phi(M)) < f/2 \). By Theorem 9, 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 can be defined by an additive matrix, and all edges in the model tree have positive length. (See Chapter 7 for information about these models.) 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 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 \( A \) that is as close as possible to the input dissimilarity matrix \( d \), with respect to some way of defining distances between matrices. Natural examples of these distances include \( L_1 \), \( L_2 \), and \( L_{\infty} \):

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

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 \( WLS(d, A) = \sqrt{\sum_{ij} \frac{(d_{ij} - A_{ij})^2}{v_{ij}}} \), where \( v_{ij} \) is the variance of \( d_{ij} \). WLS-based approaches were originally suggested in (66; 64).

Yet, finding the best tree (i.e., the best additive matrix) with respect to any of these criteria is NP-hard ((46) showed this for \( L_1 \) and \( L_2 \), and (2) showed this for the \( L_{\infty} \) norm). Agarwala et al. (2) also showed that finding an arbitrarily close approximation is NP-hard, but provided a polynomial time 3-approximation algorithm for the \( L_{\infty} \)-nearest tree. Here we discuss the safety radius (see Definition 20) of an exact and this 3-approximation algorithm for this problem.

Theorem 11 Suppose that \( \Phi \) is an algorithm that solves 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 \((T, w)\), and that \( L_{\infty}(d, D) < f/4 \), 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 \( w' \). Hence, an exact algorithm for the \( L_{\infty} \)-nearest tree problem has a safety radius of \( f/4 \), where \( f \) is the length of the shortest internal edge in the model tree.

Proof: Since \( \Phi \) solves the \( L_{\infty} \)-nearest tree problem, \( D' = \Phi(d) \) is an additive matrix that satisfies \( L_{\infty}(D', d) \leq L_{\infty}(d, D) < f/4 \). By the triangle inequality,

\[ L_{\infty}(D', D) \leq L_{\infty}(D', d) + L_{\infty}(d, D). \]
Therefore,
\[ L_\infty(D, D') < f/4 + f/4 = f/2. \]

By Theorem 9, \( D \) and \( D' \) correspond to the same tree topology. Since the output of the algorithm \( \Phi \) is an additive matrix, it follows that \( \Phi \) can be used to compute the tree \( T \) associated to the matrix \( D \) that is close to the input dissimilarity matrix \( d \).

**Theorem 12** Suppose that \( \Phi \) is an algorithm that yields a 3-approximation to 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 \((T, w)\), and that \( L_\infty(d, D) < f/8 \), 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 \( w' \). Hence, the polynomial time 3-approximation algorithm provided in (2) for the \( L_\infty \)-nearest tree has a safety radius of \( f/8 \).

The proof follows along the same lines as that for Theorem 11, and is omitted.

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 \( A \) minimizing \( \text{dist}(d, A) \)

The problem depends on how distances are computed between two matrices, 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 (54). For example, for the OLS problem (ordinary least-squares), Vach (151) described an \( O(n^3) \) algorithm, and Gascuel (71) and Bryant and Waddell (24) gave \( O(n^2) \) algorithms. Desper and Gascuel (54) also provide formulae for optimal branch lengths under the OLS criterion.

### 5.10 Minimum Evolution

The minimum evolution approach (originally proposed in (90)) 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 (54) 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 above. 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 the weighted \( L_2 \) (weighted least-squares) distances. As discussed above, finding optimal branch lengths of a given tree can be performed exactly in polynomial time for the OLS criterion.

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 whether 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, Bryant, and Denis (72) showed that ME with WLS branch lengths is not statistically consistent. However, a variant of the WLS branch lengths was used in the FastME algorithm, a method that runs in \( O(n^2 \log n) \) time, and that is statistically consistent (54). The updated FastME software (95) has an improved search strategy over the earlier versions. For details on FastME and related techniques, see (54).
5.11 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 8. 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 (153; 31; 30). However, most of the alignment-free methods do not have statistical guarantees under sequence evolution models that include indels. However, (44) provide an interesting method that they prove is statistically consistent for estimating the tree topology under the TKF91 (Thorne-Kishino-Felsenstein 1991) (149) model of sequence evolution in which indels of single nucleotides occurs.

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 9). 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 (41). Supertree methods (see Chapter 6.3), 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 11.2. See also (56; 57; 162; 110; 43; 123; 124) for some afc distance-based methods.

5.12 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?

### 5.13 Homework problems

1. Compute the CFN distance matrix between all pairs sequences in the set \( \{s_1 = 0011011111, s_2 = 0011000111, s_3 = 0011111111\} \)

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 \( H(s, s') \), where \( H(s, s') \) is the Hamming distance between \( s \) and \( s' \) (i.e., \( H(s, s') \) is the number of positions in which the two sequences differ).

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 \( \lambda(e_D) = 3.2 \).

   (a) Prove or disprove: UPGMA on Hamming distances is statistically consistent for estimating the unrooted tree topology for \( T \)

   (b) 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, $\mathcal{L} = \{L_1, L_2, L_3, L_4\}$, 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 $n \times n$ matrices $D$ and $D'$ where $D$ is 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 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 following distance-based method, $\Phi$. Given an additive matrix $D$ corresponding to tree $T$ and edge weighting $w$, the will return the tree $T$ with edge weighting $w$. Given an $n \times n$ matrix $d$ that is not additive, $\Phi(d)$ will return the tree $(s_1, (s_2, (s_3, (\ldots, s_n))))$ with all branch lengths all set to $1$.
   - Consider the two properties we discussed (identity on additive matrices, and continuous on a neighborhood of every additive matrix). Does this method satisfy either of these properties? Both? Neither?
   - If we use the CFN distance correction, and then run $\Phi$ on the resultant matrices, is this a statistically consistent technique for estimating CFN model tree topologies?

33. Consider the function $\Phi$ that maps dissimilarity matrices $d$ to $2d$ (i.e., $\Phi(d)$ is the matrix $d'$ such that $d'_{ij} = 2d_{ij}$).
   - Does $\Phi$ map dissimilarity matrices to additive matrices?
   - How could you use $\Phi$ to estimate the tree topology?

34. 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?

35. Suppose $\Phi$ maps dissimilarity matrices $d$ to $2D$, where $D$ is an optimal solution to the $L_\infty$-nearest tree. 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. Prove Theorem 12.

37. 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?

38. 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?

39. Prove the following: If $C$ is a set of binary characters that evolve without homoplasy on a tree $T$, then the Hamming distance matrix $H(i,j)$ is additive.

40. 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.

41. 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?

42. 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?

43. 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 $S$?

44. 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 $S$?

45. Suppose that $T$ is an unrooted tree that is not fully resolved (i.e., not binary). Thus, some quartet trees in $T$ are star trees, and lack internal edges. Can you use the All Quartets Method to construct $T$ from $Q(T)$? If not, show how to modify the All Quartets Method.

46. Recall how we defined CFN distances. What happens if we have two sequences of length $k$ that are different in exactly $k/2$ positions? What about if they are different in more than $k/2$ positions? 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.)
Chapter 6

Analyzing sets of trees

6.1 Introduction

In this chapter we discuss techniques for analyzing sets of trees. Depending on how the set $X$ of trees was computed, these analyses can have different purposes.

For example, when a maximum parsimony analysis is performed, many equally good trees may be found, all having the same “best” score; in this case, a consensus tree (typically the greedy consensus, or sometimes the majority consensus) is returned. Related to this, when a Bayesian MCMC (see Chapter 7.6) analysis is performed, then a random sample of the trees is examined, and a consensus tree (again, typically the greedy consensus, but sometimes the majority consensus) is returned on this set. The objective in constructing a consensus tree is to identify the features that are either universally shared, or that are mostly shared, among the different trees in $X$. For consensus trees, we require that all the trees in $X$ have the same leafset, $S$, and the consensus tree also has the same leafset.

Although there are many different consensus tree methods, only a few are in common use: the strict consensus, majority consensus, and the greedy consensus (also known as the extended majority consensus); these three are described in Section 6.2. For an entry into the literature on other consensus methods, see (21). There are two other types of analyses of sets of trees, which differ based on their purpose:

- **Supertree construction.** Here we assume that the trees in $X$ are all estimates of the species tree for a different subset of $X$, and the objective is to recover the species tree for $X$ using these estimated subset trees.

- **Agreement subtrees.** The concept of agreement subtrees is similar to the consensus trees, but instead of computing a consensus of the source trees, a common subset of the leafsets of the different trees is sought so that the source trees agree with each other on the subset.

- **Exploring differences among trees.** In this case, our objective is to explore the different phylogenetic signals in a set, rather than to estimate a single tree (the supertree problem), or to characterize the shared signal (consensus tree or agreement tree).

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.

6.2.1 Strict consensus

To construct the strict consensus, we write down the bipartitions that appear in every tree in the input (the “profile”). The tree that has exactly that set of bipartitions is the “strict consensus”. Note that the strict consensus is a contraction
of every tree in the input (though if all the trees are identical, then it will be equal to them all).

**Definition 22** 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, \(C(T) = \cap_{i=1}^{k} C(T_i)\).

### 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” (note that we mean strict majority).

**Definition 23** 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 2** The majority consensus is either equal to the strict consensus, or it refines the strict consensus, since it has every bipartition that appears in the strict consensus.

### 6.2.3 Greedy consensus

We now define the greedy consensus, 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 (160).

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 3** 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**.

**Example 15** 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
As a simple example, if \( T_1 \) and \( T_2 \) are unrooted trees with the same leaf set, since \( T_2 \) itself is a refinement of \( T_1 \) and \( T_2 \), this definition also extends to setwise compatibility:

**Definition 24** Two unrooted trees \( T_1 \) and \( T_2 \) that are on the same leaf set \( S \) are said to be compatible if there exists a tree \( T \) such that \( T \) refines both \( T_1 \) and \( T_2 \).

As a simple example, if \( T_1 \) is a star tree, and so has no internal edges at all, then \( T_1 \) and \( T_2 \) are always compatible.

**Definition 25** A set of trees \( \{T_1, T_2, \ldots, T_k\} \) (all on the same leafset) is compatible if there is a tree \( T \) that refines every tree \( T_i \), \( i = 1, 2, \ldots, k \).

We will often be interested in determining whether a set of trees is compatible, and in computing a common refinement of the set of trees when they are. A tree that is a minimal common refinement of a set of trees is called the compatibility tree.

**Definition 26** Let \( \{T_1, T_2, \ldots, T_k\} \) be a set of unrooted trees all on the same set of leaves. If there exists a tree \( T \) that is a common refinement of all the \( T_i \), then there is also a minimally resolved tree \( T \) that is a common refinement of the set. The tree \( T \) is called the compatibility tree.

**Example 16** As an example, the following trees are compatible:

- \( T_1 \) given by \( C(T_1) = \{(abc|defg)\} \), and shown in Figure 6.1.
- \( T_2 \) given by \( C(T_2) = \{(abcd|efg), (abcde|fg)\} \), and shown in Figure 6.2.

We can see they are compatible, because the tree in Figure 6.3 is a common refinement of each of the trees. However, there are other common refinements of these two trees: for example, see the tree in Figure 6.4.

Now consider the strict consensus of all the common refinements of trees \( T_1 \) and \( T_2 \). What does it look like? What bipartitions must it have? This minimal common refinement of these two trees is called the compatibility tree, and its character encoding is identical to the union of the character encodings of the two trees! Thus, we can construct that minimal common refinement by computing the tree whose character encoding is that union, using the algorithm given in the previous sections.
Figure 6.1: Tree $T_1$

Figure 6.2: Tree $T_2$
Figure 6.3: Tree that is compatible with $T_1$ and $T_2$

Figure 6.4: Another common refinement of $T_1$ and $T_2$
Lemma 4  If a set of trees \( \{T_1, T_2, \ldots, T_k\} \), each leaf-labelled by the same set \( S \) of taxa, has a compatibility tree \( T \), then \( C(T) = \bigcup_i C(T_i) \).

More generally, to see if a set of trees is compatible, we write down their bipartition sets, and then we apply the algorithm for constructing trees from bipartitions to the union of these sets. This will produce the compatibility tree, if it exists. If it fails to construct a tree, it proves that the set is not compatible.

Theorem:  A set \( T = \{T_1, T_2, \ldots, T_k\} \) of trees on the same leaf set is compatible if and only if the set \( \bigcup_i C(T_i) \) is compatible.

6.3 Supertrees

6.3.1 Why supertree construction is important

Supertrees are constructed from sets of smaller trees, and these constructions are based on many different criteria and using many different methods. To understand the various methods, it helps to consider the reasons that supertrees are constructed.

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 10.3. 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.

In Chapter 3 we discussed methods for constructing trees from very small source trees (i.e., rooted triples or unrooted quartet trees). When the input is a set of rooted source trees, then each source tree can be encoded by its set of rooted three-leaf trees (called “rooted triples”, or “triplet trees”). Similarly, when the input is a set of unrooted source trees, then the source trees can each be encoded by its set of unrooted quartet trees. In each case, methods for constructing supertrees from these rooted triples or unrooted quartet trees can then be used. In the remainder of this chapter, we discuss supertree methods that are designed explicitly for larger source trees.

Because the input to a supertree method will generally be estimated trees, they will likely have some estimation error. This is true whether the source trees are rooted or unrooted, and whether the source trees are supposed to be species trees or gene trees. Therefore, rather than trying to find a compatibility supertree (see Section 3.3), 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.

In the supertree literature (see, for example, (19)), the input to the supertree problem (i.e., the set of source trees) is called a profile. We will use that term here as well. For the rest of this chapter, we will assume that the profile given as input to the supertree problem is \( T = \{t_1, t_2, \ldots, t_k\} \), and each \( t_i \) is unrooted. We will let \( S_i \) denote the leafset of tree \( t_i \), and we let \( S = \bigcup S_i \). The objective is a supertree \( T \) (i.e., a tree on leafset \( S \)), typically one that has an optimal score with respect to some optimization problem. Nearly always the optimization problems are NP-hard, and so finding
the best supertree will depend on the specific heuristics used to solve the problem. Thus, supertree methods are, like most problems in phylogenetics, understood both by the theoretical properties of the optimization problems on which they are based, also by the details of their implementations, and finally by how well they perform on data in terms of finding good solutions to their optimization problems. Finally, although most of these optimization problems are NP-hard, there are special cases that can be solved in polynomial time, and some of the divide-and-conquer strategies we will see that utilize supertree methods to combine trees on smaller subsets specifically aim to create instances of the supertree problem that can be solved in polynomial time (e.g., see (114)).

6.3.2 Matrix Representation with Parsimony (MRP)

We begin with the Matrix Representation with Parsimony (MRP) (11) supertree problem, by far the most well known (and most popular) supertree optimization problem. MRP has variants depending on whether the input trees are rooted or unrooted; here we define the version where the input trees are unrooted. We begin with the Matrix Representation with Parsimony (MRP) (11) supertree problem, by far the most well known

6.3.2 Matrix Representation with Parsimony (MRP)

We begin with the Matrix Representation with Parsimony (MRP) (11) supertree problem, by far the most well known (and most popular) supertree optimization problem. MRP has variants depending on whether the input trees are rooted or unrooted; here we define the version where the input trees are unrooted. We begin with the Matrix Representation with Parsimony (MRP) (11) supertree problem, by far the most well known (and most popular) supertree optimization problem. MRP has variants depending on whether the input trees are rooted or unrooted; here we define the version where the input trees are unrooted.

Recall that the input to MRP is a profile \( \{t_1, t_2, \ldots, t_k\} \) of unrooted trees, that \( t_i \) has leafset \( S_i \), and that \( S = \bigcup S_i \). 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 \( t_i \).

The matrix for source tree \( t_i \) on taxon set \( S_i \subseteq S \), has a row for every element of \( S \) and a column for every internal edge in \( t_i \). To define the column associated to the internal edge \( e \) in \( t_i \), we observe 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 assign ? to every \( s \in S - S_i \). Thus, if \( |S| = n \), then each edge in \( t_i \) is represented by a \( n \)-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 \( MRP(T, T) \), the MRP score of a tree \( T \) with respect to the profile \( T \). Then, \( MRP(T, T) = \min\{MP(T, M') : M' \in \mathcal{M}\} \). 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.

**Theorem 13** Let \( T = \{t_1, t_2, \ldots, t_k\} \) be a compatible set of source trees (i.e., a set of trees that has a compatibility supertree). Then the MRP matrix defined on this input has a perfect phylogeny (see Chapter 4.1). Hence, if maximum parsimony is solved exactly on the MRP matrix, then any solution it finds will be compatible with all the source trees and will be a compatibility supertree.

Thus, although finding the compatibility supertree is an NP-hard problem when the sets \( S_i \) are allowed to be different from each other, using MRP we are able to solve this NP-hard problem. In other words, if MRP is solved exactly, then the MRP solution determines if the input trees are compatible, and returns the compatibility supertrees if they exist.

6.3.3 Matrix Representation with Likelihood (MRL)

The MRL problem (introduced in (116) is nearly identical to the MRP problem, with the following change: the same MRP matrix is computed, but then the optimization problem is the optimal solution to maximum likelihood under the CFN model, treating ?'s as missing data.
6.3.4 Robinson-Foulds Supertrees

The objective of the Robinson-Foulds Supertree problem (9; 35; 34; 92), 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 begin by defining the RF distance in this setting between a source tree $t_i$ and a supertree $T$.

We modify $T$ by removing all the leaves other than those in $S_i$; this creates an unrooted tree that could have some nodes of degree two. We then suppress all the nodes of degree two, and obtain a binary tree $T|S_i$ that is only on leafset $S_i$. Finally, we now compute the RF distance between $T|S_i$ and $t_i$.

**Example 17** Let $T = (1, (((6, 7), (4, (3, 5, 2))), (5, (3, 2), (6, 7))))$ and $t_i = (5, (3, 2), (6, 7)))$. To compute the RF distance between $T$ and $t_i$ we modify $T$ by deleting the taxa that are not in $i$, and then suppressing nodes of degree two; this gives us $T|S_i$, which is $((6, 7), (3, (5, 2)))$. The non-trivial bipartitions of $T|S_i$ are $(235|67)$ and $(25|367)$. The non-trivial bipartitions of $t_i$ are $(235|67)$ and $(23|567)$. The RF distance between $T$ and $i$ is therefore 1.

The RF supertree ($T_{RFS}$) is the tree that minimizes $\sum_{i=1}^{k} RF(T|S_i, t_i)$, i.e.,

$$T_{RFS} = \arg\min_{T} \sum_{i=1}^{k} RF(T|S_i, t_i).$$

Finding the RF supertree is NP-hard, since the compatibility supertree (if it exists) would be an optimal solution to the RF supertree problem, and determining whether a profile of trees is compatible is itself NP-complete. Local search heuristic methods, such as PluMiST (93), Robinson-Foulds Supertrees (10), and MulRF (34), were developed to attempt to construct the Robinson-Foulds Supertree.

6.3.5 Maximum Quartet Support Supertrees

In Section 3.4, we described optimization problems for constructing trees from a set of quartet trees, which could be given with weights, and the objective is to construct a tree that maximizes the total (weighted) quartet tree support. This problem can be extended to the supertree context, where it can be very useful.

We begin by defining the quartet support between a tree $t_i$ and a larger tree $T$, under the assumption that the leafset $S_i$ of $t_i$ is a subset of the leafset $S$ of $T$. For a given leaf-labelled tree $t$, we denote by $Q(t)$ the set of quartet trees of $t$. Hence, $Q(T)$ is the set of quartet trees for $T$ and $Q(t_i)$ is the set of quartet trees for $t_i$. We define the quartet support of $t$ for $T$ to be $|Q(t) \cap Q(T)|$. Then, the quartet support of a profile $T = \{t_1, t_2, \ldots, t_k\}$ 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 (142).

6.3.6 SuperFine

SuperFine (147) is a “meta-method” that can be used with a preferred supertree method. SuperFine operates in two steps: first, the source trees are combined into a constraint tree (which tends to be incompletely resolved) using the “Strict Consensus Merger”, and then this tree is resolved into a binary tree. The resolution step is obtained by resolving each high degree node (i.e., polytomy) in the strict consensus merger tree, one by one. To resolve a polytomy of degree $d$, SuperFine creates a new set of source trees, each with at most $d$ leaves, and then applies the base supertree method to this set of smaller source trees. The supertree that is constructed on the smaller source trees is then used to refine the polytomy. As shown in (147), using SuperFine with standard heuristics for MRP in PAUP* (148) and TNT (74) as the base method improved the topological accuracy of the constructed supertrees. Furthermore, SuperFine has been used with heuristics for MRL (116) in RAxML (139) and FastTree-2 (119), and also with Quartets Max Cut (QMC) (136),
and shown to improve the topological accuracy and running times of the constructed supertrees (116; 146). Thus, SuperFine is a divide-and-conquer technique that has been shown to improve the accuracy and speed of a number of different supertree methods.

6.3.7 Popular supertree methods

In the previous section, we presented a few of the optimization problems that have been used to construct supertrees: Matrix Representation with Parsimony (MRP), Matrix Representation with Likelihood (MRL), Robinson-Foulds Supertree (RFS), and Maximum Quartet Support (MQS). Each of these optimization problems is formulated in terms of unrooted source trees, but have equivalent formulations for rooted source trees. Other optimization problems have been posed, including MinCutSupertree (134), Modified MinCutSupertree (118), MinFlip Supertree (36; 37) and the MRF Supertree (28).

Nearly all the optimization problems are NP-hard, since the special case where all the source trees are compatible would allow the set of compatibility trees to be the set of optimal solutions to the optimization problem, and determining if that set is empty or not is an NP-complete problem. Hence methods for finding optimal solutions to the supertree optimization problems have generally relied upon local search heuristics to find good solutions. In addition, quartet-tree amalgamation methods and triplet-tree amalgamation methods can be repurposed as supertree methods by encoding the source trees as sets of quartet trees (for unrooted source trees) or as sets of rooted triplet trees (for rooted source trees). For example, the Quartets MaxCut method (136) has been studied as a supertree method in (146).

Supertree methods with an explicitly statistical orientation include (4; 5; 128).

6.4 Agreement subtrees

The two main techniques of this type are maximum agreement subtree (MAST) and maximum compatibility subtree (MCST).

6.4.1 Maximum Agreement Subtree

The maximum agreement subtree (MAST) problem is as follows. Given a set \( T \) 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 \( T \) agree on \( X \) (i.e., they all induce the same tree on \( X \)).

The MAST problem was posed in (65). 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 (140), 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 (60). Polynomial time algorithms for three or more bounded-degree trees were presented in (6), along with an NP-hardness proof for MAST on three unbounded degree rooted trees.

6.4.2 Maximum Compatibility Subtree

The objective of the maximum compatibility subtree (MCST) problem 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. This is the MCST problem. (The MCST problem is also referred to as the maximum refinement subtree problem, or MRST, (78).)

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 (78), 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. Polynomial time fixed parameter algorithms to compute the MCST of three or more trees were presented in (69). Algorithms for approximating the size of the complement were presented in (70).

6.5 Exploring differences among trees

Sometimes the objective in exploring a set of trees, each of them computed on the same dataset, is to determine the different phylogenetic histories they suggest. For example, some datasets are formed by running techniques (typically heuristics) for maximum likelihood, maximum parsimony, or other optimization problems, and the set of all optimal or near-optimal trees found during the search is then explored. If that set clearly clusters around a single topology, then the most likely hypothesis is that single tree, and a consensus tree (such as a greedy consensus tree or majority consensus tree) for that set is likely to be a good estimate of the tree. However, when the set contains two or more very different clusters of trees, then no single consensus tree and no single “best tree” is likely to be a good estimate of the true tree. For cases like these, techniques that can explore sets of disparate trees and represent the different phylogenetic signal in the data are helpful.

6.6 Further reading

6.6.1 Statistical aspects of supertree estimation

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 (143; 26; 47; 128; 40).

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 (143) (see discussion in (26)). Therefore, the local search heuristics for the RF Supertree that have been developed, such as PluMiST (93), Robinson-Foulds Supertrees (10), and MulRF (34), can be considered heuristics for the ML Supertree problem.

6.7 Review questions

1. Suppose you have ten trees on the same leafset.
   • 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. Define the MRP problem, and explain how to write down the MRP matrix.

4. 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?

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

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

7. 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?

8. 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 \(T'\)?

6.8 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 \(T\) 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 \(T\)
   • The majority consensus and an arbitrary tree in \(T\)

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.
(a) For each unrooted tree below, draw the tree, and write down the bipartitions that are false positives and false negatives with respect to $T_0$.

- $T_1 = (f, (g, (a, (b, (c, (d,e))))))$.
- $T_2 = (g, (f, (c, (d, (e, (a,b))))))$.
- $T_3 = (g, (f, (a, (b, (c, (d,e))))))$.

(b) 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 $T$ 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 $T$. 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. Suppose that you have a set $T$ of unrooted trees defined as follows. You take an unrooted binary tree $T$ on leafset $S$, and then you add a new leaf $x$ into $T$ in each of the possible positions to create a new unrooted binary tree on leafset $S \cup \{x\}$.

- How many trees can you create in this fashion (i.e., how big is $|X|$)? Express this as a function of $|S|$.
- What is the strict consensus of $T$?
- What is the majority consensus of $T$?
- What is the maximum topological distance (using the RF distance) between any two trees in $T$?

10. 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 $X$?

11. Suppose you have 1000 trees, each with 100 leaves, and 5000 taxa overall. How big is the MRP matrix?

12. Consider the caterpillar tree $T$ on 6 taxa, and let $T$ 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 $T$?

13. 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.

14. 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?
Chapter 7

Statistical gene tree estimation methods

7.1 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.

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.

7.2 Models of site evolution

The Cavender-Farris-Neyman model: We briefly define the Cavender-Farris-Neyman (CFN) model of binary sequence evolution. Site evolution down the tree is i.i.d., and the root has state 0 or 1, each with probability 0.5. The state of a site can change on edge e, and the number of changes is a Poisson random variable with expected value \( \lambda(e) \).

As we discussed in Chapter 1, a CFN tree on four leaves can be estimated from its sequence dataset using the Four Point Method, which is a very simple distance-based method. We now discuss how to estimate CFN trees for larger numbers of sequences.

Recall the Naive Quartet Method we described for estimating trees from distance matrices. In this method, we showed that if we had an additive matrix \( \lambda \) of pairwise distances between the leaves in a tree (and in which the branch weights are all positive), we could calculate the tree topology correctly using a quartet-based method (i.e., we compute the tree on every quartet, and then we combine all the quartet trees). Since the CFN model tree definition is that the expected number of changes of a site is always strictly greater than 0, this means that its additive matrix satisfies the
assumptions given above. Hence, the Naive Quartet Method can construct the model CFN tree \( T \) given the additive matrix of model distances.

We also showed that if the input distance matrix was sufficiently close to an additive distance matrix, then we would return the tree associated with the additive distance matrix. (Specifically, if \( f > 0 \) is the length of the shortest internal edge in the tree \( T \) and \( L_\infty(d, \lambda) < \frac{f}{2} \), then the Naive Quartet Method applied to matrix \( d \) will also return tree \( T \).)

Recall that we said that \( d_{i,j} \) converges in probability to \( \lambda_{i,j} \) for all \( i, j \) as the sequence length increases; this means that for any \( \epsilon > 0 \), there is a sequence length \( K \) so that the distance matrix \( d \) will satisfy \( L_\infty(d, \lambda) < f/2 \) with probability at least \( 1 - \epsilon \) given sequence length at least \( K \). Hence, given sequences of length at least \( K \), then with probability at least \( 1 - \epsilon \), the Naive Quartet Method will reconstruct the true model tree \( T \).

There are other distance-based phylogeny estimation methods for which we also know they will reconstruct the true tree whenever the distance matrix \( d \) is within \( f/2 \) of the additive matrix defining the model tree. For example, this theorem was proved by Kevin Atteson (7) for Neighbor Joining (131), one of the most frequently used distance-based methods for estimating phylogenetic trees in biology. See also the result for the \( L_\infty \)-nearest tree, and the 3-approximation to this problem, in Chapter 5.9.

**The Jukes-Cantor model:** We now discuss statistical models of sequence evolution for DNA sequences. Because there are four nucleotides, A,C,T, and G, 4-state Markov models are used to describe nucleotide sequence evolution.

The Jukes-Cantor (JC) model is the 4-state version of Cavender-Farris, and so models the evolution of a single site. Thus, the state at the root 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 JC 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 \( \lambda(e) = -\frac{4}{3}ln(1 - \frac{4}{3}p(e)) \), 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 \( \lambda(e) > 0 \).

Note that the JC model only addresses how a single site evolves; therefore, to extend the JC model to sequence evolution, we assume that all the sites evolve under the same process, and independently. Hence, \( \lambda(i,j) \), the expected number of changes of a random site on the path \( P_{i,j} \) between \( s_i \) and \( s_j \) in the model tree \( T \), can be estimated by

\[
D_{i,j} = -\frac{3}{4}ln(1 - \frac{4}{3}\frac{H(i,j)}{k}).
\]

This is called the **Jukes-Cantor distance correction.** As for the CFN model, Jukes-Cantor trees can be estimated using the Naive Quartet Method (and other distance-based methods such as neighbor joining), applied to the Jukes-Cantor corrected distances.

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). 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, (32)). Therefore, most estimation methods are based on models that enforce this assumption.

However, the third assumption – that all nucleotide substitutions are equiprobable – can be relaxed without losing identifiability, and there are many other sequence evolution models that have been developed. These different models vary in what they relax, and are discussed in most textbooks on molecular evolution (such as (81; 96)), and on the web (e.g., at (52)). We describe just a few of these models below, focusing on ones that are commonly used in phylogenetic inference (GTR) or in simulation studies (GTR and K2P).
**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 pyrmidine. Thus, the K2P model has two free parameters, while the JC model has only one free parameter. The K2P and JC models are often used in simulation studies, but not as frequently used in phylogenetic estimation.

**The General Time Reversible Model:** The General Time Reversible (GTR) model is the most commonly used nucleotide sequence evolution model for phylogenetic estimation purposes. The GTR model assumes there is a $4 \times 4$ substitution matrix $M$ that defines the probability of changing from one nucleotide to another nucleotide. This matrix can be quite general, but must satisfy some additional constraints in order to ensure identifiability. For the GTR model, statistically consistent distance-based tree estimation is also possible using the logdet distance correction (141).

**The General Markov Model:** A more general model is the General Markov Model (141), which allows each edge $e$ to have its own substitution matrix $M(e)$ that satisfies the same constraints as for the GTR model. Tree estimation under the General Markov model can be performed using distance methods, using the logdet distance correction (as described in (141)). 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., $M(e) = M(e')$ 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.

Some of the popular amino acid models are the JTT (87) and WAG (163) models, but there are others as well. To choose between different amino acid models, a biologist generally uses a statistical test, such as prottest (1), that evaluates the fit of each of the different models to the biological dataset he or she wishes to analyze. Since not all amino acid models are nested, the choice between models is often made using the Akaike Information Criterion (AIC), Bayesian Information Criterion (BIC), or Decision Theory Criterion (DT) scores.

### 7.3 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 the case of Cavender-Farris-Neyman and Jukes-Cantor models, for example, the unrooted tree topology is identifiable, the substitution probabilities are identifiable, but the location of the root 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, 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.

### 7.4 Calculating the probability of a set of sequences

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. Specifically, 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 and where the characters have \( r \) states (i.e., \( r = 2 \) 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 nodes in the tree. 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. 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 \( r = 2 \).

We can do this more efficiently by using a simple dynamic programming algorithm, quite similar to the algorithm used for maximum parsimony. This dynamic programming approach yields an \( O(r^2 n) \) 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.

### 7.5 Maximum Likelihood

Maximum likelihood phylogeny estimation is possibly the most common 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 parameters 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 numerical parameters. Maximum likelihood estimation for the CFN model takes as input a set \( S \) of sequences, each of the same length, that are assumed to have evolved down some unknown CFN model tree \( (T_0, \theta_0) \) and seeks the CFN model tree \( (T^*, \theta^*) \) that maximizes the probability of generating the observed sequence data (i.e., maximizes \( \text{Pr}(S|T^*, \theta^*) \)). The "maximum likelihood score" of the tree \( T \) is then \( \sup_\theta \{ \text{Pr}(S|T, \theta) \} \), and so the objective is to find the tree \( T \) with the highest maximum likelihood score.

Note that 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.)

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, note that it is possible that \( T_0 \neq T^* \) and \( \theta_0 \neq \theta^* \). In other words, 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.

Maximum likelihood thus seeks not only the unrooted tree \( T \) but also the best parameter values on the tree. 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 (JC) model would seek the JC 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 JC 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.

One thing to note, therefore, is that 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.

Maximum likelihood (ML), if run exactly (so that optimal solutions are found), is also statistically consistent under the Jukes-Cantor model, as well as under the Generalized Time Reversible (GTR) model and General Markov (GM) model; see (33). However, finding an optimal ML tree is an NP-hard problem (122), and so heuristics are used instead of exact solutions.

### 7.6 Bayesian methods

Bayesian methods, by definition, are based upon parametric models of evolution. They are also “likelihood-based” in that they calculate likelihoods of trees based upon that explicit parametric mathematical model of evolution. They differ from maximum likelihood methods in that they do not attempt to estimate the parameters of the evolutionary model in order to maximize the probability of producing the data. Instead, 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 character data for each model tree it visits, and then accepting the new model tree if the probability is larger. If the probability is smaller, then the new model tree may be accepted with some non-zero probability (but less than 1). Bayesian methods thus have to operate for a very long time, doing many proposals, until “stationarity” is reached. When the MCMC chain reaches stationarity, a sample of the model trees it visits is then taken, and the summary statistics of that sample are calculated. These give estimates of the support for the various parameters of the evolutionary process. Mostly, however, these summary statistics are used to estimate support for the different branches of the estimated tree(s).

Bayesian methods, if run properly (i.e., so that they reach the stationary distributions) are known to be statistically consistent under the Jukes-Cantor model, as well as under more general models (such as the General Time Reversible model) (144). The challenge is being able to run long enough to reach stationarity.

### 7.7 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 asserted (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 these models of evolution? What about under the Cavender-Farris-Neyman model?

Here the story is not so positive. Joe Felsenstein (62) 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 MP 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:** $(u = v = 0 \text{ and } w = x = 1)$ or $(u = v = 1 \text{ and } w = x = 0)$. 

99
• 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 \(i\) will have cost \(1\) on \(T_i\) and cost \(2\) on the other trees. Hence, to understand how MP 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 MP will return with probability increasing to 1 as the number of sites increases is \(T_2\). However, the model tree is actually \(T_1\)!

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 MP and MC 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 MP and MC 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.

### 7.8 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 (150) 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. Tuffley and Steel called this the No Common Mechanism Model (NCM).

Consider a simple version of this model where we have binary sequences, and so every edge \(e\) and site \(i\) has a probability of change, \(p(e, i)\). The state at the root is randomly selected from \(\{0, 1\}\) with equal probability.

**Theorem 14** Let \(S\) be a set of \(n\) binary 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.

See (150) for proof.

Note that the NCM model for binary sequences contains the CFN model as a special case. Therefore, if we evolve sequences under a CFN model tree for which maximum parsimony is not statistically consistent, and then estimate ML trees under the No Common Mechanism model, we will not provably converge to the true model tree as the sequence length increases. In other words, Tuffley and Steel showed:

**Theorem 15** Maximum likelihood under the NCM model is not statistically consistent.

### 7.9 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 (80), which showed model conditions and sequence lengths under which some statistically inconsistent methods were more accurate than some statistically consistent methods! However, some studies have provided mathematical analyses of the amount of data that are needed to obtain good accuracy with high probability and expressed these in terms of the model tree parameters (56; 57).

7.10 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 $A \mid S - A$ 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 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 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.
7.11 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.

**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.

**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, and 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 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.

### 7.12 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 (JC) model. What is a JC model tree? What is the JC distance correction?
5. State the Generalized Time Reversible (GTR) model.
7. What is the relationship between the GM, GTR, and JC 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 JC 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 JC 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 JC model. Is this statistically consistent?
17. Is maximum parsimony statistically consistent under the JC 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?

### 7.13 Homework problems

1. Prove or disprove: Every GTR model tree is a JC model tree.

2. Prove or disprove: Every JC model tree is a GTR model tree.

3. Prove or disprove: Every CFN model tree is a JC 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 JC model, then it is also statistically consistent under the GTR model.

7. Consider the following algorithm for estimating JC model trees from sequence data. Given a set of sequences, we compute JC 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 JC 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 \( e_1 \), and we set \( p(e_1) = .4 \), 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 \( Pr(S \mid (T, P)) \)?

- \( \Theta(nk) \)
- \( \Theta(2^n k) \)
- \( \Theta(2^k n) \)
- \( \Theta(n^2 k) \)

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:

(a) Of the three parsimony-informative character patterns, identify which one(s) would appear most frequently for tree \( T_1 \).

(b) Of the three parsimony-informative character patterns, identify which one(s) would appear most frequently for tree \( T_2 \).

(c) For each of these model trees, do you think maximum parsimony would be statistically consistent? Why?

(d) For each of these model trees, do you think UPGMA on CFN distances would be statistically consistent? Why?

(e) 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 \( e \).

(a) 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?

(b) Same question as above, but suppose the dataset consisted of four sequences \( A, B, C, D \) of length 10, where

- \( A = 0100100111 \)
- \( B = 0000000000 \)
- \( C = 0010101001 \)
- \( D = 0000000000 \)
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 JC model tree, is this a statistically consistent method?)

17. Recall the Cavender-Farris-Neyman (CFN) model, and consider three methods: maximum likelihood under CFN, maximum parsimony, and UPGMA on CFN distances.

   (a) 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.

   (b) 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 \( p(e_I) = 0.4 \) 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.

   (a) 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?
   - 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?

   (b) 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.)

   (c) 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?

(d) 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.

(e) 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?
Chapter 8

Multiple sequence alignment

8.1 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 a pair of sequences that are related by evolution can be “aligned”, using the evolutionary process that separates them.

Figure 8.1 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; furthermore, the two nucleotides may be identical (e.g., both $A$, or both $T$, etc.), or they may be different (e.g., one $A$ and one $T$). When a column contains only one nucleotide, then the other entry is a dash (−), representing the case of an indel. Interpreting whether the indel is the result of an insertion or a deletion, however, depends on knowing the ancestral state; if it is occupied by a nucleotide then the dash corresponds to a deletion, whereas if it is occupied by a dash, then the sequence containing a nucleotide is the result of an insertion. Another way of interpreting a pairwise alignment is that it defines a binary relation on the nucleotides in the sequences, with each column containing two nucleotides being one element of the binary relation.

The pairwise alignments relating sequences at the endpoints of edges in a tree define the true multiple sequence alignment on the set of sequences at the leaves, in a natural way: 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; hence, if the true history is known for every edge in the tree, then the transitivity of the binary relations defined for each edge of the tree yields the true multiple sequence alignment.

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 9.2 for this two-phase process.

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
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 8.1: 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!)
Two-phase tree estimation:
First align, then construct the tree

S1 = AGGCTATCACCTGACCTCCA  S1 = -AGGCTATCACCTGACCTCCA
S2 = TAGCTATCAGCCGCGC  S2 = TAGCTATCAGCCGCGC
S3 = TAGCTAGCCGCGC  S3 = TAGCTAGCCGCGC
S4 = TCACGACCGACA  S4 = --------TCACGACCGACA

S1
S4
S2
S3

Figure 8.2: 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.

alignments can result in quite different theoretical and empirical performance.

8.2 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 \( S = \text{AAT} \) and \( S' = \text{CAAGG} \). There are several transformations of \( S \) into \( S' \) of minimum cost. For example, we could insert \( C \) in front of \( s \), obtaining \( S_1 = \text{CAAT} \). We could then change the \( T \) into a \( G \), obtaining \( S_2 = \text{CAAG} \). Finally, we could add one more \( G \), obtaining \( S' = \text{CAAGG} \). 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 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 8.1.

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, \( S = \text{AATTAGATCGAATTAG} \) and \( S' = \text{CATTAGATTGAACAATTATACA} \)? 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. The first one we will introduce is the Needleman-Wunsch algorithm (113), which addresses the problem we described above (the “global alignment” of two sequences).

8.2.1 Needleman-Wunsch

The dynamic programming algorithm we will present in this section is essentially the same as the one proposed by Needleman and Wunsch (113), 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 8.2? 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 27** Let $S$ be a string of length $n$, with $S = s_1s_2s_3 \ldots s_n$, so that $s_i$ is the $i^{th}$ letter in $S$. The $i^{th}$ prefix of $s$ is the string $S_i = s_1s_2 \ldots s_i$, 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 8.1, the sites indicate the following events that transform sequence $S = AAT$ into sequence $Y = CAAGG$
the third case involves 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 \( i^{\text{th}} \) prefix of \( X \) and \( Y_j \) denotes the \( j^{\text{th}} \) 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 \( \text{Cost}(i, j-1) \), \( \text{Cost}(i-1, j) \), and \( \text{Cost}(i-1, j-1) \). 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 Cost(i, j - 1), Cost(i - 1, j) and Cost(i - 1, j - 1), the we can set Cost(i, j) to be the minimum of the three possible costs (using the above analysis). In other words, we set

\[
Cost(i, j) = \min\{Cost(i - 1, j - 1) + Hamming(x_i, y_j), Cost(i - 1, j) + 1, Cost(i, j - 1) + 1\}
\]

**Filling in the DP matrix.** We need to compute Cost(i, j) for all 0 ≤ i ≤ n and 0 ≤ j ≤ m. We can compute these entries in any way we like, as long as we don’t try to compute 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 ≤ i ≤ n and 1 ≤ j ≤ m, Cost(0, j) = j and Cost(i, 0) = i.
- For i = 1 to n DO
  - For j = 1 to m DO
    * Cost(i, j) = \min\{Cost(i - 1, j - 1) + Hamming(x_i, y_j), Cost(i - 1, j) + 1, Cost(i, j - 1) + 1\}
  - Return Cost(n, m)

Note that how 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 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 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 Cost(i, j) to the box that gave you the smallest value. For example, if Cost(i, j) was set to be Cost(i - 1, j - 1) + 1, then put an arrow from Cost(i, j) to Cost(i - 1, j - 1). Then, at the end of the computation, there will be a path from Cost(n, m) all the way back to 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 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 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.

**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.
Table 8.3: A multiple sequence alignment on three sequences with SOP score 8, where each indel or substitution has unit cost

| s1 | A | - | - | C |
| s2 | A | T | A | C |
| s3 | C | - | A | G |

Table 8.4: A second multiple sequence alignment on the same three sequences from Table 8.3

| s1 | - | - | A | C |
| s2 | - | A | T | A | C |
| s3 | C | A | - | - | G |

8.2.2 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, the Needleman-Wunsch algorithm can be modified to fit this setting.

8.3 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.

**Sum-of-Pairs Cost.** Given the cost function $c$ and a multiple sequence alignment $A$ for a set $S$ of $n$ sequences $s_1, s_2, \ldots, s_n$, we extend the cost as follows. First, we look at the induced pairwise alignments $A_{ij}$ defined by $A$ on each pair $s_i$ and $s_j$. Each of these has a cost $c_{ij}$, as defined by $c$. Then, we add up all these costs to obtain the cost of the multiple sequence alignment $A$. This sum-of-pairs cost is also referred to more simply as the SOP cost.

As an example, consider three sequences $s_1, s_2, s_3$ in a multiple sequence alignment given in Table 8.3. Suppose that every insertion, deletion, and substitution has unit cost. Then, examining the multiple sequence alignment in Table 8.3, 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. Question: is this an optimal multiple sequence alignment for this set of three sequences? Finding the best multiple sequence alignment for a set of $n$ sequences, based on a given cost function $c$, is NP-hard (156). 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.

### 8.3.1 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 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 18** Supposed the set of sequences is $S = \{s_1, s_2, s_3\}$, where $s_1 = ATA$, $s_2 = AAT$, and $s_3 = CAA$, 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 19** As a second example, suppose the input is $s_1 = AC$, $s_2 = ATAC$, and $s_3 = CAG$, 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 $s_2$ 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 optimal 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 (77; 132; 133) allows internal nodes to be labelled by arbitrary strings, and is NP-hard even for simple gap penalty functions (156; 159). In contrast, the maximum parsimony problem on a fixed tree can be solved exactly in polynomial time using dynamic programming (see Chapter 4.6). Thus the Tree Alignment problem is harder than the maximum parsimony problem. However, approximate solutions can be found (158; 155; 157), which have bounded error; for example, (155) 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.

### 8.3.2 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 $S$:
Lemma 5 \hspace{0.3em} Let \( w \) be the Minimum Spanning Tree algorithm and let \( w \) be the weight of edge \((s, s')\) the edit distance between \( s \) and \( s' \).

Proof: \hspace{0.3em} Note that \( w \) the triangle inequality, it follows that \( w \) vertices. For example, if \( P \) is a four-tuple for the distribution of nucleotides in the earlier positions.

Second column of the profile, etc. In each case, the generation of the nucleotide for a given position does not depend on the nucleotide in the earlier positions.

8.4 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.

8.4.1 Profiles

We begin with how multiple sequence alignments define profiles. Given a multiple sequence alignment (MSA), it is straightforward to compute the distribution of nucleotides present in a given position within the MSA. Thus, for each position \( i \), we can associate a 4-tuple giving the proportion of the times each of the four nucleotide appears. Thus, an MSA with \( k \) sites (none of which is entirely gapped) defines a profile of length \( k \), with the \( i^{th} \) position occupied by the four-tuple for the distribution of nucleotides in the \( i^{th} \) site. In Table 8.5, we show a single MSA with five sequences and its associated profile.

8.4.2 Gap-free Hidden Markov Models

Given a profile, we can consider the distribution defined by the profile. So, imagine you generate a sequence of the same length as the alignment (i.e., a DNA sequence of length 4), in which the first position draws its nucleotide from the distribution in the first column of the profile, its second position draws its nucleotide from the distribution for the second column of the profile, etc. In each case, the generation of the nucleotide for a given position does not depend on the nucleotide in the earlier positions.

117
We can construct a graphical model to generate sequences under this distribution, too. There is a start state (also called a “begin” state) and an end state. 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 \( A \) to \( B \) 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.

### 8.4.3 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 are of length 10, and 10 of length 7. Furthermore, 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” to 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.

One of the interesting consequences of this quite general model is that even if we know the graphical model – we cannot know for sure the sequence of states that were taken to generate any given sequence. That is – we might have followed a path that only used the match states, or we might have used some combination of matches, insertions, and deletions, to produce the given sequence.

However, if we know the model and its associated transition probabilities and emission probabilities for the insertion and match states, we can at least make an informed guess about the most likely path through the model for a given sequence.

8.4.4 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 matrix, it is easy to compute the probability of any sequence generated by the path. 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, we can compute the probability of generating a given sequence by multiplying the probabilities of generating each letter in the sequence (as defined 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. We may wish to know the overall probability of generating a sequence, which would involve summing probabilities over all the different paths, or the most likely path that would 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.

Given a profile HMM and a sequence, we are therefore interested in two different questions:

- Which path through the model is most likely to have generated the sequence?
- What is the probability of generating a given sequence?

Each of these problems can be addressed using dynamic programming.

8.5 Popular methods for multiple sequence alignment

8.6 Quantifying error in estimated multiple sequence alignments

Suppose \( S \) is a set of sequences and \( A \) is the reference alignment on the set \( S \). For example, \( A \) could be the true alignment, which is known to us either 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 \( \mathcal{A} \) as the “true” alignment, and evaluate error of estimated alignments by comparing them to \( \mathcal{A} \).

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 \( S = ACATTA \) has three copies of \( A \) and these are not considered identical. Now suppose that \( S' = TACA \) and that the true alignment between these \( S \) and \( S' \) is the alignment shown in Table 8.6. Suppose we have an estimated alignment, shown in Table 8.7.

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 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 (105).

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 28** Let \( \mathcal{A} \) be the reference multiple sequence alignment, and \( \mathcal{A}' \) 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. Treating \( \mathcal{A} \) as the reference alignment and \( \mathcal{A}' \) as the estimated alignment:

- **The SPFN rate** is the fraction of the truly homologous pairs (in \( \mathcal{A} \)) that are not present in \( \mathcal{A}' \); this is the false negative rate.

- **The SPFP rate** is the fraction of the homologous pairs in \( \mathcal{A}' \) that are not present in \( \mathcal{A} \); 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.
8.7 Divide-and-conquer alignment strategies

8.8 The impact of multiple sequence alignment on tree estimation

8.9 Co-estimation of alignments and trees

8.10 Current challenges

8.11 Review questions

8.12 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 8.2. 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 \( s' = TTAGGC \). 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 \( S = ATGTATA \) and \( S' = TAGTAGC \), 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. Consider the gap-free alignment given for sequence dataset \( s_1 = AACTAAG, s_2 = AATATAG, s_3 = ATAAAAG, s_4 = TTATTAG, \) 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.)
- What is the probability of generating sequence \( AACTAAG? \)
- What is the probability of generating sequence \( CTAAAAG? \)

10. Suppose I have an unknown Profile Hidden Markov Model, and I use it to generate a collection of sequences.

- Suppose the first 10 sequences I generate are all the same length. Can I infer from this that the model doesn’t have any insertion or deletion states?
- Suppose that number was 100?
- Suppose the number was 3?

11. Suppose you have a profile HMM with insertions always of only single indels – and so the graph has no self-loops. Suppose you want to compute the most probable path through the model. Note, in this problem you do not have a sequence \( s \) and so the question is independent of the sequence. Prove or disprove: the most probable path is \( X_0, X_1, X_2, \ldots, X_n \), where \( X_0 \) is the start state, \( X_n \) is the end state, and \( X_i \rightarrow X_{i+1} \) is the transition edge leaving \( X_i \) with highest probability. (In other words, do you think a greedy algorithm would find the most probable path?)

12. Give an example of a sequence \( s \) and a profile HMM without self-loops, where the most probable path is not the one that is most likely to generate the sequence.
Chapter 9

Constructing species trees under the multi-species coalescent model

9.1 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.
9.2 Theoretical foundations

9.2.1 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 \( l(e) \rightarrow \infty \), the probability of two lineages coalescing on the edge approaches 1, while as \( l(e) \rightarrow 0 \) the probability of two 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?

9.2.2 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 (49; 129; 50; 130). 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 16** 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.

### 9.2.3 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) 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 (125) 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.

9.3 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.

9.3.1 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 16, 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 $T = \{t_1, t_2, \ldots, t_k\}$ be the gene trees we observe. How can we estimate $T$ given $T$? 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 $T$ to $A$. This defines a set of rooted gene trees, each defined only on the set $A$ of species, which we denote $T_A$. By Theorem 16, 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 $T_A$. 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 algorithms (such as the ASSU algorithm (3)) 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.** 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 $T$.
- **Step 2:** Apply the ASSU algorithm (3) to the set $T$ of rooted three-leaf gene trees.
  - If $T$ is compatible, then the ASSU algorithm outputs a tree $T$, which we return.
  - Else $T$ is not compatible, and the ASSU algorithm does not return a tree; in this case, we output “Fail”.

126
Theorem 17 Let $T$ be a model rooted species tree on $n \geq 3$ 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) (101), STEM (91), STAR (102), and GLASS (108), are all summary methods that estimate species trees from rooted gene trees.

9.3.2 Summary methods that use unrooted 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.

Recall Theorem 16, 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 $T$.
- Step 2: Apply the All Quartets Method from Chapter 3.2 to the set $X$.
  - If the set $T$ of four-leaf trees is compatible, then the All Quartets Method will return a tree $T$, which we return.
  - Else the set $T$ 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 18 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 $\epsilon > 0$, 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 $1 - \epsilon$. 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 (104; 106) and BUCKy-pop (the population tree computed by BUCKy) (94), 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 \( T = \{ t_1, t_2, \ldots, t_k \} \), and each gene tree is on taxon set \( S \). Let \( d(T,t_i) \) be the number of four-taxon subsets of \( S \) on which \( T \) and \( t_i \) differ, and let \( d(T,T) = \sum_i d(T,t_i) \). Then the quartet-median tree of \( T \) is the tree \( T \) that has the smallest total quartet distance to \( T \); i.e.,

\[
T_{\text{median}} = \arg\min_T d(T,T).
\]

So consider the optimization problem that takes as input the set \( T \) 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 19** Let \( T \) be a set of \( k \) true gene trees and let \( T^* \) be the true species tree. Then for any \( \epsilon > 0 \) 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 \( 1 - \epsilon \).

**Proof:** Note that the quartet-median tree \( T \) of \( T \) maximizes \( \sum_{t \in T} \text{Sim}_q(T,t) \), where \( \text{Sim}_q(T,t) \) is the number of four-taxon subsets of \( S \) on which \( T \) and \( t \) agree. Note then that \( \sum_{t \in T} \text{Sim}_q(T,t) = \sum_X N(T,X) \), where \( X \) is a four-taxon subset of \( S \) and \( N(T,X) \) is the number of trees in \( T \) that agree with \( T \) on set \( X \). Finally, for any \( \epsilon > 0 \), there is a \( K > 0 \) so that given \( k > K \) genes then with probability at least \( 1 - \epsilon \) the dominant gene tree on any set \( X \) of four taxa will be equal to the species tree on \( X \). For such datasets, the true species tree \( T^* \) is the quartet-median tree of \( T \), since \( \text{Sim}_q(T^*,t) \) is the maximum possible for all \( t \). The proof follows. \( \Box \)

Hence,

**Corollary 2** 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 (85). 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 (104; 106; 23), 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 \( T \) 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 \( T \), 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 \( X = \bigcup_{t \in T} C(t) \) then \( |X| = O(nk) \). Furthermore, as shown in (104; 106; 23), the optimal constrained quartet-median tree can be found in time that is polynomial in \( n = |S|, k = |T|, \) 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 20 (From (106)) : Let $T$ be a model species tree, and let $T$ be a set of gene trees generated at random by $T$. If $\bigcup_{t \in T} \mathcal{C}(t) \subseteq X$, 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 (100).

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 21 (From (100)): 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 (100) was the first method to use such an approach, and was based on the neighbor joining (131) 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 (152) showed that using FastME (53) improved the topological accuracy of the species tree. Furthermore, when the matrix of internode distances has missing data, using the BioNJ* from (41) also results in improved accuracy. Thus, ASTRID (152), 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 (152), 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 estimation 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 (126) for a discussion of this issue.
9.4 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 (25), METAL (42), SMRT (super-matrix rooted triples) (48), and SVDquartets (38). 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 22** Let $T$ be a model species tree, let $T$ 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.

9.5 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 (79). As shown in (14), 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.

9.6 Estimating species trees in practice

9.6.1 Popular methods

The methods described above have included some simple theoretically motivated methods that are easily shown to be statistically consistent (e.g., SRSTE and SUSTE), as well as some sophisticated statistical methods, such as *BEAST, for co-estimating species trees and gene trees. However, in practice, species trees tend to be estimated using summary methods, due to their speed and ease of use. There is a great interest in using *BEAST, due to the potential for improved accuracy through the co-estimation paradigm. However, *BEAST is very computationally intensive, and so is not used as much on large datasets.

9.6.2 Algorithmic design techniques to improve running time and accuracy 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 (14), 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 (166) 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 (166), 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 (106). An iterative divide-and-conquer strategy was developed to improve the scalability of MP-EST to large numbers of taxa in (12). 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 (114) 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 a supertree method (e.g., SuperFine+MRL (147; 116)) to combine the smaller estimated species trees together. After iterating a few times, the quartet support (the optimality criterion used in ASTRAL) is computed for each of the different species trees, and the species tree with the best quartet support is returned. As shown in (12), “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.

9.7 Review questions

9.8 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 $\epsilon > 0$.

   • 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 $\epsilon$). For what values of $\epsilon$ 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 $\epsilon$). For what values of $\epsilon$ is this greater than 0.5?

   • Compute the probability of the rooted gene tree $t_3 = ((a, (b, d)), (c, b))$ under the multi-species coalescent (this will be a function of $\epsilon$). For what values of $\epsilon$ is this greater than 0.5?

2. Consider the model species tree from the previous problem. Prove or disprove: $\exists \epsilon > 0$ 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 $\epsilon > 0$. Prove or disprove: $\exists \epsilon > 0$ 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:
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 \( T \) 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 \( T \) (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 \( T \) 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 \( T \) 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 \( T \) 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 \( T \) 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?

\[ T = \{ ((a, (b, (c, d))), (a, (c, (b, d))), (b, (a, (c, d))), (b, (c, (a, d))) \} \]
Chapter 10

Designing methods for large-scale alignment and phylogeny estimation

10.1 Overview

10.2 Algorithm design for large-scale multiple sequence alignment estimation

10.2.1 Managing long sequences

10.2.2 Managing large numbers of sequences

10.2.3 Aligning alignments

10.2.4 Adding sequences to alignments

10.2.5 Ensembles of Hidden Markov Models

10.3 Algorithm design for large-scale phylogeny estimation

10.3.1 Heuristic search strategies

TBR, SPR, NNI, and p-ECR moves

Randomness

The ratchet

10.3.2 Using divide-and-conquer and/or iteration

DACTAL was originally described as a method for estimating a tree from unaligned without needing to align the full dataset. Figure 10.3.2 shows how the same framework can be used as a generic technique to enable computationally intensive methods to be applied to large datasets.
Figure 10.1: 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. 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.
Figure 10.2: DACTAL: divide-and-conquer trees (almost) without alignments. The input is a set of unaligned sequences. In the first step, the dataset is decomposed into overlapping small subsets of a desired size. Then alignments and 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 sequence dataset into subsets using the tree, computes alignments and trees on subsets, and combines the subset trees using the supertree method.

10.3.3 Adding sequences to trees: phylogenetic placement
10.3.4 Fast likelihood calculations
10.3.5 Fast distance-based optimization
10.3.6 Alignment-free (and almost alignment-free) phylogeny estimation
10.4 Review questions
10.5 Homework problems
Chapter 11

Advanced topics in biological phylogenetics

In this chapter we present some advanced topics in phylogenetics, including phylogenetic placement, “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, computing forests rather than trees.

11.1 Phylogenetic placement

11.2 “Fast-converging methods”

11.2.1 Sequence length requirements

11.2.2 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.

In this section we describe a more sophisticated technique for computing trees from dissimilarity matrices that can be accurate even when the NQM method will not be. The foundation of the approach is the concept of a short quartet in an edge-weighted tree.

**Definition 29** Let $T$ be a binary tree and $w : E(T) \to R^+$ 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 $Q_{\ast\text{short}}(T)$.

**Example 20** 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

$$Q_{\text{short}}(T) = \{\{1, 2, 3, 4\}, \{2, 3, 4, 5\}, \{1, 3, 4, 5\}, \{3, 4, 5, 6\} \ldots \{96, 97, 98, 99\}, \{96, 97, 98, 100\}, \{97, 98, 99, 100\}\}$$
a set with 99 quartets. The set of short quartet trees of $T$ is the set of quartet trees induced in $T$ by these quartets. 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, the short quartet trees on $12|34$ and $23|45$ can only be combined into one tree on five leaves, which is $(1, (2, (3, (4, 5)))$. Then, adding in $34|56$ to this tree uniquely determines $(1, (2, (3, (4, (5, 6))))).$ Thus, if the short quartet trees are processed in a proper order, the tree $T$ can be reconstructed, and it is clear that no other possible tree can be constructed that is consistent with these quartet trees.

In other words, the following theorem can be proven:

**Theorem 23** Let $T$ be a 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 $Q_{\text{short}}^*(T) \subseteq T'$, then $T' = T$.

This theorem was proven in (56), 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 (56) and the witness-antiwitness method (57), 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. This issue is revisited in Chapter 11.2, where we develop the mathematical framework for evaluating the sequence lengths that suffice for accuracy with high probability.

We describe the Dyadic Closure Method here, to establish why the short quartet trees of a tree can be used to construct the tree. The basic idea is to repeatedly compare two trees at a time from the input set, to see if any additional quartet trees can be inferred from the quartet trees (58). The arguments we use to infer additional quartet trees assume that the set $Q \subseteq Q(T)$ for some tree $T$ – in other words, we assume the quartet trees in $Q$ are true quartet trees (for some tree $T$), and we want to find other quartet trees that must be true for $T$, if $Q$ only contains true quartet trees. Note that we will consider $ab|cd$ to be the same quartet tree as $ba|cd, ba|dc$ and $ab|dc$. Consider the following three rules for inferring additional quartet trees from a set $Q$ of quartet trees:

- **Rule 1:** If $ab|cd$ and $ac|de$ are in $Q$ (which is assumed to be a subset of $Q(T)$), then $ab|ce, ab|de$ and $bc|de$ are also in $Q(T)$. 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 $Q$ (which is assumed to be a subset of $Q(T)$), then $ab|de$ must be in $Q(T)$. Hence, if $ab|de$ is missing from $Q$, we can add $ab|de$ to $Q$.

It should be 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. There are also higher order rules whereby quartet trees can be inferred by combining three or more quartet trees, such as the following:

- **Rule 3:** If $ab|cd, ab|ef$ and $ce|df$ are in $Q$ (which is assumed to be a subset of $Q(T)$), then $ab|df$ is also in $Q(T)$. Hence, if $ab|df$ is missing from $Q$ then we can add $ab|df$ to $Q$.

The Dyadic Closure Method, however, only needs the first two rules – and hence is able to be reasonably computationally efficient.

Given a set $Q$ of quartet trees, apply Rules 1 and 2 to every pair 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 $cl_2(Q)$. In (58; 56), the following theorem was proven:

**Theorem 24** If $Q_{\text{short}}^*(T) \subseteq Q \subseteq Q(T)$ for some tree $T$, then $cl_2(Q) = Q(T)$.
In other words, $T$ is the true tree, then the assumption of the theorem states that the set $Q$ of quartet trees has no incorrect quartet trees (i.e., $Q \subseteq Q(T)$) but also contains all the short quartet trees (i.e., $Q_{\text{short}}(T) \subseteq Q$). Under these two assumptions, the theorem states that the two rules will never add any incorrect quartet trees into $Q$, and that the dyadic closure will contain the correct quartet tree on every four leaves.

**Corollary 3** If $Q \subseteq Q(T)$ for some tree $T$, then $\text{cl}_2(Q) \subseteq Q(T)$. Hence, if $\text{cl}_2(Q)$ contains two or more trees on any quartet, then $Q \not\subseteq Q(T)$.

Based on the theorem and corollary, a simple algorithm for constructing a tree from a dissimilarity matrix $d$ was developed in (56).

**Dyadic Closure Method** We describe a simple version of the dyadic closure method presented in (58).

- **Input**: $n \times n$ dissimilarity matrix $d$
- **Output**: a tree $T$ or else fail.

- For each $x \in [d_{ij}]$,
  - 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 $\text{cl}_2(Q_x^*)$ and call it $Trees_x$.

- If there is an $x \in [d_{ij}]$ such that $Trees_x = Q(T)$ for some tree $T$, then return $T$. Else, return Fail.

As shown in (56), 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 (56), the Dyadic Closure Method runs in $O(n^5 \log n)$ time, where the input matrix is $n \times n$.

### 11.2.3 Other fast-converging methods

#### 11.3 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 (86; 112; 111; 165; 164), 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 (107), 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 (107)):

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 (76; 84; 107) for some textbooks about phylogenetic networks.
11.4 Genome rearrangements

11.5 Genome-scale multiple sequence alignment

11.6 Phylogenetic forests
Chapter 12

Constructing trees and networks for historical linguistics

12.1 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.

12.2 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 some 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’). Longer-term 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.

12.3 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. However, standard approaches are not reliable. For example, assuming that if a state is shared by more than half the known languages, it will be shared by the language at the root, is not accurate. Knowing the tree, however, can help with these estimations. For example, in the Indo-European tree, under the assumption that Anatolian is the first child of Proto-Indo-European (PIE), then the only estimations that can be reliably made are for those characters for which some Anatolian has one state and some other non-Anatolian language also has that state. Then that state must also be exhibited at the root, unless the character evolved homoplastically.

12.4 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. But, just as the strong 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.

12.5 Character-based estimation

Maximum parsimony and maximum compatibility can also be used to estimate language trees. Maximum compatibility is motivated by the idea that properly selected and coded characters ought to be compatible on the true tree, assuming there is a true tree (as opposed to a network in which taxa evolve with borrowing as well as with genetic descent). This idea follows from the selection of characters that are unlikely to evolve with homoplasy. And while
all characters can exhibit homoplasy, especially if there are mistakes in character encoding (that is, the assignment of character states), some characters are less likely than others. Thus, maximum weighted compatibility is also a relevant optimization problem in linguistic phylogenetics, where the different characters are weighted based on their resistance to being transferred laterally.

Maximum weighted compatibility

**Input:** Matrix $M$ as above, but with characters given with positive weights, $c_1, c_2, \ldots, c_k$.

**Output:** Tree $T$ on the set of taxa so as to maximize the sum of the weights of the compatible characters on $T$.

It is clear that the assessment of the relative probability of homoplasy involves a great deal of linguistic expertise and, of course, personal opinion. Thus, assigning weights to characters is best done by a linguist skilled in the language family. (Assigning states to taxa for different characters also takes linguistic expertise, for that matter!)

As with Maximum Compatibility, weighted maximum compatibility is optimized on binary trees. Thus, any heuristic for solving weighted maximum compatibility need only examine completely resolved trees.

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 are 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.

Maximum compatibility is an approach used in linguistic phylogeny, and is based on the assumption that in the absence of lateral transfer, that properly selected and coded linguistic characters will be compatible on the true evolutionary tree. However, selecting the correct characters can involve some effort. The process of identifying and removing problematic characters, and then repeating the phylogenetic analysis makes sense from a linguistic point of view, but presents several challenges. First, sometimes the dataset is large, making this process a potentially very long one. Second, the identification of problematic characters in itself involves a great deal of expertise, and unless the identification of these characters is based upon solid linguistic grounds, the removal of problematic characters may simply lead to reinforcement of the linguist’s biases.

### 12.6 Gray and Atkinson’s approach

A very different type of approach to phylogeny estimation for languages was presented by Gray and Atkinson (75). 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.

**Binary encoding of multi-state characters** 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.
12.7 Geoff Nichols’ approach: Bags of words

12.8 Perfect phylogenetic networks

12.9 Models of linguistic character evolution

Evans and Warnow proposed a parametric model of linguistic character evolution in (161).

12.10 Performance on data

12.11 Controversies

12.12 Review questions

12.13 Homework problems
Appendices
Appendix A

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.

A.1 Discrete mathematics

A.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 \( < a, b > \) or \( a \rightarrow b \) to indicate the directed edge (also called an **arc**) from \( a \) to \( b \) in a digraph.

### A.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 \( < a, b > \in R \) means that \( a \) divides \( b \) without remainder. Hence, \( < 3, 6 > \in R \) but \( < 3, 5 > \notin R \), and \( < 6, 3 > \notin R \). 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.

### A.1.3 Hasse Diagrams and partially ordered sets

A **partial order** is a binary relation \( R \) on a set \( S \) satisfying

- \( < A, B > \in R \) and \( < B, C > \in R \) implies that \( < A, C > \in R \).
- \( < A, A > \in R \) for all \( A \in S \)
- \( < A, B > \in R \) and \( < B, A > \in R \) 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 **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 \( < z, y > \in R \).

### A.1.4 Equivalence relations

A binary relation \( R \) on the set \( S \) is said to be an **equivalence relation** if it satisfies the following properties:

- \( < a, a > \in R \) for all \( a \in S \); this is called the **reflexive property**.
- If \( < a, b > \in R \), then \( < b, a > \in R \); this is called the **symmetric property**.
- If \( < a, b > \in R \) and \( < b, c > \in R \), then also \( < a, c > \in R \); this is called the **transitivity property**.

\(^{1}\)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 (154)
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 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.

A.1.5 Transitive Closure

Given a binary relation $R$ on a set $S$, the transitive closure of $R$ is the binary relation $R^+$ that is obtained by repeatedly adding $<A, B>$ to $R$ whenever $R$ contains both $<A, C>$ 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.

A.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).

A.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.

A.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 25 \ 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 \( 2 \times 3^{1-1} = 2 \) 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. \( \square \)

A.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 26 \ 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. \( \square \)

A.3 Running time analysis

A.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 \( \leftarrow \)). You will need to have symbols that express comparisons of variables (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 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. Graph representations include adjacency matrices (that use \( \theta(n^2) \) to represent the graph) and adjacency lists (that use \( \theta(m + n) \) to represent the graph). 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.

A.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 running 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.

A.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
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:
can be solved in polynomial time, we are interested in whether we can
decision problems, since they are problems in \( \text{NP} \).

A.6 The \( \text{NP} \)-complete problems

A.6.1 Introduction

Informally, the \( \text{NP} \)-Complete problems are the hardest problems in \( \text{NP} \). As a result, \( \text{NP} \)-complete problems are
decision problems, since they are problems in \( \text{NP} \), and \( \text{NP} \) is a set of decision problems. But what do we mean by
“the hardest problems in \( \text{NP} \)?”

Saying that a problem \( X \) is at least as hard as any problem in \( \text{NP} \) means that if \( X \) could be solved in polynomial
time, then every problem in \( \text{NP} \) 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 \( \text{NP} \)-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 \( \text{NP} \) 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 show 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.

### A.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 \( X \)
- 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 \( X \)

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 \( poly(n) \) vertices, where \( n \) is the number of vertices in \( G \) and \( poly(n) \) 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.

\subsection{A.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 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.

\section{A.8 General algorithm design techniques}

\subsection*{A.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 \( O(n^2) \) time and \( O(n^3) \) 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.

\subsection*{A.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.

A very simple example of a dynamic programming algorithm is one that computes the \( n \text{th} \) 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 \textit{defined recursively}. To compute the \( n \text{th} \) Fibonacci number, we design a dynamic programming (DP) algorithm, as follows.

The input to the problem is \( n \), and the output is the \( n \text{th} \) Fibonacci number \( F(n) \). We let the \( j \text{th} \) 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
The algorithm computes these values in increasing order for \( j \), and stores the results in an array \( F[i] \). We first compute \( F[3] \), then \( F[4] \), etc., until we obtain \( F[n] \). To compute \( F[i] \) given all the previously computed values, we set \( F[i] \) to the sum of the previous two values. Thus, computing \( F[i] \) takes constant time, as long as we calculate \( F[i-1] \) and \( F[i-2] \) before we compute \( F[i] \). 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.


For \( j = 3 \) up to \( n \) DO:


End(For)

Return \( F[n] \)

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 \( F[1] \) and \( F[2] \)). Then, every time the loop is entered, we use four operations (two to look at the values of \( F[j-1] \) and \( F[j-2] \), one operation to add those values, and then one operation to set the value of \( F[j] \)). The loop is entered \( n - 2 \) times. Hence, the total time is \( O(n) \), where \( n \) is the input.

### A.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

\[
t(n) = Cn + t(n - 1)
\]

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:

- If \( n \) is not a positive integer, return Null
• Else, if \( n \leq 2 \) return 1
• 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 \( t(n) > g^n \) 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!)

### A.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 \( O(n \log n) \).

### A.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!

### A.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. 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 = \( \frac{TN}{FP+TN} \)

### A.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 A.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.
   
   (a) 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.
   
   (b) What properties does this relation always hold, independent of the graph \( G \)?
   
   (c) 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 \( y \).
   (a) Prove that \( R \) is an equivalence relation.
   (b) 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 A.3.2, by finding the positive constants \( C, C' \).

18. Sort the following functions in terms of their asymptotic behavior (from slowest growing to fastest): \( \log n, n^2, n \log n, \log(n^4), n^4 - 500n^3, n!, 5n^2, n^n, 3^n, (\log n)^3, 1.1^n, 2n^2 \)

19. 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.

20. 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.

21. Use the fact that maximum clique is NP-hard to prove that maximum independent set is NP-hard.

22. 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?

23. Design a DP algorithm, and provide a running time analysis, that computes the longest increasing subsequence in an input array of \( n \) integers.

24. Design a DP algorithm, and provide its running time analysis, that computes the longest common subsequence given two arrays of integers.

25. 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.)
26. 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?

27. 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 $n > 2$

28. Find and prove (using induction) a closed form formula for $t(n)$ (as a function of $n$), for $t(n)$ defined as follows:

- $t(1) = 5$
- $t(n) = t(n - 1) + 3$ for $n > 1$

29. Find and prove (using induction) a closed form formula for $t(n)$ (as a function of $n$), for $t(n)$ defined as follows:

- $t(1) = 2$, $t(2) = 2$
- $t(n) = 5t(n - 1)$ for $n > 1$

30. 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 $n > 1$

31. 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.)

32. 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 $w[1..2n-1]$ 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 $1 \leq i \leq n$.)

33. 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 $s_k$.

34. 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 $Longest[1..2n-1]$, 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 $n + 1 \leq i \leq 2n - 1$.

35. 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:
(a) 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 \( \text{Winner} \), where \( \text{Winner}[i,j] \) 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.

(b) Give the matrix for \( \text{Winner} \) where \( p = q = 5 \).

36. 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.

(a) 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 \( \text{Winner} \), where \( \text{Winner}[i,j] \) 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.

(b) Give the matrix for \( \text{Winner} \) where \( p = q = r = 3 \).

37. Design an exact algorithm for maximum clique and analyze its running time.

38. Design an exact algorithm for maximum independent set and analyze its running time.

39. Design an exact algorithm for three-colorability and analyze its running time.

40. Design an exact algorithm for Hamiltonian Path and analyze its running time.

41. 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?

42. 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 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 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 B

Guidelines for evaluating 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 rules here all aim to get at one or more of the following basic objectives: (a) clarity of exposition (so that both you and the reader understand what you’ve done and can draw correct inferences from the data), (b) reproducibility (so that the experiment can be performed by someone else, using the exact same methods and data), (c) rigor (so that what you infer makes sense), and (d) scientific relevance (so that what you generate is relevant to some real data analysis). This is only a start, though, and a very brief introduction to these ideas.

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 C

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 A, guidelines for evaluating computational methods, before starting your project. Not all the Rules may be 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?

Long projects

Read Appendix A, guidelines for evaluating computational methods, before starting your project. Not all the Rules may be 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 $M$. 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 distances, and also evaluate multiple ways of computing trees on the resultant distance matrices. Compare the resultant phylogenies to each other.

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 (97)), 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 (39) 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 A, guidelines for evaluating computational methods, before starting your project. Not all the Rules
may be 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 likeli-
hood. 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 in a dataset. Multiple
sequence alignment and phylogeny estimation methods assume that all the sequences in the input are homolo-
gous, 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* (79) 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.
Bibliography


