Extracting and Exploiting Signals in Genetic Sequences

A thesis presented in partial fulfilment

of the requirements for the degree of

Doctor of Philosophy

in Mathematics

at Massey University

Walton Timothy James White

2011

Abstract

As DNA databases continue to grow at an exponential rate, the need for more efficient solutions to basic problems in computational biology grows ever more pressing. These problems range from the principal questions driving evolutionary science—How can we accurately infer the history of genes, individuals and species? How can we separate the signal from the noise in our data? How can we visualise that signal?—to the purely practical: How can we efficiently store all this data? With these goals in mind, this thesis mounts a computational combination attack on a variety of topics in bioinformatics and phylogenetics:

- A program is designed and implemented for solving the Maximum Parsimony problem—in essence, finding phylogenetic trees having the fewest mutations. This program generally outperforms existing highly optimised programs when using a single CPU, and unlike these earlier programs, offers highly efficient parallelisation across multiple CPUs for further speedup.

- A program is designed and implemented for compressing databases of DNA sequences. This program outperforms general-purpose compression by taking advantage of the special "treelike" structure of DNA databases, using a novel data structure, the "leaky move-to-front hashtable", to achieve speed gains.

- A data visualisation technique is introduced that concisely summarises the "treelikeness" of phylogenetic datasets on a ternary plot. Each dataset is represented by a single point, allowing multiple datasets, or multiple treatments of a dataset, to be displayed on a single diagram.

- We demonstrate problems with a standard phylogenetic analysis methodology in which a single tree is assumed *a priori*. We argue for a shift towards network methods that can in principle reject the hypothesis of a single tree.

- Motivated by a phylogenetic problem, a fast new algorithm is developed for finding the mode(s) of a multinomial distribution, and an exact analysis of its complexity is given.

Acknowledgements

I would like to thank my supervisors, Mike Hendy, David Penny and Barbara Holland, for their support (both financial and moral), guidance and patience during my candidature. I feel extremely lucky to have had the chance to work with such all-round high-calibre people. Several years ago, in a combination attack of their own, these three gradually persuaded me (against my own (worse) judgment) to upgrade from a Masters degree to a PhD. I'm grateful that they knew me well enough, and thought it worthwhile, to keep nudging me until I realised what was good for me!

Many people have had a positive impact on my life during the course of my PhD. A somewhat biased random sample includes:

Past and present members of the Allan Wilson Centre at Palmerston North, whom I have found to be without exception friendly, intelligent and helpful.

My flatmates at the venerable 53 Te Awe Awe St. estate—Matt, Rachel, Michael, both Karens, Richard and Scott—for providing a thoroughly enjoyable place to live, where I always felt comfortable, *even* when Richard was around. As always, it seems fitting to mention that Scott knows a guy who once slept *on a door*.

Tony Dale of the BlueFern supercomputer group, for always responding quickly and helpfully when something went wrong.

The loose association of People Who Have Dinner and Play Board Games on Saturday Nights—to a first approximation, Rogerio, Klaus, Simon, Gillian, Bennet, Barbara and Robin—for tolerating my sense of humour, and occasionally affording me the chance to repay their gastronomic generosity with a few least significant bits of my own.

My good friend Sylvia, for many things, including for teaching me that people from very different backgrounds can find the same kinds of people extremely funny-looking.

Jing, for being an ideal officemate. Trish, for being a tireless force for good. Joy, for returning my wisecracks with interest.

Everyone over the years who helped me *test the graphics cards* on the computers in the Allan Wilson Centre.

And of course my family, whose love and support has been a constant in my life.

Contents

Abstract i

Acknowledgements ii

1 Introduction **1**

1.1 Overview 1

1.2 Common Themes 2

1.2.1 Strengthening the Fundamentals 3

1.2.2 Information: A Common Currency 3

1.2.3 Combination Attacks 3

1.3 Background: Phylogenetic Inference 7

1.3.1 Fundamentals 7

1.3.2 Starting points for inference 8

1.3.3 Algorithmic treebuilding approaches 9

1.3.4 Optimality criteria 9

1.3.5 Search methods 11

1.3.6 Heuristic algorithms 12

1.3.7 Statistical models 12

1.3.8 Errors and consistency 14

1.3.9 Maximum Parsimony and consistency 14

1.4 Computational Complexity Primer 15

1.4.1 NP Completeness 16

2 Faster Exact Maximum Parsimony Search with XMP **20**

2.1 Introduction 20

2.2 Correctness of Steiner Tree Lower Bounds for Ambiguous Nucleotides 30

2.2.1 Without Ambiguous Nucleotides 30

2.2.2 With Ambiguous Nucleotides 31
2.3 Zero-length Edges 32
2.3.1 Sufficiency of Examining Fully Refined Trees 33
2.3.2 Avoiding the Deluge 33
2.3.3 Minimally Refined Trees 35
2.3.4 XMP Tree Representation 35
2.3.5 Identifying ml0 Edges 37
2.3.6 Contracting ml0 Edges 37
2.3.7 A Counterexample 38
2.4 Zharkikh's Rules 39
2.5 Visualising MPI Communication 40
2.6 Lower Bound Comparison 41

3 Compressing DNA Sequence Databases with coil 45

3.1 Introduction 45
3.2 Pentium IV Optimised `find_edges` 61
3.3 Sequence Buffering System 63
3.4 Data Compression as Quick and Dirty Science 64
3.5 General-purpose vs. Specialised Compression 66
3.6 Erratum 66

4 Treeness Triangles: Visualizing the Loss of Phylogenetic Signal 67

4.1 Introduction 67
4.2 Why does distance correction amplify residual signals? 69
4.3 Clarification 70

5 A Bias in ML Estimates of Branch Lengths in the Presence of Multiple Signals 82

5.1 Introduction 82
5.2 Testing Robustness 83
5.2.1 Does "the" internal edge of a mixture of two trees really exist? 83
5.2.2 Shared parameter values 84
5.2.3 The edges of a mixture model 84
5.3 Later Developments 85

5.4 Connection to Multinomial Modes 86

6 A Fast and Simple Algorithm for Finding the Modes of a Multinomial Distribution 91

6.1 Introduction 91

6.2 Motivation and Connection to Phylogenetics 91

6.2.1 The Problem with PAUP* 92

6.2.2 Choosing a Best Representative 93

6.2.3 An Alternative to Least Squares 94

6.3 Alternate Proof of Correctness 95

6.4 Problem Instances 97

7 Conclusion 105

7.1 XMP 105

7.2 COIL 107

7.3 Treeness Triangles 108

7.4 ML Bias 109

7.5 Multinomial Modes 110

7.6 How Much Optimisation is the Right Amount? 110

7.7 Other Directions 112

7.8 Summary 112

A Work Breakdown 113

A.1 XMP 113

A.1.1 Author Contributions 113

A.2 COIL 116

A.2.1 Author Contributions 116

A.2.2 Previously Examined Work 116

A.3 Treeness Triangles 121

A.3.1 Author Contributions 121

A.4 ML Bias 123

A.4.1 Author Contributions 123

A.5 Multinomial Modes 125

A.5.1 Author Contributions 125

Chapter 1

Introduction

1.1 Overview

This thesis addresses a selection of topics in bioinformatics and phylogenetics that are unified by a focus on effective computational techniques. Each topic is presented in a chapter of its own that contains a paper published in a peer-reviewed journal. Briefly, they are:

- Chapter 2: `xmp` is a fast new program for exactly solving the Maximum Parsimony problem using multiple CPUs in parallel. On a single CPU, `xmp` generally outperforms existing highly optimised programs such as PAUP* and TNT; a carefully implemented *work stealing* parallelisation strategy enables efficient use to be made of many CPUs, even on distributed-memory systems such as clusters of networked PCs.

- Chapter 3: `coil` is a program for achieving high-level compression of DNA sequence databases. `coil` beats general-purpose compression programs like `gzip` by taking advantage of the special "treelike" structure of DNA databases, using a novel lossy data structure, the "leaky move-to-front hashtable", to achieve speed gains. Note: As elaborated in section A.2.2 of appendix A, part of the work described in this chapter has been previously examined.

- Chapter 4: A Treeness Triangle is a novel data visualisation method that concisely summarises the "treelikeness" of phylogenetic datasets on a ternary plot. Each dataset is represented by a single point, allowing multiple datasets, or multiple treatments of a dataset, to be displayed on a single diagram.

- Chapter 5: It is common practice to assume a single tree gave rise to all sequences in a dataset, and then attempt to estimate this tree using Maximum Likelihood. We demonstrate how this can lead to biased inferences, and argue for a shift towards network methods that can in principle reject the hypothesis of a single tree.

- Chapter 6: `GreedyModeFind` is a fast new algorithm for finding the mode(s) of a multinomial distribution. An exact analysis of its complexity is given. Although motivated by a phylogenetic problem in the previous chapter, this algorithm is widely applicable, and fits within the theme of effective computational techniques.

For some topics, journal space limitations meant that not all relevant information could be covered in sufficient depth in the published paper. Where this is the case, additional material is provided as extra sections in the chapter. For simplicity's sake, page number references in the main body of the thesis always refer to page numbers in the thesis itself, not to the "internal" page numbers of papers included within the thesis.

The papers contained in this thesis were written in collaboration with other authors, though no other author is present on every paper. Appendix A, "Work Breakdown", describes my contribution to each paper. This appendix also contains the *DRC 16 Statement of Contribution to Doctoral Thesis Containing Publication* forms for each paper.

1.2 Common Themes

Although each chapter can be considered self-contained, there are several themes that run through them all.

1.2.1 Strengthening the Fundamentals

As a discipline matures and original motivating questions find answers, a tendency emerges for ever more esoteric research questions to be proposed. Doubtless these questions are important, but it is just as important to ensure that the bread and butter tasks performed by scientists in the field continue to receive attention. All of the topics in this thesis concern fundamental problems in the fields of phylogenetics and bioinformatics: how to do things we can already do, but do them better (chapters 2, 3, and 6), and how to make sure that we are justified in doing them (chapters 4 and 5).

1.2.2 Information: A Common Currency

This thesis approaches information from two different angles:

- Practical: `coil` (chapter 3), like all compression programs, is concerned with extracting the underlying information from a dataset containing redundancy, for the explicitly practical purpose of saving resources (storage and network bandwidth).
- Inferential: Chapter 2 is concerned with inferring important information in the form of model parameters from a dataset, while chapters 4 and 5 examine the circumstances under which such inferences are meaningful.

Section 3.4 ("Data Compression as Quick and Dirty Science") elaborates on the connection between these two related concepts.

1.2.3 Combination Attacks

Chapters 2, 3 and 6 share an additional theme.

What makes computational research challenging is the breadth of knowledge that must be brought to bear on a problem in order to produce an efficient solution. Clearly an understanding of the problem domain is required, as are strong programming skills—the ability to turn a given algorithm into an actual, working program in some programming language. In addition, two overlapping categories of computational expertise are needed:

- Theoretical. This is the knowledge that comes from computer science: algorithms and data structures, proofs of correctness, computational complexity (Big O notation). The choice of algorithm literally decides the limiting performance level of a program as input sizes increase. Without a firm grasp of these concepts, programs are produced that run beautifully on trivial inputs, but bog down horribly as soon as realistic-size inputs are tried. There is a need both for familiarity with the universe of existing algorithms and the ability to synthesise new ones when necessary.

  A clear case for the practical importance of theoretical knowledge is Le Gall (2003)'s lack of awareness of the *heap* data structure. This is the principle reason for her program's dire performance in comparison to our `GreedyModeFind`, and also the barrier to determining useful time complexity bounds for it.

  Theoretical knowledge provides *strategy*.

- Practical. Computer science attempts to describe the behaviour of computer programs using abstract models of computation; these models inevitably make simplifying assumptions that fail to capture important characteristics of real-world computers. Perhaps the most central is to "forget about" the constant factors that multiply terms in asymptotic analyses of algorithm complexity. A textbook example is the existence of *caching*: data recently acquired from large, slow storage can be temporarily kept in a smaller amount of fast storage, enabling rapid access if it is soon required again. This principle operates at multiple levels in every modern computer—hard disks cache data in RAM, RAM caches data in L2 cache, L2 caches data in L1 cache—and is highly effective in practice. Failing to take advantage of caching can lead to slowdowns of several orders of magnitude. Since the goal of any computational analysis is to produce correct answers quickly *in the real world*, it is important to be aware of these limitations and ways to mitigate them.

  An important aspect of practical algorithm design that features heavily in this thesis is parallel computation. SIMD (Single-Instruction-Multiple-Data) parallelism, available through special instructions on modern CPUs, can in theory multiply execution speed by a constant factor, but only if computations can be rearranged to avoid inter-word dependencies. SIMD parallelism is applied to `coil`'s `find_edges` program, `xmp`'s Fitch algorithm inner loop, and the Fast Hadamard Transform used for Treeness Triangles.

One of the main advances made by `xmp` is its highly efficient implementation of MIMD (Multiple-Instruction-Multiple-Data) distributed-memory parallelism, enabling effective use of a wide range of computational resources, ranging from the multiple CPUs available on modern workstations, to many hundreds of CPUs in a networked cluster of PCs or purpose-built supercomputers. Extracting the most power from both SIMD and MIMD forms of parallelism requires detailed attention to data dependencies, access patterns, and sources of latency. Examples include:

– Under the original work distribution strategy, the boss process retained the subproblem representing all remaining work, and any time a worker process requested additional work, a smaller subproblem was split off and returned to it. This strategy sometimes led to a positive feedback loop in which a worker was given a series of progessively smaller problems, resulting in enormous communications overheads. This behaviour was rectified by replacing this strategy with work stealing, in which a worker process that runs out of work randomly picks another worker process to "steal" more work from.

– The most straightforward way to have implemented the boss process main loop would have been to process requests from different workers sequentially, forcing each request to wait for the previous request to be satisfied; this would have introduced a slowdown at least linear in the number of outstanding worker requests. Our more sophisticated approach separates request handling into a fixed number of phases, in each of which the handling of all worker requests can proceed concurrently, thereby eliminating this drop in performance.

– Horizontal packing of nucleotides into machine words (in the sense of Ronquist, 1998) was chosen over vertical packing because it results in a smaller effective block size, meaning less wasted time and memory for narrow datasets.

  – Modern CPUs are notoriously slow at executing conditional branch instructions, which correspond to `IF ... THEN` statements in a high-level programming language. `xmp`'s innermost loop avoids the costly data dependencies introduced by these instructions by calculating both possible results of conditional expressions, and combining them using bit masking techniques.

Chapter 2 covers these issues in detail.

Floating-point arithmetic is another way in which computers fail to behave in an ideal manner. Being poor imitations of the real numbers enamoured of mathematicians, floating-point numbers have finite precision, which in general is eroded by each computation performed on a value. This can lead to such "obviously true" identities as `x + y - x == y` failing to hold in some cases, and the need for elaborate algorithms for tasks as simple as accurately adding a list of numbers (Kahan, 1965). For this reason, one of the primary concerns of any seasoned programmer when working with floating-point arithmetic is to eliminate long chains of calculations, as they tend to accumulate roundoff error. Here `GreedyModeFind` improves on earlier algorithms by computing all needed floating-point values "afresh" on each loop cycle, instead of reusing "tarnished" values from earlier loop cycles.

Finally, designing and implementing practical algorithms usually involves tradeoffs. Experience and experimentation are needed to make appropriate choices. These tradeoffs are everywhere, but conspicuous examples include: in `xmp`, the design of heuristics for lower bounds that are reasonably effective while remaining fast to compute; in `coil`, the leaky move-to-front hashtable data structure, which forgoes exact calculation of edit distances between sequences in exchange for a much faster approximation of the top b distances.

Practical knowledge provides *tactics*.

A key aspect of `xmp`, `coil` and `GreedyModeFind` is their focus on attacking computational problems with both theoretical (high-level) and practical (low-level) engineering in order to maximise performance.

1.3 Background: Phylogenetic Inference

The following background information is particularly relevant to the Parallel MP Search, Treeness Triangles and ML Bias subprojects (chapters 2, 4 and 5), though it also informs the core algorithm underlying the Sequence Database Compression subproject (chapter 3). For an in-depth introduction to the topic, suitable texts include Felsenstein (2004) and Swofford *et al.* (1996). Semple & Steel (2003) contains a more formal mathematical treatment.[1]

1.3.1 Fundamentals

A *phylogenetic tree* is a representation of the evolutionary history of a set of *taxa* (biological species, groups or individuals). It comprises a tree whose *leaves* (or tips) are labelled by those taxa, and may be *rooted* (in which case all edges are considered to be directed away from the root in the "forward time" direction) or *unrooted.* A tree in which every internal vertex has degree 3 (apart from a possible root vertex of degree 2) is called a *binary* or *bifurcating* tree, while other trees are *multifurcating.* A tree may additionally specify a positive[2] real-valued *branch length* for each edge (or branch); interpretation of branch lengths depends on context, though they frequently represent either time durations or (expected or actual) numbers of mutations. The term *topology* is often used to describe the edge structure of a labelled tree—i.e. the tree considered without regard to any branch lengths it may have.

Another useful concept is the notion of a *split* or *bipartition* of a taxon set X into an (unordered) pair of disjoint subsets Y and Z such that $X = Y \cup Z$. This is written $Y|Z$. Every edge in a tree defines a split, and it is sometimes useful to think of a split as a "potential edge in a tree".

Phylogenetic inference is the problem of finding the phylogenetic tree or trees that most closely approximate the *true tree* that models the process that produced the taxa, using data sampled from the taxa. (Of course, we are assuming here that

[1]The Felsenstein (2004) and Semple & Steel (2003) books could not be more different in style: the former brimming with enthusiasm and at times conversational; the latter so taut with precisely encoded meaning that its contents would, one suspects, be very slightly *expanded* by a compression program.

[2]Edges having length zero are sometimes permitted, allowing non-binary trees to be treated as special cases of binary trees. This is usually harmless, but theoretically creates a problem for inference: a model class that allows zero-length edges on trees is not identifiable, since multiple such trees correspond to the same multifurcating tree having strictly positive edge lengths.

the taxa were produced by a treelike process—this assumption is discussed further in chapter 5, and chapter 4 describes a way to visualise the extent to which it holds.) In general, the data sample consists of a set of *characters*, each of which is a mapping from a taxon to a state; in the specific case of DNA data, characters correspond to columns or *sites* in an alignment of DNA sequences, and states correspond to one of the nucleotides adenine, cytosine, guanine or thymine (A, C, G or T respectively). As the word “sequence” suggests, DNA sequences confer an ordering on these characters, but this is usually ignored by current methods—for example, statistically motivated methods generally regard characters as independent and identically distributed (i.i.d.) observations. Usually, DNA data is available only for the tips of the tree, which represent present-day taxa—although exceptions do exist, such as when analysing samples gathered from rapidly evolving pathogens over time (Drummond & Rodrigo, 2000), or ancient DNA recovered from preserved remains (Shepherd & Lambert, 2008).

1.3.2 Starting points for inference

Given an alignment containing n DNA sequences, one for each taxon, phylogenetic inference may proceed directly from the source alignment data, or it may first entail conversion of the source data to some intermediate form. The most commonly used intermediate form is a *distance matrix*, an $n \times n$ matrix containing estimates of pairwise distances between each pair of taxa. Distance estimates are usually calculated from the source alignment according to some maximum likelihood (ML) model (see section 1.3.4, “Optimality criteria”). Distance-matrix-based methods are often faster than other methods and some have attractive statistical properties, but it is important to note that they cannot be as sensitive as methods that consider the original source data as they discard all information about interactions between three or more taxa.

Of crucial importance in phylogenetic reconstruction is the notion of *consistency*. An inference method is consistent if, given sufficient data, the probability that it recovers the true model tends towards 1 as the amount of input data tends towards infinity.

Consistency is a property of an estimator; a related concept is *identifiability*, which is a property of a model class. A model class is *non-identifiable* if it contains two distinct models (that is, two non-identical choices of parameter values) θ_1 and θ_2 such that the probability distributions on all possible outcomes from these models are equal. Any other model class is *identifiable*. Intuitively, if a model class is non-identifiable then there is no way to tell which of several models within it gave rise to a given dataset, even if an infinite amount of data is available.

1.3.3 Algorithmic treebuilding approaches

Plainly all treebuilding methods can be described in terms of algorithms, but some can be defined only in this way, while others have additional properties.

The most striking example of a purely algorithmic approach is the neighbour-joining method (Saitou & Nei, 1987) and its variants. This method takes a distance matrix, and repeatedly joins together two taxa into a single "virtual" taxon until only a single taxon remains. Neighbour-joining is interesting because it combines speed (we will soon see that $O(n^3)$ for n taxa is very fast in the world of phylogenetic inference) with an impressive level of accuracy. It was not until more than 10 years after its initial discovery that it was conclusively shown to converge to the model tree when enough data is supplied (Atteson, 1999).

1.3.4 Optimality criteria

An *optimality criterion* provides a way to assign a numeric score to any given tree. It is then possible to search for the tree or trees that maximise this score.

The optimality criterion approach has advantages over purely algorithmic tree building techniques:

- Obviously an exhaustive search through treespace (the set of all possible trees) will produce provably optimal trees according to the chosen criterion.
- Since a score can be calculated for any tree, it is simple to compare two arbitrary trees by comparing their scores.

Popular choices of optimality criterion include:

1. Maximum parsimony (MP) (Fitch, 1971): the score of a tree is the fewest number of nucleotide substitutions required to produce the observed sequences

at the tips. MP can operate only on discrete character data, such as DNA. Chapter 2 describes an efficient implementation of exact MP search.

2. Maximum likelihood (ML) (Neyman, 1971; Felsenstein, 1981): the score of a tree is its *likelihood*, namely the probability that the input data would be observed given a statistical model that includes the tree topology among its parameters.[3] Usually the model includes other parameters as well, such as edge lengths for each edge in the tree, and parameters governing the process by which nucleotides mutate. ML is a very general inference method that can be applied to sequence data. See section 1.3.7 for more details.

3. Bayesian maximum *a posteriori* probability (MAP) (Rannala & Yang, 1996): the score of a tree is its *posterior probability*, namely the probability of a statistical model being true given that the input data was observed. As with ML, the statistical model incorporates the tree topology plus other parameters; the difference is that Bayesian inference additionally requires an *a priori* probability distribution, or *prior*, on models to be specified. This is needed so that the probability of a model being true *before observing any data* can be determined. (ML hails from the frequentist school of statistics, where all calculations are implicitly performed "as if" the particular model being considered were true, and the concept of "the probability of a model being true" is nonsensical.) Bayesian inference is usually performed using the Markov Chain Monte Carlo (MCMC: Metropolis *et al.*, 1953; Hastings, 1970) technique, which was adapted for phylogenetic inference by Mau *et al.* (1999).

4. Least-squares distance methods, such as the original unweighted least-squares method (Cavalli-Sforza & Edwards, 1967), or the weighted method of Fitch & Margoliash (1967): the score of a tree is the sum of squared *residuals* for all pairs of taxa. The residual for a pair of taxa is the (possibly weighted) difference between the distance given for them by the input distance matrix and the length of the unique path that connects them on the tree.

[3]It now seems amusing that one of the caveats acknowledged by Felsenstein (1981, p. 369) at the time he presented his ML computational technique for phylogenetic trees was the "paucity of DNA sequence data available for analysis" with it.

1.3.5 Search methods

Approaches to solving the inference problem using an optimality criterion can be broken into two categories:

- Exact algorithms
- Heuristics

Given a particular tree topology, it is often the case that an optimality criterion can be optimised quickly:

- The Fitch (1971) algorithm can determine an MP tree for n taxa and k sites in O(nk) time
- For least-squares methods, matrix algebra techniques are used to compute branch lengths from a pairwise distance matrix so as to minimise the score

For ML and Bayesian MAP on the other hand, maximising the score is a challenging problem even when the tree topology is given. In practice, general-purpose numerical optimisation heuristics such as Nelder-Mead optimisation (Nelder & Mead, 1965) are used for optimising branch lengths and other parameters, so these searches cannot be said to be exact.

Unfortunately, optimality criterion approaches have one big disadvantage: for most criteria, the problem of searching treespace for optimal trees is NP-hard (see section 1.4.1), meaning that it is unlikely that any algorithm exists for solving the problem more efficiently than an exhaustive enumeration of all possible trees. Since there are $(2n-5)!!$ unrooted bifurcating trees on n taxa (where $k!!$ denotes the double factorial $k(k-2)(k-4)\ldots$, which has $\lfloor k/2 \rfloor$ factors), the search space grows super-exponentially with the number of taxa, meaning that in practice, exact solutions are possible only for small numbers of taxa.

Although the branch and bound (B&B) technique (used in e.g. Hendy & Penny, 1982) can be applied to prune off provably suboptimal parts of the search space and thereby increase the size of problems that can be solved in a reasonable timeframe, its efficacy depends heavily on how well the data fit a tree, and it provides no general time-complexity improvements. Nevertheless, empirically speaking, branch and bound techniques substantially improve the performance of many exact search problems. B&B benefits from high-quality, conservative heuristics for estimating the minimum additional cost that must be incurred by a partial solution on its way to becoming a complete solution; developing such heuristics for maximum parsimony is one of the four prongs of attack listed as bullet points on p. 21 in chapter 2.

1.3.6 Heuristic algorithms

A heuristic algorithm produces a result that, empirically, is reasonable most of the time, but about which no guarantees can be made. The advantage of heuristic algorithms is that they tend to be fast, enabling us to quickly find reasonably good trees; the disadvantage is that we can never be certain that the tree(s) found are truly optimal.

Heuristic methods often operate by taking an initial tree and repeatedly modifying it in some way, keeping the resulting tree when such a "move" increases the score and discarding it otherwise. This *hill-climbing* procedure allows a search program to find a local optimum, though there is no guarantee that it is also a global optimum. If the procedure is repeated many times starting from different random trees, and the same local optimum is frequently found, confidence that a global optimum has been found is increased.

One of the earliest heuristic methods is *branch-swapping*: nearby taxa on a given tree can be moved around so as to produce a new tree that might have a better score. Nearest-neighbour interchange (NNI), subtree pruning and regrafting (SPR) and tree bisection and reconnection (TBR) are progressively more thorough ways to rearrange a tree topology (Swofford *et al.*, 1996, pp. 407–427).

1.3.7 Statistical models

Following the framework laid out by Penny *et al.* (1992), in the context of ML or Bayesian phylogenetic inference, a *model* refers to the following triple:

1. A tree with branch lengths (which in general must be rooted; but see the discussion of reversibility below)

2. Initial conditions—specifically, the distribution of bases at the root, plus whatever other parameters are supported by the model class in question, such as relative rates of change along each branch (for non-clocklike models; see below), and/or the distribution of rates across sites

3. A mechanism of change—in the case of DNA, a stochastic Markov model giving the probability of moving from any nucleotide to any other nucleotide as a function of time

Each of these three components is comprised of one or more parameters that may be either given *a priori*, inferred (and then possibly ignored), or integrated over. Steel & Penny (2000) provides a excellent assessment of the various flavours of ML that arise from considering the different choices available.

Inference of a set of parameters is performed by finding the combination of parameter values that maximises the likelihood (for ML) or posterior probability (for Bayesian inference).[4] The number of parameters, and their structure, are determined by the choice of model.

Many statistical models of evolution have the useful property of *reversibility*, meaning that the probability of generating a given set of sequences at the tips does not depend on the location of the root. This implies that an unrooted tree can be specified instead of a rooted tree. When searching the space of all trees, only all unrooted topologies (and not the greater number of all rooted topologies) need be examined.

Any general class of models can be converted to a class of models that is *clocklike*, or is said to obey the *molecular clock*, if a root vertex is specified and the tree is constrained to be *ultrametric*—that is, the sum of branch lengths along any path leading away from the root is constrained to be the same. Clocklike models stipulate that the rate of mutation is the same at any point in the tree, and consequently edge lengths can be interpreted as time spans. Informally, a dataset may be said to be

[4]It is important to note that while Bayesian inference can be used to infer a single point estimate using MAP, doing so does not make full use of the method. Bayesian inference is more usually conducted with the aim of inferring the posterior *distribution* of the parameters, since the shape of this distribution supplies additional information on the confidence with which parameter estimates can be regarded. The MCMC algorithm produces a sample from this distribution, which can be used to produce a *confidence set* of trees or other parameters if desired.

clocklike if inference using a non-clocklike class of models produces a tree that can be rooted somewhere to produce a (near-)ultrametric tree.

1.3.8 Errors and consistency

Error from statistical inference methods accrues from at least two sources:

1. Sampling error: random error caused by the fact that a finite sample cannot estimate a population parameter precisely. Sampling error reduces with increasing sample size.

2. Systematic error: error introduced by incorrect assumptions in the inference process. Systematic error does not reduce with increasing sample size—it may in fact increase.

All ML methods are consistent when applied to datasets that were generated using their models: given sufficient data, they will converge upon the true tree (Fisher, 1922; Chang, 1996). If a distance matrix corresponds exactly to the total path-lengths between leaves on some tree, then that tree is unique (Buneman, 1971); furthermore several distance-matrix methods, including neighbour-joining (Saitou & Nei, 1987), can recover the true tree whenever the maximum measurement error in any distance is less than half the length of the smallest edge, which is sufficient to establish consistency for these methods (Atteson, 1999).

Of course, we cannot truly know what model produced a set of sequences unless we generate them ourselves; hence the importance of examining the *robustness* of models used for inference, as we do in chapter 5.

1.3.9 Maximum Parsimony and consistency

After an early surge in popularity due to its intuitive simplicity, maximum parsimony fell from grace in phylogenetics. This can largely be attributed to the discovery that it is in general inconsistent: for some model trees, even with infinite data, the probability of recovering the true tree does not go to 1—in fact, in some cases, it goes to 0, while the probability of a particular incorrect tree goes to 1 (Felsenstein, 1978). (The region of the parameter space in which this phenomenon occurs is sometimes called the "Felsenstein Zone".) The construction provided by Felsenstein (1978) could be considered pathological, however Hendy & Penny (1989) demonstrated that MP can be inconsistent even for a "typical" tree that obeys a molecular clock.

Despite this gloomy state of affairs, there remains hope for parsimony. The root of the problem was identified by Steel *et al.* (1993) as being the lack of a correction for multiple changes at a site. The apparent inability to perform this correction was due to the fact that any such correction needs to act on distances between taxa, while MP requires sequence data. However, as pointed out by Steel *et al.* (1993), the ability of Hadamard conjugations (Hendy & Penny, 1993) to invertibly convert between distances and sequence spectra means that this difficulty can be overcome; the resulting method, corrected parsimony, is consistent.

Other special cases in which uncorrected MP is known to be consistent include when no common mechanism can be assumed for all sites in an alignment, and when the number of states is large in comparison to the number of sites. Steel & Penny (2004) have shown that for *ample* (or 1-connected) datasets—those sets of sequences in which every pair of sequences are connected by a path through sequences in the set, and each edge in the path has length 1—all MP trees are "Most Parsimonious Likelihood" (MPL) trees and vice versa. MPL is the ML variant that results when instead of integrating over character states at internal nodes of a tree as is done for the "usual" Maximum Average Likelihood criterion, the character states for these nodes are optimised so as to maximise the likelihood of the resulting model.

A precise description has been given by Tuffley & Steel (1997) of a useful region of the parameter space in which MP is in fact a consistent estimator.

1.4 Computational Complexity Primer

In addition to knowing that an algorithm is correct, it is practically important to know that it runs fast and requires little memory. Because computers run at different speeds, simply measuring running time on a particular computer is not enough to characterise an abstract algorithm's performance. Instead, the speed of an algorithm can be described using *Big O notation*, which gives asymptotic bounds on running time as the size of the input increases to infinity. A similar analysis can be performed for memory usage.

An algorithm is usually characterised by its worst-case performance, although average performance across all possible inputs is also of interest.[5] When many operations will typically be performed in succession, such as a series of insertions and lookups on a key-value map data structure, *amortised analysis* (Tarjan, 1985) can prove useful. This more general approach allows guarantees to be made about the worst-case performance of a sequence of n operations, without constraining the time taken by each individual operation.

The most common kind of complexity analysis uses $\mathrm{O}(\,\cdot\,)$, which expresses an "eventual" upper bound on a function. Given two nonnegative functions $f(x)$ and $g(x)$, $f(x) = \mathrm{O}(g(x))$ means that, for sufficiently large values of x, $f(x)$ is always less than or equal to some fixed multiple of $g(x)$. (An equals sign is often used to indicate the relationship, although $\in$ would be more correct.) Formally, $\exists a, c \in \mathbb{R}$ such that $\forall x \geq a, f(x) \leq cg(x)$.

The definition of $\mathrm{O}(\,\cdot\,)$ allows all coefficients, and all but the highest-degree term in a sum of polynomial terms, to be discarded, since if $f(x) = \mathrm{O}(Ax^n + Bx^m)$ and $m < n$ then $f(x) = \mathrm{O}(x^n)$ can be had by increasing a and c as necessary. This permits simple descriptions of common algorithms. A table of common complexities is given in Table 1.1.

The same notation can be used to describe an algorithm's memory usage.

1.4.1 NP Completeness

The previous section discussed asymptotic performance characteristics of particular algorithms for solving computational problems. It is useful to abstract further and consider the performance of the best-possible algorithm for solving a given problem: this allows the problems themselves to be characterised through lower bounds on the computational resources (time and memory) required to solve them. This section provides just a brief overview; for a full treatment, the interested reader should consult Garey & Johnson (1979).

[5] The utility of average-case performance can be hampered if the distribution of problem instances encountered in real applications differs strongly from the uniform distribution, as is often the case. Although this could in principle be addressed using weighted averages, doing so raises the tricky question of which distributions deserve such special attention.

Class	Description	Example
O(1)	Constant	• Array lookup • Comparison of integers • Addition and multiplication of integers[a]
O($\log n$)	Logarithmic	• Find given element in sorted list
O(n)	Linear	• Find given element in unsorted list • Find location of length-n string occurrence using suffix tree • Calculate parsimony score of n-taxon tree using Fitch algorithm
O($n \log n$)	Log-linear	• Sort a list using merge sort or heap sort
O(n^2)	Quadratic	• Find longest common substring of 2 length-n strings • Sort a list using quicksort, bubble sort or insertion sort
O(n^3)	Cubic	• Find shortest paths between all pairs of vertices in n-vertex graph using Ford-Warshall algorithm • Calculate neighbour-joining tree
O(2^n)	Exponential	• Find assignment of variables satisfying Boolean proposition or determine none exists • Fast Hadamard Transform (O($n2^n$))

[a]Under the usual *unit cost RAM model*, it is assumed that all integers fit within a single machine register and that most operations on machine registers take unit time.

Table 1.1: Common algorithm complexities

The most important practical distinction to make is between the complexity classes P (Polynomial-Time) and NP-hard (NP stands for Nondeterministic Polynomial-Time). A problem is in P if an algorithm exists that can solve every instance of the problem in polynomial time—i.e. in $O(x^k)$ time, for some fixed k. A problem is in NP if a solution to the problem can be checked for correctness in polynomial time; NP thus includes all problems in P, as well as potentially other problems that are hard to solve but easy to verify when a solution is already available.

The most difficult problems in NP are called *NP-complete*. These problems have the property that, if any polynomial-time algorithm can be found for one of them, then *every* other problem in NP can also be solved in polynomial time. (Suppose the two problems are ϕ and ρ and the polynomial-time algorithm to solve ϕ is a_ϕ: in polynomial time, ρ can be transformed into a polynomial number of instances of ϕ, each of which can be solved by calling a_ϕ as a subroutine.) The related class *NP-hard* includes all NP-complete problems, and also problems outside of NP such as the Halting Problem: determining whether a given program eventually completes or continues running forever.

The first problem proved NP-complete was Boolean satisfiability: the problem of determining whether there exists an assignment of true/false values to variables that makes a given propositional logic formula true (Cook, 1971). In the same paper, Cook also introduced the standard technique for proving a problem NP-hard via *reduction* from an NP-complete problem. A raft of NP-completeness results for other well-known problems soon followed (Karp, 1972).

Roughly speaking, problems in P are considered "efficiently solvable", while problems outside of it are "intractable".[6] (Note that this definition of "efficient" differs from the notion of statistical efficiency, which measures the accuracy of an estimation technique for a given sample size.) It is a longstanding open question whether P is a strict subset of NP, or whether instead P=NP—but the fact that decades of computer science research has failed to produce a polynomial-time algorithm for any NP-complete problem is taken as strong evidence that P$\neq$NP. For all practical

[6]This description is not always apt; it's certainly possible for an $O(n^3)$ algorithm to run infeasibly slowly on typical inputs, while an algorithm that is nominally $O(2^n)$ runs quickly on nearly all inputs. The Simplex algorithm for linear programming is an example of an exponential-time algorithm that in practice has better performance on typical problems than the polynomial-time ellipsoid method (Klee & Minty, 1972).

purposes, proving an algorithm NP-hard means that the best algorithm that can be expected to solve the problem is exponential-time.

Chapter 2

Faster Exact Maximum Parsimony Search with XMP

2.1 Introduction

This chapter presents the paper “Faster Exact Maximum Parsimony Search with XMP”, which was published in volume 27, issue 10, pp. 1359–1367 of the journal *Bioinformatics* in 2011.

All software associated with this paper is now freely available via Subversion from `http://xmpfast.sourceforge.net`.

BIOINFORMATICS ORIGINAL PAPER Vol. 27 no. 10 2011, pages 1359–1367
doi:10.1093/bioinformatics/btr147

Phylogenetics Advance Access publication March 27, 2011

Faster exact maximum parsimony search with XMP

W. Timothy J. White[1],* and Barbara R. Holland[2],*

[1]Institute of Fundamental Sciences, Massey University, Palmerston North, New Zealand and [2]School of Mathematics and Physics, University of Tasmania, Hobart, Tasmania, Australia

Associate Editor: David Posada

ABSTRACT

Motivation: Despite trends towards maximum likelihood and Bayesian criteria, maximum parsimony (MP) remains an important criterion for evaluating phylogenetic trees. Because exact MP search is NP-complete, the computational effort needed to find provably optimal trees skyrockets with increasing numbers of taxa, limiting analyses to around 25–30 taxa. This is, in part, because currently available programs fail to take advantage of parallelism.

Results: We present XMP, a new program for finding exact MP trees that comes in both serial and parallel versions. The serial version is faster in nearly all tests than existing software. The parallel version uses a work-stealing algorithm to scale to hundreds of CPUs on a distributed-memory multiprocessor with high efficiency. An optimized SSE2 inner loop provides additional speedup for Pentium 4 and later CPUs.

Availability: C source code and several binary versions are freely available from http://www.massey.ac.nz/~wtwhite/xmp. The parallel version requires an MPI implementation, such as the freely available MPICH2.

Contact: w.t.white@massey.ac.nz; barbara.holland@utas.edu.au

Supplementary information: Supplementary data are available at *Bioinformatics* online.

Received on October 20, 2010; revised on March 16, 2011; accepted on March 17, 2011

1 INTRODUCTION

Of all the criterion-based approaches to evolutionary tree selection, the maximum parsimony (MP) criterion is the most intuitive: ‘Select the tree or trees that require the fewest DNA substitutions’. The early popularity of MP was dampened by the discovery that it can be statistically inconsistent: in the ‘Felsenstein Zone’, increasing amounts of data will lead to increasingly certain recovery of the wrong tree (Felsenstein, 1978). Later, more general conditions leading to inconsistency were described (see Schulmeister 2004 for an extensive review). Elucidation of the assumptions made by MP led to the understanding that it is consistent when the expected number of changes per site, both across the tree and on any edge, are sufficiently small (Felsenstein, 2004; Steel, 2001, pp. 97–102). Steel and Penny (2000) chart the many variants of ML and their connections to MP. Of particular relevance is that MP is a most-parsimonious likelihood estimator for ‘ample’ datasets in which all taxa can be connected by a tree with maximum edge length 1, which is of practical significance when dealing with population (intra-species) data where taxa are highly similar (Holland *et al.*, 2005).

An often-overlooked fact is that the primary criticism levelled at MP—lack of statistical consistency in the general case—can be rectified through Hadamard conjugation (Penny *et al.*, 1996; Steel *et al.*, 1993). In particular, the Kimura three-parameter model (Kimura, 1981) and all its submodels can be corrected for directly using Hadamard conjugation, while any reversible model at all for which additive distances can be estimated can be dealt with via the Distance Hadamard transformation (Hendy and Penny, 1993). The resulting two-step approach, called corrected parsimony, is statistically consistent.

Although powerful heuristics for the MP problem have been developed (e.g. Goloboff 1999; Nixon 1999; Roshan *et al.* 2004; see also Felsenstein 2004, Chapter 4), they necessarily come without guarantees. We believe that a freely available, high-performance implementation of exact MP search is conspicuously absent, and would benefit the phylogenetics community. Our program, XMP, fills this gap. XMP offers:

- a parallelization strategy that is simultaneously highly portable across different computer architectures and highly efficient, scaling to hundreds of processors. This is the main new contribution;
- a new lower bounding approach for pruning unpromising regions of the search space, inspired by the MinMax Squeeze (Holland *et al.*, 2005);
- streamlined Fitch inner loop calculations using optimizations not published elsewhere; and
- a hand-optimized SSE2 assembly language implementation of the Fitch inner loop for Intel Pentium 4 and later CPUs, offering a potential 4-fold speed improvement.

The latter three improvements are also available in a single-processor version. We compare this version with a popular exact MP program and find that XMP is much faster in almost every case.

1.1 Existing implementations

Two branch and bound algorithms for exact maximum parsimony search were proposed by Hendy and Penny (1982). Their Algorithm I, which adds taxa to a partial tree one at a time, forms the basis of most current implementations of exact MP search (Bader *et al.*, 2006; Felsenstein, 1989). TurboTree (Penny and Hendy, 1987) took a different approach, inserting characters one at a time instead of taxa.

Hennig86 (Farris, 1989) was one of the earliest widely used parsimony programs offering an exact search feature.

*To whom correspondence should be addressed.

ExactMP (Bader *et al.*, 2006) is an exact MP search program designed to run in parallel on a shared-memory multiprocessor. For five 'hard' datasets, the authors achieve an average parallel speedup of 7.26 on eight processors using a queue-based locking and work distribution mechanism. In terms of absolute speed, Figure 4 of Bader *et al.* (2006), which compares the speed of ExactMP running on eight processors with PAUP* running on one processor, suggests that PAUP* is approximately four times faster than ExactMP on a single processor. We note that while ExactMP requires a shared-memory computer to run, XMP runs efficiently on both shared-memory computers and distributed-memory computers such as the BlueGene BG/L supercomputer or a networked cluster of commodity PCs.

PAUP* is a popular program for performing many types of phylogenetic inference. PAUP*'s exact MP inference is very fast—it is one of two programs that we use as a benchmark for XMP. While some of the strategies used by PAUP* have been published (Swofford *et al.*, 1996), unfortunately many algorithmic details have not.

PHYLIP (Felsenstein, 1989) is a freely available implementation of many phylogenetic inference methods, and includes the program dnapenny for performing exact maximum parsimony analysis. This program is typically much slower than PAUP*—on the same hardware, dnapenny took slightly over 20 min to find the unique minimal tree for the mt-10 dataset, while PAUP* took 44 s and the plain (non-SSE2) version of XMP took 7.3 s. The SSE2-optimized version of XMP took just 2.2 s.

TNT (Goloboff *et al.*, 2008) is a freely available maximum parsimony program explicitly focusing on efficient heuristic methods for large datasets. TNT also provides a fast exact search facility, which we compare with XMP.

Althaus and Naujoks (2006) take quite a different approach to the usual implicit enumeration scheme. Instead of adding individual taxa to a partial tree, they build sets of candidate rooted monophyletic groups in a first pass, and in a second pass form unrooted trees from all legal combinations of three such groups. Because the inputs to the second pass are monophyletic groups, the authors are able to draw on a variety of rules developed for general Steiner tree construction to eliminate tree topologies that cannot possibly be optimal. They order groups cleverly so as to minimize the number of legality tests performed.

Sridhar *et al.* (2008) take a different approach again, providing two different integer programming formulations of the MP problem: one that contains only a polynomial number of variables and constraints, and one that in the worst case may require an exponential number of both, but which in practice is faster to solve. They demonstrate impressive solution times, in one case solving a 34-taxon problem in less than one min. However, their program is restricted to alignments containing binary characters, and produces only one MP tree, rather than all minimal trees. Although the program currently requires the commercial mixed integer solver CPLEX, they provide free access to a web-based front-end at http://www.cs.cmu.edu/~imperfect/index.html.

In the remainder, we assume as input an aligned DNA dataset having n taxa and k sites. Informally, a dataset with many taxa is *tall*; a dataset with many sites is *wide*. The *length* of an edge or tree is the minimum number of point substitutions it requires; the *parsimony score* or *MP score* of a set of taxa is the minimum length of any tree interconnecting those taxa. XMP accepts ambiguous nucleotides and gaps, the latter being interpreted as 'any nucleotide'—the same as in PAUP* under default settings (GAPMODE=MISSING).

2 METHODS

2.1 Branch and bound for maximum parsimony

Although the maximum parsimony (MP) score can be determined in $O(nk)$ time using the Fitch–Hartigan algorithm (Fitch, 1971; Hartigan, 1973) when a tree is given, the problem of finding a tree whose MP score is minimal is NP-complete (Graham and Foulds, 1982), so it is unlikely that any algorithm exists that is asymptotically faster than enumerating and scoring all possible trees. Nevertheless, branch and bound (B&B) algorithms can offer a substantial improvement in practice. B&B is a general strategy for solving optimization problems that operates by exploring a *search graph* in which each node corresponds to a subproblem, and each arc links a subproblem with a child subproblem formed by adding constraints to the parent. This search graph is usually not represented explicitly but rather is implicit in the recursion structure of the program. In the usual formulation of B&B for MP search, first introduced by Hendy and Penny (1982), a subproblem is a binary tree on a subset of the full taxon set, and there is a child subproblem for each edge in the tree, indicating where the next taxon will be added. (For now, we leave aside the question of which taxon will be added next.) Full trees correspond to feasible solutions. The algorithm begins by calculating an upper bound on the parsimony score, often by building a heuristic MP tree. Then evaluation of subproblems takes place, starting with the original unconstrained problem, represented by the unique binary tree on some chosen set of three taxa. For each subproblem visited, a lower bound on the MP score of any solution reachable via that node is computed. Clearly adding taxa to a tree can never decrease its length, so it is acceptable to use parsimony scores of partial trees as lower bounds; later we look at stronger bounds. The utility of B&B hinges on the following observation: when a node is visited whose lower bound exceeds the current upper bound, it follows that it cannot lead to an optimal solution, so evaluation of its descendants can be skipped. This event is called a *cutoff* and the descendant nodes are said to be *pruned*, *bounded out* or *fathomed*. As the algorithm proceeds and more-parsimonious full trees are discovered, the global upper bound is reduced, further accelerating performance. In practice, the improvement over exhaustive enumeration is highly dataset dependent, but significant for typical biological datasets.

2.2 Branch and bound in XMP

The above description of B&B leaves several questions unanswered:

- Which three taxa should be chosen for the initial tree?
- In what order should taxa be added to the tree?
- In what order should subproblems be evaluated?

B&B involves adding taxa to a current partial tree in some order. At any point, given a partial tree containing $m < n$ taxa, we must choose (i) the next taxon to add and (ii) the order in which we should add that taxon to each of the $2m-3$ edges in the tree. As a general rule, the 'worst' taxon should be added next (so as to produce bound cutoffs as early in the search tree as possible), and that taxon should be added in its 'best' position first (so that new, tighter upper bounds are discovered sooner, possibly leading to earlier pruning of search trees stemming from other placements of the taxon). One important decision is whether to use a static or dynamic taxon addition order. A static order always adds taxa to the tree in the same order, while a dynamic order decides the taxon to add next just before the addition is to take place, so it can depend on the topology of the current tree. Although Purdom *et al.* (2000) indicates that using a dynamic taxon addition order can improve running times for tall datasets, we note that performance is highly variable across datasets, and it complicates the calculation of lower bounds on the length added by the remaining taxa. Also it is difficult to efficiently decide which taxon should be added next. We chose to use a fixed taxon addition order for XMP, which has the advantage of making lower bound calculations extremely fast (see Section 2.6). Where different implementation strategies are in conflict, we have generally favoured those that leave the B&B inner loop as streamlined as possible.

The order in which taxa are added to the tree is critical to performance (Hendy and Penny, 1982; Purdom *et al.*, 2000), with different orders easily leading to running time disparities of many orders of magnitude. XMP uses the standard max–mini approach (Nei and Kumar, 2000): first, an exhaustive search is used to find the three-taxon tree whose length is greatest; then $n-3$ rounds take place, in each of which the taxon t_i whose minimum length increase across all edges is maximum across all remaining taxa is added to the tree by inserting it at its minimum-length edge. This heuristic attempts to order taxa so that every tree built on any initial sequence of taxa has length as great as possible, in order to force early cutoffs.

Regarding the order in which the chosen taxon is added to the edges of the current partial tree, XMP takes the simple approach of using a preorder depth-first search (DFS) tree traversal, which is convenient for recursion. In the typical case where the initial upper bound is optimal (which was the case for all the datasets used for performance testing), the order in which placements are tried has no bearing on the set of search nodes evaluated, so this simple order suffices.

2.3 Parallelization strategy

A simple way to subdivide the original B&B problem is to have a set of worker processes that request work from a boss process whenever they are idle, while the boss enumerates complete trees down to some small depth (number of taxa) and sends each subproblem to a requesting worker to solve, until all starting trees have been exhausted. However, the subproblems generated this way may vary greatly in difficulty, leading to enormous imbalances in solution time: it can happen that all but one process finish work in milliseconds and sit idle, while one process continues working for months. One way around this is to arrange for the boss never to hand off the last subproblem, but rather subdivide it by inserting the next taxon at each edge, and hand off one of the generated subproblems instead. We found that although this sometimes improves behaviour, a different problem can arise: after the boss begins additional subdivision, subproblems issued to workers become smaller and smaller to the point where running time is dominated by communication between boss and workers. In effect, the situation approaches a serial B&B enumeration in which every node evaluation requires round-trip communication between a boss and a worker. Since communication latency typically dwarfs the time needed for a single node evaluation, performance drops dramatically.

Bader *et al.* (2006) confine themselves to a shared-memory multiprocessor, which affords them the ability to investigate advanced locking and lock-bypassing priority queues for managing subproblems. Approaches based on priority queues guarantee a minimum number of node evaluations, but increase the time needed for each node to be evaluated and complicate load balancing. Provided that the initial upper bound is tight, a simple depth-first search will evaluate exactly the same nodes (most likely in a different order), but is simpler, enables fast incremental tree modifications and has better locality properties. So XMP forgoes priority queues entirely, trusting that the initial TBR search phase (see Section 2.6) will bring us a near-optimal upper bound, and opts for a simple load-balancing approach that works on both shared-memory and distributed-memory multiprocessors: *work stealing* (Blumofe and Leiserson, 1999). A single boss process starts with the original tree on three taxa, and hands the entire problem to the first worker process that requests work. Every subsequent work request made to the boss causes it to choose a non-idle *victim* worker at random and steal a job (subproblem) from it to pass back to the requesting worker, or *thief*. Intuitively, work stealing never performs unnecessary communication, and provided steal victims are chosen randomly, it has excellent performance characteristics in terms of expected total execution time (Blumofe and Leiserson, 1999). All requests in XMP go via the boss, which simplifies termination detection and makes it easier to guarantee starvation-free servicing of workers. Although boss-free, truly distributed work stealing can be achieved on a distributed-memory computer by using a one-sided communications protocol that does not require synchronization with the destination, the necessary hardware support (Remote Direct Memory Access or RDMA) is often inadequate and must be emulated at additional cost: tellingly, in their study on the performance of distributed work stealing, Dinan *et al.* (2009) set aside one of every eight CPUs (apparently without adjusting their efficiency measurements) for a 'data server' process just to emulate efficient one-sided communication, despite the fact that their network technology, InfiniBand, nominally supports RDMA. XMP's centralized strategy does place an upper limit on scalability, though as our results show, surpassing this limit requires hundreds of processors.

The parallel version of XMP uses the industry standard MPI message-passing interface for handling all communication between processes. Using MPI rather than a thread-based approach enables XMP to compile and run on a wide variety of systems. For maximum efficiency, we use only non-blocking sends and receives, allowing computation and communication to proceed simultaneously whenever possible, and the boss uses `MPI_Waitsome()` to prevent starvation of workers. For portability, the program makes no assumptions about shared accessibility of files across processes.

In addition to requesting work when they are idle, workers announce improved upper bounds they discover to the boss, which broadcasts them. Workers poll `MPI_Test()` to detect incoming UB changes and steal requests. We use the adaptive polling interval technique advocated by Dinan *et al.* (2008) to balance computational throughput with responsiveness, using parameters $i_{max}=1024, i_{min}=1, i_{inc}=1, i_{dec}=2$. Briefly, the polling interval increases linearly up to a maximum while no message is received, halving upon receipt of a message.

Despite its simplicity, our boss worker formulation maps neatly to both 'big' and 'small' multiprocessors:

- on computers with only a few processors, such as modern desktop workstations, the boss process spends almost all its time waiting, and consequently takes up very little CPU time. This effectively leaves an 'extra' CPU free to allocate to a worker—i.e. in order to use the entire capacity of such a machine, XMP should be run with the number of processes set to one more than the number of CPUs. As our results show, the task switching that occurs when the boss needs to service a request requires very little overhead; and
- computers with many processors, such as the IBM BlueGene series of supercomputers, commonly mandate a fixed allocation of processes to CPUs. In this case, time that the boss process spends waiting is necessarily wasted. However, due to the scale of these systems, the boss has many more requests to service and thus spends little time idle, so that typically only a fraction of one CPU is wasted.

2.4 Subdivision into jobs

It seems natural to define subproblems as partial trees on $m<n$ taxa: expanding a subproblem is then done by inserting the $(m+1)$-th taxon at each of the $2m-3$ edges in turn. Although ideal for serial computation, this scheme causes problems for parallel implementations because it is difficult for steal victims to 'break off' large *jobs* (sets of subproblems) to send back to thieves, which is necessary for minimizing communication overhead. This problem can be overcome by adopting a finer notion of subproblem that constrains the set of edges at which the next taxon can be inserted, along with a novel representation for sets of subproblems.

XMP uses a *remaining edge pair list* (REPL) to compactly encode a family of subproblems formed by stripping taxa in reverse order from a base partial tree. A job whose base tree contains m taxa is represented by a list of $m-2$ integer pairs; the elements of the i-th pair identify a range of edges in a partial tree of size $i+2$. Edges are numbered in DFS preorder. To understand the REPL, first note that given (a) an initial tree on three taxa, (b) a list of remaining taxa to be inserted in order, and (c) a fixed policy for numbering edges, then a list of m edge indices uniquely identifies a tree on the first $m+3$ taxa by interpreting each edge index as identifying the edge to insert the next taxon at. The set of subproblems represented by an REPL containing m pairs $(L_i, R_i), 1 \le i \le m$ may be described recursively as follows:

(1) If $L_m > R_m$, then no subproblems are included; otherwise, construct the $(m+2)$-taxon tree from the edge index list $L_{1...m-1}$ as described above. The subproblem corresponding to this partial tree, with taxon $m+3$ constrained to be inserted at an edge having index in the range $L_m \dots R_m$, is included; and

(2) If $m > 1$, remove the final pair from the list and increase L_{m-1} by 1. All subproblems that would be included by this new REPL are also included.

The initial problem—a tree on three taxa in which the fourth can be inserted at any edge—is given by $(0,2)$. Each worker maintains an REPL as it enumerates its B&B tree: every time a taxon, say the m-th, is added to a partial tree containing $m-1$ taxa, a pair $(0, 2m-2)$ is appended to the list, representing the $2m-3$ edges at which the next taxon can now be inserted; whenever the m-th taxon is removed, the last pair in the list is removed, and the first element of the new final pair is incremented.

REPLs can be quickly updated during local B&B search, and their compactness reduces the size of job messages. But their primary advantage, and the key to dealing with steal requests effectively, is that an REPL can be easily partitioned into two REPLs representing disjoint sets of subproblems having the following properties:

- The B&B recursion exploring the original REPL can continue exploring one of the new REPLs; and
- The other REPL contains a largest possible subproblem (smallest possible partial tree).

When a steal request arrives, the victim can quickly discover a subproblem corresponding to the smallest partial tree that it has available by scanning its REPL for the first edge range (i,j) with $i<j$. Suppose this is the m-th pair. The REPL to send back consists of the first $m-1$ edge ranges (which necessarily have both edge indices equal) plus the edge range (j,j). The victim then sheds this job from its own workload by decrementing R_m in its own REPL. This can be done in $O(m)$ time.

Upon receipt of a new job REPL, a thief assembles the base tree from the list by 'fast-forwarding' the usual B&B enumeration process—at each tree size, taxon insertion is simply skipped for any edge having index less than the corresponding L_i. Normal B&B then resumes.

2.5 Coping with complexity

A number of factors contribute to complexity in the parallel code: tracking the states of workers, the necessity for both worker-initiated and boss-initiated communications, reliably detecting termination and our desire to use non-blocking I/O for efficiency. That complexity led to bugs. Bugs in parallel software can be nightmarish due to the difficulty in reproducing them, so we decided to model the communicating system of processes using the MPI-Spin extension to the model checker Spin (Holzmann, 1997; Siegel, 2007). This excellent tool caught one obvious, and two extremely subtle bugs which we subsequently fixed. After heavy optimization of the Spin model, the final run examined 1.4 billion state transitions and required 52 min and 44 GB of RAM to confirm that every possible interleaving of execution sequences involving one boss and two workers is free of deadlocks and other assertion violations—a very strong indication that the program works correctly with any number of workers.

The remaining subsections concern topics that apply to both serial and parallel versions of XMP.

2.6 Upper and lower bounds

By default, XMP initially attempts to find a tight upper bound on the MP score by performing greedy (hill-climbing) TBR branch swapping on 100 trees produced by random addition order. A minimum spanning tree is also calculated, although this usually produces a loose bound.

During the B&B phase, lower bounds on MP scores are needed for each partial tree examined. The MP score of the partial tree is an admissible but usually suboptimal bound; XMP employs several strategies for improving on this that are described in the following subsections. In each case, we are given a partial tree on $m<n$ taxa and tasked with finding a lower bound on the length that must be added by inserting the remaining $n-m$ taxa in some fashion; this can be added to the MP score of the partial tree to get a lower bound on the MP score of any full tree that is reachable from it. XMP takes the standard approach of calculating bounds that depend only on the taxa present in the tree, and not on the tree topology. Because XMP uses a fixed taxon addition order, only $n-2$ different subsets of taxa will ever be encountered, meaning that all lower bounds can be precomputed and stored in a lookup table for speed.

The parallel version of XMP does not parallelize the initial computation of upper and lower bounds. Instead each worker permutes sites randomly before evaluating bound heuristics, and the overall best bounds of each kind are retained. This occasionally produces superlinear speedups.

2.7 Single column discrepancy lower bound

Consider a partial tree containing some subset of the taxa, and a site containing d distinct nucleotides. If only $e<d$ distinct nucleotides appear at that site among the taxa in the tree so far, then the remaining $d-e$ nucleotides must be added by the remaining taxa, with each distinct nucleotide incurring a cost of at least 1 substitution. These lower bounds can be summed over all sites to produce an overall *single column discrepancy* (SCD) lower bound that is cheap to compute and leads to speedups of typically 1.3 to 2.4 for static taxon addition order (Purdom *et al.*, 2000).

In practice, the algorithm is complicated slightly by ambiguous nucleotides. To handle these, we split the problem into computing an upper bound on a given subset of taxa, and a lower bound on the entire taxon set: the difference is a lower bound on the length that must be added by the remaining taxa. Upper bounds for a single site are found by representing ambiguous nucleotides as *state sets* of unambiguous nucleotides, and subtracting the frequency of the most commonly occurring nucleotide from the number of taxa. This is equivalent to determining the most frequently occurring nucleotide x, then constructing a new, unambiguous site in which every state set containing x is replaced with x itself and every other state set S is replaced with any nucleotide $y \in S$, and finally applying equation 2 of Steel and Penny (2005) to this new site. This formula produces optimal upper bounds in the absence of ambiguous nucleotides, and good quality bounds in other cases. Tight single-site lower bounds can be found by solving a maximum set packing problem. We performed this in an offline step for each of the 2^{2^4-1} possible sets of distinct ambiguous nucleotides that could be present at a site, and stored the results in a lookup table. The SCD lower bound can be requested with the `-Bd` option to XMP.

2.8 Incompatibility lower bound

The I-bound of Holland *et al.* (2005) exploits the fact that every non-overlapping pair of incompatible sites must increase the length of an MP tree by at least 1, and sometimes more. This bound has the advantage that it can be added to the SCD bound to produce a stronger lower bound. XMP provides a similar bound via a greedy approximate maximum matching algorithm for finding incompatible site pairs, which is available using the `-Bi` option.

2.9 PARTBOUND lower bound

The MinMax Squeeze (Holland *et al.*, 2005) attempts to produce provably optimal MP trees by pushing a lower bound on MP scores up until it meets the length of trees found heuristically. That work extends the lower bounding technique first established as the Partition Theorem by Hendy *et al.* (1980), which essentially states that the MP score of a dataset must be at least the sum of the MP scores of each part in a sitewise partition of the dataset. XMP contains a new lower bounding technique based on this approach, PARTBOUND, available via the `-Bp` option. Rather than maximize the overall lower bound on the first m taxa, we seek partitions that maximize the sum of the *final scores* for each part. The *raw score* of a part π is $LB(\pi,n)-UB(\pi,m)$, where $LB(\pi,i)$ is a lower bound on the MP score of any tree on the first i taxa, restricted to the sites in π, and $UB(\pi,i)$ is an upper bound defined similarly. Raw scores may be negative. The final score of a part is the greater of the raw score and the SCD bound for the first m taxa summed over all sites in the part. Final scores are always non-negative, and their sum can be safely added to the MP score of any tree built on the first m taxa. Starting from the trivial partition (one site per part), XMP searches partition space with a greedy site-swapping heuristic that attempts to increase the total final score—or, when impossible, the total raw score—until no final score improvement has been made for two iterations.

Part upper bounds are calculated as for the SCD bound. The challenge is to calculate good lower bounds: XMP uses several strategies, choosing the best for a given part. After identical sequences are collapsed, the Kruskal algorithm (Kruskal, 1956) is used to find the largest 1-connected components. If one or two components result, the length of the minimum spanning tree (MST) is used; this bound dominates the D-bound of Holland *et al.* (2005) when one of the edges has length >2. If three or more components result, then the lengths of Steiner trees (which may or may not have a single Steiner vertex) on all possible sets of three components are calculated, and the longest chosen. (Again, ambiguous nucleotides prove an annoyance since distances between sequences containing them may violate the triangle inequality; nevertheless, it can be shown that these constructions yield valid lower bounds—see the Supplementary Material.) Finally, the length of an MST is at most twice the length of a Steiner tree in any metric space, so finding an MST and dividing the length by 2 yields another lower bound on the MP score (Proposition 5.4.1 in Semple and Steel, 2003). For pairs of sites containing no ambiguous nucleotides, there is no need to divide by 2 (Bruen and Bryant, 2008).

2.10 Fast fitch parsimony

Ronquist (1998) describes how Fitch parsimony operations can be accelerated by encoding state sets as bit vectors and storing multiple sites in a machine word. In the terminology of that paper, XMP horizontally packs four-bit state sets into machine words of 32, 64 or 128 bits in width. He also observes that modern superscalar, deeply pipelined CPUs penalize unpredictable conditional branches in program code—a trend that has become more severe in the years since. The Fitch algorithm (Fitch, 1971) tests whether two state sets have a non-empty intersection and thus appears to require such a branch; however, he offers several algorithms that cleverly sidestep the problem by using bit masking techniques.

Although Algorithm 8 of Ronquist (1998) increases performance by using only predictable conditional branches, it does not achieve the full potential of this approach because it still loops over each possible state a site may take. The algorithm on p. 271 of Goloboff (2002) improves matters slightly by 'unrolling' the loop, but XMP boosts speed further by eliminating all per-state calculations. As the following C code shows, the trick involves exploiting the carry produced by binary addition:

```
u = ((((x & y & 0x77777777) + 0x77777777) |
      (x & y)) & 0x88888888) >> 3;
z = (x & y) | ((x | y) &
      ((u + 0x77777777) ^ 0x88888888));
```

Here `x` and `y` are 32-bit words each containing blocks of 8 input state sets from two child sequences, and `z` is assigned the resulting block of 8 output state sets for the parent sequence. `u` will have each 4-bit nibble set to 0001 if the corresponding site necessitated a mutation, and 0000 otherwise; it is also used for computing length increases. `&`, `|`, `^` and `>>` are the operations AND, OR, XOR and right-shift, respectively, and numbers beginning with `0x` are in hexadecimal. We deliberately leave in common subexpressions like `(x & y)`, trusting the compiler to do a better job of deciding when and how to evaluate them than we could.

To understand the process, consider a single site (nibble). We compute the intersection of the state sets `(x & y)` and mask out the leftmost bit; adding the binary value 0111 to this value will produce a 4-bit sum whose leftmost bit is 1 if and only if any of the remaining 3 bits are 1, thereby detecting whether `x` and `y` share any states in the set `{A,C,G}` with a single machine instruction. ORing this value with the original intersection nibble produces a 4-bit value whose leftmost bit is 1 if and only if any of the original 4 intersection bits were 1, i.e. if `x` and `y` share any states at this site. The reason for masking out the leftmost bit of the intersection nibble is to guarantee that adding 0111 cannot cause a carry into the nibble to the left: this permits a single 32-bit addition to perform shared-state detection for all 8 input state sets in parallel. This general technique was apparently first discovered by Lamport (1975), who credits D.E. Knuth.

With knowledge of whether the two taxa share any states at a given site now stored in the leftmost bit within the corresponding nibble, further masks and shifts can be used to compute `u` and the resulting Fitch state sets `z`. To turn the `u` values into the required all-1 or all-0 masks, XMP uses the O(1) calculation `((u + 0x77777777) ^ 0x88888888)`, which is faster than the repeated shifting and ORing described by Ronquist (1998) and Goloboff (2002). Because all operations respect nibble boundaries, the entire Fitch calculation can be performed in parallel across all eight sites as with earlier algorithms. We speculate that some of these techniques are already used in some existing MP programs—for example, in a personal communication note Goloboff (2002, p. 272) attributes to Farris unspecified optimizations that produce 'similar results …with about half the operations'—but they do not seem to be explicitly described in the phylogenetics literature.

In an initial step, XMP discards parsimony-uninformative sites and condenses the remaining groups of equivalent site patterns into individual weighted columns. By sorting sites in decreasing order of weight, we enable two additional shortcuts: (i) we can exit the innermost loop as soon as the length added to the tree exceeds the current bound, which is more likely to happen early on; (ii) we can swap over to a faster bit-counting algorithm for computing the remaining cost as soon as all weights in a block have dropped to 1, since from that point on all sites must have this weight. By default, the number of 1-bits in a machine word is counted using a multiply-mask-shift sequence, although an alternative implementation can be selected that sums adjacent nibbles, then adjacent pairs of nibbles, and so on. Both approaches take a small, fixed time and are likely to be faster than either of the iterative schemes detailed in Ronquist (1998). The lookup table approach suggested by Moilanen (1999) and used by Bader *et al.* (2006) may be faster, but we concluded that using a large chunk of memory to hold a lookup table was likely to degrade cache performance unnecessarily.

Goloboff (1993) describes a way to speed up parsimony searches by avoiding a complete first-pass Fitch optimization for each taxon insertion, enabling amortized $O(k)$ Fitch scoring of taxon insertions. For correct handling of ambiguous nucleotides at leaf nodes, a workaround is required (Goloboff, 1996, pp. 204–205), though this does not impact the time complexity. XMP uses a similar scheme by Yan and Bader (2003) that handles this situation without additional bookkeeping. We enhance this by applying

Shortcut C of Goloboff (1996, pp. 209–211) to eliminate unnecessary second-pass recursion. Finally, we noticed that as Fitch performance increased, proportionally more execution time was spent on bookkeeping overhead. Exploiting the fact that all memory allocations occur in LIFO order during the main B&B phase, we allocate a block of memory beforehand and thereafter use single pointer additions and subtractions for quickly allocating memory from this block when needed.

2.11 SSE2 optimized version

The SSE2 instruction set, available on Pentium 4 and later CPUs, includes instructions for operating on 128-bit quantities. Particularly on Core2 and later CPUs where most of these instructions execute in 1 clock cycle, this offers a potential 4-fold performance improvement over the usual 32-bit operations. We developed an optimized version of the Fitch inner loop using hand-coded SSE2 assembly language, which, like the regular C-code version, avoids conditional branches for maximum performance. Due to syntax differences, the SSE2-optimized version is currently only available for Windows compilers.

3 RESULTS

XMP was run on the real and synthetic datasets from Bader *et al.* (2006) as well as three other real datasets ranging in height, width and difficulty level. The leftmost five columns in Table 1 summarize the datasets. Performance was measured in the following environments: a quad-processor 2.66 GHz Core2 Windows XP PC using MPICH2, an 8-processor 3.2 GHz Linux PC using MPICH2 and the BlueFern BlueGene/L (BG/L) supercomputer at the University of Canterbury, using its proprietary implementation of MPI. BlueFern is a distributed-memory supercomputer, affording the chance to test XMP's ability to scale up across hundreds of CPUs without the benefit of fast access to a central memory store. For brevity, we report only a subset of results; see the Supplementary Material for a fuller picture.

Table 1. Datasets and performance of XMP `-Bdi` on 8-CPU SMP

Dataset	Taxa	Sites	MP length	Trees	T1	T8W	S/Up
h1	12	64	364	1	42.73	5.82	7.34
h2	12	64	367	2	34.43	4.72	7.29
h3	12	64	359	1	5.31	0.77	6.90
h4	13	64	397	2	215.24	29.16	7.38
h5	13	64	396	3	264.88	35.91	7.38
mh1	20	64	124	1134	14.86	2.16	6.88
mh2	20	64	192	3	7.03	1.09	6.45
mh3	20	64	118	*	873.94	123.97	7.05
mh4	20	64	303	5	9.97	1.47	6.78
mh6	20	64	128	612	24.41	3.45	7.08
e1	24	500	593	6	0.13	0.16	0.81
e3	24	500	589	36	0.11	0.18	0.61
e4	24	500	584	3	0.07	0.13	0.54
e5	24	500	577	3	0.12	0.14	0.86
e6	24	500	579	2	0.09	0.15	0.60
Eukar	27	2461	3512	60	1.55	1.35	1.15
rbc14	14	759	963	2	22.44	3.06	7.33
Metaz	20	1008	825	3	23.81	3.33	7.15
its36	36	607	233	62370	1552.64	192.92	8.05
mt-10	10	10539	16179	1	4.84	3.29	1.47
32hum	32	202	95	*	2269.51	301.66	7.52

Leftmost five columns describe datasets used; rightmost three columns give performance of XMP `-Bdi` on 8-CPU SMP. T1: Total elapsed time in seconds for serial version. T8W: total elapsed time in seconds for 8-worker parallel version (9 MPI processes). S/Up: speedup (T1/T8W).
* indicates more than 100 000 trees; only the first 100 000 were saved.

3.1 Serial performance

Figure 1 compares the times of the single-processor version of XMP with PAUP* and TNT on Windows XP. Four variants of XMP are considered: '-Bd' indicates that only the SCD bound was used; '-Bdi' indicates the sum of this bound and the incompatibility bound; '-Bp' indicates the new PARTBOUND bound; and '-Bp SSE2' indicates the SSE2-optimized implementation of this bound (all others use the portable C version). For parity with XMP's upper bounding strategy, PAUP* runs first used a corresponding HSEARCH command to find upper bounds; TNT does not allow the initial upper bound to be specified for an exact search, so no corresponding attempts could be made to improve initial upper bounds for this program. We note that both XMP and PAUP* found a tight upper bound in their initial TBR phase each time. An upper limit of 100 000 trees was set.

As Figure 1 shows, on all but three datasets XMP `-Bdi` is faster than PAUP*, often by a considerable margin. For mh6, 32hum and its36, XMP `-Bdi` is 50, 19 and 14% slower, respectively. For its36, XMP `-Bp` is faster, beating PAUP* by 18%, while for the other two datasets, `-Bp` is around 1% slower than `-Bdi`.

On all but two datasets, XMP `-Bdi` is faster than TNT, again often by a significant margin. For mh2, XMP `-Bdi` is 8% slower, while for mh3, TNT is dramatically faster than both XMP `-Bdi` and PAUP* at 260.61 s versus 1299.92 s and 8673.11 s, respectively. On the other hand, while XMP `-Bdi` and PAUP* take just 35.12 s and 23.42 s for mh6, respectively, TNT takes 2922.51 s. TNT regularly runs in under half the time of PAUP* for mid-range datasets, but performs badly on the largest two datasets.

We find that the performance on many datasets is actually affected very little by the choice of lower bound strategy. This seems surprising at first, since in the tables of lower bounds computed for each strategy, in all cases the entries near the start (being the bounds for small partial trees) show the `-Bdi` bound to greatly exceed the corresponding `-Bd` bound, and the `-Bp` bound to be greater still (data not shown). However as the partial tree size increases, the bounds for each strategy necessarily decrease, eventually becoming equal at some tree size. If a partial tree grows to that size under the `-Bdi` or `-Bp` bound without being eliminated, then it and its subproblems will be evaluated as with the simple `-Bd` bound; it is in this region of the search space that B&B presumably spends most of its time.

PARTBOUND (`-Bp`) is clearly a loss for very wide datasets such as mt-10 and Eukar. By comparison, `-Bdi` never shows excessive overhead, and is beaten by `-Bp` only for the its36 and rbc14 datasets, where it does not do much worse.

Unsurprisingly, the SSE2-optimized version of XMP is everywhere faster than the portable C version. Generally, the speedup increases with the width of the dataset, since this translates to a greater proportion of time spent in the innermost loop. The clearest example is the mt-10 dataset, although the total running time as shown on Figure 1 is dominated by the PARTBOUND calculation. Considering just the B&B time components for the mt-10 `-Bp` runs, the portable C version requires 6.05 s while the SSE2 version requires just 1.38 s—an improvement of roughly 4.4 times,

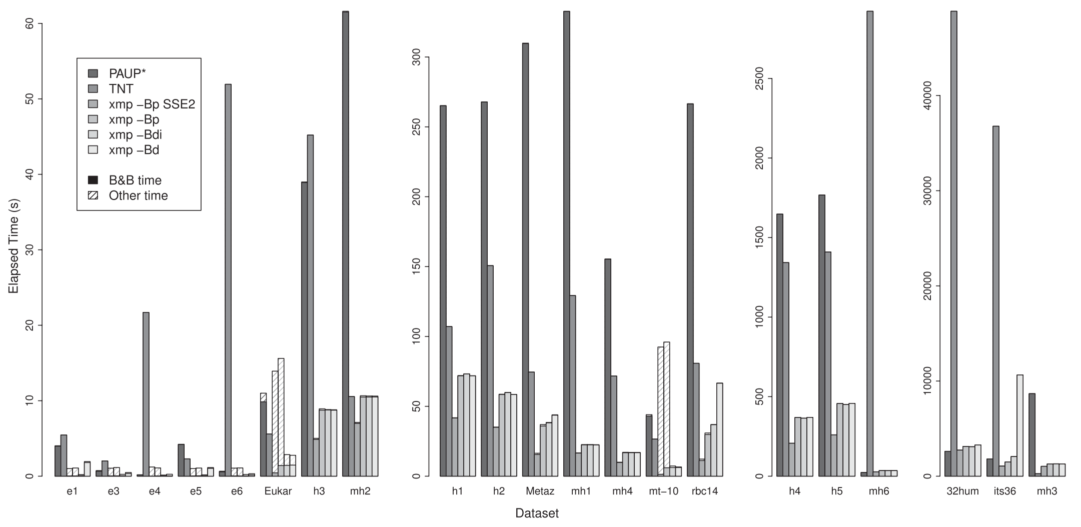


Fig. 1. Execution time of XMP versus PAUP* and TNT on 1 CPU. Each named dataset is analysed using exact search with PAUP*, TNT and four variants of XMP. The time taken for each run is shown as a sum of B&B time (solid) and other time (lined) components; for most runs, other time is insignificant. Separate time axes show detail for short, medium, long and very long runs. The underlying data are available in the Supplementary Material.

exceeding the 'theoretical' limit of 4-fold improvement. This is presumably due to inefficiencies in the compiler-generated code for the portable C version.

3.2 Parallel performance on eight CPUs

The rightmost three columns of Table 1 show the performance of XMP when run on the 8-CPU Linux machine using the `-Bdi` option. The T1 column shows the total time in seconds for the serial XMP version, and the T8W column shows the total time for the parallel version using 9 MPI processes (1 boss and 8 workers). The S/Up column is the speedup or ratio. This setup is similar to that used by Bader *et al.* (2006). For the five datasets labelled 'hard' in that paper, we find an average speedup of 7.258, almost exactly equalling the result obtained there. However, the 'moderate' and 'real' groups fare much better with XMP, achieving speedups of 6.847 and 5.210, respectively, while Figure 3 of Bader *et al.* (2006) shows that ExactMP obtains speedups of less than 6 and less than 5, respectively. The slowdown observed for the 5 'easy' datasets is simply a consequence of the fact that the B&B phase for these datasets takes much less than 1 s, meaning that overall runtime is dominated by one-time overheads such as communicating input data to all workers. In our article, all speedups and efficiencies are calculated with respect to the serial version.

Interestingly, the its36 dataset experiences a superlinear speedup due to the discovery of better lower bounds. We found this phenomenon occurred only rarely in our experiments.

3.3 Parallel performance on hundreds of CPUs

We ran each dataset using various numbers of CPUs on the BG/L in Virtual Node mode, which gives XMP control of both CPUs on each BG/L compute node. Figure 2 shows the efficiency of each

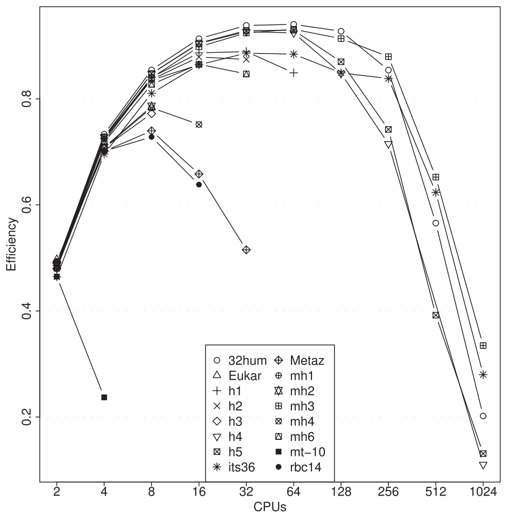


Fig. 2. Efficiency of XMP `-Bp` on the BG/L supercomputer as the number of CPUs is increased. An efficiency of 1.0 indicates overhead-free parallelization with respect to the serial version. Each dataset is shown separately; only runs taking >5 s are shown.

run that required more than 5 s. Efficiency is defined as $t_1 \div (mt_m)$, where m is the number of CPUs, t_1 is the elapsed time used by the serial version and t_m is the elapsed time used by the m-CPU parallel version. Because BG/L CPUs are always assigned exactly one process to run, the boss process necessarily consumes one full

CPU, so efficiency starts out near 0.5 for two CPUs and climbs. We show only results for the `-Bp` lower bound strategy; this can be considered conservative, since all lower bound processing takes place in serial beforehand, and `-Bp` has the highest overheads for this phase.

For the five hardest datasets (mh3, its36, 32hum, h5 and h4), we find that efficiency remains well above 0.8 for up to 128 processors, and for three of these datasets it remains above this level even for 256 processors. Above this point, performance begins to drop off sharply, presumably due to saturation of the boss node.

As a final test, we compared the running times of XMP `-Bp` on 256 BG/L CPUs and PAUP* on a 2.66 GHz Core2 CPU, using progressively larger subsets of the 53humans dataset from Holland *et al.* (2005). (Our 32hum dataset consists of the first 32 taxa from this dataset.) We find that analysing the first 39 taxa takes 5452.56 s (roughly 1.5 h) to complete with XMP in this configuration, while the same dataset takes PAUP* 168341 s (roughly 46 h, 45 min). The ratio of elapsed times improves from 27.17 at 32 taxa to 30.87 at 39 taxa, suggesting that the parallel version of XMP continues to become more efficient as problems grow in size. Adding one more taxon to the dataset results in XMP taking 19 h, 55 min; the corresponding PAUP* analysis was aborted after 2 weeks, but is projected to require more than 25 days—an impractical amount of time for most researchers to spend on a single analysis.

4 DISCUSSION AND CONCLUSIONS

XMP is at least as scalable on shared-memory multiprocessors as ExactMP, is faster in absolute terms and also runs efficiently on distributed-memory multiprocessors where ExactMP will not run at all. On almost all datasets tested, the serial version of XMP convincingly beats TNT, which in turn is faster than PAUP* on a majority of datasets, although there remain cases where PAUP*—a program now 9 years old—still holds out. Naturally, we were very interested in discovering the internal workings of TNT and PAUP*, but very little solid information could be found. We agree with Goloboff (1993) and Ronquist (1998) in calling for details of fast computational techniques to be made public, with the goal of advancing the state of the art: it seems likely that a person with knowledge of all three programs could design a program that outperforms all of them. We hope that the strategies, algorithms and tricks described in this article, and the free source code to XMP, contribute towards this goal.

Because of the superexponential complexity of exact MP search, the improvements realized in XMP will not usually allow many more taxa to be analysed, but they do dramatically increase the speed of existing searches. For example, our analysis of the its36 dataset on a modern Linux PC takes 17 min, 14 s using 1 CPU. This drops to 2 min, 23 s when all 8 CPUs of the machine can be used. Using 256 CPUs on a BG/L, the analysis takes only 38.55 s. These speed increases will also accelerate heuristic searches for large datasets that internally rely on exact searches on subsets of taxa, such as Rec-I-DCM3 (Roshan *et al.*, 2004), or the sectorial search of Goloboff (1999) when configured to use exact search for small sectors.

While raw performance is important, it must be noted that mature programs like PAUP* and TNT offer a fuller set of features than XMP currently does. Perhaps the most important feature absent from XMP but offered by both PAUP* and TNT is the ability to collapse edges according to various criteria, which can lead to sizeable reductions in output. This can still be done by an external program after an XMP run completes, but it is an inconvenience for the user. Other advantages of these existing programs include the ability to impose topology constraints, and to save suboptimal trees.

Holland *et al.* (2005) suggest an application of the MinMax Squeeze to B&B search, which XMP fulfils. The `-Bp` partition bound achieves modest speedups on two datasets, but otherwise does not materially improve execution times, and for wide datasets like mt-10 actually produces an overall slowdown. This is despite the fact that the lower bounds produced for each tree size are always greater than or equal to those produced by other bounds (data not shown). These results suggest that `-Bdi` is a good default setting, with `-Bp` as an option to consider for larger datasets.

4.1 Possible future directions

Section 4 of Bachrach *et al.* (2005) describes a lower bound based on a circular ordering approximation algorithm for the Path-Constrained Travelling Salesman Problem, a generalization of the TSP. This bound is unlike the lower bounds used in XMP in that it depends on the topology of the partial tree, which makes it not only potentially stronger but also much slower to compute. It would be interesting to incorporate this bound into XMP.

Felsenstein (2004; pp. 65–66) discusses rules for reducing the search space needed for exact MP search, which he attributes to a Russian language paper by A.Zharkikh, 1977. However, some of these rules do not appear to be compatible with XMP's enumeration scheme, which produces trees that are always binary but may contain branches that have length zero under every possible assignment of mutations to edges (A.Zharkikh, J.Felsenstein, personal communication). In order to accommodate Zharkikh's rules, two approaches seem possible: either avoid introducing zero-length edges into partial trees in the first place or create a collapsed copy of each partial tree considered and apply the rules to it. The former approach can be realized by performing binary tree B&B as usual but ignoring all zero-length edges (and, in particular, never inserting a taxon into such an edge). This would speed computation by potentially reducing the number of partial tree subproblems spawned by any given parent subproblem, but it can be shown that doing so sometimes causes (collapsed) MP trees to be missed. In contrast, the latter approach is safe, but whether the overhead entailed would pay for itself is likely to be dataset dependent.

Regarding applications, we look to corrected parsimony (Hendy and Penny, 1993; Penny *et al.*, 1996; Steel *et al.*, 1993). Despite outperforming other non-ML methods in simulation tests (Charleston *et al.*, 1994), it appears that no reconstruction-accuracy comparison of corrected parsimony with ML has yet been done. We think this is an important oversight. Although XMP currently requires integer site pattern weights, the non-integer weights involved in corrected parsimony can be accommodated by scaling and truncation. In contrast to ML methods, which rely on heuristics like the Nelder–Mead algorithm (Nelder and Mead, 1965) to optimize the final ML score, this approach presents the intriguing possibility of recovering trees in which the error in the optimality criterion (total tree length) is *bounded*. Higher scaling and lower truncation thresholds produce tighter bounds at the expense of wider datasets and increased running time.

ACKNOWLEDGEMENTS

We wish to thank Joe Felsenstein, Andrey Zharkikh and Mike Hendy for helpful discussion concerning when an edge must, or must not, appear in all MP trees; David Bader and Rouven Naujoks for helpfully supplying datasets and program code; and the University of Canterbury's BlueFern supercomputing facility.

Conflict of Interest: none declared.

REFERENCES

Althaus,E. and Naujoks,R. (2006) Computing steiner minimum trees in Hamming metric. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, ACM, Miami, Florida.

Bachrach,A. *et al.* (2005) Lower bounds for maximum parsimony with gene order data. In *Comparative Genomics*, Vol. 3678 of *Lecture Notes in Computer Science*, pp. 1–10.

Bader,D.A. *et al.* (2006) ExactMP: an efficient parallel exact solver for phylogenetic tree reconstruction using maximum parsimony. In *Proceedings of the International Conference on Parallel Processing*, pp. 65–73.

Blumofe,R.D. and Leiserson,C.E. (1999) Scheduling multithreaded computations by work stealing. *J. ACM*, **46**, 720–748.

Bruen,T.C. and Bryant,D. (2008) A subdivision approach to maximum parsimony. *Ann. Combinatorics*, **12**, 45–51.

Charleston,M.A. *et al.* (1994) The effects of sequence length, tree topology, and number of taxa on the performance of phylogenetic methods. *J. Comput. Biol.*, **1**, 133–151.

Dinan,J. *et al.* (2008) A message passing benchmark for unbalanced applications. *Simul. Model. Pract. Theory*, **16**, 1177–1189.

Dinan,J. *et al.* (2009) Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, Portland, Oregon.

Felsenstein,J. (1978) Cases in which parsimony or compatibility will be positively misleading. *Syst. Zool.*, **27**, 401–410.

Farris,J.S. (1989) Hennig86, version 1.5. *Cladistics*, **5**, 163.

Felsenstein,J. (1989) PHYLIP – Phylogeny Inference Package (Version 3.2). *Cladistics*, **5**, 164–166.

Felsenstein,J. (2004) *Inferring Phylogenies*. Sinauer Associates, Inc., Sunderland, MA.

Fitch,W.M. (1971) Toward defining the course of evolution: Minimum change for a specified tree topology. *Syst. Zool.*, **20**, 406–416.

Goloboff,P.A. (1993) Character optimization and calculation of tree lengths. *Cladistics*, **9**, 433–436.

Goloboff,P.A. (1996) Methods for faster parsimony analysis. *Cladistics*, **12**, 199–220.

Goloboff,P.A. (1999) Analyzing large data sets in reasonable times: Solutions for composite optima. *Cladistics*, **15**, 415–428.

Goloboff,P.A. (2002) Optimization of polytomies: State set and parallel operations. *Mol. Phylogenet. Evol.*, **22**, 269–275.

Goloboff,P.A. *et al.* (2008) TNT, a free program for phylogenetic analysis. *Cladistics*, **24**, 774–786.

Graham,R.L. and Foulds,L.R. (1982) Unlikelihood that minimal phylogenies for a realistic biological study can be constructed in reasonable computational time. *Math. Biosci.*, **60**, 133–142.

Hartigan,J.A. (1973) Minimum mutation fits to a given tree. *Biometrics*, **29**, 53–65.

Hendy,M.D. and Penny,D. (1982) Branch and bound algorithms to determine minimal evolutionary trees. *Math. Biosci.*, **59**, 277–290.

Hendy,M. and Penny,D. (1993) Spectral analysis of phylogenetic data. *J. Classif.*, **10**, 5–24.

Hendy,M.D. *et al.* (1980) Proving phylogenetic trees minimal with l-clustering and set partitioning. *Math. Biosci.*, **51**, 71–88.

Holland,B.R. *et al.* (2005) The minmax squeeze: guaranteeing a minimal tree for population data. *Mol. Biol. Evol.*, **22**, 235–242.

Holzmann,G.J. (1997) The model checker SPIN. *IEEE Trans. Softw. Eng.*, **23**, 279–295.

Kimura,M. (1981) Estimation of evolutionary distances between homologous nucleotide sequences. In *Proc. Natl Acad. Sci. USA*, **78**, 454–458.

Kruskal,J.B.J. (1956) On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. Am. Math. Soc.*, **7**, 48–50.

Lamport,L. (1975) Multiple byte processing with full-word instructions. *Commun. ACM*, **18**, 471–475.

Moilanen,A. (1999) Searching for most parsimonious trees with simulated evolutionary optimization. *Cladistics*, **15**, 39–50.

Nei,M. and Kumar,S. (2000) *Molecular Evolution and Phylogenetics*. Oxford University Press, Oxford.

Nelder,J.A. and Mead,R. (1965) A simplex method for function minimization. *Computer J.*, **7**, 308–313.

Nixon,K.C. (1999) The parsimony ratchet, a new method for rapid parsimony analysis. *Cladistics*, **15**, 407–414.

Penny,D. and Hendy,M.D. (1987) Turbo tree: a fast algorithm for minimal trees. *Comput. Appl. Biosci.*, **3**, 183–187.

Penny,D. *et al.* (1996) Corrected parsimony, minimum evolution, and hadamard conjugations. *Syst. Biol.*, **45**, 596–606.

Purdom,P.W. *et al.* (2000) Single column discrepancy and dynamic max-mini optimizations for quickly finding the most parsimonious evolutionary trees. *Bioinformatics*, **16**, 140–151.

Ronquist,F. (1998) Fast fitch-parsimony algorithms for large data sets. *Cladistics*, **14**, 387–400.

Roshan,U.W. *et al.* (2004) Rec-I-DCM3: a fast algorithmic technique for reconstructing large phylogenetic trees. In *Proceedings of the IEEE Computational Systems Bioinformatics Conference, Stanford, CA*. IEEE Computer Society, CA, USA.

Schulmeister,S. (2004) Inconsistency of maximum parsimony revisited. *Syst. Biol.*, **53**, 521–521.

Semple,C. and Steel,M. (2003) *Phylogenetics*. Oxford Lecture Series in Mathematics. Oxford University Press, Oxford.

Siegel,S.F. (2007) Model checking nonblocking MPI programs. *Proceedings of Verification, Model Checking, and Abstract Interpretation*, Vol. 4349 of *Lecture Notes in Computer Science*, pp. 44–58.

Sridhar,S. *et al.* (2008) Mixed integer linear programming for maximum-parsimony phylogeny inference. *IEEE-ACM Trans. Comput. Biol. Bioinformatics*, **5**, 323–331.

Steel,M. (2001) Sufficient conditions for two tree reconstruction techniques to succeed on sufficiently long sequences. *SIAM J. Discrete Math.*, **14**, 36–48.

Steel,M. and Penny,D. (2000) Parsimony, likelihood, and the role of models in molecular phylogenetics. *Mol. Biol. Evol.*, **17**, 839–850.

Steel,M.A. and Penny,D. (2005) Maximum parsimony and the phylogenetic information in multistate characters. In Albert,V.A. (ed) *Parsimony, Phylogeny and Genomics*. Oxford University Press, Oxford, pp. 163–178.

Steel,M.A. *et al.* (1993) Parsimony can be consistent. *Syst. Biol.*, **42**, 581–587.

Swofford,D.L. *et al.* (1996) Phylogenetic inference. In Hillis,D.M., Moritz,C. and Mable,B.K. (eds) *Molecular Systematics*, 2nd edn., Sinauer Associates, Sunderland, MA, pp. 407–514.

Yan,M. and Bader,D.A. (2003) Fast character optimization in parsimony phylogeny reconstruction. *Technical report TR-CS-2003-53 from the University of New Mexico*. Available at http://www.cs.unm.edu/research/tech-reports/.

2.2 Correctness of Steiner Tree Lower Bounds for Ambiguous Nucleotides

In section 2.9 ("PARTBOUND Lower Bound") of the paper (p. 25), we mention that the calculation of lower bounds for collections of 3 or more sequences is complicated by the existence of ambiguous nucleotides. This section proves the correctness of these bounds.

2.2.1 Without Ambiguous Nucleotides

For the 3-component Steiner tree case, call the 3 1-connected components A, B, C and suppose initially that there are no ambiguous nucleotides. Let $d(u, v)$ be the Hamming distance (number of different nucleotides) between two sequences u and v, and let

$$x = \min(d(u, v)) \text{ for any } u \in A \text{ and } v \in B$$

$$y = \min(d(u, v)) \text{ for any } u \in B \text{ and } v \in C$$

$$z = \min(d(u, v)) \text{ for any } u \in A \text{ and } v \in C$$

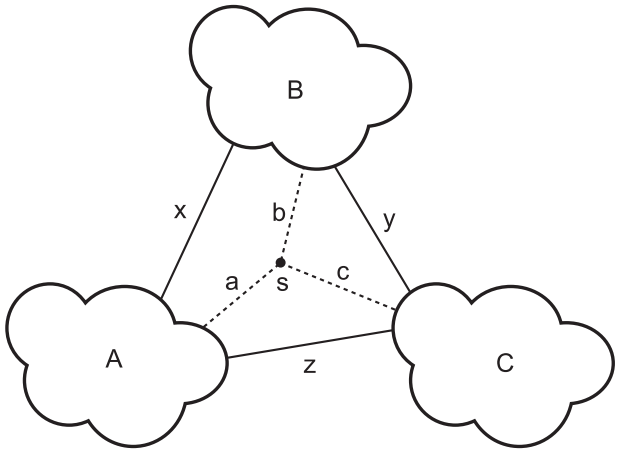


Figure 2.1: Introducing a Steiner point s

Now suppose that we introduce a Steiner point s, and connect all 3 components to this new sequence as shown in Figure 2.1. Let a be the shortest distance between s and any sequence in A, b be the shortest distance between s and any sequence in B, and c be the shortest distance between s and any sequence in C.

These are true distances, so by the triangle inequality, we have that:

$$a + b \geq x$$
$$b + c \geq y$$
$$a + c \geq z$$

Summing both sides and dividing by 2, we get:

$$a + b + c \geq \frac{1}{2}(x + y + z)$$

$a+b+c$ is the (unknown) length of the edges added to connect the 3 components. The above inequation says that $\frac{1}{2}(x + y + z)$ (which we can readily compute) is a lower bound for this quantity, *provided* that there are no ambiguous nucleotides. (This is actually a weaker bound than that given by Hendy *et al.* (1980, section 4).)

2.2.2 With Ambiguous Nucleotides

An ambiguous nucleotide can be regarded as a set of specific nucleotides. In order to generalise the above lower bound to work with sequences containing ambiguous nucleotides, we need to consider all of the quantities x, y, z, a, b, c as functions of a function ϕ that maps every ambiguous nucleotide P in the input dataset to a specific nucleotide $p \in P$. Call these ϕ-dependent quantities $x_\phi, y_\phi, z_\phi, a_\phi, b_\phi, c_\phi$ respectively.

Under any valid assignment ϕ of specific nucleotides to ambiguous nucleotides:

$$a_\phi + b_\phi + c_\phi \geq \frac{1}{2}(x_\phi + y_\phi + z_\phi)$$

Given two possibly ambiguous nucleotides P and Q, define

$$\text{mindist}(P, Q) = \begin{cases} 1 & \text{if } P \cap Q = \emptyset \\ 0 & \text{otherwise} \end{cases}$$

If $|P| = |Q| = 1$ (i.e. when both nucleotides are unambiguous), the function mindist gives the distance between the two nucleotides; in other cases it gives a lower bound on this distance, optimistically assuming that a common nucleotide will be assigned whenever possible. mindist can be extended to sequences by summing over corresponding nucleotides.

We seek a lower bound for $\min(a_\phi + b_\phi + c_\phi)$, where the minimum is taken over all possible assignments ϕ. (In the worst case, when every nucleotide in the $n \times k$ dataset is $\mathtt{N} = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$, there are 4^{nk} possible assignments, so enumerating them is infeasible.)

Let

$$\begin{aligned}
\hat{x} &= \min(\text{mindist}(u, v)) \text{ for any } u \in A \text{ and } v \in B \\
\hat{y} &= \min(\text{mindist}(u, v)) \text{ for any } u \in B \text{ and } v \in C \\
\hat{z} &= \min(\text{mindist}(u, v)) \text{ for any } u \in A \text{ and } v \in C
\end{aligned}$$

Clearly $\hat{x} \leq x_\phi \; \forall \phi$, and similarly for $\hat{y}$ and $\hat{z}$. Thus for all ϕ,

$$\frac{1}{2}(\hat{x} + \hat{y} + \hat{z}) \leq \frac{1}{2}(x_\phi + y_\phi + z_\phi) \leq a_\phi + b_\phi + c_\phi$$

so in particular

$$\frac{1}{2}(\hat{x} + \hat{y} + \hat{z}) \leq \min_\phi(a_\phi + b_\phi + c_\phi)$$

giving the desired lower bound.

2.3 Zero-length Edges

In section 4.1 ("Possible Future Directions") of the paper we mention "maintaining collapsed edges during B&B" as a possible performance enhancement that we found to be unworkable. The following two sections discuss this issue more thoroughly.

The B&B enumeration performed by `xmp` effectively considers all $(2n-5)!!$ possible unrooted binary trees on n taxa in which each taxon labels a unique leaf—what Semple & Steel (2003) call *binary phylogenetic X-trees*. We call these trees *fully refined*, and all other trees (those containing multifurcations, internal nodes labelled with taxa, or nodes labelled with multiple taxa) *partially refined*. `xmp` generates each fully refined tree at most once, skipping only those trees for which a lower bound on the parsimony score exceeds a known upper bound: this guarantees that every fully refined tree that could possibly be an MP tree is tested. The Fitch algorithm (Fitch, 1971), which operates on rooted binary trees, is accommodated by "folding" the edge from an arbitrarily chosen reference taxon and placing the *calculation root* at this new node.

The weight (or length, or parsimony score) of an edge e or tree T is designated w_e or w_T respectively. Semple & Steel (2003) formulate an assignment of hypothetical sequences to the internal nodes of a tree as an extension of a partial function; following this we shorten "assignment of sequences to internal nodes" to *extension*.

2.3.1 Sufficiency of Examining Fully Refined Trees

Any partially refined tree T with $m < 2n - 3$ edges can be *refined* into a fully refined tree T' by a suitable series of $2n - 3 - m$ node expansions in which a node v is replaced with a *refining edge* xy whose endpoints partition both the neighbours and the label set of v. $w_{T'} \leq w_T$ because it is always possible to assign to both x and y the same sequence that was assigned to v, and thereby ensure that all inserted edges have length 0; when refining a multifurcation, sometimes multiple parallel substitutions on the spokes can be swapped for a single substitution on the new edge, reducing $w_{T'}$ below w_T. (In general refinement is not unique, and different refinements may produce trees with different weights.) From this it follows that contracting an edge can never reduce the parsimony score, so there is no need to examine partially refined trees to discover all fully refined MP trees.

2.3.2 Avoiding the Deluge

The downside of `xmp`'s fully refined tree search is that, even if every refinement T' of a partially refined MP tree T has $w_{T'} = w_T$, all these refinements will be returned as individual MP trees. In this situation all refining edges are purely artefacts of the B&B process and contain no biological information—they are in fact misleading. Particularly troubling is the fact that this situation occurs more frequently for datasets having few mutations, such as intraspecific datasets, which is where MP is most accurate and therefore most appealing. Clearly it would be preferable to return just T in this case, which expresses the same information more compactly.

At first it seems this could be done by somehow enumerating progressively further refined trees, halting when no further refinement can produce a tree with lower weight. However, such an approach must be able to detect any lower-weight trees lurking among the possible refinements of any given partially refined tree, and it's not clear how this could be accomplished faster than by trying all such refinements—a process that essentially mirrors the explicit enumeration of fully refined trees.[1]

Therefore a better approach is to continue enumerating fully refined trees as before, while looking for edges that may be safely contracted. Two major types of edges present themselves as candidates:

1. *Minimum-length-zero* edges: Contract any edge e for which there exists an MP extension yielding $w_e = 0$.

2. *Maximum-length-zero* edges: Contract any edge e for which there does not exist an MP extension yielding $w_e > 0$.

The difficulty with minimum-length-zero edges is that contracting one such edge may cause another such edge to require at least 1 substitution under any MP extension. Thus in general there is no unique maximally contracted form of a tree with respect to this criterion.

In contrast, contracting a maximum-length-zero (hereafter *ml0*) edge does not disturb the set of MP extensions (when restricted to the remaining nodes), meaning that all such edges in a given tree T can be contracted in any order to produce a unique maximally contracted tree collapse(T). We call this operation *collapsing*, and we say that a tree devoid of ml0 edges is *dense*. Clearly $w_{\text{collapse}(T)} = w_T$. If T is an MP tree, then every refinement of collapse(T) is also an MP tree, since each refining edge added can neither increase the MP score (see Section 2.3.1) nor decrease it (since that would contradict the minimality of T). This makes collapse(T) a compact way to exactly represent a set of fully refined MP trees.

[1]An exception exists for the rare case in which a *perfect phylogeny* exists for the dataset—that is, a tree having weight equal to the SCD bound. Then there is exactly 1 minimally resolved MP tree, which can be discovered in O(n) time using the Tree-Popping algorithm (Meacham, 1981).

2.3.3 Minimally Refined Trees

Call a tree T *minimally refined* if every refinement T' of T has $w_{T'} = w_T$, and no edge in T can be contracted to produce a tree with this property. Ideally `xmp` should report only minimally refined MP trees to reduce the amount of repetitive output. We note that contracting ml0 edges does not in general produce minimally refined MP trees: Figure 2.2a shows a 5-taxon dense MP tree with weight 3, which could in principle be reported as the sole MP tree because no refinement of it has lower weight; however the process of contracting ml0 edges in fully refined trees will not discover this tree because for each of its 3 fully refined MP refinements there exists an MP extension that involves a substitution on a refining edge, as shown in Figure 2.2b. All this means is that collapsing ml0 edges is not maximally effective in concisely representing all MP trees.

Nevertheless, culling ml0 edges is effective in stopping the worst excesses. Particularly when taxa are highly similar, the number of ml0 edges in a tree can become high. In the worst case—n identical taxa—there are $(2n - 5)!!$ fully refined MP trees, each consisting exclusively of ml0 edges. Even in real datasets, retention of these edges is an inconvenience: the its36 dataset examined in the paper has 62370 distinct fully refined trees, but after collapsing all ml0 edges, just 320 distinct trees remain.

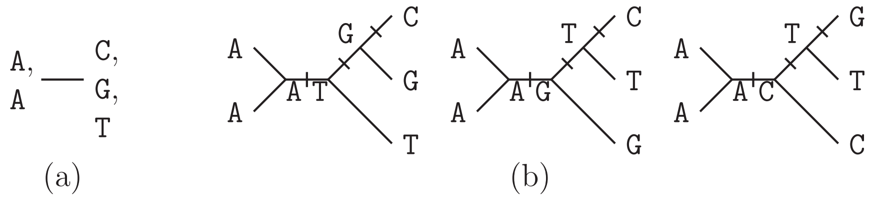


Figure 2.2: (a) A minimally refined MP tree and (b) its 3 full refinements

2.3.4 XMP Tree Representation

Before discussing how ml0 edges can be identified, it is first necessary to explain how `xmp` represents trees during B&B.

The main work of `xmp`'s B&B inner loop consists of:

1. Calculating the weight that would be added by inserting a given taxon at a given edge

2. Updating the tree representation to reflect a newly inserted or removed taxon

The Fitch algorithm (Fitch, 1971) allows the weight of a tree to be calculated in a single postorder traversal that constructs, for each internal node v, a *preliminary state set* giving the set of nucleotides that may appear at v in any MP extension for the subtree at and below v. The naive approach of recalculating the entire tree using the first pass of the Fitch algorithm every time a taxon is added or removed performs much unnecessary recomputation; an obvious improvement is to record weights for each subtree, and to recompute only nodes on the path between a newly inserted or removed taxon and the root. Going further in this direction, it is possible to traverse this path for each site independently, halting recalculation of a site as soon as a state set is produced that is equal to the previously calculated version (Gladstein, 1997). In practice the time saved by this "data-driven" approach is outweighed by the overhead of performing tree traversal k times, and performance is hampered further if multiple sites are packed into a machine word, so we never considered it for `xmp`.

Both improvements to the basic Fitch algorithm described above still require $O(n)$ time to calculate how much weight is added to a tree by inserting a given taxon at a given edge. In contrast, the algorithm of Yan & Bader (2003) enables each insertion to be evaluated in amortised constant time, by maintaining a binary tree in which any edge can behave as the Fitch calculation root.[2] This is the algorithm `xmp` uses.

Accordingly, `xmp` maintains a binary tree in which each edge is labelled by three sequences: the *postorder*, *preorder* and *final* Fitch sequences for that edge. Extending the notation of Yan & Bader (2003), for an edge xy we denote these sequences $\overrightarrow{xy}$, $\overleftarrow{xy}$ and $\widehat{xy}$, respectively. $\overrightarrow{xy}$ is the preliminary state set of x from the standard Fitch algorithm; $\overleftarrow{xy}$ is the preliminary state set of y that would result if the Fitch calculation root were on xy; $\widehat{xy}$ is the preliminary state set of a hypothetical Fitch calculation root on xy. As with the standard Fitch algorithm, a calculation root is created by choosing an arbitrary reference taxon and "folding" the edge from that

[2]Although Goloboff (1993) also claimed amortised constant time for his algorithm, Goloboff (1996, p. 204) concedes that errors can occur if the dataset contains ambiguous nucleotides. In general, the concise and formal description given by Yan & Bader (2003), which includes a proof of correctness, inspires greater confidence than do Goloboff's sometimes-ambiguous explanations.

taxon; all edges are considered to be directed towards this root.

2.3.5 Identifying ml0 Edges

The next step is to identify ml0 edges from the tree representation `xmp` uses during B&B. For simplicity we consider a single site; since parsimony scores are additive across sites, an edge is ml0 iff the edge for each site is ml0.

Theorem 2.1 explains the connection between $\overrightarrow{xy}$, $\overleftarrow{xy}$ and ml0 edges.

Theorem 2.1. *For a given edge xy, $\overrightarrow{xy} \cap \overleftarrow{xy} \neq \emptyset \iff xy$ is ml0.*

Proof. By the correctness of the Fitch algorithm, for any edge xy and nucleotide s, there exists an MP extension for the subtree including and below x in which x is assigned nucleotide s iff $s \in \overrightarrow{xy}$. Similarly, for any edge xy and base s, there exists an MP extension for the subtree including and above y in which y is assigned nucleotide s iff $s \in \overleftarrow{xy}$. Let $Z = \overrightarrow{xy} \cap \overleftarrow{xy}$. If $Z \neq \emptyset$ then any $s \in Z$ can be assigned to both x and y to produce a zero-length edge connecting two minimal-length subtrees, which is therefore a tree of minimal length; any extension for which xy has length $\delta > 0$ increases the total tree length by at least δ and thus cannot be minimal. Therefore $Z \neq \emptyset \implies xy$ is ml0.

To show the reverse direction, suppose there is an ml0 edge xy for which $\overrightarrow{xy} \cap \overleftarrow{xy} = \emptyset$. In this case, all extensions in which xy is maintained at zero length by assigning the same nucleotide to both x and y incur a length increase of at least 1: choosing any $s \in \overrightarrow{xy}$ increases the length of the subtree including and above y by at least 1, while choosing any $s \in \overleftarrow{xy}$ increases the length of the subtree including and below x by at least 1. The remaining possibility is to assign any $a \in \overrightarrow{xy}$ to x and any $b \in \overleftarrow{xy}$ to y, allowing both subtrees to retain minimal length but incurring 1 substitution on xy. This is clearly a minimal extension (although not necessarily uniquely minimal), contradicting the assumption that xy is ml0. Therefore $\overrightarrow{xy} \cap \overleftarrow{xy} \neq \emptyset \iff xy$ is ml0. □

2.3.6 Contracting ml0 Edges

Now that edges that can safely be contracted have been identified, two broad strategies can be considered:

1. Perform B&B enumeration of fully refined trees as before. In each fully refined MP tree discovered, contract all ml0 edges and report only the unique trees that result.

2. Adjust the enumeration algorithm so that these edges are never generated in the first place.

The first approach is eminently correct, but unsatisfying because it involves doing a great deal of work and then discarding most of it. The sheer number of meaningless bifurcations evidenced by the ratio of fully refined MP trees to collapsed MP trees for datasets like its36 suggests that there is structure in the problem that is not being fully exploited by the B&B algorithm. In particular it appears that it should be possible to avoid inserting taxa at ml0 edges during B&B without compromising the search. We call this approach *precollapsed enumeration*. Unfortunately, this turns out not to be the case.

For precollapsed enumeration to be correct, we require that any MP tree that can be produced by a sequence of taxon insertions involving at least one insertion into an ml0 edge, collapses to a tree that can also be produced by inserting taxa into only non-ml0 edges, and collapsing the result.

2.3.7 A Counterexample

Figure 2.3 shows a counterexample to the correctness of precollapsed enumeration involving 6 single-site sequences, most of which are ambiguous nucleotides. Trees on the left are fully refined trees, with ml0 edges represented by circles; trees on the right are the corresponding dense (collapsed) trees. Figure 2.3a shows an initial 5-taxon tree T, and Figures 2.3b 2.3c show the result of inserting a new taxon having sequence `C` at two different ml0 edges in T, producing trees T_1 and T_2 respectively. Both these trees have weight 2, which is optimal for the 6-taxon dataset. However they collapse to distinct dense trees, and there is only one non-ml0 edge in T, so at least one of collapse(T_1) and collapse(T_2) cannot be recovered by inserting the taxon only at non-ml0 edges in T. It follows that precollapsed enumeration does not guarantee recovery of all dense MP trees. For this reason, `xmp`'s fully refined B&B tree enumeration was left unchanged.

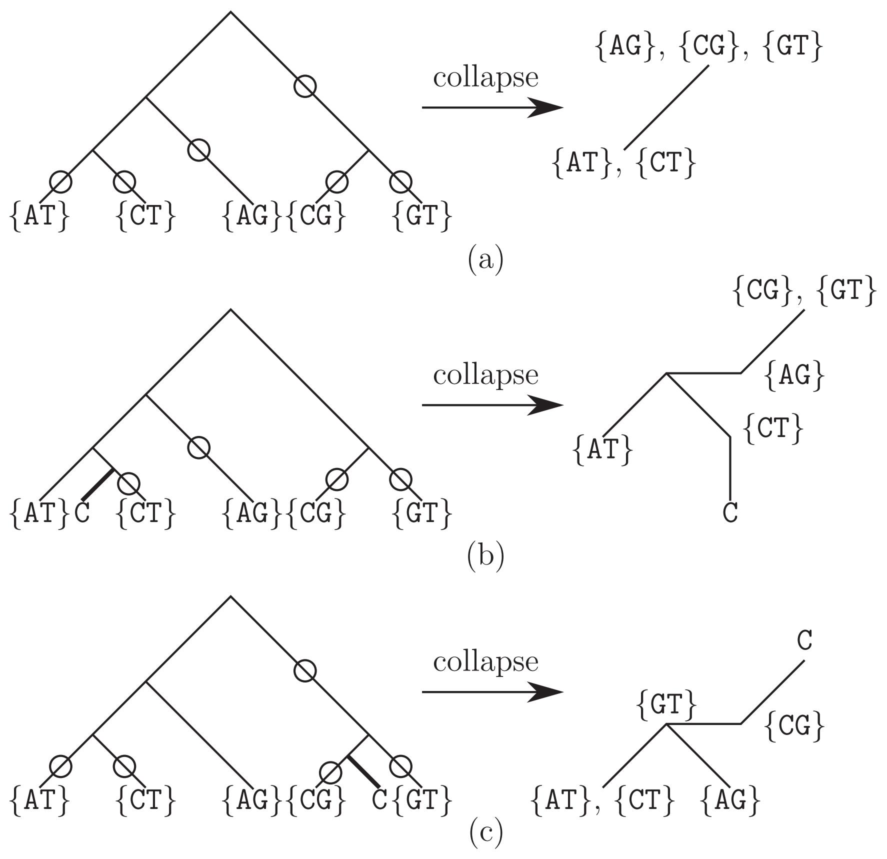


Figure 2.3: Counterexample showing trees missed by precollapsed enumeration

2.4 Zharkikh's Rules

Felsenstein (2004, pp. 66–67) describes several rules for reducing the search space needed for exact MP search, which he attributes to a Russian-language paper by Zharkikh (1977). Rule #3 on p. 66 is particularly appetising, because if applicable it effectively reduces the taxon count by collapsing similar taxa into groups:

> Look at all states of all characters. For each one, let the state define the membership of a group S. Calculate the number of states (over all characters) that are shared by members of group S but that do not appear anywhere else. Call this number $n_0(S)$. Compute the distances between all pairs of species i and j that are in S. The distance is in this case the number of characters that differ between the species. If the largest value of D_{ij} among all these pairs of species is less than $n_0(S)$, then the group S must appear on all most parsimonious trees. It can now be collapsed to a single fictional species, which has its state computed from a Fitch parsimony algorithm. Thus any states that are shared by

all members of the group appear in the new species, and otherwise its state is an ambiguity between some of the possibilities within the group.

However, the tree in Figure 2.4a is an MP tree with 2 substitutions, even though the two `A` taxa are not grouped together—this contradicts a straightforward interpretation of Zharkikh's rule.

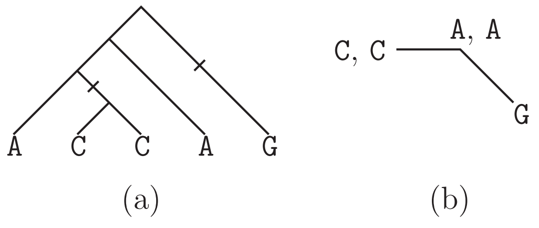


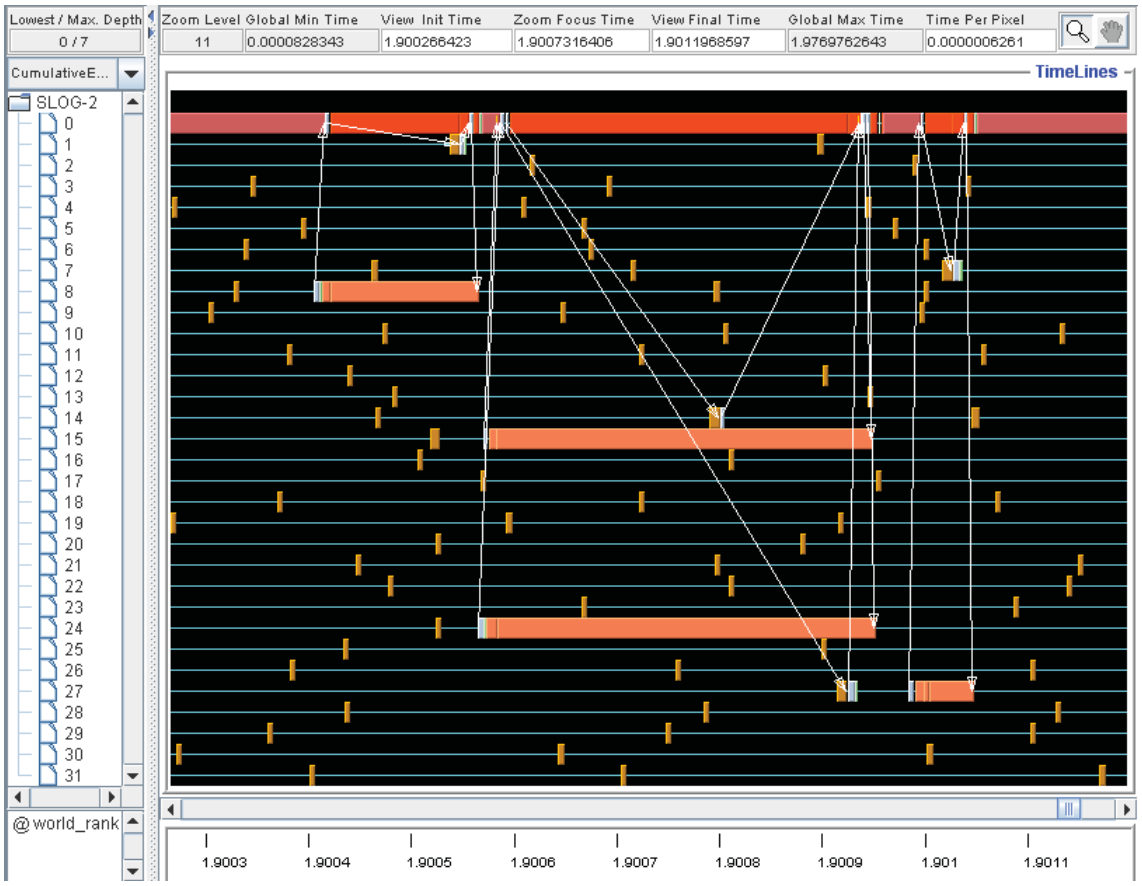
Figure 2.4: Two representations of a tree that contradicts Zharkikh's rule #3

Personal communication with Felsenstein and Zharkikh suggests that the problem stems from the existence of ml0 edges in the tree. Certainly the dense tree in Figure 2.4b that results from collapsing the tree in Figure 2.4a does not exhibit this problem. However as previous sections have established, we were unable to find a way to safely and efficiently enumerate only dense trees, and so this rule has not been implemented in `xmp`.

2.5 Visualising MPI Communication

The MPI message-passing interface includes a useful tracing facility that produces logfiles that can be graphically displayed with the program Jumpshot.

Figure 2.5 shows the activity of each process during a 800µs time segment in a 32-process run of the parallel version of `xmp`. Each process occupies a row, with the boss at the top. Time in seconds runs along the x axis. Narrow brown boxes represent periodic `MPI_Test()` calls performed by busy workers to poll for new upper bounds or steal requests; white arrows are messages sent between processes. Peach-coloured boxes represent worker calls to `MPI_Waitany()`, during which the worker is stalled, waiting for work to arrive. During the time segment shown, processes 8, 24, 15 and 27 request more work from the boss, which steals from the randomly selected processes 1, 14, 27 and 7 to satisfy these requests. Steal requests from processes 24 and 15 arrive in quick succession and are forwarded in tandem by the boss. On this


Lowest / Max. Depth 0 / 7

Zoom Level 11

Global Min Time 0.0000828343

View Init Time 1.900266423

Zoom Focus Time 1.9007316406

View Final Time 1.9011968597

Global Max Time 1.9769762643

Time Per Pixel 0.0000006261

Figure 2.5: 800µs segment of Jumpshot trace for 32-process parallel `xmp`

occasion all steal victims respond by sending back jobs, although it is possible for the boss to attempt to steal from a worker that has just run out of work itself, in which case the victim will deny the steal request and the boss will attempt to steal from a different worker.

Figure 2.6 shows the last 25ms of a ~2s run, clearly showing how work is running out and worker processes spend increasing amounts of time waiting for work to arrive. The blue boxes depict the synchronous send operations used by workers to send tree data back to the boss for output after the main B&B phase has completed. In this case, 13 of the 31 worker processes had optimal trees to report.

2.6 Lower Bound Comparison

Before the B&B phase begins, for each partial tree size `xmp` calculates a lower bound on the weight that must be added by all remaining taxa. (Recall that `xmp` uses a static taxon addition order, so the number of taxa on a partial tree determines which

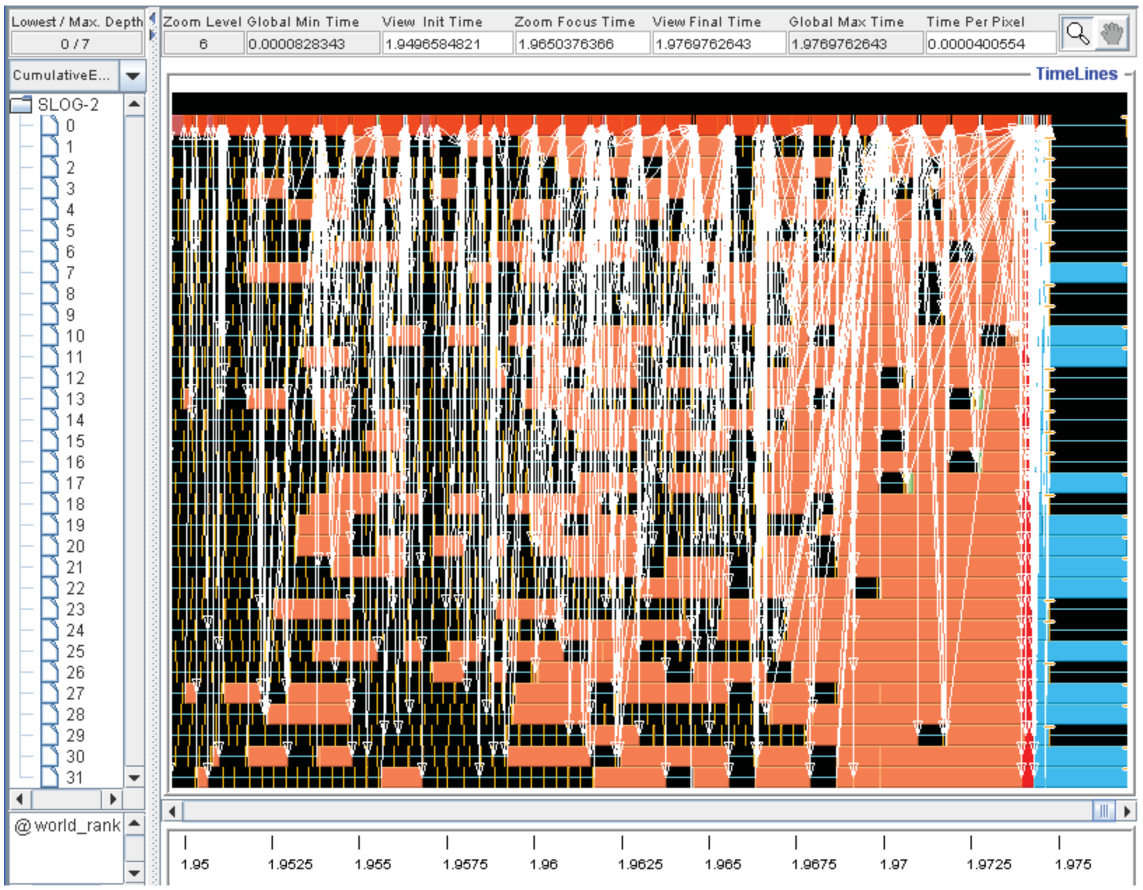

Lowest / Max. Depth: 0 / 7
Zoom Level: 6
Global Min Time: 0.0000828343
View Init Time: 1.9496584821
Zoom Focus Time: 1.9650376366
View Final Time: 1.9769762643
Global Max Time: 1.9769762643
Time Per Pixel: 0.0000400554

Figure 2.6: Jumpshot trace of last 25ms of 32-process parallel `xmp`

taxa are present.) For each partial tree considered during B&B, the corresponding lower bound will be added to the tree's weight to determine whether an optimal tree can possibly result from adding more taxa to this partial tree, so it is important to be able to quickly compute high lower bounds to avoid wasting time evaluating trees that cannot possibly be optimal. Figure 2.7 compares the three different strategies `xmp` provides for computing these lower bounds on the 21 datasets examined. `-Bp`, `-Bd` and `-Bdi` correspond to the newly introduced PARTBOUND bound, the single column discrepancy (SCD) bound and the sum of the SCD and incompatibility bounds respectively. Partial tree size is shown on the x axis, with the corresponding minimum additional weight shown on the y axis.

In general, `-Bp` dominates, although `-Bdi` is better in 3 isolated cases: 10 (vs. 9) for 10-taxon trees on 32hum, 8 (vs. 7) for 11-taxon trees on 32hum, and 12 (vs. 11) for 14-taxon trees on e4. As discussed in the paper, the superiority of the `-Bp` and `-Bdi` bounds at small tree sizes seldom has a significant effect on overall runtime.

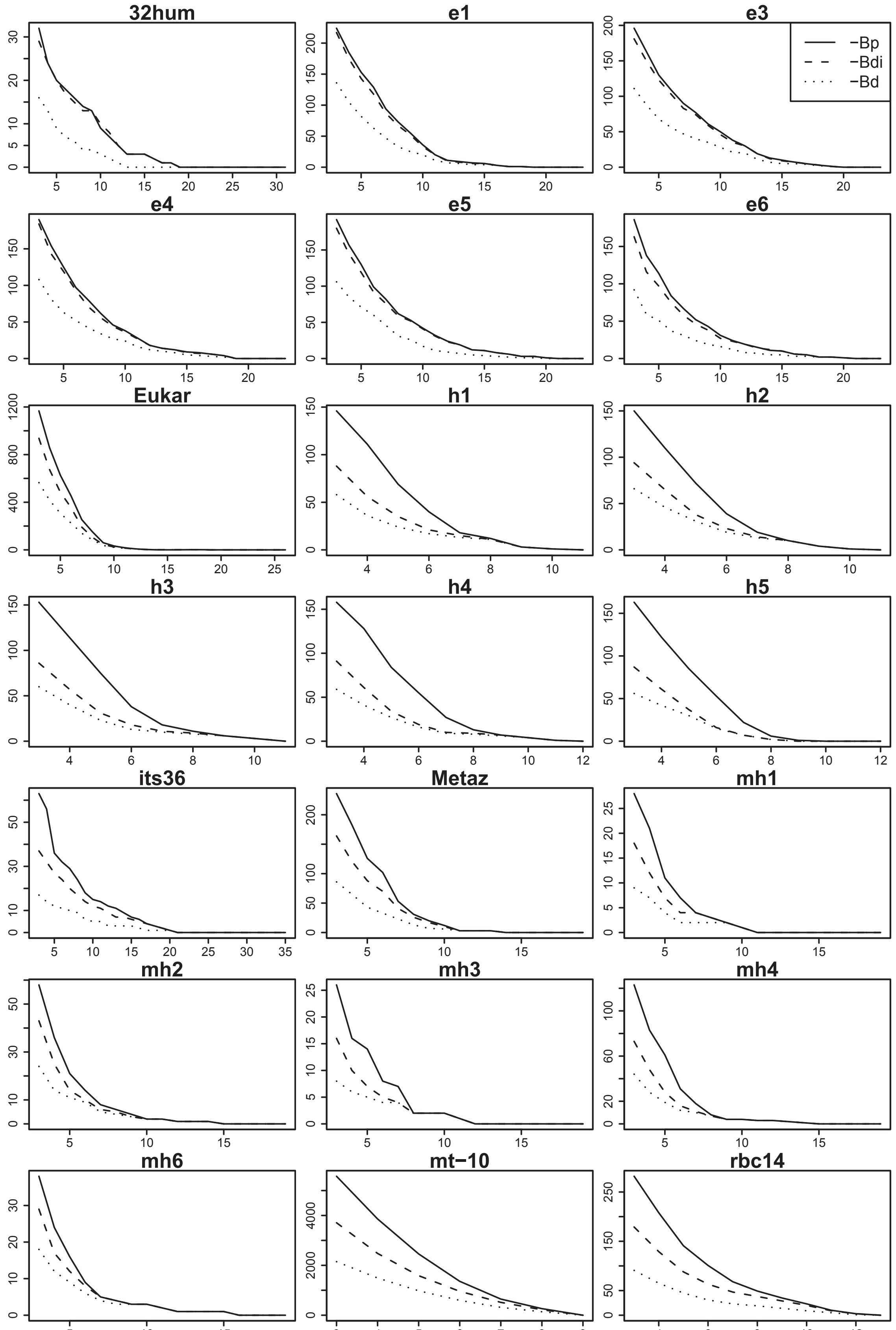


Figure 2.7: Lower bounds for the weight added by all remaining taxa (y axis) vs. partial tree size (x axis), computed using 3 lower-bounding strategies. Results for each dataset are shown on separate plots. Higher values are associated with improved search space pruning and faster search times.

Chapter 3

Compressing DNA Sequence Databases with coil

3.1 Introduction

This chapter presents the paper "Compressing DNA Sequence Databases with `coil`", which was published in the journal *BMC Bioinformatics* in 2008.

All software associated with this paper is now freely available via Subversion from `http://coildna.sourceforge.net`.

BMC Bioinformatics

BioMed Central

Software

Open Access

Compressing DNA sequence databases with coil

W Timothy J White* and Michael D Hendy

Address: Allan Wilson Centre for Molecular Ecology and Evolution, Massey University, Palmerston North, New Zealand

Email: W Timothy J White* - w.t.white@massey.ac.nz; Michael D Hendy - m.hendy@massey.ac.nz

* Corresponding author

Published: 20 May 2008

BMC Bioinformatics 2008, **9**:242 doi:10.1186/1471-2105-9-242

Received: 23 October 2007
Accepted: 20 May 2008

This article is available from: http://www.biomedcentral.com/1471-2105/9/242

Abstract

Background: Publicly available DNA sequence databases such as GenBank are large, and are growing at an exponential rate. The sheer volume of data being dealt with presents serious storage and data communications problems. Currently, sequence data is usually kept in large "flat files," which are then compressed using standard Lempel-Ziv (gzip) compression – an approach which rarely achieves good compression ratios. While much research has been done on compressing individual DNA sequences, surprisingly little has focused on the compression of entire databases of such sequences. In this study we introduce the sequence database compression software coil.

Results: We have designed and implemented a portable software package, coil, for compressing and decompressing DNA sequence databases based on the idea of *edit-tree coding*. coil is geared towards achieving high compression ratios at the expense of execution time and memory usage during compression – the compression time represents a "one-off investment" whose cost is quickly amortised if the resulting compressed file is transmitted many times. Decompression requires little memory and is extremely fast. We demonstrate a 5% improvement in compression ratio over state-of-the-art general-purpose compression tools for a large GenBank database file containing Expressed Sequence Tag (EST) data. Finally, coil can efficiently encode incremental additions to a sequence database.

Conclusion: coil presents a compelling alternative to conventional compression of flat files for the storage and distribution of DNA sequence databases having a narrow distribution of sequence lengths, such as EST data. Increasing compression levels for databases having a wide distribution of sequence lengths is a direction for future work.

Background

The advent of the Sanger sequencing method enabled DNA sequence data to be collected and manipulated on computers, paving the way for explosive growth in the new field of bioinformatics. Publicly available DNA sequence databases such as GenBank play a crucial role in collecting and disseminating the raw data needed by researchers in the field. This database currently contains 168 Gb of sequence data [1] section 2.2.8, and is expected to continue to grow at an exponential rate, doubling in size roughly every 14 months [2]. The volume of data being dealt with now presents serious storage and data communications problems. Currently, sequence data is usually kept in large "flat files," which are then compressed using standard Lempel-Ziv compression [3] (e.g. with gzip [4]). Unfortunately this approach rarely achieves good compression ratios: typically, gzip fails to match the "compression" afforded by simply encoding each base using 2 bits [5].

Previous work concerning the compression of biological (DNA or protein) sequences can be divided into two categories: techniques developed for efficiently compressing sequence data for the sake of reduced resource consumption (disk space or network usage) [5-9]; and investigations of the usefulness of compressibility as a measure of information content, for the purpose of making inferences about sequences (such as the relatedness of two sequences) [10,11]. In this article we will focus on work in the former category. Examining this body of work reveals two distinct approaches:

• Compressing individual biological sequences

• Compressing databases of biological sequences

Compressing individual biological sequences

It is now widely recognised that DNA data is inherently difficult to compress below the level of 2 bits per base achievable through direct encoding [5,6,9]. Much research has gone into developing algorithms for more effectively compressing individual DNA sequences. These include BioCompress [6], BioCompress-2 [7], GenCompress [8], the CTW+LZ algorithm [5], and DNACompress [9]. Perhaps the best of these is DNACompress, which employs the PatternHunter [12] sequence search algorithm to discover patterns of approximate repeats or approximate palindromic repeats in sequence data. DNACompress achieved compression averaging 13.7% on a sample set of DNA sequences and is substantially faster than earlier algorithms. Grumbach and Tahi [7] allude to a "vertical" mode of compression for compressing multiple sequences in a database, however they do not elaborate on how this might be accomplished.

While these single-sequence algorithms are interesting from a theoretical point of view, and are certainly becoming increasingly practical in the modern world of genome-scale analysis, a great deal of everyday bioinformatics work continues to entail the communication and storage of multiple sequences, and the modest compression gains afforded by these algorithms are ultimately not sufficient to justify their adoption for large databases.

Compressing databases of biological sequences

Strelets and Lim [13] describe a program, SAGITTARIUS, for compressing PIR-format [14] protein sequence databases. Their system uses standard dictionary-style compression of sequence entry metadata, and a novel alignment-based compression strategy for the protein sequence data itself. A small number of sequences is maintained in memory as the *reference sequence accumulator*, and each sequence in the database is aligned with each sequence in this list. If any alignment produces a strong match, the input sequence is recoded using symbols describing insertions and deletions to enable recovery from its close match in the accumulator; otherwise, the sequence is output verbatim and added to the accumulator, overwriting the oldest incumbent sequence if the accumulator is full. Sequences to be output are compressed using run-length encoding and Huffman encoding, and the shorter of the two encodings is chosen. Thus the accumulator represents a window of recently encountered interesting sequences. The authors set the size of the accumulator at three sequences, and were able to achieve 2.50:1 compression, significantly better than PKZIP® [15] at 2.13:1.

Strelets and Lim [13] were interested in producing a compressed database that could be used interactively in much the same way as the original database. This was facilitated in part by the fact that their approach never requires recursive decoding of sequences – each sequence is encoded in terms of at most one other sequence, which is itself available "as-is," (i.e. not compressed in terms of another sequence). While useful for interactive operations, it is clear that avoiding recursive encoding must limit the overall level of compression obtained. Since we are targeting maximum compression, coil differs from that of [13] in this respect. Another difficulty arises in the assumption that similar sequences are likely to appear near each other in the input file. This is crucial in order to be able to limit the size of the accumulator and thereby the runtime. The authors found that increasing the size of the accumulator past three sequences increased the runtime but made no substantial improvement in compression, which appeared to justify their assumption. Unfortunately, while this neat localisation of similar sequences may have been true of the PIR database in 1995, it is certainly not true of the large nucleotide databases of today, and we chose not to make this assumption.

Li, Jaroszewski and Godzik have taken a similar approach to the related problem of producing non-redundant protein databases with their CD-HI [16] and CD-HIT [17] packages. More recently, Li and Godzik have extended this approach to DNA sequences with the cd-hit-est program [18]. Their main advance over [13] is in employing short-word filters to rapidly determine that two sequences cannot be similar, which significantly reduces the number of full alignments necessary. Despite impressive speed on small-to-medium datasets, they report that clustering 6 billion ESTs at 95% similarity takes 139 hours [18].

The program nrdb [19] locates and removes exact duplicate sequences from a DNA database in FASTA format. While this program is clearly a step in the right direction,

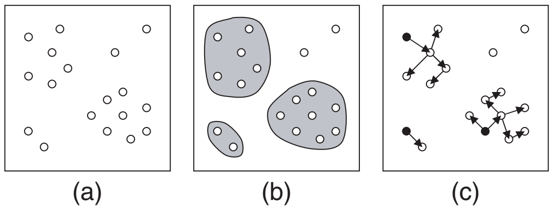


Figure 1
Edit-tree coding of similar sequence groups. Circles represent DNA sequences in a database; the straight-line distance between circles represents the edit distance between sequences. Initially (a) we are presented with the input database. In the first step (b), groups of similar sequences are discovered. In the second step (c), each group is edit-tree coded independently by determining a reasonable tree, selecting a root sequence (coloured black) and recording the necessary edits along each edge. Some sequences are not sufficiently similar to any other sequence to be delta-encoded – these sequences will be recorded verbatim.

many sequences in a typical database are almost but not quite exact duplicates of other sequences (perhaps differing at one or two positions), highlighting opportunities for further improvements.

Compression and the maximum parsimony criterion

A *phylogenetic tree* is a Steiner tree estimating the evolutionary history of a set of *taxa* (species, genes or individuals). *Maximum parsimony* is a criterion for building phylogenetic trees from DNA sequences that aims to identify the tree or trees containing the fewest *point mutations* along their edges, where a point mutation, or *edit*, is an insertion or deletion of a single nucleotide or a substitution of one nucleotide for another. (Most implementations of maximum parsimony search consider only point substitutions, for reasons of computational efficiency.) We then note:

If we are given the complete sequence at one node of a tree, as well as all edge mutations, we can reconstruct the sequences at all the nodes.

Thus a maximum parsimony tree represents an optimal solution to storing sequence data in the form of a list of edit operations on a tree rooted at a single reference sequence. This can be an efficient compression if the sequences are closely related, so that the number of edit operations is small in comparison with the total sequence length. Within a large database, it is expected that there will be large groups of closely related sequences – for example, the DNA encoding a particular gene from many different species. More precisely, we expect that many sequences will be highly similar to at least one other sequence in the database. If this is the case a considerable saving in storage space can be achieved by identifying such groups, determining good trees for them, and encoding each group as a single root sequence plus a series of "deltas" along the tree edges. We have called this approach *edit-tree coding*. Figure 1 illustrates how sequences within a database are processed according to this scheme.

In practice, it soon becomes apparent that even a heuristic maximum parsimony search on subsets of sequences is not computationally feasible for a large database. The standard maximum parsimony tree evaluation algorithm requires all sequences to be *aligned*. Both alignment and the subsequent parsimony searches are hard problems [20,21]. Fortunately, when dealing with data compression we are not concerned about exactly maximising some function – our requirement is a method which is fast and performs well on typical cases. A practical alternative to maximum parsimony search is to construct an approximation to the *minimum spanning tree* on the sequences, where the metric is the *edit distance* between two sequences – the number of single-character insertions, deletions or replacements required to transform one sequence into the other. Unlike Steiner trees, minimum spanning trees do not introduce new internal vertices, and computation is fast: an algorithm having time complexity almost linear in the number of edges exists [22]. The total tree length is bounded by twice that of the maximum parsimony tree. By judicious selection of algorithms and data structures, we have developed heuristics and approximations that make this task feasible for databases having sizes in the gigabyte range, despite having essentially quadratic time complexity in the size of the database.

Goals of coil

Our goal was to develop a software package, coil, for compressing and decompressing DNA sequence databases based on edit-tree coding. The primary intention is to reduce the bandwidth required to transmit large amounts of DNA sequence data from a central repository to many recipients, and also to reduce disk space requirements for archival storage of such data. While it is desirable to enable efficient searching of a compressed database, and progress has been made in this area [23-25], we have not attempted to do so here. Instead, coil is geared towards maximising compression ratios. This is achieved at the expense of execution time and memory usage – but note that the compression time represents a "one-off investment" whose cost is quickly amortised if the resulting compressed file is transmitted many times. Decompression requires little memory and takes O(D) time for data sets of size D nucleotides.

coil primarily targets sequence databases containing many short sequences of roughly equal length, such as

Expressed Sequence Tag (EST) databases. Many such databases exist: at the time of printing, GenBank contains 23 Gb of EST sequence data in FASTA format, comprising 43,380,458 sequences in total [1]. Targeting these databases simplifies design and implementation: many operations on pairs of sequences take time quadratic in the length of their operands, so shorter sequences are important for high performance; also, assuming reasonably short sequences means that little attention need be paid to the intra-sequence positions of regions of similarity between two sequences.

coil reads and write FASTA format databases [26]. This simple format is easy to work with manually, easy to program input and output routines for, and widely used. Additionally, the format contains a minimum amount of additional context information for each sequence, which allows us to focus on compressing the sequence data.

Implementation

coil consists of a small group of C programs that perform the compression and decompression steps described below, as well as a Perl script which simplifies the compression process by automating sequences of steps and providing sensible defaults where helpful. Use of these programs is described later.

Overview

Hereafter, unless otherwise qualified, D denotes the number of nucleotides (characters) in a database, N the number of sequences, and $L = D/N$ the average length of a sequence. As in the C language, the notation $a \% b$ is used to indicate taking the remainder of a modulo b, for some non-negative integer a and positive integer b.

Conceptually, the process of compressing a database with coil proceeds through the following stages:

1. Creating a *similarity graph* that pairs sequences of high similarity. The similarity graph is an edge-weighted undirected graph in which vertices represent sequences and edges exist between highly similar sequences, with edge weights indicating similarity strength.

2. Extracting an *encoding graph* from the similarity graph. The encoding graph is a set of rooted directed trees (formally, arborescences) whose arcs correspond to a subset of the edges in the similarity graph.

3. Encoding each tree in the encoding graph. For each tree, the root sequence is stored verbatim (*raw-encoded*); an in-order traversal is then used to *delta-encode* each other sequence in the tree in terms of its parent sequence.

4. A multi-platform general-purpose compression program, such as gzip or bzip2 [27], is applied to extract further compression gains.

A typical usage pattern in a Unix-like environment would be to use the tar archive program to collect the files of step 3 together, and pipe the resulting file through bzip2 -9 (the -9 command-line option requests maximum compression).

Decompression of a coil archive amounts to inverting the delta-encoding of the final compression stage: for each encoded tree, the root sequence is written out, following which an in-order traversal recovers every other sequence using the (already recovered) parent sequence and the encoded delta information. This takes place after the general-purpose compression step is undone. Note that in general, the *order* of sequences in the recovered FASTA file will be different than in the original FASTA file; if this is undesirable, program options can be set to restore the original order (i.e. an exact copy will be produced).

All of these steps are explained in more detail below.

Edit distances and similarity graphs

A common way to quantify the similarity between two strings a and b is to compute the *Levenshtein distance*: the smallest number of single-character insertions, deletions and substitutions required to transform a into b. Ideally, we would compute exact distances between every pair of sequences in the database and output a complete graph with perfect similarity information. But since computation of the Levenshtein distance between two strings of length m and n takes $O(mn)$ time [28] in the general case, a database of size D containing N roughly equal-length sequences would require $O(N^2)O(D^2/N^2) = O(D^2)$ comparisons. When database sizes are in the gigabyte range, quadratic-time algorithms are not viable.

Instead, coil uses a more efficient related similarity measure derived by counting the number of length-k substrings, or k-tuples, two strings have in common. For small k, calculation of k-tuple similarity scores can be made very fast by using a k-tuple index data structure (described below) to obtain a list of all sequences in the database that contain a given k-tuple in constant time.

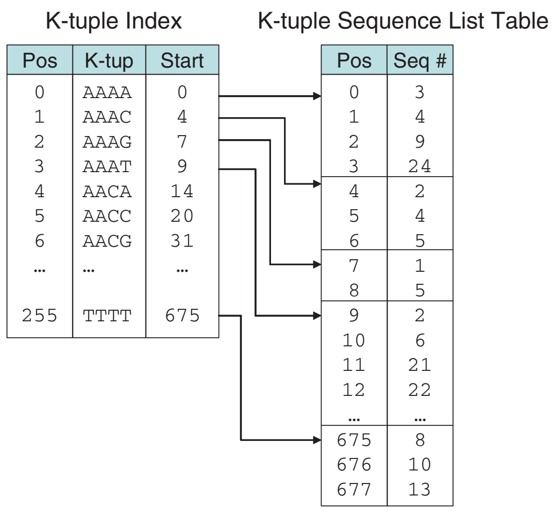
The **k***-tuple index*

A nucleotide (A, C, G or T) can be encoded as a 2-bit integer, and consequently a k-tuple of nucleotides has a natural representation as an integer of $2k$ bits. In coil, the leftmost nucleotide occupies the most significant bits. The k-tuple index data structure, which is prepared in a preprocessing step using the program make_index, consists of two files: a k*-tuple sequence list file* ending with the extension .ktl, which contains $(D - N(k - 1))/s$ integer sequence numbers (*seqnums*); and a k-*tuple index file* ending with the extension .kti, which contains 4^k integer offsets into the first file. s is a "slide" parameter used to reduce the size of the *k*-tuple sequence list file, at the cost of reduced accuracy: only *k*-tuples beginning at sequence positions divisible by s are entered into the index. As Figure 2 shows, the ith entry in the *k*-tuple index file points to the beginning of the list of seqnums that contain *k*-tuple i, which continues until the seqnum list for the $(i + 1)$th entry begins.

These files are built in $O(D + 4^k)$ time using a bucket sort algorithm that performs two passes over the raw sequence data. The algorithm is similar to that used to build the *k*-tuple indices used by SSAHA [29]. Note that unlike in SSAHA, we do not record the intra-sequence positions of *k*-tuples in the *k*-tuple index, nor do we ever record a given seqnum more than once in a given *k*-tuple's seqnum list; instead we rely on our assumption that the database contains short sequences to ensure that there is a low probability of a sequence containing more than one instance of a particular *k*-tuple. Should this not be the case, the efficacy of the algorithm will be reduced, however correctness will not be compromised.

The bucket sort algorithm requires both files to be able to fit in memory simultaneously. If this is not possible, make_index produces multiple pairs of output files: each pair is an index on a segment of the database that will just fit in the amount of memory specified.

It is worth mentioning that empirically, the sizes of seqnum lists in a *k*-tuple index built from DNA sequence data are highly nonuniform, with some *k*-tuples appearing several orders of magnitude more frequently than others. These *k*-tuples cause many spurious hits that slow down the similarity graph construction step. coil follows the smart practice described in [29] of completely eliminating *k*-tuple seqnum lists that exceed a user-specified size: this has the double effect of reducing index file sizes and dramatically improving the selectivity, and hence the speed, of the next stage.



Pos	K-tup	Start
0	AAAA	0
1	AAAC	4
2	AAAG	7
3	AAAT	9
4	AACA	14
5	AACC	20
6	AACG	31
…	…	…
255	TTTT	675

Pos	Seq #
0	3
1	4
2	9
3	24
4	2
5	4
6	5
7	1
8	5
9	2
10	6
11	21
12	22
…	…
675	8
676	10
677	13

Figure 2
Example *k*-tuple index structure for *k* = 4.

Creating the similarity graph
Constructing the similarity graph is the main compression bottleneck in coil. The main contribution made by coil is in engineering an algorithm to efficiently compute pairwise approximate sequence similarity scores using a combination of the raw sequence data and the *k*-tuple index, which is implemented in the find_edges program. We first introduce a "naïve" comparison algorithm, and several variants which each proved unsatisfactory.

The naïve algorithm is parameterised by k, s and b. b is a small integer which is used to limit the total number of edges in the similarity graph to bN; it is necessary to avoid storing $O(N^2)$ edges in the similarity graph. In most testing, b was set to 10. The pseudocode for the algorithm follows:

```
• For each query sequence $q$ in the database:
    Create an empty linked list of (seqnum, hit count) pairs, $M$.
    For each k-tuple $t$ in $q$:
        ▪ Look up the list of sequences that contain $t$ starting at a position divisible by $s$ using the k-tuple index.
        ▪ Merge this list into $M$.
        ▪ Keep track of the number of times each sequence in $M$ has had a k-tuple in common with $q$.
    For each pair $(i, c)$ of the $b$ pairs having the highest hit counts in $M$:
        ▪ Create the edge $(q, i)$ in the similarity graph and assign it weight $c$.
```

The seqnum lists read from the *k*-tuple index are in seqnum order, and M is maintained in this order also. The merge step is the usual list merge, except that whenever pairs having the same seqnum are to be merged, the result is a single pair whose hit count is equal to the sum of the hit counts of the arguments to the comparison. The intuition here is that if two sequences share many *k*-tuples, they are likely to be similar. In fact, it is relatively straightforward to show that if a string of length *n* is at edit distance *d* from another string, then the two strings must share at least $n - (d + 1)k + 1$ *k*-tuples [30]; so we can reasonably expect a correlation in the reverse direction. The output of the algorithm is a representation of the similarity graph in edge-list format.

Unfortunately, the above algorithm suffers from severe performance degradation due to random *k*-tuple matches clogging *M* and slowing down list merges. *M* soon fills with many pairs containing low hit counts, representing sequences that are not significantly similar to *q* but share one or two *k*-tuples with *q* by chance. In fact it can be shown that under reasonable assumptions about the distribution of *k*-tuples in the database, the repeated list merging introduces an $O(D^3)$ factor into the running time.

SSAHA [29] overcomes the clogging problem by choosing *k* to be high enough that very few chance matches occur; however this is only a viable approach if enough memory is available as memory requirements are exponential in *k*. The impressive search speeds described in [29] were obtained on a computer with 16 Gb of RAM and with *k* set to 14 or 15. Requiring this amount of memory for coil would immediately put the program out of range of almost all computers in use today.

Another way to ameliorate the situation is to convert the list *M* into a form of *hashtable* by maintaining *r* separate pair lists $M_0 \dots M_{r-1}$, and merging the seqnum list for the *i*th *k*-tuple in *q* into the list $M_{i\%r}$. After all *k*-tuples have been scanned, a final merge step combines the *r* lists. Even better, partition by seqnum rather than *k*-tuple position: send each seqnum *i* to the list $M_{i\%r}$. The latter technique is more resilient to variations in seqnum list sizes. Choosing $r = 2^h$ for some positive integer *h* enables fast calculation of the remainder through bitwise operations. While these modifications do improve the running time of the naïve algorithm, as Table 1 shows, there remains much work to be done before this algorithm will be feasible for gigabyte-sized databases. We describe below a way to eliminate the time spent processing unpromising hits

Table 1: Execution time for find_edges variations on a small dataset

Algorithm	Parameters	Execution Time (s)
SSAHA	maxGap = 0, maxInsert = 0	118.58
	maxGap = 20, maxInsert = 20	118.72
Basic		97.98
Batch merging	*c* = 16	113.03
	c = 32	84.35
	c = 64	71.09
Recursive merging		80.55
Hashtable	*h* = 12	56.02
	h = 13	56.24
	h = 14	57.92

The dataset used, month.est_mouse, is a monthly update of the Genbank Mouse EST dataset comprising 31,401 sequences having average length 438 nucleotides.

Letting go of perfection: the leaky move-to-front hashtable

The algorithms described in the preceding subsection all compute the complete list of (seqnum, hit count) pairs for a given query sequence *q*, including the "noise" matches with small hit counts, even though we end up keeping only the best *b* such matches. To avoid getting bogged down with noise matches, we modify the seqnum-hashing hashtable from the previous subsection by replacing each of the *r* variable-length linked lists in the hashtable with a small fixed-size array of size *f*. Instead of maintaining these arrays in seqnum order, a move-to-front discipline is used: whenever a seqnum *i* arrives, we scan the array $M_{i\%r}$ for an occurrence. If it is found, it is moved to the front of the array, its hit count is incremented and all preceding elements are shunted down one position. If it is not found, it is inserted at the front of the array with hit count 1; all existing elements are shunted down one position to make room, with the last pair being deleted ("pushed off the end").

Intuitively, the success of this algorithm hinges on the following key assumption:

If a database sequence is genuinely similar to the query sequence, its seqnum will turn up often enough that it will never be pushed off the end of the list.

There are several reasons for the improved performance of this algorithm:

Frequently occurring seqnums are found more quickly and require fewer updates

A frequently occurring seqnum *x* is more likely to have been recently referenced and hence is more likely near the front of its array. Thus when *x* next occurs, the scan will not need to proceed very far down the array. Also note that only those elements that precede *x* in the array need to be shunted back – later elements remain in their original positions.

Packing the seqnum and count values

A "side effect" of the hashtable structure is that there is no need to explicitly record the low h bits of each seqnum in each array, since every seqnum in array M_i must have these bits equal to i. Thus these bits are available for other uses. Since each array entry consists of a (seqnum, count) pair, and since count rarely exceeds L/s, it makes sense to store the count value in these bits. For our default choice of k and s parameters, this is reasonable for $h \geq 10$. (If it is important to deal properly with the rare occasions that that more 1023 k-tuples match between two sequences, we can simply saturate the count at $2^h - 1$.)

Fixed-size multidimensional arrays require very low memory overhead

The leaky move-to-front hashtable data structure is a $2^h \times f$ array. Because of their highly regular structure, accessing data items in fixed-size multidimensional arrays requires only multiplications and additions using the fixed dimension sizes, and does not require any pointers or special list termination symbols to be explicitly stored in memory, reducing the memory allocation overhead.

Computers like fixed-size arrays

Linked lists are efficient in theory, but in practice, computer hardware has long been designed for efficient processing of contiguous arrays of elements, and the "pointer-chasing" inherent in working with linked lists inevitably introduces comparatively large overheads. In particular, items at consecutive positions in a linked list may occupy widely separated memory addresses – a problem known as *poor spatial locality of reference*. In these circumstances, when iterating through the items in a linked list, memory cache hardware cannot predict which bytes will be read or written next, and performance suffers. In contrast, a scan through an array accesses memory bytes in sequential order, and will benefit from cache line fills that read contiguous blocks of memory into cache.

Assuming pessimistically that every seqnum must endure a full f comparison and move operations, analysis gives a time complexity of $O(fD^2/s4^k)$ for this algorithm.

Pentium 4 optimised version

Many modern CPUs use *pipelining* to increase instruction throughput. We have developed an implementation of the find_edges algorithm optimised for the heavily pipelined Pentium 4 processor [see Additional file 1].

Extracting the encoding graph

Once a similarity graph has been created with find_edges, the next step is to extract from it an encoding graph that distinguishes groups of similar sequences and describes how they are to be encoded. Then each group is compressed independently. Both steps are performed by the program encode. For the time being, we assume the availability of a subroutine for delta-encoding one sequence in terms of another that produces a "black box" block of data bytes; this algorithm is described in the subsection "Delta-encoding Sequences".

First we note some structural properties of the encoding graph. Each sequence in the database will be either raw-encoded, or delta-encoded in terms of one other sequence: this implies that each vertex in the encoding graph will have at most one in-edge. Then by prohibiting cycles it is easy to show that the encoding graph will be a forest of directed trees, each having edges directed away from a root vertex. Since the decision about which vertex to choose as the root has little bearing on the speed or compression level achieved for a tree, coil selects the lowest-numbered sequence.

To be effective in compressing sequence databases, coil needs to produce an encoding graph in which highly similar sequences are linked by an edge whenever possible. More precisely, we want to maximise the total similarity score of the encoding graph, subject to the constraints that it be a subgraph of the similarity graph, and also a forest. This is the *maximum spanning forest* problem, which is equivalent to the *minimum spanning forest* problem using negative edge weights, which in turn is a generalisation of the heavily studied *minimum spanning tree* (MST) problem. Happily, several algorithms exist for efficiently solving these problems [22,31,32].

Since the similarity graph produced by find_edges is very sparse (containing at most bN edges) and is already in edge-list format, we employ Kruskal's $O(|E|\log|E|)$ algorithm [31]. Kruskal's algorithm is very simple to state:

1. Read in the similarity graph edge list.

2. Sort edges by similarity score.

3. For each edge in the sorted list:

    • If this edge would not introduce a cycle, add it to the encoding graph forest.

Importantly, both sorting and cycle-testing can be performed efficiently. Edge sorting is accomplished in $O(|E| + \max(score))$ time using bucket sort. (Note that the score of an edge between sequences of lengths x and y is at most $\min(x, y)/s$). Cycle-testing is performed using the fast union/find data structure described in [33]. This data structure manages an equivalence relation on a set: here, the classes are the connected components of the encoding graph, which combine as edges are added. Determining whether an edge would induce a cycle amounts to testing whether its two endpoints are in the same class. m such tests can be performed on a set of size n in $O(m\alpha\ (n, m))$ time [34], where $\alpha(n, m)$ is the extremely slow-growing inverse Ackermann function: effectively constant time per test.

Delta-encoding sequences

Once the encoding graph has been created, encoding of the individual trees can begin. For each tree, the root sequence is output verbatim, and an in-order tree traversal then delta-encodes every other sequence in terms of its parent. It is well-known that such a traversal can be used to encode a rooted tree as a string containing only vertex identifiers and parentheses. Conversion in both directions can be accomplished without requiring random access to the characters of the string, implying that an in-memory tree data structure can be efficiently "streamed" to or from a sequential storage medium (such as a disk file) in time linear in the number of nodes.

An *edit script* is a list of edit operations, which we here take to be single-character insertions, deletions and substitutions. Our task is to find a minimal-length edit script for converting one string a of length n into another string b of length m. This problem can be solved in $O(nm)$ time using a straightforward dynamic programming approach, in which we successively compute optimal edit scripts for pairs of prefixes of a and b in terms of previously computed solutions. Several algorithms exist that are asymptotically faster for certain input distributions [28,35]. In particular, an algorithm of Myers [35] can solve a variant of this problem in which only insertions and deletions are allowed in $O(nd)$ time and space, where d is the edit distance (and thus the size of the edit script). Since the encoding stage of coil deals only with sequences already deemed to be similar by heuristics, this algorithm was chosen for implementation. Another attractive feature of the Myers algorithm is that it considers possible edit scripts in increasing order of edit distance, and can be terminated when the edit distance reaches some predetermined maximum distance $d_{\max}$. In coil's encoding stage, this is used to bound the runtime of the algorithm: if the edit distance between a pair of sequences exceeds a user-specified figure (defaulting to 150), the algorithm terminates early and a trivial edit script having length $a + b$ is produced.

Once an edit script has been found, it must be compactly encoded into data bytes. coil uses a simple scheme in which the most significant bit (MSB) of a byte specifies whether an insertion or deletion is to take place, and the remaining seven bits specify the offset (with respect to the source string) from the previous edit operation. If an edit operation is more than 126 characters along from the previous edit operation, a special code byte, having its lowest seven bits equal to 127, is emitted, indicating that the next byte should be read and 126 should be added to that byte's value to form the offset. This code byte may occur multiple times, adding 126 to the total offset each time. Since deletion operations identify character positions within the source string while insertion operations identify positions *between* characters, special care must be taken to handle string positions and offsets in a manner that permits unambiguous decoding.

In the case of an insertion operation, the character to be inserted is not recorded in-place but written to a separate file. This breaks the "edit script as black box" design principle, however separating the edit script and nucleotide data streams in this way makes the distributions of bytes in each stream more predictable, resulting in compression gains that cannot be overlooked.

Although the encoding described is fairly compact, it is clearly not optimal: for example, we expect position offsets to be tightly clustered around zero, implying that an encoding in which lower offset values were represented with fewer bits would yield higher space savings. However, this and any other detectable redundancy will be eliminated when the coil archive files are passed through an external general-purpose compression program.

Sequence buffering

We have developed a simple buffering system that enables maximally efficient random access to sequence data [see Additional file 2].

Incremental compression

Large sequence databases such as GenBank [36] are not static. They are being updated daily, and there is a need for database users to access the latest versions. The solution found by most organisations distributing these databases is to make available daily or weekly updates in the form of deltas – lists of sequences added, changed or removed from the original database release. End users who already have the main database installed can download the updates and apply these changes to their local database copies to produce up-to-date versions. These database deltas are much smaller files, often less than 100 Mb in size, and coil's usual mode of compression performs poorly on such small files.

A common update performed on a database is the addition of one or more new sequences. coil therefore supports *incremental compression*: the ability to efficiently encode one sequence database, the *increment*, in terms of another *baseline* database. We presume a user who downloads a database delta already has the original baseline database, so we can "refer back" to baseline sequences from within the encoding graph of the compressed increment. This approach makes available a large pool of candidate root sequences that can be used for efficient delta-encoding of each sequence in an increment.

In the remainder of this section we will refer to the mode of compression discussed in previous sections as *standalone* compression. For concreteness we will talk about compressing an increment database named incr.fasta in terms of a baseline database named base.fasta. It may be helpful to first read the subsection "Using coil", which describes the overall workflow and the individual files read and written by the various programs in the coil package.

Implementation of incremental compression

Suppose that there are *B* sequences in the baseline database base.fasta, and *I* sequences in the increment incr.fasta. Compression of the baseline entails ordinary standalone compression; the only difference is that the *k*-tuple index files, and also the base.coil.seqnos file, need to be retained for compression (though not for decompression) of the increment. To compress the increment, *k*-tuple index files are produced from the increment database itself. However, we require the seqnums of the increment to be distinct from those of the baseline, so the -i *B* command-line option must be used with the make_index and find_edges programs to offset the starting seqnum by *B*.

The encode program is then run with the command-line option -i base, to indicate that the input file should be encoded in terms of the coil archive base. When this option is chosen, the Kruskal maximum spanning forest algorithm is modified to avoid adding an edge between two components which both contain baseline sequences. This is easy to accomplish, since all baseline sequences have seqnums less than *B*, and as components are identified by their lowest-numbered seqnum any component that contains a baseline sequence will be represented by the seqnum of that sequence, effectively limiting the involvement of baseline sequences to being the roots of components in the encoding graph.

Once encode has produced an encoding graph, it needs access to the sequence data so that delta-encoding can be performed. Obtaining the sequence data for an increment seqnum can be accomplished in the usual way (via the sequence buffering system), but this is not the case for baseline sequences.

The necessary baseline sequences are obtained from the baseline coil archive by decompressing the entire baseline database in-memory, but writing out only those baseline sequences which are the roots of trees in the increment encoding graph (which we call *buds*). During this step, baseline sequences will be visited in *decode order*; by first sorting the list of required baseline seqnums into this order, extraction can take the form of a list merge. The sorting step involves inverting the permutation of baseline seqnums recorded in the base.coil.seqnos file (a linear-time and -space operation), hence the requirement that this file be retained after compression of the baseline database. Then the encode program traverses and encodes all trees rooted in baseline sequences in decode order. For each such tree, the (strictly increasing) position of the baseline root sequence in the decode order is written to the file incr.bud, which is included in the increment archive to facilitate decompression. Finally, the program traverses and encodes all trees rooted in increment sequences in the usual fashion.

When decompressing an increment with the program decode, the command-line option -i base is used to specify that the coil archive base should be used as the baseline. decode first decompresses the baseline sequences at the positions listed in incr.bud and uses these sequences as roots for decoding the initial segment of the increment archive; then the remaining trees are decompressed as usual.

Using coil

Compressing a FASTA database using coil involves running several C programs that work together to produce a number of output files. Some of these files, collectively termed the coil *archive*, are required for recovering the original data, while the remainder may be discarded once compression is complete. Alternatively, the user may run a Perl script which automates these steps. The final step requires the files comprising the coil archive to be compressed by a general-purpose compression program. Figure 3 shows the complete process of using coil for standalone compression of a database; incremental compression is similar, but requires that all output files produced during baseline compression (including the baseline coil archive itself) are also available. Incremental compression produces one additional, small output file ending with the extension .bud which must be included in the final archive.

Decompression

Decompressing a coil archive is simple: first "undo" the general-purpose compression used to compress the archive, then run the program decode. Decompression of an increment requires the name of the baseline database be specified on the command line with the -i switch. The process takes $O(D)$ time and requires $O(\max(seqLen) * \max(treeDepth))$ memory. $\max(treeDepth)$ is typically small, but could be bounded using a simple adjustment to the Kruskal algorithm if necessary.

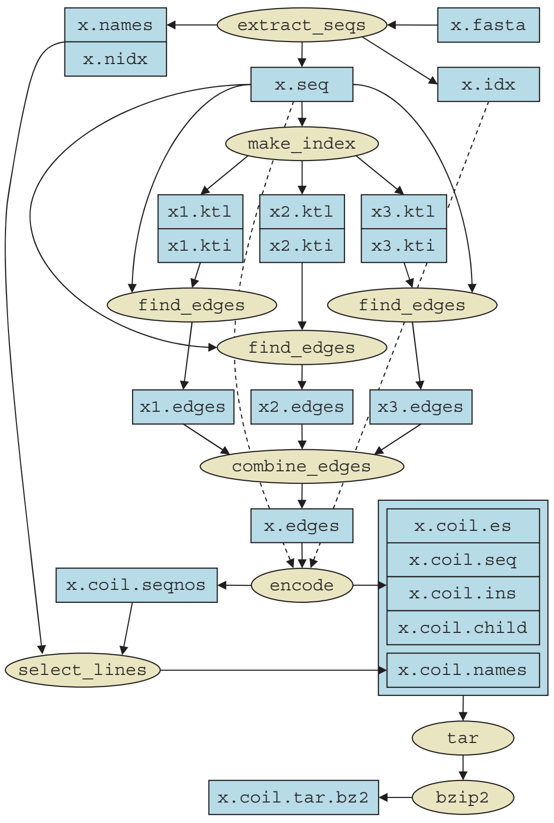


Figure 3
Using coil to compress a FASTA database. As few two-file *k*-tuple index segments are produced as memory allows.

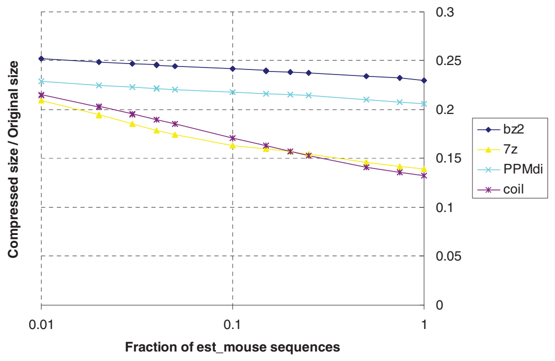


Figure 4
Compression ratio vs. DB size. The compression ratios of all tested algorithms increase as the input size increases; those of coil and 7-Zip increase faster than the rest.

If the x.coil.seqnos file has been stored with the coil archive, then it is possible to recover the original sequence order at decompression time using the -x switch to the decode program; otherwise, sequences in the FASTA output file will appear in a different order.

For maximum portability across platforms, all files containing binary integer data use the little-endian storage format. All reading and writing of such files occurs through platform-specific load_vector() and save_vector() functions.

All programs accept the -h and --help switches, which can be used to display usage information. A brief description of each program can also be found in the README.txt file included in the coil software package.

Results and discussion

To investigate the compression ratio achieved and running time required by coil for datasets of various sizes, tests were performed on randomly chosen subsets of sequences from a version of the GenBank est_mouse database, which contains 1,729,518,522 nucleotides in 3,852,398 sequences. Twelve dataset sizes were examined, with three test datasets produced for each size. Each dataset having a name of the form ems*n* contains 3,852,398 × *n*/100 sequences randomly selected from the est_mouse database. For the 100% size level, a single dataset (the original est_mouse database) was run three times, giving an indication of the noise level involved in execution time measurements.

A number of alternative compression programs were tested in addition to coil:

1. **bz2**: The general-purpose compressor bzip2 [27] with compression level 9.

2. **nrdb+bz2**: Elimination of duplicate sequences with the nrdb program, followed by bzip2 with compression level 9.

3. **PPMdi**: The PPMd general-purpose compressor variant I described in [37], with model order 8 and RAM usage 256 Mb (the most allowed by the program).

4. **7z**: the LZMA compression mode of the freely available general-purpose compression program 7-Zip [38]. This was the only other program we found that was capable of utilising 1 Gb of RAM during compression.

We also attempted to compress our datasets with the program DNACompress [9], however we found that we were unable to compress datasets larger than 14.5 million bases using this program. Unfortunately this is smaller than the smallest dataset we used in testing, and we were forced to abandon this attempt.

The tests on the ems*n* datasets specified restrictions for 7-Zip and coil to use at most 1 Gb of RAM. It should be noted that the other programs all use substantially less memory than this; in particular, PPMdi is limited to 256 Mb. To enable a fairer comparison with the PPMdi program, a further series of tests was carried out on the 140 Mb FASTA dataset rfam_full using 256 Mb settings for each program, and the -x option to coil.pl to enforce in-order sequence recovery. Due to problems attempting to compile nrdb on Windows, the nrdb+bz2 measurements were performed on a different computer: a Linux 3.2 GHz Xeon machine with 4 Gb of RAM. To measure decompression speed for nrdb+bz2, a simple C program, unnrdb, was written to expand the multi-header FASTA files produced by nrdb.

All coil runs used the parameter values k = 12, s = 8, f = 4 and h = 10, and were run on a 2 GHz Intel Core 2 Duo computer with 2 Gb RAM running Microsoft Windows XP. Most parameter values were chosen by earlier experimentation, however the choice of f and h received extra attention. Since it is important for the speed of the find_edges program that its hashtable data structure fit in cache memory, and it is not obvious how to trade off the f and h parameters for a fixed memory size, preliminary testing was conducted with several values of these parameters, suggesting (f = 4, h = 10) is best for the case where the hashtable is limited to 16 Kb in size – small enough to fit in the first-level cache of any modern computer system.

Table 2 shows the sizes of the resulting compressed files. We immediately see that it is a race between coil and 7-Zip's LZMA compression mode: these two compressors easily outstrip all others, with the gap widening as file sizes increase. This is depicted graphically in figure 4. It appears that the 1 Gb of RAM available to both these compressors makes a big difference when compressing files containing many sparse repeats. There is never more than a 6% difference in file sizes between these two compressors, with 7-Zip performing best on smaller files. coil edges out 7-Zip on the larger files, eventually claiming a 5% improvement on the largest dataset tested, ems100.

Looking at the execution times in Table 3, a similar transition takes place: coil is faster than 7-Zip up until around ems50, at which point the quadratic nature of find_edges starts to dominate. coil compresses the ems25 datasets faster and better than does 7-Zip by a small margin.

Finally, the decompression times shown in Table 4 show that coil is somewhat slower than the other programs, though still essentially linear-time as expected. Not shown in the table is coil's frugal memory usage during decompression – the maximum memory usage while decompressing ems100 is just 4.5 Mb, in comparison to the 89 Mb used by 7-Zip and the 270 Mb used by PPMdi.

With respect to the rfam_full datasets, coil outperformed the nearest competition – again, 7-Zip – by around 3% in terms of compression ratio, though requiring more than twice as much time to do so. PPMdi performed poorly, producing a file more than twice the size of that produced by coil or 7-Zip. This is especially surprising given that these other programs were operating with the same 256 Mb RAM constraints as PPMdi for this dataset. bzip2 does substantially better with only 8 Mb of RAM at its disposal.

Surprisingly, although the optimised Pentium 4 version of find_edges produced a speed improvement of 25% on the Pentium 4 computer on which we performed initial testing, using this version of the program actually decreased performance by 6% on the Core 2 Duo platform. Only one test was run using this version of the program, indicated by a row with an asterisk in Tables 2, 3 and 4; all other results shown use the regular version of find_edges.

Conclusion

We have demonstrated that the concept of edit-tree coding can be applied to produce a practical compression tool for sequence databases. The execution time required is not negligible and appears to grow quadratically with database size, but adequate compression on large EST databases can nevertheless be achieved on "everyday" modern computers. Furthermore, concern over compression time diminishes when it is considered to be amortised over the many decompressions that may take place in the targeted field of *one-source-many-sinks* operations. Decompression is acceptably fast, uses very little memory and can be performed on any computer with a C compiler. Source code portability and binary compatibility of compressed files has been tested on two widely used platforms, Linux and Win32.

There remains a wide scope for experimentation with coil and fine-tuning of algorithms and parameters. For example, one avenue not pursued here is the extent to which filtering of common k-tuples affects execution time and matching accuracy. It may be that the most commonly occurring 80% of k-tuples can be removed without dramatically affecting overall compression. While this kind of search space pruning would never be acceptable in a program like SSAHA that is specifically designed to find matches between sequences, we only care about accuracy

Table 2: Compressed file sizes

Dataset	FASTA	bz2	nrdb+bz2	7z	PPMdi	coil
ems1	23292780	5876747	5871445	**4870989**	5331953	4990193
	23199910	5853780	5852865	**4854519**	5311350	4981279
	23201245	5852837	5852772	**4857631**	5312411	4988747
ems2	46519702	11576074	11574420	**9057588**	10475531	9432789
	46428669	11557030	11556573	**9023980**	10454826	9410376
	46390115	11547516	11549117	**9036246**	10445740	9426594
ems3	69631679	17211495	17205793	**12922145**	15537092	13607729
	69647486	17212318	17208461	**12907737**	15543489	13592739
	69715954	17231912	17225610	**12920294**	15558845	13623246
ems4	92905691	22841127	22810035	**16600091**	20601712	17625302
	93012024	22868732	22849091	**16611294**	20629724	17655369
	92850447	22813494	22799324	**16587471**	20585812	17584008
ems5	116125238	28428297	28415051	**20245345**	25636473	21509065
	116249077	28451622	28426520	**20260429**	25663621	21547174
	116117128	28413464	28397742	**20239745**	25630456	21496207
ems10	232365230	56136032	56054164	**37932764**	50662993	39774087
	232226017	56101887	56085818	**37910566**	50643774	39711435
	232230440	56099503	56030860	**37871106**	50622855	39685294
ems15	348404276	83539894	83461996	**55591757**	75411889	56758484
	348435883	83529794	83463158	**55594352**	75435650	56771053
	348292392	83453434	83396104	**55580937**	75374710	56768193
ems20	464825178	110838776	110755872	72989089	100113255	**72984372**
	464778933	110777795	110650470	**72991749**	100083039	73004561
	464532828	110766213	110653180	**72918789**	100046482	72978434
ems25	581105516	137940393	137814551	89636246	124600275	**88816000**
	580758935	137898843	137748733	89647136	124521398	**88829572**
	580693026	137884675	137756070	89594767	124526386	**88745435**
ems50	1161787240	272394718	271857439	169833915	244302747	**164139098**
	1161908810	272481687	271896055	169824808	244355206	**164069331**
	1161582289	272310746	271844248	169812165	244255108	**164093038**
ems75	1742471477	405262890	404293340	247835911	362403056	**236517596**
	1742664959	405243466	404268271	247921410	362419128	**236506851**
	1742458336	405281768	404397179	247684455	362394809	**236572552**
ems100	2323234744	533757352		324292321	478735224	**308211386**
	2323234744	533757352		324292321	478735224	**308211685**
	2323234744	533757352		324292321	478735224	**308211677**
ems100*	2323234744					308212275
rfam_full	140518668	4413613		4113889	9504648	**3995880**
	140518668	4413613		4113889	9504648	**3996447**
	140518668	4413613		4113889	9504648	**3995925**

All sizes are in bytes. The FASTA column shows the size of the original uncompressed FASTA file. The smallest file in each row is shown in bold. * This row shows the result of using version of find_edges optimised for the Pentium 4. nrdb+bz2 failed to compress the ems100 dataset because the size of the FASTA file exceeded 2 Gb. All coil runs performed on the rfam_full dataset used the -x option to enable in-order recovery of sequences. nrdb+bz2 was not used with the rfam_full dataset because it is incapable of restoring this order.

of sequence matches where it noticeably improves compression.

Availability and requirements

Project name: coil

Project home page: http://awcmee.massey.ac.nz/Data/wtwhite/coil-1.1.2

Operating system(s): Tested on Windows and Linux. Binaries are additionally provided for Windows. Source code should compile in any UNIX-like environment.

Programming languages: ANSI C, Perl

Other requirements: Perl 5.6 or higher

Licence: BSD-style license

Table 3: Compression execution time

					coil				
Dataset	**bz2**	**nrdb+ bz2**	**7z**	**PPMdi**	find_edges	encode	tar+bz	other	**total**
ems1	5.6	7.7	43.3	5.6	3.5	9.3	1.2	10.8	24.8
	5.5	9.2	48.2	4.7	3.6	8.1	1.0	10.4	23.1
	5.7	10.2	43.5	4.3	3.6	8.2	1.2	12.5	25.5
ems2	10.3	15.1	96.5	9.9	9.5	20.0	1.0	19.1	49.5
	10.3	17.7	101.4	8.5	9.7	20.2	1.0	19.0	49.8
	10.1	16.8	95.6	8.3	9.7	20.3	1.1	20.8	51.9
ems3	15.2	22.6	154.8	14.7	17.0	36.5	1.9	27.7	83.1
	16.7	24.6	162.9	12.7	17.3	35.4	2.2	28.0	82.9
	15.4	22.4	154.2	12.8	17.3	34.5	3.1	29.2	84.2
ems4	20.3	32.0	216.5	20.0	25.8	50.6	3.6	34.3	114.3
	20.5	33.4	221.1	17.4	26.7	50.3	3.0	37.1	117.1
	20.0	31.1	215.0	17.0	26.0	49.1	3.3	39.3	117.7
ems5	30.3	39.1	276.7	25.3	35.5	65.4	4.2	42.7	147.7
	25.5	43.5	280.4	21.5	35.7	65.5	4.1	45.2	150.5
	25.3	38.7	275.9	21.4	35.8	64.6	4.1	46.2	150.7
ems10	62.5	85.1	573.8	49.8	100.5	179.3	8.4	100.4	388.6
	60.7	88.0	580.8	45.5	102.1	176.0	9.0	84.4	371.4
	50.6	80.1	575.4	43.5	100.5	160.8	9.4	87.7	358.4
ems15	94.3	117.8	871.1	69.0	197.9	271.8	12.6	118.2	600.6
	76.7	136.5	876.5	64.7	198.5	276.9	13.7	130.5	619.5
	89.8	119.6	869.5	64.5	196.1	275.8	13.8	133.4	619.1
ems20	101.5	169.9	1163.0	92.8	317.1	393.5	16.7	176.0	903.5
	101.7	179.7	1169.3	86.9	321.6	393.5	18.5	212.2	945.8
	120.5	158.8	1161.3	84.9	319.6	399.7	16.9	215.6	951.8
ems25	133.0	207.7	1482.2	116.0	471.7	503.2	22.5	280.8	1278.1
	152.3	220.7	1438.9	105.9	470.9	467.2	22.3	218.4	1178.8
	171.0	196.3	1456.4	106.0	468.0	504.4	23.4	248.0	1243.8
ems50	306.2	411.2	2882.0	215.4	1657.4	1172.3	105.4	716.3	3651.4
	340.0	452.4	2893.3	209.2	1658.2	1170.7	104.5	583.7	3517.1
	291.1	411.6	2888.3	207.2	1655.5	1174.7	107.9	671.8	3609.9
ems75	500.7	712.4	4328.8	314.4	3517.1	1814.9	167.8	1173.4	6673.2
	506.8	618.1	4304.5	311.9	3502.1	1810.2	164.7	992.9	6469.8
	508.7	593.5	4298.9	317.7	3490.7	1798.6	165.0	1116.4	6570.6
ems100	668.6		5760.8	408.5	6064.4	2552.4	223.9	1421.8	10262.6
	634.1		5707.5	404.1	6042.3	2524.8	219.3	1429.0	10215.3
	689.2		5773.6	403.3	6114.1	2496.4	217.5	1546.2	10374.1
ems100*					6446.3	2515.4	218.6	1505.8	10686.1
rfam_full	32.8		75.8	7.9	114.8	12.8	4.0	40.7	172.3
	29.6		75.3	7.9	113.9	12.3	4.3	38.2	168.7
	29.6		75.5	7.8	114.5	12.4	4.2	36.0	167.1

All durations are in seconds. The rightmost five columns break down the execution of coil by its main component programs; the "other" column includes the time needed for the programs extract_seqs, make_index and select_lines.
*This row shows the result of using the Pentium 4-optimised version of find_edges – surprisingly, this version of find_edges is actually about 6% slower than the original version on this CPU.

Abbreviations

EST: Expressed Sequence Tag. In order to produce a protein, a cell first copies the segment of DNA encoding that protein (the *gene*) to a complementary *messenger RNA* molecule. ESTs are DNA sequences obtained by extracting and sequencing the messenger RNA molecules from a cell. Because only a single sequence read is performed on each molecule, ESTs are limited to approximately 800 bases in length.

FASTA: A simple text file format for storing multiple DNA or protein sequences. Each sequence begins with a single line starting with the character ">" and containing the sequence name, followed by any number of lines containing the sequence data.

MST: Minimum Spanning Tree. Given a connected, edge-weighted graph, a minimum spanning tree of the graph is a subgraph that (a) contains all vertices of the graph, (b)

Table 4: Decompression execution time

Dataset	bz2	nrdb+bz2	7z	PPMdi	coil
ems1	1.8	3.0	1.5	4.6	3.6
	1.8	3.1	1.5	3.9	4.5
	1.8	3.1	1.5	4.3	4.5
ems2	3.5	6.2	3.2	8.7	6.5
	3.5	6.1	3.2	7.9	6.6
	3.4	6.0	3.1	8.6	7.1
ems3	5.2	9.0	4.8	13.2	10.4
	5.2	9.3	4.9	11.7	11.0
	5.2	9.0	4.9	13.2	10.3
ems4	6.9	12.0	6.5	17.5	14.3
	7.0	12.3	6.4	15.6	13.6
	6.9	12.0	6.5	17.6	13.7
ems5	8.7	15.2	7.8	22.1	16.8
	8.6	15.3	7.3	19.7	17.7
	8.6	14.9	8.1	22.0	18.3
ems10	17.0	29.8	15.1	44.9	36.3
	17.0	30.6	14.8	40.1	36.8
	17.1	29.9	17.6	44.5	36.9
ems15	25.2	44.6	22.1	65.9	52.3
	25.5	46.0	22.0	59.1	53.7
	25.5	44.4	24.4	65.5	52.8
ems20	33.8	59.5	30.3	87.1	68.2
	34.0	60.7	29.8	78.1	70.2
	34.1	59.4	32.0	86.2	69.1
ems25	41.9	74.4	36.7	107.6	91.1
	42.1	76.3	38.9	100.9	85.9
	42.4	74.0	36.2	106.7	86.8
ems50	126.6	147.7	71.7	210.4	286.1
	128.6	152.0	71.9	202.8	274.9
	129.0	148.4	74.5	209.7	280.9
ems75	187.7	223.2	110.3	312.0	511.3
	190.5	228.3	111.3	306.7	471.3
	191.2	221.5	116.3	352.4	464.9
ems100	247.1		142.1	324.2	646.1
	248.6		137.6	404.9	674.2
	252.4		143.5	531.9	700.8
ems100*					649.9
rfam_full	9.6		7.9	9.1	60.7
	6.3		6.4	9.1	59.9
	6.1		6.2	9.1	60.2

All durations are in seconds. *This row shows the result of using the Pentium 4-optimised version of find_edges.

retains connectivity and (c) has minimum total weight among all such subgraphs. It follows that in a graph with positive edge weights, such a subgraph is always a tree.

Authors' contributions

WTJW conceived the concept of edit-tree coding, designed and implemented the coil software, performed performance measurement, and produced an early draft of the manuscript. MDH provided design advice and made significant contributions to the final version of the manuscript.

Additional material

Additional file 1

Appendix 1 – Pentium IV optimised find_edges. Describes the version of the find_edges program optimised for the Pentium IV processor.
Click here for file
[http://www.biomedcentral.com/content/supplementary/1471-2105-9-242-S1.doc]

Additional file 2

Appendix 2 – Sequence Buffering System. Describes the system used for efficiently obtaining random access to sequence data in memory-constrained environment.
Click here for file
[http://www.biomedcentral.com/content/supplementary/1471-2105-9-242-S2.doc]

Acknowledgements

The authors would like to acknowledge the helpful comments of D. Penny in preparing this manuscript.

References

1. NCBI: **NCBI-GenBank Flat File Release 159 Release Notes.** [ftp://ftp.ncbi.nih.gov/genbank/release.notes/gb159.release.notes].
2. NCBI News: **GenBank Passes the 100 Gigabase Mark.** *NCBI News* [http://www.ncbi.nlm.nih.gov/Web/Newsltr/V14N2/100gig.html].
3. Ziv J, Lempel A: **Universal Algorithm for Sequential Data Compression.** *IEEE Transactions on Information Theory* 1977, **23:**337-343.
4. Gailly J, Adler M: **gzip (GNU zip) compression utility.** [http://www.gnu.org/software/gzip/].
5. Matsumoto T, Sadakane K, Imai H: **Biological sequence compression algorithms: December 18-19; Tokyo.** Universal Academy Press; 2000:43-52.
6. Grumbach S, Tahi F: **Compression of DNA sequences: 30 March-2 April; Snowbird, Utah.** Edited by: Storer JA and Cohn M. IEEE Computer Society Press; 1993:340-350.
7. Grumbach S, Tahi F: **A New Challenge for Compression Algorithms - Genetic Sequences.** *Inf Process Manage* 1994, **30:**875-886.
8. Chen X, Kwong S, Li M: **A compression algorithm for DNA sequences.** *IEEE Engineering in Medicine and Biology Magazine* 2001, **20:**61-66.
9. Chen X, Li M, Ma B, Tromp J: **DNACompress: fast and effective DNA sequence compression.** *Bioinformatics* 2002, **18:**1696-1698.
10. Li M, Badger JH, Chen X, Kwong S, Kearney P, Zhang HY: **An information-based sequence distance and its application to whole mitochondrial genome phylogeny.** *Bioinformatics* 2001, **17:**149-154.
11. Kocsor A, Kertesz-Farkas A, Kajan L, Pongor S: **Application of compression-based distance measures to protein sequence classification: a methodological study.** *Bioinformatics* 2006, **22:**407-412.
12. Ma B, Tromp J, Li M: **PatternHunter: faster and more sensitive homology search.** *Bioinformatics* 2002, **18:**440-445.
13. Strelets VB, Lim HA: **Compression of Protein-Sequence Databases.** *Comput Appl Biosci* 1995, **11:**557-561.
14. Wu CH, Yeh LSL, Huang HZ, Arminski L, Castro-Alvear J, Chen YX, Hu ZZ, Kourtesis P, Ledley RS, Suzek BE, Vinayaka CR, Zhang J, Barker WC: **The Protein Information Resource.** *Nucleic Acids Res* 2003, **31:**345-347.
15. Katz P: **PKZIP.** 1.1th edition. 1990 [http://www.pkware.com/]. Milwaukee, WI, USA, PKWARE, Inc.
16. Li WZ, Jaroszewski L, Godzik A: **Clustering of highly homologous sequences to reduce the size of large protein databases.** *Bioinformatics* 2001, **17:**282-283.

17. Li WZ, Jaroszewski L, Godzik A: **Tolerating some redundancy significantly speeds up clustering of large protein databases.** *Bioinformatics* 2002, **18:**77-82.
18. Li WZ, Godzik A: **Cd-hit: a fast program for clustering and comparing large sets of protein or nucleotide sequences.** *Bioinformatics* 2006, **22:**1658-1659.
19. **nrdb** [http://blast.wustl.edu/pub/nrdb/]
20. Thompson JD, Higgins DG, Gibson TJ: **Clustal-W - Improving the Sensitivity of Progressive Multiple Sequence Alignment through Sequence Weighting, Position-Specific Gap Penalties and Weight Matrix Choice.** *Nucleic Acids Res* 1994, **22:**4673-4680.
21. Foulds LR, Graham RL: **The Steiner problem in phylogeny is NP-complete.** *Advances in Applied Mathematics* 1982, **3:**43-49.
22. Chazelle B: **A minimum spanning tree algorithm with Inverse Ackermann type complexity.** *Journal of the ACM* 2000, **47:**1028-1047.
23. Ferragina P, Manzini G: **Indexing compressed text.** *J ACM* 2005, **52:**552-581.
24. Russo LMS, Oliveira AL: **A compressed self-index using a Ziv-Lempel dictionary.** In *String Processing and Information Retrieval, Proceedings Volume 4209*. Berlin, SPRINGER-VERLAG BERLIN; 2006:163-180.
25. Foschini L, Grossi R, Gupta A, Vitter JS: **When indexing equals compression: Experiments with compressing suffix arrays and applications.** *ACM Trans Algorithms* 2006, **2:**611-639.
26. Lipman DJ, Pearson WR: **Rapid and Sensitive Protein Similarity Searches.** *Science* 1985, **227:**1435-1441.
27. Seward J: **bzip2 and libbzip2 - A program and library for data compression.** 1.0.3 edition. 1997.
28. Hunt JW, Szymanski TG: **A Fast Algorithm for Computing Longest Common Subsequences.** *Communications of the ACM* 1977, **20:**350-353.
29. Ning ZM, Cox AJ, Mullikin JC: **SSAHA: A fast search method for large DNA databases.** *Genome Res* 2001, **11:**1725-1729.
30. Burkhardt S, Karkkainen J: **One-gapped q-gram filters for Levenshtein distance.** In *Combinatorial Pattern Matching Volume 2373*. Berlin, SPRINGER-VERLAG BERLIN; 2002:225-234.
31. Kruskal JB Jr.: **On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem.** *Proceedings of the American Mathematical Society* 1956, **7:**48-50.
32. Prim RC: **Shortest Connection Networks and Some Generalizations.** *Bell System Technical Journal* 1957, **36:**1389-1401.
33. Moret B, Shapiro H: **Algorithms from P to NP: Design and Efficiency.** Redwood City, CA, Benjamin/Cummings; 1991.
34. Tarjan RE: **Efficiency of a Good but Not Linear Set Union Algorithm.** *J ACM* 1975, **22:**215-225.
35. Myers EW: **An O(ND) Difference Algorithm and its Variations.** *Algorithmica* 1986, **1:**251-266.
36. **GenBank Sequence Database** [http://www.ncbi.nlm.nih.gov/Genbank/index.html]
37. Shkarin D: **PPM: One Step to Practicality.** 2002:202-211.
38. **7-Zip** [http://www.7-zip.org]

3.2 Pentium IV Optimised `find_edges`

Many modern CPUs use pipelining to increase instruction throughput. In a pipelined CPU, each CPU instruction is fed through an execution pipeline consisting of several stages, so that typically each stage is processing a different instruction at any given time. The Intel Pentium 4 has 20 pipeline stages. Performance is maximised when all pipeline stages are kept occupied with instructions, although this is not always possible: whenever one instruction's behaviour depends on the outcome of an earlier instruction which has not yet completed executing, that instruction must "wait" for the earlier instruction to complete. In particular, a conditional branch instruction may change the memory address that the next instruction should be read from, depending on the outcome of some test (such as whether a particular CPU register is equal to zero) which is not yet known. Despite the existence of sophisticated branch prediction hardware in the Pentium 4, pipeline flushes due to mispredicted branches can dramatically reduce performance, especially when test outcomes are close to random.

Nearly all `if ... else` constructs in compiled languages generate conditional branch instructions, so it follows that eliminating as many such constructs as possible from the inner loops of computationally intensive code will improve the performance of that code on pipelined processors. Since the Pentium 4 is a very popular processor, we felt it was worthwhile to develop an optimised version of the central subroutine in the `find_edges` program for this processor. Our subroutine, which is written in assembly language, performs the step of merging a seqnum into a leaky move-to-front hashtable as described in the paper, with the hardcoded restriction that $f = 4$. The code uses the SSE2 extended instruction set to manipulate 128-bit quantities representing vectors of four 32-bit numbers. The fact that the code fits all relevant quantities into the eight SSE2 on-chip registers helps performance, however the most remarkable feature of this subroutine is that it manages to accomplish its task without a single conditional branch instruction. This is possible due to creative use of the `PCMPEQD` comparison instruction in concert with various bit-shifting and logical SSE2 instructions; for examples of the general approach the interested reader may refer to Chapter 5 of the *Intel 64 and IA-32 Architectures Optimization Handbook*, available from `http://www.intel.com/products/processor/manuals/`. We feel that the 35 instructions of “straight-line” code we finally arrived at is close to the fastest possible implementation of this functionality on this processor. Performance results are described in the Results and Discussion section.

The actual sequence of SSE2 instructions follows:

```
mov     eax,[bucket_addr]
movdqa xmm0,[XMMPTRTYPE PTR eax]     // Load original ring buffer
movd    xmm1,[seqno]   // Load new seqno into low 32 bits
pshufd xmm1,xmm1,0     // Propagate low 32 bits to all 4 dwords
pand    xmm1,xmm6      // Mask out counts in new seqno
movdqa xmm2,xmm0
pand    xmm2,xmm6      // Mask out counts in ring buffer
pcmpeqd xmm2,xmm1      // Find match, if any

// Compute the new dword to go at the front of the buffer
movdqa xmm3,xmm0
pand    xmm3,xmm2
por     xmm1,xmm3      // OK since if there is a match, the seqno.s must be equal
pshufd xmm3,xmm1,0xF9
por     xmm1,xmm3
pshufd xmm3,xmm1,0xFE
por     xmm1,xmm3      // Now xmm1 contains dword to go at front, including count, in
    // (at least) the rightmost dword
pand    xmm1,xmm7      // Extract just the rightmost dword

// Compute the other 3 dwords
pslldq xmm2,4  // Also ensures low 32 bits are zero
pshufd xmm3,xmm2,0x90
por     xmm2,xmm3
pshufd xmm3,xmm2,0x40
por     xmm2,xmm3      // Now xmm2 contains mask extending from one−to−the−left−of
    // match out to the left
pshufd xmm3,xmm0,0x90  // xmm3 contains shifted version of buffer
pand    xmm0,xmm2      // Keep only the part that should not be shifted from xmm0
pandn   xmm2,xmm3      // Put only the part that should be shifted in xmm2
por     xmm0,xmm2      // Combine them
pand    xmm0,xmm5      // Extract just the remaining 3 dwords

// Combine the two parts
```

```
por     xmm1,xmm0        // Combine them
psubd   xmm1,xmm7        // Increment the score
movdqa  [XMMPTRTYPE PTR eax],xmm1     // Store the result

// Keep track of the largest count that this k-tuple has incremented
// This could also be done in standard assembler, since we are only
// interested in a single 32-bit quantity.
pslld   xmm1,32-LOG2RINGBUCKETS // All we want is to be able to compare counts
movdqa  xmm2,xmm4
pcmpgtd xmm4,xmm1       // Is the current biggest still the biggest? (Signed compare, so
    // top bit of count must be off!)
pand    xmm2,xmm4
pandn   xmm4,xmm1
por     xmm4,xmm2
```

3.3 Sequence Buffering System

The `encode` program must frequently deal with sequences which are widely separated in the input file, so it requires efficient random access to the sequence data. Although all POSIX-compatible operating systems provide an `fseek()` system call to allow a program to position the file pointer at any byte in a file, the subsequent read will almost certainly require the disk read/write head to physically move, a process which may consume several milliseconds. When performed several million times, this comparatively slow mechanical movement is inefficient.

This problem is addressed in `coil` by a buffering system which takes advantage of the fact that although random file seeks are slow, in-memory sorts are fast. Instead of performing delta-encoding during each preorder tree traversal, two passes through the entire forest are performed. Before the first pass, the sequence data file is scanned to extract an array of sequence file offsets, and a memory buffer of user-specified size (defaulting to 100Mb) is allocated. Then during the first pass through the encoding forest, tree traversals simply record the seqnums that need to be retrieved, adding each seqnum, along with its target offset in the memory buffer (computed using the file offset array), to the array `ordered_seq_list`. During the second pass through the encoding forest, the function `get_next_seq()` is repeatedly called to actually read the next sequence from the memory buffer and perform delta-encoding or raw-encoding as necessary. This function performs sequence buffering in a “lazy” fashion: if a sequence is available in the buffer, it will be returned; otherwise, the `buffer_seqs()` function is first called, which determines how many sequences will fit in the buffer, sorts this portion of `ordered_seq_list` by seqnum and reads the sequences into memory. Each sequence is placed in the memory buffer at the location determined by its position in the traversal order. The benefit of sorting the `ordered_seq_list` array is that sequences are read in file order, minimising disk seeks: in the limit case where the entire database fits in the buffer, no seeks at all are performed.

3.4 Data Compression as Quick and Dirty Science

Both compression and scientific analysis have at their heart the quest for discovery of *patterns* in the data. It is interesting to contrast the demands placed on algorithms used for finding these patterns in these two cases.

For scientific analysis, accuracy of the pattern model—the extent to which the model reflects real-world processes—is paramount. It is important to discover patterns that are biologically plausible. Thus, when aligning protein sequences we take care to set up the amino acid scoring matrix so that groups of amino acids sharing known physical properties are marked as similar. Also, it is important to have a model whose assumptions and limitations can be stated precisely—or better yet, quantified. For this reason, when building phylogenetic trees we often choose the expensive maximum likelihood approach over the typically faster maximum parsimony search, since it allows the specification of a rich variety of models of sequence evolution.

Data compression also operates by attempting to model input data. Compression becomes possible only when the input data contains patterns that can be described in fewer bytes than can the raw data itself—in particular, random data is not compressible (Li & Vitányi, 2008).[1]

[1]Information theory tells us that random data is incompressible *in expectation* (or equivalently, that a sufficient amount of random data is incompressible). Any scheme that compresses simple patterns like homopolymer runs must expand some other patterns (e.g. to “escape” the symbol used to indicate the start of a run).

Many powerful compression algorithms work according to the *predictor-corrector* principle. Based on the data seen so far, the next unit of data is predicted; then an "error term" is written out to correct the prediction to produce the actual data unit. As a simple example, a digital sound recording consisting of a series of amplitude samples can be compressed by predicting the i^{th} sample value using a quadratic extrapolation from the previous three data points; the error term is then the difference between the prediction and the actual value. When the predictor is accurate, the error term is usually low; more precisely, the distribution of error terms is tightly clustered around a single value, and so has low entropy. This low-entropy stream of error values can then be encoded using a small number of bits by a statistical compression algorithm such as Huffman coding.

Usually, as the accuracy demanded of a predictor increases, both the computational complexity and the sheer computational effort required to produce that accuracy increase drastically. Thus it is usually the case that wringing the last few drops of accuracy out of a predictor requires a staggering amount of computation, which simply isn't worthwhile when we are only looking to save some disk space or speed up a file download. Certainly, the more accurate the model, the better the level of compression that will be achieved; but generally speaking, a low-order approximation to the real-world process provides an ample compression gain for modest time expenditure.

With scientific analysis, we want to find the model that best explains the data, ideally in terms of meaningful parameters. With data compression, we just want to find a model that is "usually better than nothing at all"—we don't particularly care how or why it works, so long as it can be found quickly and works well, most of the time. Thus, despite the difference in emphasis, it should come as no surprise that an algorithm developed to tackle a problem in one field is useful to the other. As computer hardware and algorithms continue to improve, making feasible ever more accurate compression schemes, it will be interesting to see how these two fields continue to come together in the future.

3.5 General-purpose vs. Specialised Compression

One issue that doesn't receive attention in the published paper is the maintenance burden associated with choosing a specialised compression program such as `coil` over a widely available general-purpose compressor such as `gzip`. Can a 5% improvement in compression justify switching to a new program, with the possibility of bugs, stability and portability issues that doing so entails?

Clearly, if you will be storing only a small amount of sequence data, the answer is "No". But `coil` is designed for archiving large databases of DNA sequences, and its compression levels are expected to increase with increasing dataset size. Fig. 4 on p. 55 corroborates this, indicating that `coil`'s compression slowly begins to outperform the general-purpose compressor 7-Zip as the dataset grows in size, despite identical memory requirements. This suggests that `coil` will produce more substantial compression gains on the ever-larger databases now being produced by next-generation sequencing systems.

The fact that I am currently the only person maintaining the `coil` software is a legitimate cause for concern for would-be adopters, but in fairness, almost every software project—including the now-ubiquitous general-purpose compressor `bzip2` (Seward, 1997)—began life this way. By making `coil` freely available and placing it in the online SourceForge repository, access is ensured for the foreseeable future. Finally, in an environment where work with datasets of this type and size is routinely carried out, there are probably already a variety of other bioinformatics tools in use as well—tools which face similar stability and availability challenges. In other words, it is quite likely that the people who would be in a place to use `coil` are already familiar with managing an ecosystem of evolving analysis software.

3.6 Erratum

The paper refers several times to an "in-order" traversal in the `encode` and `decode` steps. In fact, these steps use a preorder traversal—that is, a depth-first traversal in which a parent node is always processed before its children.

Chapter 4

Treeness Triangles: Visualizing the Loss of Phylogenetic Signal

4.1 Introduction

This chapter presents the paper "Treeness Triangles: Visualizing the Loss of Phylogenetic Signal", which was published in the journal *Molecular Biology and Evolution* in 2007.

All software associated with this paper is now freely available via Subversion from `http://treetri.sourceforge.net`.

A phylogenetic dataset, such as a sequence alignment or pairwise distance matrix, contains a great deal of information; sometimes, too much for the purpose at hand. This can occur for example when many datasets need to be compared. In such cases, it is useful to be able to *summarise* a dataset.

A useful summary captures the "essence" or important aspects of the dataset, while discarding less relevant information. In the case of phylogenetic analysis, a relevant question is: To what extent does this dataset fit a phylogenetic tree?

Because evolution involves stochastic mutations, it is expected that the same mutation may occur on different branches in the tree, and that a mutation in one branch may be reversed by a later mutation in the same branch. These events, termed *homoplasies* and *reversals* respectively, introduce signals into the resulting sequences that contradict the signals corresponding to the true tree. Although they present a difficulty for tree reconstruction, provided that enough data is available, they can be dealt with by adopting an explicitly stochastic model of sequence evolution that accounts for their occurrence. This is possible because these signals are expected to be uncorrelated and smaller than the signals for the true tree.

On the other hand, processes that are not restricted to a single phylogenetic tree introduce a different and more interesting class of problems for reconstruction. Examples of such processes include datasets formed from a mixture of different trees, lineage sorting, hybridisation, and horizontal gene transfer. Perhaps more subtly, *selection*—a fundamental evolutionary process—can skew the results of molecular sequence tree reconstruction. This is because it introduces a new possible explanation for why two sequences are similar: it could be because they diverged recently (the usual explanation), *or* because they diverged long ago but now encode beneficial traits in both species, arrived at independently through a process of convergent evolution. Since the goal of phylogenetic analysis is to gain an accurate understanding of the distant past, it is vitally important to estimate the extent to which the primary simplifying assumption of a single phylogenetic tree actually applies to a given dataset.

The Background section of the paper describes a variety of existing visualisation techniques. One approach not mentioned there is δ plots (Holland *et al.*, 2002), which enable identification of problematic taxa. While useful, sometimes it is necessary to display datasets more compactly. Approaches that summarise a dataset down to a single scalar goodness-of-fit value are certainly possible, and have the advantage of being highly amenable to statistical analysis (Goldman, 1993). This approach allows many datasets to be summarised in a single diagram, but it limits the amount of information that can be conveyed about each dataset. However it is possible to keep slightly more information without sacrificing the intuitive visualization. Treeness Triangles accomplish this by turning a phylogenetic dataset into a single point on a ternary plot, where the co-ordinates are given by the proportions of the signal that correspond to internal edges on the best tree, external edges on the best tree, and residual signal that does not correspond to any edge on the best tree. Importantly, the dimensionality of the resulting summary is low enough to enable easy interpretation of regions on the plot, while remaining high enough to distinguish important features of the dataset.

This paper demonstrates the usefulness of the Treeness Triangle approach, both on simulated and on real chloroplast datasets. The method itself is very general, requiring only (a) any kind of data that can be used to build a set of *splits* (such as an alignment or pairwise distance matrix) and (b) either a particular tree or a tree-building algorithm. The choice of which methods to use to turn the dataset into (a) a set of splits and (b) a tree are unrestricted—it is not required that the splits in the tree form a subset of those produced by the dataset. As such, Treeness Triangles can be used not only to compare multiple datasets, but also to compare the effect of multiple different models or tree-building methods on a single dataset.

4.2 Why does distance correction amplify residual signals?

Paragraph 3 on p. 78 contains some interesting observations regarding the effects of distance correction that deserve further explanation. That correcting distances for unobserved changes should lead to an apparent increase in internal edge signal component is welcome, but the observation that such corrections also appear to amplify residual signals is at first worrying. However, both phenomena have the same underyling cause.

Distance correction for unobserved changes increases all distances in a distance matrix, but, being nonlinear, it increases large distances the most. The longest distances are usually those corresponding to paths between taxa in the true tree that contain many internal edges; the shortest distances are usually those corresponding to paths between taxa in the true tree that contain few internal edges, such as the 2-edge paths between neighbouring taxa. Thus the splits supported by these short distances—being predominantly the external edges of the true tree—are given the lowest boost by distance correction. All other splits—being predominantly those corresponding to internal edges, and splits that are incompatible with the true tree—are given a relatively greater boost. After normalising all split weights, we should therefore expect to find that both the internal and residual components are relatively higher than for an uncorrected dataset.

4.3 Clarification

On p. 79, the paper discusses lineage sorting and lateral gene transfer under the rubric of “non-treelike evolutionary processes”. This is perhaps an awkward choice of words, since in the absence of recombination events, both of these processes can indeed be represented on a tree. The confusion can be cleared up by keeping in mind the context of the paragraph, which begins, ‘The issues become distinctly more problematic if we entail the further assumption that “all” gene sequences in a particular given chromosome are related by one and the same tree’. In other words, this paragraph concerns itself with processes that generate entire chromosomes, rather than individual genes. At this level, it is sensible to talk about lineage sorting and lateral gene transfer being “non-treelike”, since the chromosome cannot in general be modelled by a single tree when these processes are active—even though each of the individual gene trees, considered in isolation, is of course treelike.

Treeness Triangles: Visualizing the Loss of Phylogenetic Signal

W. T. White,[1] *S. F. Hills,*[1] *R. Gaddam,*[1] *B. R. Holland,* and David Penny**

*Allan Wilson Center for Molecular Ecology and Evolution, Massey University, Palmerston North, New Zealand

It is well known that molecular data "saturates" with increasing sequence divergence (thereby losing phylogenetic information) and that in addition the accumulation of misleading information due to chance similarities or to systematic bias may accompany saturation as well. Exploratory data analysis methods that can quantify the extent of signal loss or convergence for a given data set are scarce. Such methods are needed because genomics delivers very long sequence alignments spanning substantial phylogenetic depth, where site saturation may be compounded by systematic biases or other alternative signals. Here we introduce the Treeness Triangle (TT) graph, in which signals detectable by Hadamard (spectral) analysis are summed into 3 categories—those supporting 1) external and 2) internal branches in the optimal tree, in addition to 3) the residuals (potential internal branches not present in the optimal tree). These 3 values are plotted in a standard ternary coordinate system. The approach is illustrated with simulated and real data sets, the latter from complete chloroplast genomes, where potential problems of paralogy or lateral gene acquisition can be excluded. The TT uncovers the divergence-dependent loss of phylogenetic signal as subsets of chloroplast genomes are investigated that span increasingly deeper evolutionary timescales. The rate of signal loss (or signal retention) varies with the gene and/or the method of analysis.

Introduction

Estimating phylogenies for deep divergences with sequence data is known to be a mathematically hard problem for a number of reasons. Over timescales on the order of about 600 Myr or more, the historical signal contained in sequences will be obscured by random noise (Penny et al. 2001). The theoretical results of Mossel and Steel (2004, 2005) demonstrate that under standard Markov models, as currently employed in molecular phylogenetics, primary sequences should lose all information about divergences approaching 1 billion years in age. For example, following theorem 14.2 of Mossel and Steel (2005), we can calculate that for 4 sequences of length 1000 evolving under a Jukes–Cantor model of nucleotide substitution with a mutation rate per nucleotide of about 10^{-8} per year, that if all 4 lineages existed as far back as 1 billion years ago, the probability of correctly estimating the tree would be 1/3 plus 0.002 (where the 1/3 term is just the chance of guessing correctly). With this model and substitution rate, it requires sequences of ~100,000 bp to have a 50% chance of recovering the correct tree for just 4 taxa. This calculation assumes ideal conditions; any sources of conflicting information would require longer sequences to compensate, and hence, the calculation places an upper bound on the expected result for the case of a simple, known model.

A related complication is that commonly used models of sequence evolution assume that, across the entire tree, each site is evolving in the same rate class. This includes the widely used general time reversible (GTR) model and its extension to models where a distribution of rates-across-sites (RAS) is assumed, with or without some sites being considered to be invariant. However, models assuming a gamma distribution require that each site must stay in the same rate class across all lineages (Steel et al. 1994). Such RAS models are only a simplified approximation of how sequences really evolve in nature (Lockhart et al. 2000; 2006), but for shorter time scales they provide a sufficiently good approximation to allow accurate phylogenetic estimation. We refer to such short to intermediate time periods (up to about 300 Myr) as the "comfort zone" because simulations reinforce the conclusion that phylogenetic inference is very powerful here (Penny et al. 2001). However, over time scales of half a billion years or more, the failure to incorporate lineage-specific processes, such as changes in nucleotide composition between taxa, may have dire consequences for phylogenetic estimation (see e.g., Ho and Jermiin 2004). Simulations allow us to predict the loss of information under specific models, but for real data sets where the actual substitution process is poorly understood, we need to be able to assess quantitatively the phylogenetic information in a given data set.

Confidence in inferred trees is often estimated by bootstrap values or posterior probabilities. Such values are useful when assessing whether or not sampling error may be influencing the results. However, bootstrap values do not detect systematic error; thus, they do not guarantee whether or not the branch in question is correct. For example, several studies of genome-scale data sets have shown that "support" in terms of bootstrap proportions (BPs) can swing from 100% for one tree to 100% for a different tree by adjusting the model of nucleotide substitution (Phillips et al. 2004; Goremykin et al. 2005). The bootstrap is generally not useful for assessing either loss or presence of phylogenetic signal for deep divergences because it does not take into account systematic error such as mutational bias (Lockhart et al. 1992; Lockhart and Cameron 2001; Buckley 2002). Stated another way, the bootstrap permits statements about site pattern frequencies, but it does not address the issue of whether or not site patterns reflect historical signal.

To determine whether systematic error is readily detectable for a given data set, tools to evaluate the goodness of fit of models of evolution are often employed. In present practice, goodness of fit is typically assessed using relative tests such as the likelihood ratio test or the Akaike Information Criterion as implemented in Modeltest (Posada and Crandall 1998), which ask whether model A fits the data significantly better than model B without, however, revealing how close model B comes to approximating

[1] These authors contributed equally to this work.

Key words: plastid genomes, spectral analysis, model misspecification, exploratory data analysis, ternary plot, Hadamard conjugation.

E-mail: D.Penny@massey.ac.nz.

Mol. Biol. Evol. 24(9):2029–2039. 2007
doi:10.1093/molbev/msm139
Advance Access publication July 13, 2007

the true model. Another class of tests has been used to answer questions about the absolute goodness of fit of models to data in a phylogenetic context (Reeves 1992; Goldman 1993; Bollback 2002; Waddell 2005; Jayaswal et al. 2005), but failure to pass such tests does not explain what aspects within the data are causing the poor fit. The parametric bootstrap is another test and can be used to compare, for example, the observed and predicted numbers of "singleton" sites, which basically correspond to the external branches of the tree (Goremykin et al. 2005; Waddell 2005).

Phylogenetic network methods also allow exploration of different, potentially conflicting, signals in the data. It is well known that there is a one-to-one correspondence between phylogenetic trees and sets of compatible splits; a binary tree with n taxa corresponds to a set of $2n - 3$ splits. Network methods allow sets of incompatible splits and correspondingly more detailed graphs. One of the first was split decomposition (Bandelt and Dress 1992a) which takes a metric (distance matrix) on n taxa and produces a set of up to $n(n - 1)/2$ weakly compatible weighted splits, as implemented in SplitsTree 4 (Huson and Bryant 2006). A useful feature is that both the proportion of the metric that is explained (graphically represented) by the split system and the residual that is not explained (undepicted) are both calculated. NeighborNet (Bryant and Moulton 2004) is a more recent method that produces a set of up to $n(n - 1)/2$ circular splits; these can always be represented on a planar graph. Other exploratory methods include spectral analysis (Hendy and Penny 1993), Lento plots (Lento et al. 1995), and consensus networks (Holland et al. 2005). These methods have proved useful for assessing conflicting signals within individual data sets (Kennedy et al. 2005; Nannya et al. 2005). The likelihood-mapping approach of Strimmer and von Haeseler (1997), which also uses a triangle plot, provides a useful graphical gauge of phylogenetic signal without recourse to assumptions about the underlying tree. Unfortunately, its output may be difficult to interpret: if most points fall near the center of the diagram, it can be concluded with confidence that the data is non-treelike, but if the points cluster at corners of the triangle, the data may or may not be treelike. More importantly, there is frequently a need to compare multiple data sets or various models on the same data set. In such cases, it is convenient to have an exploratory approach that enables rapid comparison across many data sets. Although the approaches mentioned above are useful, they are also visually complex—meaning it is hard to compare results across many data sets and treatments. For example, whereas likelihood-mapping summarizes a data set with a set of points on a diagram, a treeness triangle (TT) summarizes a data set with a single point, enabling multiple data sets, or multiple analyses of a single data set, to be compared on a single diagram. Although the dekapentagonal mapping approach of Zhaxybayeva et al. (2004) extends the quartet-based likelihood-mapping method to 5-taxon data sets, with a single point per data set, generalizing the method to n taxa appears problematic.

Building upon the concept of treeness, introduced by Andreas Dress and used in Eigen and Winkler-Oswatitsch (1981) and Eigen et al. (1988) to assess how well data fit a tree, we introduce the "Treeness Triangle" (TT) method. This assorts phylogenetic signals in aligned sequences into 3 components: signals that correspond to internal edges (branches) of a tree (I), signals that correspond to external edges of a tree (E), and the residual signals (R) that correspond to edges not present in the specified tree. These 3 values must sum to 1.0 and can therefore be plotted in a standard triangle (ternary) plot that readily reveals the relative proportion of each signal type in a given data set. We illustrate the TT with both simulated and real data—the latter from complete chloroplast (plastid) genomes. Here, we compare the redistribution of signal proportions for the same genes as a function of increasing evolutionary time from flowering plant (fp) evolution spanning roughly 160–200 Myr (Magallon and Sanderson 2005) to the early diversification of photosynthetic eukaryote lineages, including red algae, whose fossil record spans at least 1200 Myr (Butterfield 2000). It is essential to understand the extent to which sequences retain phylogenetic signal for ancient divergences and to detect conflicting signals. For the reasons given above, this chloroplast data set is a suitable test case for evaluating the TT.

Materials and Methods

Simulated Data

Random ultrametric trees were sampled from the PDA (Proportional to Distinguishable Arrangements) distribution, in which each tree topology is equally likely: the Markov model that generates these trees is in Steel and Penny (1993). Each random ultrametric tree was produced by taking a symmetric two-taxon rooted tree and randomly adding edges. Sequences were simulated on these random trees using Seq-Gen (Rambaut and Grassly 1997) and the Jukes–Cantor model, with 0.2 (figs. 2*A* and *D*), 0.4 (figs. 2*B* and *E*), and 0.6 (figs. 2*C* and *F*) expected mutations per site along any path from the root to a tip. One hundred random trees were produced, and for each tree and mutation rate, data sets were generated with 100, 200, 400, 800, 1600, 3200, and 6400 sites.

Real Data

The real data set has 30 complete chloroplast (plastid) sequences and is subdivided into 4 overlapping subsets of 12 taxa each. The first subset has 12 flowering plants (fp), and each subsequent subset contains 6 sequences from the previous data set and 6 new ones. Thus, the land plant (lp) data set has 6 fps and 6 others from conifers to bryophytes; the green plant (gp) data set has 6 lps and 6 green algae; and the plastid (pl) data set has 6 from the gp data set and 6 other algae. The taxa in each data set, together with GenBank accession numbers, are shown in table 1.

Complete annotated plastid genomes were downloaded from GenBank, and annotations for the genomes were tabulated using a Perl script. A table was generated which included information about gene sequence, protein sequence, and gene location. The sequences were then imported into a Microsoft Access database. The database allowed sequences for each gene to be accessed quickly

Table 1
The Complete Chloroplast Genomes Used in This Study and the Data Sets They Appear in. They are fp (flowering plants), lp (land plants), gp (green plants), and pl (plastids–reds, greens, and browns)

Genome	Plant Group	GenBank	Data set			
Zea mays	Poaceae (Panicoideae)	NC001666	fp	lp		
Oryza sativa	Poaceae (Ehrhatoideae)	NC005973	fp	lp		
Nicotiana tabacum	Solanaceae	NC001879	fp			
Atropa belladonna	Solanaceae	NC004561	fp			
Arabidopsis thaliana	Brassicaceae	NC000932	fp	lp		
Spinacia oleracea	Caryophyllales	NC002202	fp	lp	gp	pl
Oenothera elata	Myrtales (Rosids)	NC002693	fp			
Lotus corniculatus	Legume	NC002694	fp			
Calycanthus floridus	Magnoliid	NC004993	fp			
Panax ginseng	Asterid	NC006290	fp			
Amborella trichopoda	ANITA	NC005086	fp	lp	gp	
Nymphaea alba	ANITA	NC006050	fp	lp		
Pinus thunbergii	Conifer	NC001631		lp	gp	
Adiantum capillus-veneris	Fern	NC004766		lp	gp	pl
Psilotum nudum	Fern-ally	NC003386		lp		
Physcomitrella patens	Moss	NC005087		lp	gp	
Anthoceros formosae	Hornwort	NC004543		lp		
Marchantia polymorpha	Liverwort	NC001319		lp	gp	pl
Chaetosphaeridium globosum	Streptophyte alga	NC004115			gp	pl
Mesostigma viride	Streptophyte alga	NC002186			gp	
Nephroselmis olivacea	Chlorophyte (Prasinophyceae)	NC000927			gp	pl
Chlorella vulgaris	Chlorophyte (Trebouxiophyceae)	NC001865			gp	
Chlamydomonas reinhardtii	Chlorophyte (Volvocales)	NC005353			gp	
Pseudendoclonium akinetum	Chlorophyte (Ulvales)	AY835431			gp	pl
Odontella sinensis	Diatom	NC001713				pl
Guillardia theta	Cryptophyte	NC000926				pl
Cyanophora paradoxa	Glaucocystophyceae	NC001675				pl
Porphyra purpurea	Rhodophyte (Bangiales)	NC000925				pl
Cyanidium caldarium	Rhodophyte (Cyanidiales)	NC001840				pl
Gracilaria tenuistipitata	Rhodophyte (Florideophyceae)	NC006137				pl

across the taxa of interest. For each gene sequence, alignments were carried out in BioEdit (Hall 1999); nucleotide data was translated to protein sequences, aligned, and translated back to nucleotide sequences. Where automated alignment was carried out, Clustal X (Thompson et al. 1994) was used, together with manual editing. Distance matrices were generated using PAUP* (Swofford 2001) from the aligned gene data sets (all alignments are available from http://awcmee.massey.ac.nz/downloads.htm).

Hadamard Transformation

Although the TT could be used directly on the frequencies of splits as observed in the data, it is usual to use it after correcting for inferred multiple changes. For mathematical reasons (Hendy et al. 1994), the full Hadamard transform requires either 2-state characters with a symmetric distribution or 4-state characters for the Kimura 3ST model and its submodels, namely the Jukes–Cantor and Kimura 2ST. However, the distance Hadamard calculation can be used with more complex models, including those that are nonstationary such as the general Markov model to which the LogDet applies (Lockhart et al. 1994) and any form of maximum likelihood distances (Felsenstein 2003, p. 196–221). This method is summarized in the next section. Despite it initially appearing counterintuitive, because of the reduction of information in distances relative to sequences (Penny 1982; Huson and Steel 2004), there are some potential advantages of the distance Hadamard method over the full Hadamard. Because the distance Hadamard only uses pairwise distances, both the variance and the bias are reduced when correcting for inferred multiple changes (Hendy and Charleston 1993; Charleston et al. 1994; Waddell et al. 1994; Nei 1996). The variance and the bias on distance values both increase as the number of changes between taxa increases, and the increase in the bias is faster than linear owing to a logarithmic factor used in the correction term (Tajima 1993). Obviously, the minimum observed length of a quartet must be larger than that for the pairs contained within it, and consequently, the variance and bias of the inferred length of the quartet will be larger than for either pair. However, because of the loss of information in distances (Penny 1982; Huson and Steel 2004), we test for the effect of this loss and also use the projected Hadamard method (Waddell and Hendy 1997). This uses a separate Hadamard conjugation for each of the 3 parameters under the Kimura 3ST model. The comparison of the distance and projected Hadamard approaches is thus straightforward.

Calculation of the Distance Hadamard

The Hadamard transformation requires distance values for all subsets of taxa with an even number of members; 0, 2, 4, 6, 8, . . . n. This is an extension from quartet methods (e.g., Vinh and von Haeseler 2004) that only include subsets of 4 taxa. The values for ${}^{n}C_2 = n(n-1)/2$ pairs of taxa are standard pairwise distances and are given by the input distance matrix. The values are either observed (uncorrected or Hamming) distances calculated directly from sequences or corrected (inferred) distances. Similarly, there are ${}^{n}C_4$ possible quartets of 4 taxa and each value is the minimum of the 3 combinations of pairwise distances. For example, for the quartet $q = \{i, j, k, l\}$, the entry is $\min\{d(i, j) + d(k, l), d(i, k) + d(j, l), d(i, l) + d(j, k)\}$, where $d(x, y)$ is the pairwise distance between taxa x and y. Again, the quartet values are from observed values or from inferred distances. For all larger subsets having an even number m of sequences, the distance is determined by finding the combination of taxon pairs from this subset having minimum total distance. In practice, it suffices to examine the sums of the distance values for each pair and the remaining $m - 2$ taxa, which have already been calculated.

Treeness Triangle

The TT uses splits, subdivisions of a set of n taxa into 2 disjoint subsets, thus corresponding to an edge in a tree. In general, for n taxa there are 2^{n-1} splits including the null split. The analysis was carried out on software based on SpectroNet (Huber et al. 2002). The programs are available from http://awcmee.massey.ac.nz/downloads.htm. The main operations are indicated in supplementary figure S1 (Supplementary Material online). Nucleotide or RY-coded sequences can be translated directly into the frequency of observed splits (s vector), or, using a program such as PAUP* (Swofford 2001) or the freely available PHYLIP (Felsenstein 2004), nucleotide, RY-coded or protein sequences can be converted to either observed or corrected pairwise distances. Pairwise distance values can be expanded into full generalized distances (respectively, r for observed and ρ for inferred/corrected), which have values for all subsets with an even number of taxa (Hendy and Penny 1993; Penny et al. 1993; Hendy et al. 1994) via the distance Hadamard. The values in the s, r, ρ, and γ vectors (supplementary fig. S1, Supplementary Material online) are interconvertible by the Hadamard conjugation. Subsets of entries from either the s or γ vector can be selected, for example, those with values greater than zero. A network (Huber et al. 2002), Lento plot (Lento et al. 1995), or TT (fig. 1*A*) can then be drawn. The Lento plot (fig. 1*B*) and TT both require a tree for their calculation. In the current implementation, the tree is obtained by the closest tree method (Hendy 1991) using a standard branch and bound search (Penny and Hendy 1987), although we emphasize that the TT can be used with any methods for producing both a set of splits and a tree from a data set. For example, when working with distance data, the weakly compatible set of splits output by SplitsTree 4 (Huson and Bryant 2006) could be used as an alternative to splits generated via the distance Hadamard, and a minimum evolution algorithm (Rzhetsky and Nei 1992) could be used to generate a tree.

Of course, if the model used to build the tree is incorrect (model misspecification) or if there is insufficient data (sampling error), it is possible that the tree used as input to the TT does not match the (usually unknown) true tree. In the case of the closest tree algorithm used in this paper, the tree recovered corresponds roughly to the tree that gives the best possible treeness values, in the sense of minimising the R component. Thus, the treeness components computed for a data set will be optimistic when a tree different from the true tree better explains the data. This does not invalidate the outcome of a TT analysis: the TT faithfully evaluates the tree likeness of a data set "with respect to a tree-building method of the user's choice." Although both sampling error and model misspecification can be tested for (e.g., using bootstrapping and the absolute and relative tests of goodness of fit described in the introduction, respectively), this is probably not justified when using the TT simply as an exploratory data analysis tool.

In the TT, the values of all signals in the data sum to unity and the proportion of signal on the external (E) and internal (I) edges (branches) of the optimal tree are indicated by the first 2 of the 3 entries indicated at the 3 apices of the triangle. The sum of the residual signals (R) is the third entry. Given a set of splits and a tree as input, these 3 values are computed by classifying each split in the split set as an external edge of the tree, an internal edge of the tree, or absent from the tree and adding the split's weight to the corresponding total: E, I, or R, respectively. The final step is normalization so that the total $E + I + R$ equals 1. The upper apex represents the star tree where there are no internal edges in the data, the lower left where the data fits entirely onto internal edges of the tree, and the lower right where all signals are of equal value (there is no support for any particular tree). For a specified tree, the 3 classes of values (E, I, and R) are summed as described above, normally as the γ values, and these 3 coordinates are plotted within the TT as illustrated in figure 1*A*. This summarizes 3 signals in just one point, in contrast to a Lento plot (shown in fig. 1*B* for the fp data set). In further contrast to a Lento plot, the TT can summarize a large number of comparisons in a single graph (see figs. 2 and 3). For data that perfectly fit a tree, all points would have an R component of 0 and hence would lie on the line connecting the E and I apices.

Results

Simulated Data: TT Using the True Model

We first analyzed simulated data in order to examine the extent of the sampling error in the residuals when the model (tree plus mechanism of nucleotide change) is correct. Figure 2 shows results for simulated nucleotide data sets with 12 taxa and increasing numbers of sites. For 100, 200, 400, 800, 1600, 3200, and 6400 sites, each data set was analyzed for both the distance and projected Hadamard methods. Although in this case the true tree for each data set was available, the trees used for TT analysis were recovered from the simulated data using the closest tree algorithm as is usual for real data sets, allowing for the (realistic) possibility of recovering an incorrect tree. Additional data sets, created by shuffling the data within each alignment column, were used in order to measure the effect of complete information loss. For clarity, figure 2 only shows the results for data sets with 100, 400, 1600, and 6400 sites, and the shuffled sites of 6400 nt. Figures 2*A–C* shows the results for the distance Hadamard and figures 2*D–F* the projected Hadamard.

For figure 2*A* in particular, the points representing the same sequence lengths cluster into bands, each point

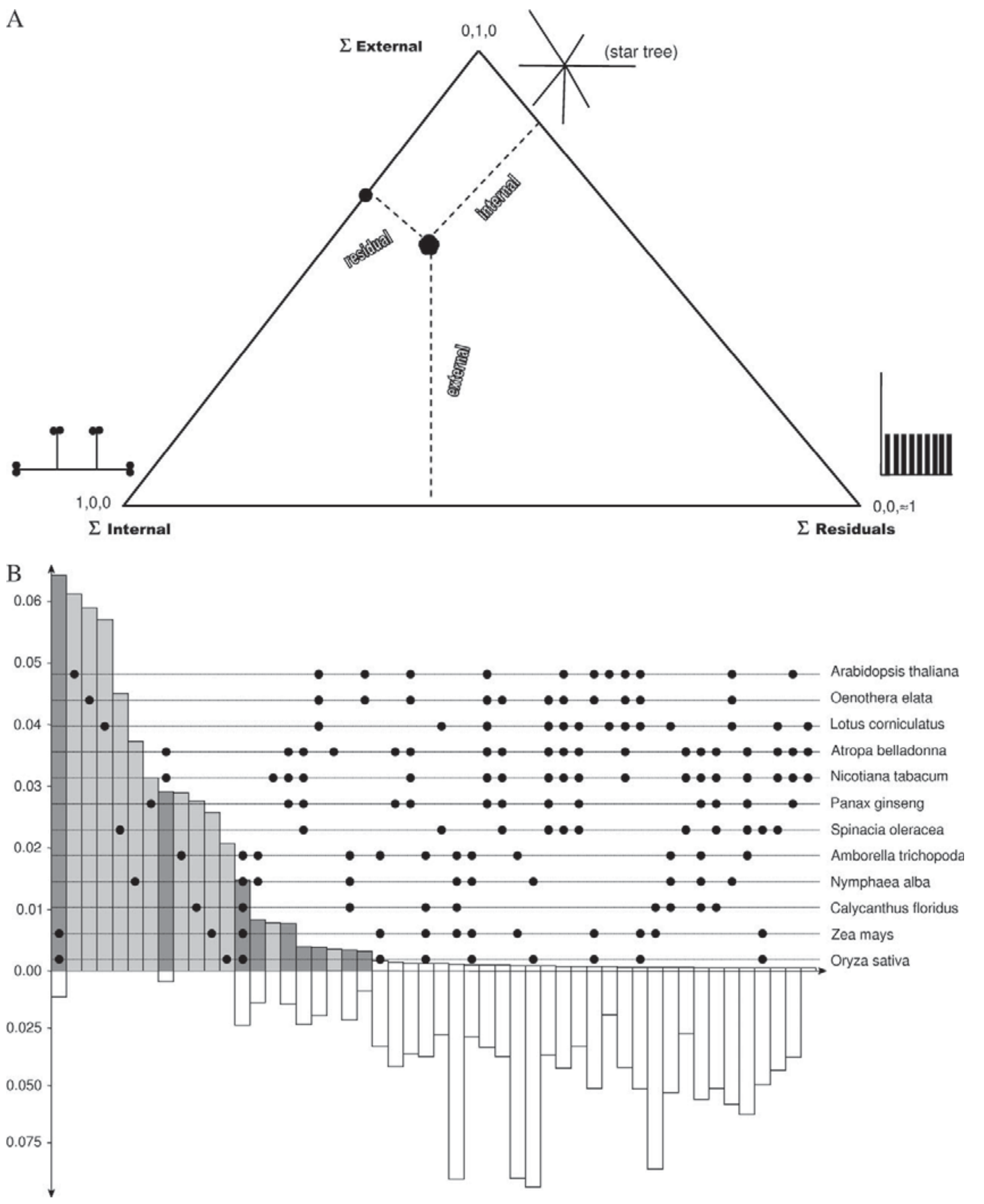


FIG. 1.—A TT (*A*) and a Lento plot (*B*). The sum of all signals in the data (γ values after a Hadamard transform) are normalized to unity. The 3 values calculated are the proportion of signal on the external and internal edges (branches) of the optimal tree and the sum of the remaining (residual) signals. The upper vertex of the triangle (*E*) is from a star tree where there are no internal edges in the data, the lower left (*I*) where the data fits entirely onto the internal edges of the tree, and the lower right (R, residuals) where all signals are of equal value (and thus the data does not represent a tree). The value plotted is 0.33, 0.53, and 0.14 for internal, external, and residuals, respectively. In principle, any tree can be used for the plot but the "closest tree" was used here because of its speed of computation and reasonable statistical properties (Hendy 1991). The Lento plot (*B*) shown is for the *atpB* gene of the fp data set. The values above the axis are the values of the signals for a split (edge or branch of a tree) and the values below the axis are the normalized sum of the values of other splits that are incompatible with that split. The bars have been shaded dark, medium, and white for splits belonging the E, I, and R classes, respectively. Note that the abscissa scale differs by a factor of 2 above and below the origin. The species, in order, are *Arabidopsis thaliana*, *Oenothera elata*, *Lotus corniculatus*, *Atropa belladonna*, *Nicotiana tabacum*, *Panax ginseng*, *Spinacia oleracea*, *Amborella trichopoda*, *Nymphaea alba*, *Calycanthus floridus*, *Zea mays*, and *Oryza sativa* (see table 1).

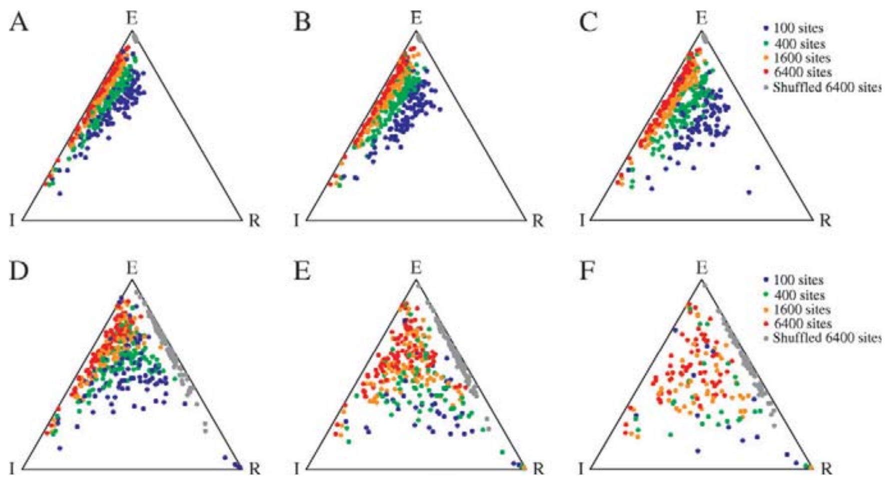


FIG. 2.—TT illustration of sampling error on simulated data. Results are for different sequence lengths and amounts of nucleotide change using the distance Hadamard (panels *A–C*) and projected Hadamard (panels *D–F*). TT points for 100 12-taxon trees under a molecular clock model were simulated with the Jukes–Cantor model and having 0.2 (panels *A* and *D*), 0.4 (panels *B* and *E*), and 0.6 (panels *C* and *F*) expected mutations per site between the root and the tip. For each point, data was generated on a different random tree consistent with a molecular clock and the optimal tree inferred by using the closest tree algorithm on the recovered split vector. The values are for 100 (blue), 400 (green), 1600 (orange), and 6400 (red) sites. Data sets created by taking 6400-site data sets and randomly shuffling nucleotides within columns are shown in gray. Note that some data sets produced split vectors that could not be analyzed using the projected Hadamard directly because of negative arguments to the log function; points for these data sets were omitted. (See also supplementary table S2, Supplementary Material online.)

indexed by 3 signal types (*I*, *E*, and *R*). As expected, with increasing sequence length the points approach the *I–E* ($R = 0$) line. Because this data is simulated on a tree and uses the same mechanism to simulate the data and recover the tree, sampling error is the only significant factor contributing to the residual component. These results can be compared later with real data where model misspecification may be significant. As expected (see Waddell et al. 1994), the residuals (*R* axis) decrease in inverse proportion to the sequence length. In contrast, there is a faster-than-linear increase in the residuals component as the rate of change increases. This trend is shown in row 1 (figs. 2*A–C*) and row 2 (figs. 2*D–F*), where the diagrams on the left, center, and right correspond to expected numbers of substitutions per site of 0.2, 0.4, and 0.6, respectively.

The spread of points along the *I–E* line occurs because each point is from a different random tree; the spread does not vary noticeably between data set sizes. The projected Hadamard (figs. 2*D–F*) retains more information from the original data and hence carries a larger residual component. This is seen by comparing each TT plot in figure 2*D–F* with the one immediately above it. This means that the distance Hadamard is still underestimating the full values of the residual component. With the projected Hadamard, there is still a significant residual component with 6400 sites for the highest rates of nucleotide change (fig. 2*F*).

Real Data

We first checked for each gene whether the parameters for the gamma distribution (Γ) of rates across sites and the proportion of variable sites (P_{var}) were reasonably constant in the 4 subsets of taxa (supplementary table S2, Supplementary Material online). To conform to the mathematical assumptions, this constancy of gamma and P_{var} should hold when going from the fp to the lp, gp, and pl data sets. Although the estimates for the gamma shape parameter vary considerably from ~0.3 to ∞ across genes and data sets, there is no clear trend with increasing divergence of the taxa. (Note that ∞ is a valid gamma value, indicating that all sites are evolving at an equal rate.) As expected, the proportion of invariant sites is significantly higher in the fp than the other 3 taxa sets. This may be a bias in estimation from having more constant sites in the fp alignments than in the 3 more divergent taxa sets. Nevertheless, the decrease in constant sites is consistent with the prediction of a relaxed covarion model that additional sites will become variable for deeper comparisons (Gaucher et al. 2002; Lockhart et al. 2006). In the 3 most divergent taxa sets (lp, gp, and pl), there is a significant positive correlation between the gamma shape parameter and the proportion of invariant sites (correlation coefficients of 0.46, 0.44, and 0.48, respectively). In other words, in models where more sites are classed as invariant, rates are close to being equal across the variable sites and in models where there are few invariant sites the rate distribution is more skewed.

Figures 3*A* and *B* are TT plots with points for each of 35 genes, calculated under the distance Hadamard (fig. 3*A*) and the projected Hadamard (fig. 3*B*). The points colored red, orange, green, and blue correspond to the fp, lp, gp, and pl subsets of taxa. The gray points are shuffled versions of each data set. For the distance Hadamard, there is a strong tendency, as expected, for the points to move closer toward the *E* (external branches) apex with older divergences (fp → lp → gp → pl). For the projected Hadamard, there is a similar tendency to move toward the E apex with increasing divergence. Compared with the distance Hadamard, the projected Hadamard yields TT points with a much larger residual value.

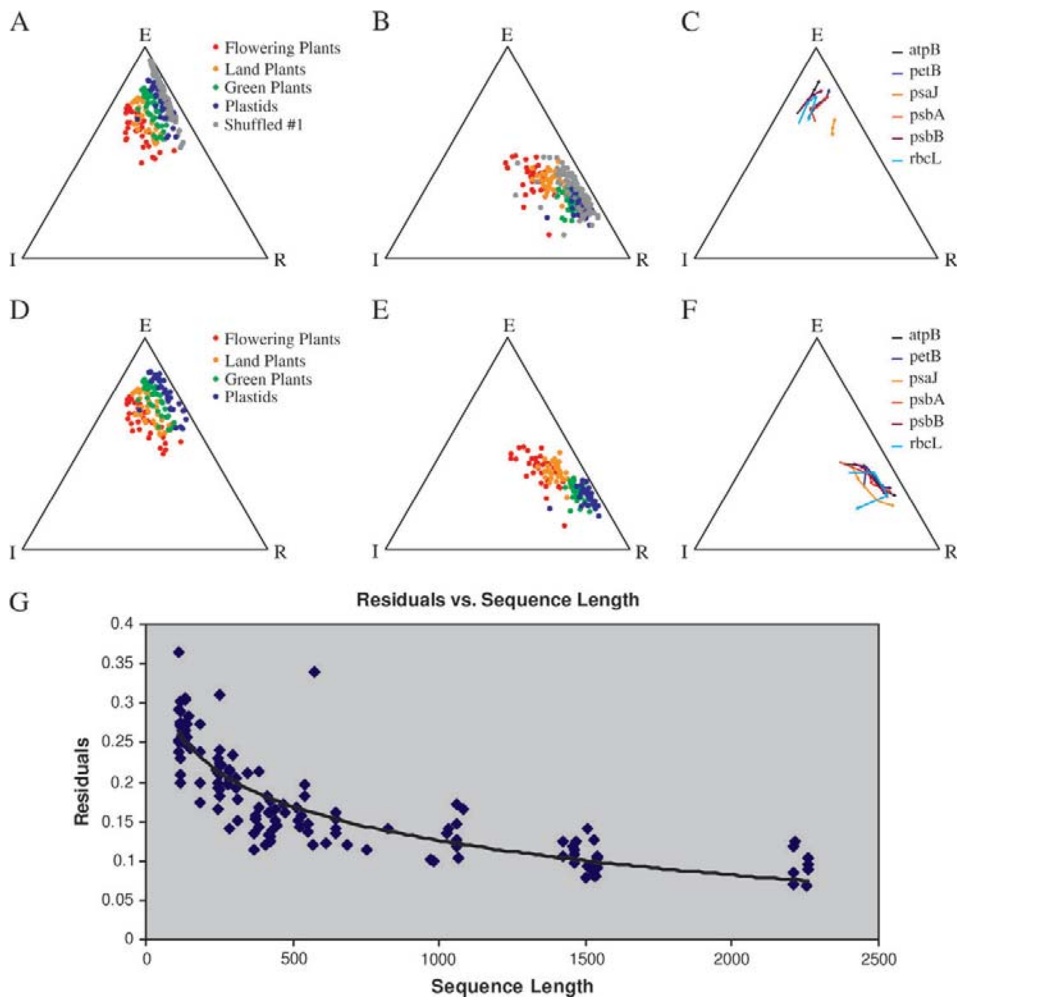


FIG. 3.—Basic results for the 4 chloroplast data sets, with the distance (panels *A* and *D*) and projected (panels *B* and *E*) Hadamard methods. Each of 35 genes from the 4 data sets (fp, red; lp, orange; gp, green; and pl, blue) are used, together with randomized (shuffled) columns for all 4 data sets in gray. In *A* and *B*, each dot represents a separate gene. In *C* and *F*, 6 genes are identified and the arrows indicate the change in TT value in going from the fp, lp, green algae, and pl data sets (fp $\rightarrow$ lp $\rightarrow$ gp $\rightarrow$ pl). There are 3 genes with >1,500 bp (*atpB*, *psbB*, and *rbcL*), 2 with >500 bp (*psbA* and *petB*) and one with <150 bp (*psaJ*). As expected, there is a decrease in signal for the internal branches on moving from the fps to the pl data set. For the pl data set (blue), there is apparently little phylogenetic signal at all on the internal branches (however, see the concatenated data set in fig. 4). This analysis shows that much more signal is retained in the projected Hadamard than for the distance Hadamard, but much of that the additional signal does not fit onto the optimal tree. *G* is the residual component of the TT plotted against sequence length for all genes and taxa sets. *D* and *E* are equivalent to 3A and 3B but are the result of using the global optimum tree (concatenated) instead of the closest trees calculated on each data set individually. As expected, the residuals are marginally larger in 3*D* and *E*.

With progressively deeper geological divergence times from ~200 Ma to ~1.2 Ga, the points in figure 4*A* migrate toward apex *E*, but there is no apparent shift toward the *R* apex, as might have been expected for random data. To understand this effect, consider the following. As sequence length tends to infinity, we expect shuffling by columns to produce homogenous genetic distances between all pairs of taxa. In other words, all entries in the resulting distance matrix, apart from the diagonal, would become equal to some constant *d*, whose value is determined by the number and nature of sequence differences in the data. Such distances can be represented exactly on a star tree with each external edge of length *d*/2; this corresponds to the upper point (the *E* apex) in the TT. However, for shuffled alignments of finite length, the values in the distance matrix only approximate *d*; hence, we see some signal mapped to internal edges and some, typically a larger component, mapped to residuals. Notably, genes for the pl data set, spanning more than 1.2 Ga, map to points in the same region of the plot as the shuffled (randomized) data. The contrast with figure 3*B* is instructive: with the projected Hadamard there is a stronger movement toward the *R* apex. Figures 3*C* and *F* track 6 individual genes, with arrows in the direction of increasingly divergent taxa sets.

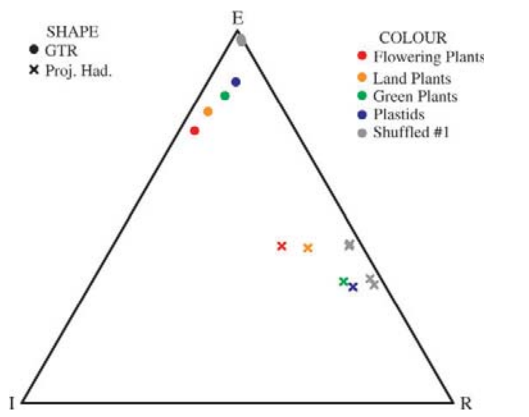


FIG. 4.—Results from concatenated genes for each of the 4 data sets (fp, red; lp, orange; gp, green; and pl, blue). Filled circles are used for distance (GTR) calculations, and crosses are used for projected Hadamard calculations on the concatenated data sets. For data sets with older divergences, the values again show a progression toward the *E* axis (longer external and shorter internal edges). Nevertheless, the points are much closer to the *I–E* (Internal–External) axis than with the individual genes (fig. 3). This is a positive result and implies that to some extent the nonphylogenetic signals observed with individual genes cancel out. In contrast, the crosses indicate that in comparison to the distance Hadamard, the projected Hadamard retains considerably more of the information in the original data, but little of the information retained corresponds to the closest tree.

In figure 3*A*, it appears that for individual genes most phylogenetic signal, corresponding to internal edges of the optimal tree, is lost in the oldest data sets comprising gp and algae, especially pl. Indeed, for many genes the residual component (the distance from the *I–E* axis) is larger than the signal for the internal branches of the optimal tree (distance from the *E–R* axis). In general, the TT reveals that for each gene taken individually, most of the phylogenetic signal for deep divergences has been lost. Figure 3*G* shows the relationship between sequence length and the residual component of the signals. As expected from the simulation results (fig. 2), the residuals component is generally smaller for longer genes.

For concatenated genes, however, the results for the distance Hadamard for each of the 4 data sets (filled circles in fig. 4) indicate a substantial component of signal that maps to internal edges. The expected migration of points toward the *E* (external) apex with increasing evolutionary time is observed in the transition from the fp → lp → gp → pl data sets. Again, for each of the corresponding data sets shuffled by columns, virtually all signal on internal edges of the tree is lost. However, it is most striking that in the concatenated data the points lie close to the internal–external (*I–E*) axis, meaning that the signal for the residual axis is both quite small and spread over many possible alternative signals. This important observation suggests that the high residual signal for individual genes differs across genes. Put another way, not only the residual signal from the individual genes could stem from both sampling effects of gene length and also lineage-specific differences in functional constraints (that might average out).

Another area where the TT allows easy comparison is across different treatments of the same data set. In figure 5, we show the effect of different distance corrections on the position of the points within the TT for the genes *atpF* (fig. 5*A*) and *petD* (fig. 5*B*); the results for the other genes are in supplementary material online. The 4 distance methods used are uncorrected *p* distances, filled diamond; Tamura–Nei corrected distances, open diamond; LogDet distances, closed circle; and GTR maximum likelihood distances, open circle. All genes, except for *rbcL* (see supplementary material online) show the fp, lp, gp, and pl progression. It is interesting that going from the uncorrected distances to any form of correction tends to increase the values of both the internal (*I*) and residuals components (*R*). This indicates that the uncorrected data underestimates the internal branches of the tree but simultaneously that the signal not conforming to an optimal tree is amplified when distances are corrected for multiple changes.

Discussion

The tendency in current phylogenetic practice is to focus attention on those aspects of a given data set that map onto a particular tree. But the issue of how well a bifurcating tree actually describes the observed properties of the data in question is at least as important. What can we really assume safely about sequence evolution? For any given "individual gene," it can probably be safely assumed that all sequences

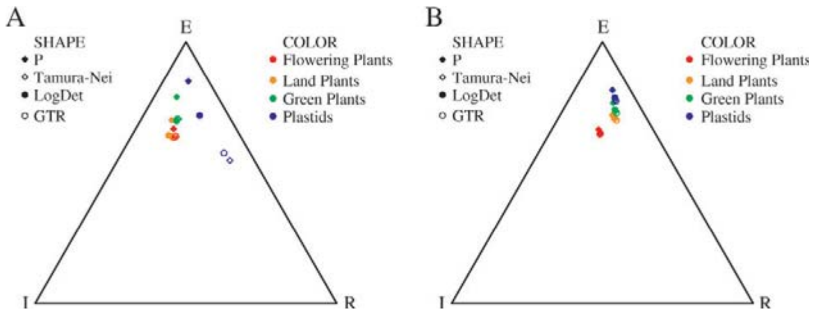


FIG. 5.—Effect of different corrections on the fit between model and data. The distance Hadamard applied under different corrections for inferred multiple changes. The 4 distance matrices used are: filled diamond, uncorrected *p* distances; open diamond, Tamura–Nei corrected distances; filled circle, LogDet distances; and open circle, maximum likelihood distances. *A* and *B* are for the proteins *atpF* and *petD*, respectively. In general, the more complex the optimal model, the better the fit between the data and the tree.

that we observe in nature are in fact related by a series of treelike lineage splits that correspond to a recurrent process of DNA duplication and mutational accumulation; the only readily imaginable exceptions to such a rule would entail intragenic recombination or gene conversion among sequence variants possessing fixed differences. If we neglect the latter 2 mechanisms, then our default assumption would be that gene and protein sequences are related by processes that in mathematical terms are well described by trees.

The issues become distinctly more problematic if we entail the further assumption that "all" gene sequences in a particular given chromosome are related by one and the same tree. This assumption is inherent to the concept that there is a single tree of life by which all things are related and that all we need to do is to identify its topology. But many evolutionary mechanisms that affect the evolution of genes are known, which are fundamentally not depictable as strictly bifurcating trees. The 4 most prominent and mechanistically best understood examples of non–treelike evolutionary processes include 1) hybridization events, as are common among flowering plants (fps); 2) gene transfers from organelle genomes in the endosymbiotic origin of organelles; 3) lineage sorting, which occurs when gene trees differ from species trees because of coalescence events occurring in a different order than speciation events (as described in e.g., Rosenberg 2002; Degnan and Salter 2005); and 4) lateral gene transfer among prokaryotic chromosomes, as mediated by 4a) transduction via phages, 4b) transformation in the case of naturally competent bacteria such as *Haemophilus influenzae*, and 4c) conjugation via plasmids, as any hospital that uses antibiotics can attest. Treelike or not, all these processes do fit within the more general mechanism of descent with modification.

In the age of genomes and phylogenomics, where gene trees are produced on an industrial scale, we often find discrepancies between trees produced for a collection of genes within a particular set of chromosomes. It has become quite popular to infer a prevalence of lateral gene transfer or other non–treelike biological process as the cause of such differences. However, from the mathematical standpoint the issue might more readily be formulated as, "How likely is it that we will infer the same tree, or even similar trees, for 2 genes from the same set of organisms even if we know exactly how molecules are evolving?" Even when the true tree and the true model of sequence evolution is known, as in computer simulated data, it is very difficult to infer the true tree for moderately diverged genes (Nei 1996; Penny et al. 2001), and only with such "perfect" data can we begin to feel how well or how poorly methods of phylogenetic inference actually perform with distantly related taxa. If our goal is to learn something about the evolutionary past from gene sequence data, we need to better understand the relationship between the data that we observe and the trees that are inferred from them. That means that there is a need to understand not only the site patterns that will fit onto a binary tree, but also those that will not (i.e., conflicting data). Networks, Lento plots, and TT plots are steps in that direction.

Here, we investigated both simulated data and real sequence data from chloroplast genomes. The reason for investigating the latter stems from the circumstance that, with the exception of *rbcL* (which has long been known to exhibit paralogy across the red algal-green algal boundary Martin et al. [1992]), there is every reason to assume that the sequences of proteins encoded in chloroplast genomes are all related by the same historical process of evolutionary bifurcations. This is because there are no known cases of gene families within chloroplast genomes, no duplicate copies of chloroplast genes (with the exception of those encoded in the inverted repeat, whose sequences are identical), and no known examples of gene replacement via lateral acquisitions (leaving *rbcL* aside). Therefore, for a given taxon sample, all chloroplast-encoded proteins should, in principle, produce the same tree in phylogenetic inference. The observation is, however, that they produce different trees, sometimes with very high BPs (Goremykin et al. 1997; Martin et al. 1998; Lockhart et al. 2000; Vogl et al. 2003). The reasons underlying the inability of current molecular phylogenetic methods to extract the same tree for different chloroplast proteins (or any other protein set where paralogy or lateral acquisitions can be reasonably excluded *a priori*), need clarification, if progress is to be made in understanding deeper evolutionary history. The problem of distinguishing between historical and other types of signal in molecular data is hard and becomes increasingly severe for deep divergence times.

The projected Hadamard (Waddell and Hendy 1997) uncovers more conflict than the distance Hadamard. There is still the option for exploring the full Hadamard on 4-state characters. However, this requires a vector with 4^{n-1} entries, rather than 2^{n-1} for the distance Hadamard and $3 \times 2^{n-1}$ for the projected. The number of signals in the residual component is large. For n taxa there are 2^{n-1} possible splits, n of which correspond to external branches of the tree, $n - 3$ to internal branches, and thus (omitting also the null split), there are $2^{n-1} - 2n - 4 = 2002$ splits for $n = 12$ taxa. In principle, both the mean and standard deviation of the support for any particular split can be calculated for the Hadamard (Waddell et al. 1994). In practice, the large number of signals means that the variance of the splits will be relatively high, and this will contribute to the higher residual values for the projected Hadamard versus the distance Hadamard.

In this paper, we have generated TT points with respect to the closest tree, although the method could be used more generally. For example, to compare the effect of different distance corrections on the weakly compatible splits systems produced by split decomposition (Bandelt and Dress 1992b), one could define a triangle point by summing up the weights of trivial splits (of the form A|B, where either |A| or |B| = 1) and assigning it to the E (external) corner, summing up the weights of the nontrivial splits and assigning it to the I (internal) corner, and assigning the split-prime residue to the R (residual) corner. When it comes to depicting signal conflicts, TT is complementary to both Lento plots and networks; Lento plots show all the conflicting signals and networks show the most important conflicting signals. A TT point shows how much conflicting signal there is, without identifying the signals, making it easy to compare the amount of signal across different data sets and treatments. TTs reveal that the vast majority of all phylogenetic signals observed in the real chloroplast data (or in simulated data) conflict with the optimal tree, rather than support it, even for comparatively short divergence times corresponding to less than about 200 Myr. Using the projected Hadamard, the difference between the shuffled and unshuffled pl data set was small. This warrants caution with regard to interpreting trees for deeper divergences.

Supplementary Material

Supplementary materials are available at *Molecular Biology and Evolution* online (http://www.mbe.oxfordjournals.org/).

Acknowledgments

This work was supported by the New Zealand Marsden Fund. We thank Bill Martin for considerable input for the design of the project at the SMBE meetings at UC Irvine, for the first chloroplast dataset used during development, and for ongoing discussions.

Literature Cited

Bandelt H-J, Dress AWM. 1992a. A canonical decomposition theory for metrics on a finite set. Adv Math. 92:47–105.

Bandelt H-J, Dress AWM. 1992b. Split decomposition: a new and useful approach to phylogenetic analysis of distance data. Mol Phylogenet Evol. 1:242–252.

Bollback JP. 2002. Bayesian model adequacy and choice in phylogenetics. Mol Biol Evol. 19:1171–1180.

Bryant D, Moulton V. 2004. Neighbor-net: an agglomerative method for the construction of phylogenetic networks. Mol Biol Evol. 21:255–265.

Buckley TR. 2002. Model misspecification and probabilistic tests of topology: evidence from empirical data sets. Syst Biol. 51:509–523.

Butterfield NJ. 2000. Bangiomorpha pubescens n. gen., n. sp.: implications for the evolution of sex, multicellularity, and the Mesoproterozoic/Neoproterozoic radiation of eukaryotes. Paleobiology. 26:386–404.

Charleston MA, Hendy MD, Penny D. 1994. The effects of sequence length, tree topology and number of taxa on the performance of phylogenetic methods. J Comp Biol. 1:133–151.

Degnan JH, Salter LA. 2005. Gene tree distributions under the coalescent process. Evolution. 59:24–37.

Eigen M, Winkler-Oswatitsch R. 1981. Transfer-RNA: the early adaptor. Naturwissenschaften. 68:217–228.

Eigen M, Winkler-Oswatitsch R, Dress A. 1988. Statistical geometry in sequence space—a method of quantitative comparative sequence-analysis. Proc Natl Acad Sci USA. 85:5913–5917.

Felsenstein J. 2003. Inferring phylogenies. Sunderland (MA): Sinauer Associates.

Felsenstein J. 2004. PHYLIP (phylogeny inference package). Seattle (WA): Department of Genome Sciences, University of Washington.Version 3.6b. Distributed by the author

Gaucher EA, Gu X, Miyamoto MM, Benner SA. 2002. Predicting functional divergence in protein evolution by site-specific rate shifts. Trends Biochem Sci. 27:315–321.

Goldman N. 1993. Statistical tests of models of DNA substitution. J Mol Evol. 36:182–198.

Goremykin VV, Hansmann S, Martin WF. 1997. Evolutionary analysis of 58 proteins encoded in six completely sequenced chloroplast genomes: revised molecular estimates of two seed plant divergence times. Plant Syst Evol. 206:337–351.

Goremykin VV, Holland B, Hirsch-Ernst KI, Hellwig FH. 2005. Analysis of Acorus calamus chloroplast genome and its phylogenetic implications. Mol Biol Evol. 22:1813–1822.

Hall TA. 1999. BioEdit: a user-friendly biological sequence alignment editor and analysis program for Windows 95/98/NT. Nucleic Acids Symp Ser. 41:95–98.

Hendy MD. 1991. A combinatorial description of the closest tree algorithm for finding evolutionary trees. Discrete Math. 96:51–58.

Hendy MD, Penny D. 1993. Spectral analysis of phylogenetic data. J Classif. 10:5–24.

Hendy MD, Charleston MA. 1993. Hadamard conjugation—a versatile tool for modeling nucleotide-sequence evolution. N Z J Bot. 31:231–237.

Hendy MD, Penny D, Steel MA. 1994. A discrete Fourier analysis for evolutionary trees. Proc Natl Acad Sci USA. 91:3339–3343.

Ho S, Jermiin L. 2004. Tracing the decay of historical signal in biological sequence data. Syst Biol. 53:623–637.

Holland BR, Delsuc F, Moulton V. 2005. Visualizing conflicting evolutionary hypotheses in large collections of trees: using consensus networks to study the origins of placentals and hexapods. Syst Biol. 54:66–76.

Huber KT, Langton M, Penny D, Moulton V, Hendy M. 2002. SpectroNet: a package for computing spectra and median networks. Appl Bioinformatics. 1:159–161.

Huson DH, Bryant D. 2006. Application of phylogenetic networks in evolutionary networks. Mol Biol Evol. 23:254–267.

Huson DH, Steel M. 2004. Distances that perfectly mislead. Syst Biol. 53:327–332.

Jayaswal V, Jermiin LS, Robinson J. 2005. Estimation of phylogeny using a general Markov matrix. Evol Bioinform Online. 1:62–80.

Kennedy M, Holland BR, Gray RD, Spencer HG. 2005. Untangling long branches: identifying conflicting phylogenetic signals a priori using spectral analysis, neighbor-net, and consensus networks. Syst Biol. 54:620–633.

Lento GM, Hickson RE, Chambers GK, Penny D. 1995. Use of spectral-analysis to test hypotheses on the origin of pinnipeds. Mol Biol Evol. 12:28–52.

Lockhart PJ, Cameron SA. 2001. Trees for bees. Trends Ecol Evol. 16:84–88.

Lockhart PJ, Huson D, Maier U, Fraunholz MJ, Van de Peer Y, Barbrook AC, Howe CJ, Steel MA. 2000. How molecules evolve in eubacteria. Mol Biol Evol. 17:835–838.

Lockhart PJ, Novis P, Milligan BG, Riden J, Rambaut A, Larkum T. 2006. Heterotachy and tree building: a case study with plastids and eubacteria. Mol Biol Evol. 23:40–45.

Lockhart PJ, Penny D, Hendy MD, Howe CJ, Beanland TJ, Larkum AWD. 1992. Controversy on chloroplast origins. FEBS Lett. 301:127–131.

Lockhart PJ, Steel MA, Hendy MD, Penny D. 1994. Recovering evolutionary trees under a more realistic model of sequence evolution. Mol Biol Evol. 11:605–612.

Magallon SA, Sanderson MJ. 2005. Angiosperm divergence times: the effect of genes, codon positions, and time constraints. Evolution. 59:1653–1670.

Martin W, Somerville CC, Loiseauxdegoer S. 1992. Molecular phylogenies of plastid origins and algal evolution. J Mol Evol. 35:385–404.

Martin W, Stoebe B, Goremykin V, Hansmann S, Hasegawa M, Kowallik KV. 1998. Gene transfer to the nucleus and the evolution of chloroplasts. Nature. 393:162–165.

Mossel E, Steel MA. 2004. A phase transition for a random cluster model on phylogenetic trees. Math Biosci. 187:189–203.

Mossel E, Steel M. 2005. How much can evolved characters tell us about the tree that generated them?. In: Gascuel O, editor. Mathematics of evolution and phylogeny. Oxford: Oxford University Press. p. 384–412.

Nannya Y, Sanada M, Nakazaki K, et al. (11 co-authors). 2005. A robust algorithm for copy number detection using high-density oligonucleotide single nucleotide polymorphism genotyping arrays. Cancer Res. 65:6071–6079..

Nei M. 1996. Phylogenetic analysis in molecular evolutionary genetics. Annu Rev Genet. 30:371–403.

Penny D. 1982. Towards a basis for classification: the incompleteness of distance measures, incompatibility analysis and phenetic classification. J Theor Biol. 96:129–142.

Penny D, Hendy MD. 1987. Turbotree—a fast algorithm for minimal trees. Comput Appl Biosci. 3:183–187.

Penny D, McComish BJ, Charleston MA, Hendy MD. 2001. Mathematical elegance with biochemical realism: the covarion model of molecular evolution. J Mol Evol. 53:711–723.

Penny D, Watson EE, Hickson RE, Lockhart PJ. 1993. Some recent progress with methods for evolutionary trees. N Z J Bot. 31:275–288.

Phillips MJ, Delsuc F, Penny D. 2004. Genome-scale phylogeny: sampling and systematic errors are both important. Mol Biol Evol. 21:1455–1458.

Posada D, Crandall KA. 1998. Modeltest: testing the model of DNA substitution. Bioinformatics. 14:817–818.

Rambaut A, Grassly NC. 1997. Seq-Gen: an application for the Monte Carlo simulation of DNA sequence evolution along phylogenetic trees. Comput Appl Biosci. 13:235–238.

Reeves JH. 1992. Heterogeneity in the substitution process of amino acid sites of proteins coded for by mitochondrial DNA. J Mol Evol. 35:17–31.

Rosenberg NA. 2002. The probability of topological concordance of gene trees and species trees. Theor Pop Biol. 61:225–247.

Rzhetsky A, Nei M. 1992. A simple method for estimating and testing minimum-evolution trees. Mol Biol Evol. 9:945–967.

Steel MA, Penny D. 1993. Distributions of tree comparison metrics - some new results. Syst. Biol. 42:126–141.

Steel MA, Székely L, Hendy MD. 1994. Reconstructing trees when sequence sites evolve at variable rates. J Comput Biol. 1:153–163.

Strimmer K, von Haeseler A. 1997. Likelihood-mapping: a simple method to visualize phylogenetic content of a sequence alignment. Proc Natl Acad Sci USA. 94:6815–6819.

Swofford DL. 2001. PAUP* phylogenetic analysis using parsimony (*and other methods). Version 4.0b8. Sunderland (MA): Sinauer Associates.

Tajima F. 1993. Unbiased estimation of evolutionary distance between nucleotide sequences. Mol Biol Evol. 10:677–688.

Thompson JD, Higgins DG, Gibson TJ. 1994. Clustal-W—improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. Nucleic Acids Res. 22:4673–4680.

Vinh S, von Haeseler A. 2004. IQPNNI: moving fast through tree space and stopping in time. Mol Biol Evol. 21:1565–1571.

Vogl C, Badger J, Kearney P, Li M, Glegg M, Jiang T. 2003. Probabilistic analysis indicates discordant gene trees in chloroplast evolution. J Mol Evol. 56:330–340.

Waddell PJ. 2005. Measuring the fit of sequence data to phylogenetic model: allowing for missing data. Mol Biol Evol. 22:395–401.

Waddell PJ, Hendy MD. 1997. Using phylogenetic invariants to enhance spectral analysis of nucleotide sequence data. In: Information and Mathematical Sciences Reports, Series B (A. Swift, ed). Massey University, Palmerston North. [cited 2007 July 23]. [Internet]. http://awcmee.massey.ac.nz/people/mhendy/pdf/ProjectedHadamardTemp.pdf.

Waddell PJ, Penny D, Hendy MD, Arnold GC. 1994. The sampling distributions and covariance matrix of phylogenetic spectra. Mol Biol Evol. 11:630–642.

Zhaxybayeva O, Hamel L, Raymond J, Gogarten JP. 2004. Visualization of the phylogenetic content of five genomes using dekapentagonal maps. Genome Biol. 5:R20.

Jianzhi Zhang, Associate Editor

Accepted June 22, 2007

Chapter 5

A Bias in ML Estimates of Branch Lengths in the Presence of Multiple Signals

5.1 Introduction

This chapter presents the paper "A Bias in ML Estimates of Branch Lengths in the Presence of Multiple Signals", which was published in the journal *Molecular Biology and Evolution* in 2008.

All software associated with this paper is now freely available via Subversion from `http://mlbias.sourceforge.net`.

One of the primary modes of scientific inquiry is the questioning of assumptions. In the field of phylogenetics, one glaring assumption dominates: that life evolved on a single tree.[1] While there is much evidence that the evolution of life is broadly tree-like, there is also evidence that some aspects of some forms of life evolved according to other, more general processes.

The goal of this paper is twofold:

1. To gauge the robustness of traditional ML tree-building to the presence of

[1]This despite the fact that Darwin himself was not committed to a single tree as the representation of the relationships between living things. He preferred instead the term "descent with modification", which merely emphasises the continuity of all life and includes processes such as lateral gene transfer that, when considered at the level of individuals (or equivalently, genomes) are more general (Penny, 2011).

treelike "noise" in the data.

2. To advocate for the use of analysis methods, such as network methods, that can in principle reject the hypothesis of a single tree.

5.2 Testing Robustness

Earlier robustness analyses, such as Sullivan & Swofford (2001), generated datasets using a relatively general model, and then inferred trees using a method that assumed a more restricted class of models. Importantly, there remained commonalities between the generating model and inference model class: they both assumed a single tree topology. This made it possible to test robustness by measuring the proportion of 4-taxon tree topologies recovered correctly.

In our paper, we generated data on a mixture model *using trees with different topologies* and fed into a single-tree inference method. Consequently it may very well be asked: What are the common parameters shared by the generating model and the inference model class? Or more plainly: What are we hoping to get?

5.2.1 Does "the" internal edge of a mixture of two trees really exist?

The main difficulty surrounds our use of the term "the internal edge" in describing the mixture of trees A and B on p. 87: it is not clear what single parameter in the generating model the inferred internal edge length can be said to be estimating.

No difficulties arise if we assume that one of the trees in the mixture, called the "true tree", has a much larger proportion than the other, which we can call the "noise tree". If the proportion p_A of tree A is large in relation to the proportion $p_B = 1 - p_A$ of tree B, and if sufficiently many sites are available that sampling error is not a concern, then it is reasonable to expect that single-tree ML will infer A's topology, so the internal branch length inferred is an estimate of A's internal branch length (and vice versa when $p_B \gg p_A$). But when $p_A \approx p_B$, this interpretation is unsatisfying.

5.2.2 Shared parameter values

In order to sensibly discuss the behaviour of inferences made using more-restricted model classes on data generated under more-general models, we need to make precise the notion of when parameter values are "shared" by different models.

Ever-present real-valued parameters like transition-transversion ratio are a simple case: if all k components of a mixture model have equal values for such a parameter, then we can sensibly describe this collection of parameters $\theta_i, 1 \leq i \leq k$ as a single parameter θ of the overall mixture model, and attempt to infer it using a more specific, single-component model. The resulting estimate, $\hat{\theta}$, can be meaningfully compared with the θ_i, and statements can be made about the estimation procedure regarding the usual parameter-dependent statistical properties like convergence (or lack thereof) and bias.

On the other hand, the structure of an edge-weighted phylogenetic tree means that it is not immediately clear how, or even whether, the parameters describing one tree T can be matched up with the parameters describing another tree U. T and U may in general have different topologies, which potentially makes their respective sets of parameters *prima facie* structurally different and thus incommensurable.

5.2.3 The edges of a mixture model

One way to overcome this is to follow the lead of the Hadamard conjugation technique (Hendy & Penny, 1993), and embed the topology-dependent parameter space of a particular tree in a larger, topology-independent space. Recall that an edge in a tree (considered without its length) is defined by a split of taxa $X|Y$, and that there are 2^{n-1} distinct splits on n taxa. This enables us to represent the $2n-3$ edge-length parameters describing any edge-weighted, unrooted binary tree T on n taxa using a vector $\mathbf{s}_T$ of 2^{n-1} parameters indexed by split: the $2n-3$ elements corresponding to edges present in T are assigned the corresponding length, while the remaining $2^{n-1}-2n+3$ elements, which correspond to edges absent from T, are assigned the value 0.[2] Now that we have a set of parameters that is structurally identical across different tree topologies, we can safely say that two trees T and U

[2]This describes the situation for an unrooted binary tree; much the same procedure works for rooted and/or multifurcating trees.

share a parameter value whenever $s_{Ti} = s_{Ui}$ for some split i.

According to this formulation, the meaning of "the internal edge" of a mixture of trees is not well defined; but whenever all components in the mixture model contain some split $A|B$ as an edge, and this edge has identical length across all components, the meaning of "the edge splitting taxon set A from taxon set B" *is* well defined, regardless of how the topologies of the components may otherwise differ. All such shared edges can be regarded as edges *of the mixture model*, capable of being inferred using a single-tree inference method (at least in principle).

It follows that the four external edges of the trees A and B analysed in the paper *are* shared parameter values. Likewise, for the 5-taxon analysis, all five external edges, plus the edge separating taxa 4 and 5 from the rest in the mixture of trees A and B, are genuine shared parameter values. Figure 1B shows that ML estimation of the four external edges in the 4-taxon analysis is indeed biased upward as the mixture approaches an even balance between the two topologies. That such a simple (and, we propose, frequently occurring) effect as a mixture of two trees is enough to distort the results of single-tree ML analysis is a persuasive argument for the use of network methods like Spectronet (Huber *et al.*, 2002), which are inherently immune to such problems.

5.3 Later Developments

Since the publication of our paper, the problem of how to deal with datasets that contain signals from multiple trees has received attention from other researchers.

Instead of modelling arbitrary mixtures of trees, Heled & Drummond (2010) propose to infer a posterior distribution on species trees using the multispecies coalescent (Degnan & Rosenberg, 2009). This approach requires a multilocus dataset containing multiple individuals per species. The gene trees are not actually inferred individually: instead the entire product space of gene trees for all of the loci is integrated over using Bayesian MCMC. Because it explicitly allow for the convergence of different gene lineages, the approach of Heled & Drummond (2010) will not be misled by multiple tree signals in the data, provided that each locus is free from recombination. As such it presents a compelling alternative to network methods whenever an alignment can be confidently partitioned into recombination-free regions *a priori*.

The PhD thesis of Schliep (2009, chapter 2) has examined the problem of inferring mixture models of trees in more detail. Unlike the simple 1-parameter mixture model we used on p. 88, in which edge lengths were specified *a priori*, Schliep (2009) considers the case in which edge lengths for both trees are to be estimated in addition to the mixing proportions. Using the phangorn phylogenetic reconstruction software (Schliep, 2011), he finds that estimates of corresponding edge lengths on the component trees are strongly negatively correlated, leading to high sampling variance of these parameters and making their accurate recovery difficult. He notes that the topic of which phylogenetic mixture models are identifiable and which are not remains an active area of research.

5.4 Connection to Multinomial Modes

The development of the `GreedyModeFind` algorithm in chapter 6 is a direct consequence of the need to produce an artefact-free plot for Figure 1A of this paper (p. 88). The details are explained in that chapter. Suffice it to say that the sentence "Because PAUP* requires integer site weights for likelihood computation, the site probabilities were multiplied by 30,001 and rounded to the nearest integer", appearing near the bottom of p. 87, considerably understates the actual difficulty of this step.

LETTERS

A Bias in ML Estimates of Branch Lengths in the Presence of Multiple Signals

David Penny, W. T. White, Mike D. Hendy, and Matthew J. Phillips[1]

Allan Wilson Center for Molecular Ecology and Evolution, Massey University, P.O. Box 11222, Palmerston North, New Zealand

Sequence data often have competing signals that are detected by network programs or Lento plots. Such data can be formed by generating sequences on more than one tree, and combining the results, a mixture model. We report that with such mixture models, the estimates of edge (branch) lengths from maximum likelihood (ML) methods that assume a single tree are biased. Based on the observed number of competing signals in real data, such a bias of ML is expected to occur frequently. Because network methods can recover competing signals more accurately, there is a need for ML methods allowing a network. A fundamental problem is that mixture models can have more parameters than can be recovered from the data, so that some mixtures are not, in principle, identifiable. We recommend that network programs be incorporated into best practice analysis, along with ML and Bayesian trees.

Phylogeneticists have been active in finding regions of parameter space where methods for inferring evolutionary trees are, or are not, reliable. Classic simulation studies include Huelsenbeck and Hillis (1993); Gaut and Lewis (1995); and Lockhart et al. (1996). However, most theoretical and simulation work has been on a data set generated on a - single tree, though we have known for a long time that real sequences have more signals than can fit onto a single tree. This is shown by networks (Bandelt and Dress 1992; Holland et al. 2004; Huson and Bryant 2006), Lento plots (Lento et al. 1995), and the treeness triangle (White et al. 2007).

There has been considerable work on the effects of model violation/misspecification (e.g. Sullivan and Swofford 2001; Buckley 2002), but most work focuses on the Markov model itself and less on whether a single tree is insufficient to describe the data. For the present analysis, we consider a scientific model as consisting of 3 parts (Penny et al. 1992):

the structure of the model (a tree in our case),
a mechanism of nucleotide or amino acid change (a stochastic Markov model), and
the initial conditions (including the nucleotide composition at the root, rates of changes, and the distribution of rates across sites).

Here, we concentrate on cases where model misspecification is the inadequacy of the tree itself, rather than the Markov model.

There are biological examples where mixture models (see Kolaczkowski and Thornton 2004; Matsen and Steel 2007) are biologically reasonable and cases where they are approximating other biological mechanisms. Realistic examples include lineage sorting, hybridization, lateral gene transfer, and with questions of orthology/paralogy among members of a gene family. In other cases, mixture models can mimic other processes, such as similar changes in G + C content leading to convergence onto an incorrect tree (see Lockhart et al. 1992). Thus, we consider that the principles illustrated by mixture models are important in understanding phylogenetic errors.

To illustrate the principles, we generate data on 2 trees, combine the sequences in defined proportions, and test the ability of standard maximum likelihood (ML) to estimate the edge (branch) lengths. Data were generated under a symmetric 2-state Hadamard conjugation (Hendy and Penny 1993) using an Excel spreadsheet (http://awcmee.massey.ac.nz/downloads.htm). This gives exact frequencies of the observed patterns, so there are no sampling errors in either the data or the edge lengths generated from it. We effectively have infinitely long sequences and so concentrate on systematic (Phillips et al. 2004), rather than sampling, errors.

The first simulations used two 4-taxon trees A (1,2)(3,4) and B (1,3)(2,4). In the examples reported here the external branches had 0.2 changes per site and the internal 0.05; only one parameter (the internal edge) is being changed. The 11 data sets had from 100% to 0% of patterns from tree A, decreasing in steps of 10%, with the remaining data coming from tree B.

PAUP* (Swofford 2001) was used on all data sets to estimate the ML values for both trees and branch lengths (4 external and one internal). The ML values for trees are in figure 1*A* and the branch lengths in figure 1*B*. As expected, ML favors tree A or B, depending on the proportion of data from each tree, and has equal ML values when the data is mixed 50:50. It is interesting that the ML value for either tree changed little with up to 30% of sites coming from the alternative tree—even though the ML value for the minor tree was increasing. As expected, there was no change in the ML value of the third possible tree (1,4)(2,3), which was not used in generating the data. Because PAUP* requires integer site weights for likelihood computation, the site probabilities were multiplied by 30,001 and rounded to the nearest integer.

Figure 1*B* has the estimates of branch lengths. The 100% A and 100% B results are important controls, in that branch lengths are estimated correctly (0.2 for the external and 0.05 for the internal). However, with the mixtures, the internal edge is underestimated and the external overestimated. The effect on the external edges is straightforward; the values increase above 0.2, with a maximum overestimate of 6% in the case of the 50:50 mixture.

[1] Present address: Centre for Macroevolution and Macroecology, School of Botany and Zoology, Australian National University, Canberra 0200, ACT, Australia

Key words: maximum likehood estimation, mixture models, multiple signals.

E-mail: d.penny@massey.ac.nz.

Mol. Biol. Evol. 25(2):239–242. 2008
doi:10.1093/molbev/msm263
Advance Access publication November 28, 2007

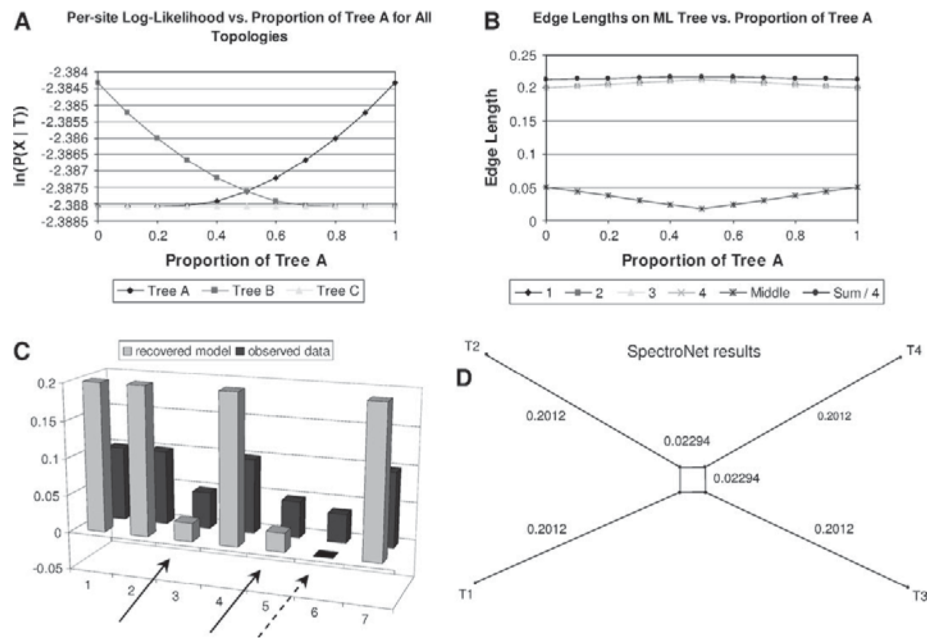


FIG. 1.—**A**, Average per-site ML values for 11 mixtures of the trees (1,2)(3,4) and (1,3)(2,4). The highest likelihoods are at the top. **B**, ML edge estimates for the same data as figure 1*A*. The external edges are overestimated and the internal edge underestimated. **C**, The Hadamard conjugation recovers the 2 signals from a 50:50 mixture of data from the 2 trees; there is a very small negative signal for the third tree. The back values are the observed frequencies of the 8 possible patterns in the data; the front values are after correcting for inferred multiple changes at a site. The 2 solid arrows indicate the signals for the 2 trees in the mixture; the dashed arrow is for the potential third tree that is not in the mixture. **D**, SpectroNet similarly recovers the 2 internal edges from the mixture and represents them as a network.

In this example, the effects are stronger on the internal branch, the estimate being over 30% too low, even considering just one of the 2 competing signals. The expected lengths of the 2 competing signals are both 0.025, and the estimate is $>$30% lower. However, if the estimate is expressed as a percentage of the sum of the 2 competing signals, we get a 66% underestimate; it is only 34% of the sum. Either way there is a severe underestimate of the length of the internal edge, which will be important when estimating dates of divergence.

There is additional information in the data that allows good recovery of both signals; the problem is forcing ML onto a single tree. The Hadamard (Hendy and Penny 1993), given data from a 50:50 mixture, recovers the 2 signals, each with a value of 0.025 (fig. 1*C*). The recovery is not exact, in that there is also a very small negative value ($-$0.000625) for tree C (1,4)(2,3). This exceeds the rounding errors from the computer storage of real numbers, which is $<10^{-16}$ in these calculations. Other network programs can recover signal for both trees as shown in figure 1*D* for SpectroNet (Huber et al. 2002) for the same 50:50 mixture.

The bias in ML is not inherent to ML but is a form of model violation from forcing the calculation onto a single tree. To show this, we tested a variant of ML with 10^6 sites from a mixture of the 2 trees. For each of the 11 data sets (fig. 1), each site was assigned to tree A or B with random starting probabilities *P* and $1 - P$; the trees and edge lengths were given a priori. The search uses a simple hill-climbing algorithm and tests new *P* values, accepting any which lead to a better log-likelihood score. The value of *P* that maximizes the log likelihood is selected. With just a single parameter to estimate, this procedure quickly converged to the correct *P* value, always resulting in a higher log-likelihood value than that produced by fitting either tree alone. Thus, there is additional information in the data that is not used by standard ML on a single tree and which can be used to help detect model violation.

We also did tests on 5-taxon trees. From figure 1*B*, we can imagine that the additional signal for the internal branch of tree B, which does not fit on tree A, might increase the lengths of edges 1 and 3, or 2 and 4, thus increasing their estimated lengths. To test this, we used 3 unrooted trees on 5 taxa: tree A (1,2),3,(4,5); tree B (1,3),2,(4,5), and tree C (1,4),3,(2,5). Again using the Hadamard, we calculated the data for each tree and made mixtures of trees A and B, and A and C. In the first case, only one internal edge was changed, and we expected that the upwards bias on the external edge to taxon 4 (in the 4-taxon case) was transferred to the internal edge that separates taxa 4 and 5 (in the 5-taxon case). In contrast, the second mixture (trees A and C, interchanging taxa 2 and 4) is expected to affect both internal edges. The expectation was that the first mixture (trees A and B) would be a more local effect and the second mixture (trees A and C) more global.

The results in figure 2 are for the 2 internal edges in the mixtures of trees A and B, and A and C. With the first mixture (A and B), there is the expected underestimate of the first internal edge, and the predicted upwards bias in the adjacent internal edge. When both internal edges are affected (in the mixture of trees A and C), then both internal

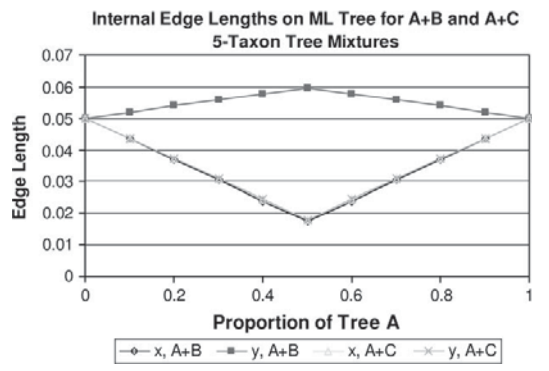


FIG. 2.—Edge lengths for the 2 internal edges x and y on mixtures of trees A and B, and A and C –(1,2),3,(4,5); (1,3),2,(4,5) and (1,4),2,(3,5) respectively. x is the internal edge leading to the taxon pair containing taxon 1; edge y is the internal edge leading to the taxon pair containing taxon 5.

edges are seriously underestimated, similar to figure 1*B*. Again, as in figure 1*C*, we used the Hadamard to recover the signals for both trees from both mixtures. For these examples, the results are accurate to the 5th decimal place. Thus, the failure of ML on a single tree to recover the signals more accurately is not a lack of signal in the data; rather it is model misspecification.

In addition to the model misspecification, there are the well-known problems associated with increased parameterization (see, Steel 2005). With mixture models, there can be more parameters than independent signals in the data. With a standard mixture model on 4 taxa (as in Matsen and Steel 2007), there are a minimum of 5 parameters for the edges of each tree, and one for the proportion of sites from each tree. Thus, there are 11 parameters for the model, which is more than the 7 independent values being estimated from the data—there are 8 patterns in the data for the 2-state symmetric model (Hendy and Penny 1993), but their frequencies must sum to 1.0. The numbers of patterns does increase rapidly with 4-state characters (Hendy et al. 1994), but this is not sufficient to guarantee identifiability. With mixture models, there is the question of identifiability (Chang 1996; Rannala 2002): whether there is in principle sufficient information in the data to uniquely identify the model. Note that in the Matsen and Steel (2007) example, the mixture of 2 sets of edge lengths on a single tree would be rejected by methods that weight against increasing the number of parameters. The mixture requires 6 additional parameters over the alternative single tree that generates the same data.

The problem is exacerbated by the recent tendency to increase the number of parameters being used in models—whether it be from variation in population size (Drummond et al. 2005), recombination (Martin et al. 2005), multiple ML optima (Chor et al. 2000), or different gene trees (Degnan and Rosenberg 2006, see also Stefankovic and Vigoda 2007). These recognize that additional parameters are leading to alternative models giving the same data, and so formal analysis on identifiability of models is urgently required (see Allman and Rhodes 2006 and Matsen and Steel 2007).

Although we expect that with real data, ML on single trees will be biased in estimating branch lengths, this should be kept in perspective. Researchers did not stop searching the space of trees when Graham and Foulds (1982) showed that the problem of evaluating all trees was NP-hard (and therefore cannot be computed exactly as the number of taxa increases). Rather, more care was taken over the search procedures in order to get the best results. We urgently need more work on ML and Bayesian methods for networks (see Strimmer and Moulton 2000; Pagel and Meade 2004). Similarly, we conclude that "best practice" in an analysis is to include a network diagram along with trees from ML and Bayesian analysis.

Literature Cited

Allman ES, Rhodes JA. 2006. The identifiability of tree topology for phylogenetic models, including covarion and mixture models. J Comp Biol. 13(5):1101–1113.
Bandelt H-J, Dress AWD. 1992. A canonical decomposition theory for metrics on a finite set. Adv Math. 92:47–105.
Buckley TR. 2002. Model misspecification and probabilistic tests of topology: evidence from empirical data sets. Syst Biol. 51:509–523.
Chang JT. 1996. Full reconstruction of Markov models on evolutionary trees: identifiability and consistency. Math Biosci. 137:51–73.
Chor B, Holland BR, Penny D, Hendy MD. 2000. Multiple maxima of likelihood in phylogenetic trees: an analytic approach. Mol Biol Evol. 17:1529–1541.
Degnan JH, Rosenberg NA. 2006. Discordance of species trees with their most likely gene trees. PLoS Genet. 2:e68.
Drummond AJ, Rambaut A, Shapiro B, Pybus OG. 2005. Bayesian coalescent inference of past population dynamics from molecular sequences. Mol Biol Evol. 22:1185–1192.
Gaut BS, Lewis PO. 1995. Success of maximum-likelihood phylogeny inference in the 4-taxon case. Mol Biol Evol. 12:152–162.
Graham RL, Foulds LR. 1982. Unlikelihood that minimal phylogenies for a realistic biological study can be constructed in reasonable computational time. Math Biosci. 60:133–142.
Hendy MD, Penny D. 1993. Spectral analysis of phylogenetic data. J Classif. 10:5–24.
Hendy MD, Penny D, Steel MA. 1994. Discrete Fourier spectral analysis of evolution. Proc Natl Acad Sci USA. 91:3339–3343.
Holland BR, Huber KT, Moulton V, Lockhart PJ. 2004. Using consensus networks to visualize contradictory evidence for species phylogeny. Mol Biol Evol. 23:848–855.
Huber KT, Langton M, Penny D, Moulton V, Hendy M. 2002. SpectroNet: a package for computing spectra and median networks. Appl Bioinformatics. 1:159–161.
Huelsenbeck JP, Hillis D, et al. 1993. Success of phylogenetic methods in the 4-taxon case. Syst Biol. 42:247–264.
Huson DH, Bryant D. 2006. Application of phylogenetic networks in evolutionary studies. Mol Biol Evol. 23:254–267.
Kolaczkowski B, Thornton JW. 2004. Performance of maximum parsimony and likelihood phylogenetics when evolution is heterogeneous. Nature. 431:980–984.
Lento GM, Hickson RE, Chambers GK, Penny D. 1995. Use of spectral analysis to test hypotheses on the evolutionary origin of the pinnipeds. Mol Biol Evol. 12:28–52.
Lockhart PJ, Larkum AWD, Steel MA, Waddell PJ, Penny D. 1996. Evolution of chlorophyll and bacteriochlorophyll: the problem of invariant sites in sequence analysis. Proc Natl Acad Sci USA. 93:1930–1934.
Lockhart PJ, Penny D, Hendy MD, Howe CJ, Beanland TJ, Larkum AWD. 1992. Controversy on chloroplast origins. FEBS Lett. 301:127–131.

Martin DP, Williamson C, Posada D. 2005. RDP2: recombination detection and analysis from sequence alignments. Bioinformatics. 21:260–262.

Matsen FA, Steel M. 2007. Phylogenetic mixtures on a single tree can mimic a tree of another topology. Syst Biol. 56:767–775.

Pagel M, Meade A. 2004. A phylogenetic mixture model for detecting pattern-heterogeneity in gene sequence or character-state data. Syst Biol. 53:571–581.

Penny D, Hendy MD, Steel MA. 1992. Progress with evolutionary trees. Trends Ecol Evol. 7:73–79.

Phillips MJ, Delsuc F, Penny D. 2004. Genome-scale phylogeny: sampling and systematic errors are both important. Mol Biol Evol. 21:1455–1458.

Rannala B. 2002. Identifiability of parameters in MCMC Bayesian inference in phylogeny. Syst Biol. 51:754–760.

Steel M. 2005. Should phylogenetic models be trying to fit an elephant? Trends Genet. 21:307–309.

Stefankovic D, Vigoda E. 2007. Pitfalls of heterogeneous processes for phylogenetic reconstruction. Syst Biol. 56:113–124.

Strimmer K, Moulton V. 2000. Likelihood analysis of phylogenetic networks using directed graphical models. Mol Biol Evol. 17:875–881.

Sullivan J, Swofford DL. 2001. Should we use model-based methods for phylogenetic inference when we know that assumptions about among-site rate variation and nucleotide substitution pattern are violated. Syst Biol. 50: 723–729.

Swofford DL. 2001. PAUP* Phylogenetic analysis using parsimony (*and other methods) ver 4.0b8. Sunderland (UK): Sinauer Associates.

White WT, Hill SF, Gaddam R, Holland BR, Penny D. 2007. Treeness triangles: visualizing the loss of phylogenetic signal. Mol Biol Evol. 24:2029–2039.

Martin Embley, Associate Editor.

Accepted November 19, 2007

Chapter 6

A Fast and Simple Algorithm for Finding the Modes of a Multinomial Distribution

6.1 Introduction

This chapter presents the paper "A Fast and Simple Algorithm for Finding the Modes of a Multinomial Distribution", which was published in the journal *Statistics and Probability Letters* in 2010.

All software associated with this paper is now freely available via Subversion from `http://greedymodefind.sourceforge.net`.

6.2 Motivation and Connection to Phylogenetics

It frequently happens that in the course of attempting to solve one problem, another problem appears. Usually the latter problem is smaller and more specific than the former. In contrast, this chapter follows the story of a small, specific, even *grubby* phylogenetic problem—surely no more than an afternoon's work!—that turned into a rather general problem, before finally succumbing, somewhat serendipitously, to a satisfying solution.

6.2.1 The Problem with PAUP*

The motivation for developing the `GreedyModeFind` algorithm described in this chapter came from the need to create Figure 1A on p. 88 in the previous chapter's paper. In this figure, inferred ML scores are shown for 11 datasets, each a mixture of two trees, with the proportion of tree A shown on the x axis. Because a 2-state symmetric i.i.d. model is used, the two trees in the mixture can each be described by a vector giving the probabilities of observing each of the $2^{4-1} = 8$ possible site patterns; each of the 11 points along the x axis represents a dataset whose site pattern probability vector is a weighted combination of the corresponding probabilities for these trees. Each of the three curves shows the ML scores calculated by imposing a different fixed topology on the inferred tree, and varying branch lengths so as to maximise the likelihood. The problem encountered in producing this figure was that the site pattern probabilities used to generate the 11 datasets are real numbers, but the program used to calculate the ML scores, PAUP* (Swofford, 2001), requires integer column weights.

The obvious expedient solution would be to multiply the site pattern frequencies by a large number and round them somehow. (Because sites are i.i.d., multiplying site pattern frequencies by a constant k amounts to raising the ML score to the power k, or equivalently to multiplying the log-likelihood score by k—a transformation that can be trivially undone.) The larger the multiplication factor, the more accurate the approximation. Experimenting with PAUP* suggested that it stores column weights as 16-bit signed integers, limiting this factor to 32767.

Unfortunately, this straightforward approach produces unimpressive results. Figure 6.1 shows an early version of Figure 1A computed by multiplying site pattern probabilities by 30000 and rounding down. (Note that the graph's vertical axis was later inverted and rescaled, and the colours changed.) The two "bump" artefacts distract noticeably from the main point the plot is trying to make: that ML inference correctly infers no trace of tree C.

The essence of the problem is that multiplying and then rounding the site pattern probabilities down (or up, or randomly) does not in general produce frequencies that sum to a fixed total, and no obvious method of getting around this suggests itself. This might not have been a practical problem, except that ML inference, when

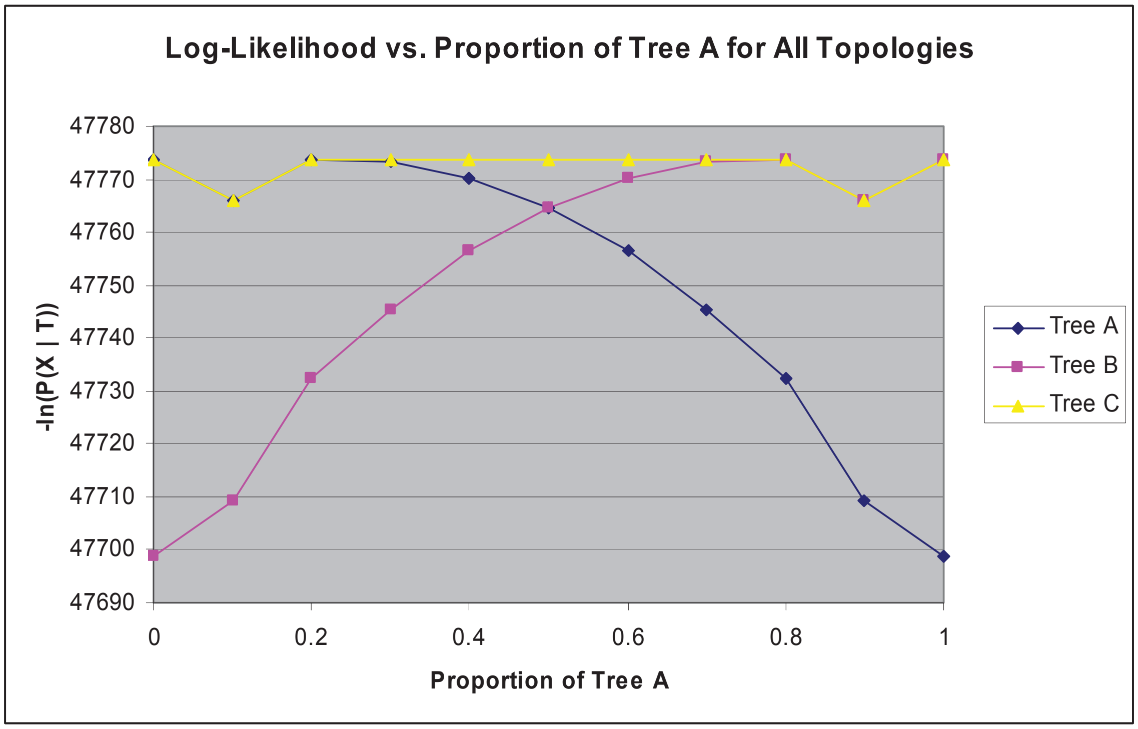


Figure 6.1: An early version of Figure 1A from chapter 5, showing unsightly bumps

constrained to integer site frequencies in this way, appears to be very sensitive to departures from exact ratios of site patterns.[1] In order to reduce the scale of these artefacts, it is necessary to consider how to apportion sites between site patterns in more detail. The task now becomes: Given a total number of sites n, r distinct site patterns, and a vector of site pattern probabilities $\mathbf{p} = (p_1, p_2, \ldots, p_r)$ that sum to 1, how to choose a vector of nonnegative integer site pattern frequencies $\mathbf{x} = (x_1, x_2, \ldots, x_r)$ such that they sum to n and $\mathbf{x}/n$ best approximates $\mathbf{p}$?

6.2.2 Choosing a Best Representative

This is actually a very general question. The first issue is to make the notion of "best approximates" more precise.

We start by considering the relative frequency vector $\hat{\mathbf{p}} = (x_1/n, x_2/n, \ldots, x_r/n)$ induced by any selection of a vector $\mathbf{x} = (x_1, x_2, \ldots, x_r)$ satisfying $\sum_{i=1}^{r} x_i = n$. In the specific application at hand (producing Figure 1A for the paper in chapter 5), we seek an $\mathbf{x}$ for some fixed n such that passing this vector to PAUP* produces a log-likelihood score that, after dividing by n, is minimally different from the true log-

[1]This may or may not be an issue that is specific to PAUP's ML inference algorithm; other ML programs were not tested.

likelihood score that would be computed using the exact vector $\mathbf{p}$. However, given the complicated numerical optimisation steps performed internally by PAUP*, this formulation is far too opaque to hope for a simple optimisation strategy.

Although PAUP* requires integer site weights, this is merely for ease of calculation. The likelihood function it computes is of course defined for fractional site weights, and it is reasonable to expect that a small change in the input site weight vector will elicit a small change in the likelihood score of the best model found. Thus we can approximately minimise the error in the log-likelihood score output by PAUP* by minimising the error in the site weight vector we supply as input to it.

The standard approach would be to minimise the error between $\hat{\mathbf{p}}$ and $\mathbf{p}$ in the least-squares sense: choose $\mathbf{x}$ so that $\sum_{i=1}^{r} (p_i - \hat{p}_i)^2$ is minimised. However the fact that all x_i are constrained to be nonnegative integers turns this problem into an integer programming problem with a quadratic cost function, rendering it highly nontrivial to solve.

6.2.3 An Alternative to Least Squares

Although least-squares appears doomed from a computational point of view, a different approach is available: set $\mathbf{x}$ to the vector that is most likely to be produced by collecting the outcomes of n independent trials, each found by choosing site pattern i with probability p_i. This $\mathbf{x}$ is the *mode* of the multinomial distribution having parameters $(n, r, \mathbf{p})$, and $\hat{\mathbf{p}} = \mathbf{x}/n$ approximates $\mathbf{p}$. But in what sense is this approximation justified?

Call a vector of probabilities $\mathbf{y}$ *representable using n individuals* if $n\mathbf{y} \in \mathbb{Z}^r$. Substituting $a = n$ in Theorem 1 of the paper shows that if $\mathbf{x}$ is a mode of $(n, r, \mathbf{p})$ then $\mathbf{x}$ is also a (in fact the unique) mode of the multinomial distribution $(n, r, \hat{\mathbf{p}})$, with $\hat{\mathbf{p}} = \mathbf{x}/n$. Thus, among all probability vectors representable using n individuals, $\hat{\mathbf{p}}$ is *nearest* to $\mathbf{p}$ in the sense that the multinomial distributions $(n, r, \mathbf{p})$ and $(n, r, \hat{\mathbf{p}})$ share a mode. In particular, when $\mathbf{p}$ is itself representable using n individuals—for example if $n = 5$ and $\mathbf{p} = (0.2, 0.2, 0.6)$—its unique mode $\mathbf{x}$ must be the vector $n\mathbf{p}$, giving $\hat{\mathbf{p}} = \mathbf{p}$ as desired.

I could find no satisfactory algorithm, and very little treatment overall, for computing the mode of a multinomial distribution in the literature. Given the fundamental importance of multinomial distributions in statistical applications, this came as a surprise. I resolved to attack the problem using a greedy heuristic approach that computed modes by incrementing elements in a frequency vector that initially contained all zeroes. Finding that these experiments yielded promising results, I began to investigate whether in fact this greedy algorithm could be proven to always give correct results. As detailed in the paper, I found that it could. This basic approach, already fast and numerically stable, was further improved by incorporating the starting point used by Finucan (1964) instead of starting from a zero vector.

6.3 Alternate Proof of Correctness

I am grateful to my coauthor Mike Hendy for suggesting a simpler proof of correctness of this algorithm. I had originally conceived a more elaborate proof of correctness that used induction to show that, at every step during construction of a mode vector, it is possible to make a choice that leads to an optimal solution, and that the algorithm always makes one of these choices. For completeness I give this original proof below.

First we establish some terminology. We call a mode vector $\mathbf{x}^*$ a *complete solution.* A *partial solution* is an r-vector whose elements sum to n or less. We say that a partial solution is *larger* or *smaller* than another if its element sum is larger or smaller, respectively. We call a partial solution $\mathbf{x}$ *admissible* if there exists a complete solution $\mathbf{x}^*$ such that $x_i^* \geq x_i \; \forall i$—in other words, $\mathbf{x}$ is admissible iff a complete solution can be produced by increasing some subset of its elements. Clearly, the r-vector consisting of all zeroes is always an admissible partial solution.

We must also consider the sequence of algorithmic steps involved in constructing partial and complete solutions. Abusing notation slightly, we will also use $\mathbf{x}^*$ to refer to a sequence of steps that creates the vector $\mathbf{x}^*$ by incrementing the elements of some admissible initial vector $\mathbf{x}_0$. Each step consists of choosing an element to increment. Note that for any k-step decision sequence D that builds an admissible partial solution vector $\mathbf{x}_k$ from an initial vector $\mathbf{x}_0$, there exists a decision sequence D^* that builds a complete solution from $\mathbf{x}_0$ and whose first k steps agree with D.

Our algorithm starts with an r-vector consisting of all zeroes and increments elements one at a time. After n iterations, a mode vector is produced. To prove this, it is sufficient to show that:

I. The initial partial solution is admissible.

II. Given an admissible partial solution after $i \geq 0$ steps, the $(i+1)^{\text{th}}$ greedy step will produce another admissible partial solution.

III. The greedy algorithm will eventually produce a partial solution whose elements sum to n.

Conditions (I) and (III) are trivial: clearly, the initial zero vector is admissible; and the sum of vector elements increases by 1 on each iteration, so the algorithm will terminate after n iterations. It remains to show that condition (II) is met. We will not take the usual approach of identifying separate Greedy Choice and Optimal Substructure requirements, instead we will present a direct inductive proof.

Assume that we have an admissible partial solution after m steps, $\mathbf{x}_m$, which can be extended to a complete solution $\mathbf{x}^*$. The greedy step will produce $\mathbf{x}_{m+1}$ by choosing the element i that maximises (6.1) below:

$$\frac{(m+1)!}{\left(\prod_{j=1}^{r} x_j!\right)(x_i+1)} \cdot \prod_{j=1}^{r} p_j^{x_j} p_i = \underbrace{\frac{(m+1)!}{\left(\prod_{j=1}^{r} x_j!\right)} \cdot \prod_{j=1}^{r} p_j^{x_j}}_{A} \cdot \underbrace{\frac{p_i}{x_i+1}}_{B} \tag{6.1}$$

A is invariant w.r.t. i and always > 0, so the i that maximises B also maximises (6.1). B will be referred to as the *score* of element i, denoted q_i.

If the optimal solution $\mathbf{x}^*$ also chooses element i at step $m+1$, then clearly $\mathbf{x}_{m+1}$ is still admissible. Otherwise, $\mathbf{x}^*$ chooses a different element $c \neq i$ at this step, and two cases arise:

a) The optimal solution $\mathbf{x}^*$ chooses element i at some later step $s > m+1$. In this case, the choices made by the optimal solution at steps s and $m+1$ can be swapped to produce a new complete solution $\mathbf{x}'$ with the same (maximal) probability, in which element i is chosen at step $m+1$. $\mathbf{x}_{m+1}$ makes the same choice as $\mathbf{x}'$ at step $m+1$, and thus is still admissible.

b) The optimal solution $\mathbf{x}^*$ does not choose element i at any later step. The remaining steps in the proof concern this case.

The optimal solution $\mathbf{x}^*$ chooses element c at step $m+1$ instead of element i. Suppose that from step $m+1$ onwards, a total of n_c choices of element c are made by $\mathbf{x}^*$. Then their contribution to the final probability product is given by (6.2) below:

$$\prod_{j=1}^{n_c} \left(\frac{m+j}{x_c+j} \cdot p_c \right) \tag{6.2}$$

If one of these choices of c was replaced by a choice of i, the expression in (6.2) (and therefore the final probability product) would be multiplied by (6.3):

$$\frac{1}{\frac{m+n_c}{x_c+n_c} \cdot p_c} \cdot \frac{m+n_c}{x_i+1} \cdot p_i = \frac{p_i\,(x_c+n_c)}{p_c\,(x_i+1)} \tag{6.3}$$

But the greedy choice of element i at step $m+1$ implies $\frac{p_i}{x_i+1} \geq \frac{p_j}{x_j+1}\ \forall j$. In particular,

$$\begin{aligned} & \frac{p_i}{x_i+1} \geq \frac{p_c}{x_c+1} \\ \Longrightarrow & \frac{p_i\,(x_c+1)}{p_c\,(x_i+1)} \geq 1 \\ \Longrightarrow & \frac{p_i\,(x_c+n_c)}{p_c\,(x_i+1)} \geq 1 \end{aligned} \tag{6.4}$$

Thus we can replace the choice of c at step $m+1$ in the optimal solution $\mathbf{x}^*$ with a choice of i, producing a new complete solution $\mathbf{x}''$ whose probability is not less than that of $\mathbf{x}^*$ and which chooses element i at step $m+1$. $\mathbf{x}_{m+1}$ makes the same choice as $\mathbf{x}''$ at this step, and thus is still admissible. Note that the "$\geq$" sign in (6.4) above is effectively an "$=$" sign, since for the LHS to be greater than 1 would contradict the optimality of $\mathbf{x}^*$.

6.4 Problem Instances

Ideally we would have tested `GreedyModeFind`'s performance on the problem instances used by Le Gall (2003). However we were not able to obtain them: the paper itself does not detail them, nor does it indicate where they might be found, and an email request to the author went unanswered. Nevertheless, given the reliable running time of `GreedyModeFind`, it can be safely assumed that these problem instances would not have produced very different results.

Statistics and Probability Letters 80 (2010) 63–68

Contents lists available at ScienceDirect

Statistics and Probability Letters

journal homepage: www.elsevier.com/locate/stapro

A fast and simple algorithm for finding the modes of a multinomial distribution

W.T.J. White [*], M.D. Hendy [**]

Allan Wilson Centre for Molecular Ecology and Evolution, Massey University, Private Bag 11-222, Palmerston North 4442, New Zealand

ARTICLE INFO

Article history:
Received 15 May 2009
Accepted 20 September 2009
Available online 4 October 2009

ABSTRACT

Suppose a trial has r possible outcomes, with the i-th outcome having probability p_i, $\sum_{i=1}^{r} p_i = 1$ and $\mathbf{p} = (p_1, \cdots, p_r)$. The outcome of n independent trials can be described by the frequencies $\mathbf{k} = (k_1, \ldots, k_r)$, $\sum_{i=1}^{r} k_i = n$, where outcome i was selected with frequency $k_i \in \mathbb{N}$. A mode (there can be multiple modes) is an outcome with maximal likelihood over all possible outcomes.

Despite the ubiquity of multinomial distributions in statistical applications, the best algorithm to date for finding modes has unknown computational complexity and highly variable performance in practice. It is also vulnerable to precision problems due to accumulated roundoff error.

We propose a new algorithm, `GreedyModeFind`, for calculating the mode(s) of this distribution, given n and $\mathbf{p}$. `GreedyModeFind` is simple, efficient and numerically robust, requiring $O(r \log r)$ time and $O(r)$ space to find one mode. A concise representation of the full set of joint modes can be found for an additional time cost of $O(r \log r)$. In practice this algorithm drastically improves on the performance of earlier algorithms. We provide a freely available C++ implementation of `GreedyModeFind`.

1. Background

Suppose with each of n independent trials, a selection process chooses one of r possible outcomes with probabilities

$$\mathbf{p} = (p_1, \ldots, p_r), \quad \left(\sum_{i=1}^{r} p_i = 1 \right).$$

The outcome of the n trials can be expressed as a vector of r non-negative integers $\mathbf{k} = (k_1, \ldots, k_r)$, where k_i is the frequency of the i-th outcome and $\sum_{i=1}^{r} k_i = n$. Let

$$\mathbf{K}_{n,r} = \left\{ (k_1, \ldots, k_r) \in \mathbb{N}^r \mid \sum_{i=1}^{r} k_i = n \right\}$$

be the set of all possible outcomes.

The probability of obtaining the outcome $\mathbf{k} \in \mathbf{K}_{n,r}$ is

$$P(\mathbf{k}) = \binom{n}{k_1, \ldots, k_r} p_1^{k_1} \cdots p_r^{k_r} = n! \prod_{i=1}^{r} \frac{p_i^{k_i}}{k_i!}. \tag{1}$$

[*] Primary corresponding author. Tel.: +64 21 0761691.
[**] Corresponding author.
E-mail addresses: w.t.white@massey.ac.nz (W.T.J. White), m.hendy@massey.ac.nz (M.D. Hendy).

0167-7152/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.spl.2009.09.013

A vector $\mathbf{y} \in \mathbf{K}_{n,r}$ is a *mode* of the joint distribution if $P(\mathbf{y})$ is maximal among all vectors in $\mathbf{K}_{n,r}$ (there can be multiple ("joint") modes). We were motivated to search for multinomial modes in order to construct simulated DNA sequences having nucleotide frequencies as similar as possible to reference sequences for a bioinformatics application; another application is to particle non-linear filtering (described in Le Gall (2003)).

The earliest algorithm for computing a mode of a multinomial distribution was suggested as Problem 28 in Feller (1957). This amounted to enumerating all possible $\mathbf{k} \in \mathbf{K}_{n,r}$ within bounds attributed in Finucan (1964) to Moran (personal communication), which restrict the number of candidate solutions. However these bounds are effective only for small r, prompting the development by Finucan of an iterative method (Finucan, 1964) that starts from an initial $\mathbf{k} \in \mathbf{K}_{n,r}$ and proceeds towards a mode (or modes) by incrementing or decrementing one frequency vector element at a time. This work also showed that if multiple modes exist, then they occur as a single "cluster".

Later Le Gall (2003) proposed a variation of Finucan (1964) that reduced memory requirements by eliminating the calculation of one floating-point vector, and made explicit the bookkeeping steps involved in selecting the next element to adjust. This work also gave an empirically obtained running time bound, as well as heuristics for rapidly solving perturbations of a given problem.

Our `GreedyModeFind` is a modification of Le Gall's algorithm where modes of $\mathbf{p}$ in $K_{n,r}$ are found for successive values of n by a greedy algorithm. It addresses several weaknesses of the earlier algorithms:

- Both Finucan's (1964) and Le Gall's (2003) algorithms accumulate terms across $O(r)$ loop iterations, leading to possible loss of numerical precision on large instances. In contrast, the longest chain of inexact arithmetic calculations in `GreedyModeFind` is fixed at two, since all needed values are computed "afresh" on each loop cycle.
- `GreedyModeFind` incorporates a straightforward bookkeeping optimisation that dramatically speeds up the determination of the next frequency vector element to adjust.
- `GreedyModeFind` is the first algorithm of its kind for which tight bounds on computational complexity have been established.

2. Preliminaries

It is useful to introduce some constructions where some components of $\mathbf{x}$ are changed to adjacent integers. Given $\mathbf{x} = (\mathbf{x_1}, \dots, \mathbf{x_r}) \in \mathbf{K}_{n,r}$ and distinct $i, j \in \{1, \dots, r\}$, let

$$\mathbf{x}^{(i)} = (x_1, \dots, x_i + 1, \dots, x_r), \qquad \mathbf{x}_{(j)} = (x_1, \dots, x_j - 1, \dots, x_r),$$

so $\mathbf{x}^{(i)} \in \mathbf{K}_{n+1,r}$, $\mathbf{x}_{(j)} \in \mathbf{K}_{n-1,r}$, and $\mathbf{x}^{(i)}_{(j)} \in \mathbf{K}_{n,r}$. The probabilities of these are easily compared to $P(\mathbf{x})$. We see from (1) that

$$P(\mathbf{x}^{(i)}) = (n+1)\frac{p_i}{x_i+1}P(\mathbf{x}), \qquad P(\mathbf{x}_{(j)}) = \frac{1}{n}\frac{x_j}{p_j}P(\mathbf{x}), \qquad P\left(\mathbf{x}^{(i)}_{(j)}\right) = \frac{p_i}{x_i+1} \cdot \frac{x_j}{p_j}P(\mathbf{x}). \tag{2}$$

For some values m there is a unique and easily determined mode of $\mathbf{p}$ in $\mathbf{K}_{m,r}$. For any positive real number a, let $w_i = \lfloor ap_i \rfloor$ for $i = 1, \dots, r$;

$$\mathbf{w}(=\mathbf{w}(a)) = (w_1, \dots, w_r); \quad \text{and} \quad m(=m(a)) = \sum_{i=1}^{r} w_i. \tag{3}$$

Theorem 1. *$\mathbf{w}$ is the unique mode of $\mathbf{p}$ in $\mathbf{K}_{m,r}$.*

Proof. Consider any $\mathbf{z} \in \mathbf{K}_{m,r}$. If $\mathbf{z} \neq \mathbf{w}$, then for some $i, j \in \{1, \dots, r\}$, $z_i < w_i$ and $z_j > w_j$, so as they are integers,

$$z_i < w_i \Rightarrow z_i + 1 \le w_i = \lfloor ap_i \rfloor \le ap_i$$

and

$$z_j > w_j \Rightarrow z_j \ge w_j + 1 = 1 + \lfloor ap_j \rfloor > ap_j.$$

Hence

$$P\left(\mathbf{z}^{(i)}_{(j)}\right) = P(\mathbf{z}) \cdot \frac{p_i}{z_i+1} \cdot \frac{z_j}{p_j} > P(\mathbf{z}) \cdot \frac{p_i}{ap_i} \cdot \frac{ap_j}{p_j} = P(\mathbf{z}).$$

Hence $\mathbf{z}$ cannot be maximal, so $\mathbf{w}$ is the unique mode of $\mathbf{p}$ in $\mathbf{K}_{m,r}$. □

We note that Le Gall (2003) proves the weaker claim that $\mathbf{w}$ is *a* mode using her Proposition 1; in fact, uniqueness is necessary for the correctness of both her algorithm and ours when considering joint modes.

Now if we are given a mode $\mathbf{w}$ of $\mathbf{p}$ in $K_{n,r}$ we can construct modes for $\mathbf{p}$ in $K_{n\pm1,r}$ by a greedy step. By Eq. (2) the maximal values among $\{P(\mathbf{w}^{(i)})\}$ occur when $\frac{p_i}{w_i+1}$ is maximal, and the maximal values among $\{P(\mathbf{w}_{(j)})\}$ occur when $\frac{w_j}{p_j}$ is maximal.

Theorem 2. *If $\mathbf{x}$ is a mode for $\mathbf{p}$ in $\mathbf{K}_{n,r}$, then $\mathbf{x}^{(k)}$ is a mode for $\mathbf{p}$ in $\mathbf{K}_{n+1,r}$, where $k \in \{1, \dots, r\}$ maximises $\frac{p_i}{1+x_i}$.*

Proof. Let $\mathbf{y}$ be a mode for $\mathbf{p}$ in $\mathbf{K}_{n+1,r}$; then $\forall i \in \{1, \dots, r\}$,

$$P(\mathbf{y}_{(i)}) = \frac{1}{n+1}\frac{y_i}{p_i}P(\mathbf{y}) \le P(\mathbf{x}),$$

and

$$P(\mathbf{x}^{(i)}) = (n+1)\frac{p_i}{1+x_i}P(\mathbf{x}) \le P(\mathbf{y}). \tag{4}$$

Thus

$$y_i \le 1 + x_i, \quad \forall i \in \{1, \dots, r\}. \tag{5}$$

As x_i, y_i are integers with

$$\sum_{i=1}^{r} y_i = n + 1 = 1 + \sum_{i=1}^{r} x_i,$$

Eq. (5) implies $y_k = 1 + x_k$ for exactly one value $k \in \{1, \dots, r\}$ and $y_j = x_j$ for each of the remaining $r - 1$ values. From Eq. (4), $P(\mathbf{x}^{(k)})$ is maximal when $\frac{p_k}{1+x_k}$ is maximal. □

However it is possible that $\frac{p_k}{1+x_k}$ may be maximal for more than one k, in which case each $\mathbf{x}^{(k)}$ is a mode for $\mathbf{p}$ in $\mathbf{K}_{n+1,r}$.

Theorem 3. *If* $\mathbf{x}$ *is a mode for* $\mathbf{p}$ *in* $\mathbf{K}_{n,r}$*, then* $\mathbf{x}_{(k)}$ *is a mode for* $\mathbf{p}$ *in* $\mathbf{K}_{n-1,r}$*, where* $k \in \{1, \dots, r\}$ *maximises* $\frac{x_i}{p_i}$.

Proof. Let $\mathbf{z}$ be a mode for $\mathbf{p}$ in $\mathbf{K}_{n-1,r}$; then $\forall i \in \{1, \dots, r\}$,

$$P(\mathbf{z}^{(i)}) = n\frac{p_i}{z_i+1}P(\mathbf{z}) \le P(\mathbf{x}),$$

and

$$P(\mathbf{x}_{(i)}) = \frac{1}{n}\frac{x_i}{p_i}P(\mathbf{x}) \le P(\mathbf{z}). \tag{6}$$

Thus

$$x_i \le 1 + z_i, \quad \forall i \in \{1, \dots, r\}. \tag{7}$$

As x_i, z_i are integers with

$$1 + \sum_{i=1}^{r} z_i = n = \sum_{i=1}^{r} x_i,$$

Eq. (7) implies $1 + z_k = x_k$ for exactly one value $k \in \{1, \dots, r\}$ and $z_j = x_j$ for each of the remaining $r - 1$ values. From Eq. (6), $P(\mathbf{x}_{(k)})$ is maximal when $\frac{x_k}{p_k}$ is maximal. However it is possible that $\frac{x_k}{p_k}$ may be maximal for more than one k, in which case each $\mathbf{x}_{(k)}$ is a mode for $\mathbf{p}$ in $\mathbf{K}_{n-1,r}$. □

3. Finding a mode

Faster bookkeeping with heaps. From a practical perspective, the primary shortcoming of Le Gall's algorithm is the unpredictable performance across problem instances (see Le Gall, 2003, Table 3 and Fig. 1(a)). This is due to Le Gall's procedure for choosing the next frequency vector element to adjust by maintaining a list of r pairs (i, q_i) sorted by q value. Choosing the next element to adjust is simple — it is always the element indexed by the i value of the first entry in the list. However, once that element has been adjusted, the entry's q value is increased and it must be repositioned in the list so as to maintain sorted order — a laborious process that may involve scanning and copying every entry in the list, implying $O(r)$ time per loop iteration. The key to speeding up Le Gall's algorithm is to replace the sorted list with a heap data structure. A heap containing r elements can be built in $O(r)$ time; the minimum element is available in $O(1)$ time; removal of the minimum element and insertion of an arbitrary new element both take $O(\log r)$ amortised time. Since each loop iteration requires extraction of the pair (i, q_i) having minimum q, removal of this element, and reinsertion of the element with updated q value, the size of the heap remains constant at r and the total amortised time per iteration is just $O(\log r)$. We use this heap-based approach in our own algorithm, though we note that it can also be directly applied to Le Gall's algorithm.

Choosing a starting point. The following bounds, attributed in Finucan (1964) to Moran (citation not provided), constrain the elements of every mode vector $\mathbf{x}^* = (x_1^*, \dots, x_r^*)$ for $\mathbf{p}$ in $\mathbf{K}_{n,r}$:

$$np_i - 1 \le x_i^* \le (n + r - 1)p_i. \tag{8}$$

Finucan concludes that a useful starting point for his iterative search is initially setting each frequency vector element to $(n + r/2)p_i$, the approximate mid-point of this range. `GreedyModeFind` intialises the search with $x_i = \lfloor (n + r/2)p_i \rfloor$ which by Theorem 1 guarantees that the resulting vector is a unique mode of $\mathbf{p}$ in $\mathbf{K}_{m,r}$ for some $m \leq n$. This will simplify the handling of multiple modes later.

Theorems 2 and 3 suggest the following simple algorithm for finding a mode for $\mathbf{p}$ in $\mathbf{K}_{n,r}$:

```
Function GreedyModeFind1(p[n]):
    m = 0
    For i from 1 to r:
        x[i] = floor(p[i] * (n + r / 2))
        m = m + x[i]

    If m <> n:
        For i from 1 to r:
            h[i].elem = i
            If m < n:
                h[i].score = p[i] / (x[i] + 1)
            Else (m > n):
                h[i].score = x[i] / p[i]

        HeapBuild(h)

        While m <> n:
            best = HeapRemoveMax(h)
            If m < n:
                x[best.elem] = x[best.elem] + 1
                best.score = p[best.elem] / (x[best.elem] + 1)
                m = m + 1
            Else (m > n):
                x[best.elem] = x[best.elem] - 1
                best.score = x[best.elem] / p[best.elem]
                m = m - 1

            HeapPush(h, best)

    Return x
```

The first loop runs r times with constant time per iteration. If we are lucky, we land directly on the desired value of m and stop there. Otherwise, the next loop also runs r times with constant time per iteration. Building the heap requires $O(r)$ time. To calculate the maximum number of iterations of the main while-loop, observe that

$$\sum_{i=1}^{r} \left(n + \frac{r}{2}\right) p_i = n + \frac{r}{2} = \sum_{i=1}^{r} \left\lfloor \left(n + \frac{r}{2}\right) p_i \right\rfloor + \sum_{i=1}^{r} d_i$$

where $0 \leq d_i < 1,\ \forall i$. This implies that $\sum_{i=1}^{r} d_i < r$, so

$$n - \frac{r}{2} < m = \sum_{i=1}^{r} \left\lfloor \left(n + \frac{r}{2}\right) p_i \right\rfloor \leq n + \frac{r}{2}.$$

The main while-loop adjusts m by 1 each iteration until $m = n$, so this loop must execute at most $\frac{r}{2}$ times. Each iteration requires a heap removal and heap insertion, both $\log(r)$ operations. Thus the time complexity is $O(r \log r)$. $O(r)$ space is required for the heap.

4. Multiple modes

Theorems 2 and 3 show that a mode can be constructed from a mode of a neighbouring distribution. The following two theorems establish that *all* modes can be constructed this way:

Theorem 4. *If $\mathbf{x}$ is a mode for $\mathbf{p}$ in $\mathbf{K}_{n,r}$, then there exist $\mathbf{y}$ and k s.t. $\mathbf{y}$ is a mode for $\mathbf{p}$ in $\mathbf{K}_{n-1,r}$ and $\mathbf{y}^{(k)} = \mathbf{x}$. In other words, there is no mode for $\mathbf{p}$ in $\mathbf{K}_{n,r}$ that cannot be constructed from some mode for $\mathbf{p}$ in $\mathbf{K}_{n-1,r}$ using Theorem 2.*

Proof. By Theorem 3, we can construct a mode $\mathbf{u}$ for $\mathbf{p}$ in $\mathbf{K}_{n-1,r}$ from $\mathbf{x}$. $\mathbf{u} = \mathbf{x}_{(i)}$ for some $i \in \{1, \ldots, r\}$. Clearly $\mathbf{u}^{(i)} = (\mathbf{x}_{(i)})^{(i)} = \mathbf{x}$, so set $\mathbf{y} = \mathbf{u}$ and $k = i$. $\square$

Theorem 5. *If $\mathbf{x}$ is a mode for $\mathbf{p}$ in $\mathbf{K}_{n,r}$, then there exist $\mathbf{y}$ and k s.t. $\mathbf{y}$ is a mode for $\mathbf{p}$ in $\mathbf{K}_{n+1,r}$ and $\mathbf{y}_{(k)} = \mathbf{x}$. In other words, there is no mode for $\mathbf{p}$ in $\mathbf{K}_{n,r}$ that cannot be constructed from some mode for $\mathbf{p}$ in $\mathbf{K}_{n+1,r}$ using Theorem 3.*

Proof. By Theorem 2, we can construct a mode $\mathbf{v}$ for $\mathbf{p}$ in $\mathbf{K}_{n+1,r}$ from $\mathbf{x}$. $\mathbf{v} = \mathbf{x}^{(j)}$ for some $j \in \{1, \ldots, r\}$. Clearly $\mathbf{v}_{(j)} = (\mathbf{x}^{(j)})_{(j)} = \mathbf{x}$, so set $\mathbf{y} = \mathbf{v}$ and $k = j$. □

We now consider the problem of recovering all modes for $\mathbf{p}$ in $\mathbf{K}_{n,r}$ starting from a mode for $\mathbf{p}$ in $\mathbf{K}_{m,r}$. We will assume for simplicity that $m \leq n$ (a symmetrical argument holds for $m \geq n$).

Suppose that for some $m \leq n$, $\mathbf{u}$ is the unique mode for $\mathbf{p}$ in $\mathbf{K}_{m,r}$. (Theorem 1 guarantees the existence of m and $\mathbf{u}$.) It then follows from Theorems 2 and 4 that any mode for $\mathbf{p}$ in $\mathbf{K}_{n,r}$ can be obtained by starting from $\mathbf{u}$ and performing some sequence of $n - m$ steps, where each step increases a maximal-score vector element by 1 and then adjusts its score appropriately. (If $m = n$ then we have the unique mode already.) Uniqueness of $\mathbf{u}$ is necessary to guarantee that all modes in $\mathbf{K}_{n,r}$ can be reached in this manner.

Evidently there is a connection between multiple modes and multiple vector elements having maximal scores. We now analyze this more carefully.

The `GreedyModeFind1` algorithm leaves open the question of what to do when more than one of the vector elements have equal scores at a given step. Since selecting one element to increment always decreases its score but has no effect on the scores of other elements, the other maximum-score elements will remain the only maximum-score elements at the next step. Consequently, if after step s there are k maximum-score elements to choose from, then steps $s + 1$ through $s + k$ will choose these k elements in some order, and that order is immaterial to the vector produced after step $s + k$. We call such a sequence of k choices of k maximum-score elements in some order a *resolved ambiguous segment*. However, the order of choices is immaterial only when at least k decisions remain to be made, i.e. when $m + s + k \leq n$. If fewer decisions remain to be made, say $g = n - m - s$, the choice of which g elements to select does impact the final vector. Theorems 2 and 3 support the choice of any of the k maximum-score elements at each of the g steps, so there are $\binom{k}{g}$ distinct solutions. Note that the particular mode initially arrived at will depend on the input order and particulars of the heap implementation.

To modify `GreedyModeFind1` to handle multiple modes, we compute g by counting the number of equal-score elements chosen during the final steps of the algorithm (i.e. steps $n - m - g + 1, \ldots, n - m$). Call the score shared by these elements q. We then continue to extract maximal-score entries from the heap until the score drops below q. Suppose b elements with score q are extracted: then $g + b = k$, the total number of equally good elements that could have been chosen in the final g steps. By recording the sequence of element choices made, the last g choices made by the algorithm can be "undone" to produce a "base vector" $\mathbf{x}_{\text{base}}$ from which all modes can be generated in a straightforward fashion using recursion.

As an efficient alternative to explicitly enumerating all modes, the vector $\mathbf{x}_{\text{base}}$ can be returned along with the value of g and a list of the k equally good elements. Because no element can appear twice in the same run of equal-score element choices, at most r elements can appear in such a sequence, meaning that a ring buffer of size $\min(n, r)$ suffices to hold the history of choices. This enables the algorithm to retain $O(r)$ space complexity when this more succinct return value representation is used. We call this version of the algorithm `GreedyModeFind`.

Approximate equality concerns. The handling of multiple modes requires tests for equality between floating-point numbers. Because such tests are susceptible to numerical error, in practice some notion of "approximate equality" between floating-point numbers must be used. This is not an ideal situation; however we do no worse than the algorithms of Finucan and Le Gall in this respect. We now analyse one criterion for approximate equality, leading to a bound on the relative error, which we here define as the ratio of the most-probable to the least-probable frequency vectors that could be reported as joint modes. Rearranging allows a suitable tolerance parameter value to be determined easily from a specified maximum allowed relative error.

One natural criterion for approximate equality is a bound on relative magnitude. Call two floating-point numbers x and y approximately equal iff

$$\frac{\max(x, y)}{\min(x, y)} \leq 1 + z \tag{9}$$

where z is a tolerance parameter that can be varied to control the strictness of the criterion. This criterion is appropriate when x and y are known to be nonzero and to have the same sign, as is the case here.

The proof of correctness ensures that the mode initially produced by `GreedyModeFind` will have maximal probability among all vectors that could be produced by the algorithm. Call the probability of this initial mode vector $p_{\max}$. Suppose that the algorithm concludes that joint modes exist — in other words, with $g - 1$ steps remaining in the algorithm, the top $k - 1$ elements on the heap all have scores approximately equal to the score of the element chosen at step $n - g + 1$. Each of the g elements chosen in a joint mode may have a score that is as low as a factor of $\frac{1}{1+z}$ below the actual maximal score, and g may be as large as $\min(n, r)$. Since the overall probability of a frequency vector is just the product of the scores of its element choices, in the worst case, a vector may be reported as a joint mode when its probability is just $\left(\frac{1}{1+z}\right)^{\min(n,r)}$ times the probability of the most-probable vector. In other words

$$\frac{p_{\max}}{p_{\min}} \leq (1 + z)^{\min(n,r)}. \tag{10}$$

Table 1
Timings for ten problems in each of nine problem sizes. Those problem instances for which a time of 0.00 is reported took less than one kernel clock tick (0.01 s on our Linux platform).

r	50,000	50,000	50,000	250,000	250,000	250,000	500,000	500,000	500,000
n	25,000	50,000	100,000	100,000	250,000	500,000	200,000	500,000	1,000,000
Min time (s)	0.00	0.00	0.00	0.03	0.01	0.02	0.07	0.04	0.04
Max time (s)	0.00	0.00	0.00	0.04	0.03	0.03	0.08	0.05	0.06

By rearranging this inequality, it is easy to determine the minimum value of the tolerance parameter z for a given acceptable level of relative error $p_{\max}/p_{\min}$:

$$z \geq \frac{p_{\max}}{p_{\min}}^{\frac{1}{\min(n,r)}} - 1. \tag{11}$$

For example, if $n = 1000$, $r = 1000$, and a maximum relative error of 10^{-6} is mandated (i.e. $p_{\max}/p_{\min} = 1.000001$), $z \simeq 10^{-9}$.

5. Performance results

Table 1 shows the performance of a C++ implementation of `GreedyModeFind` on problems of the same sizes used by Le Gall (2003) (note: they are not the same problem instances as were used by Le Gall). The table shows the minimum and maximum running times across the ten instances of each size. The platform used was a 3 GHz Pentium 4 computer running Linux, with 512 Mb of RAM and 2 Mb of L2 cache. Probability vectors were generated using random numbers read from the `/dev/urandom` device.

Assuming that the 3 GHz computer we used is 6–10 times faster than the 502 MHz computer used by Le Gall and adjusting accordingly, all our computations complete in comfortably less than 1 s — the minimum running time for any of Le Gall's computations. Some of Le Gall's runs took substantially longer (more than 27 min in one case).

6. Conclusion

We have presented `GreedyModeFind`, an algorithm for discovering the modes of a multinomial distribution that dramatically improves on earlier algorithms in terms of real-world performance on large instances. In addition, our algorithm avoids numerical precision problems to which earlier algorithms are susceptible. We provide asymptotic bounds on the time complexity of our algorithm.

A C++ implementation of `GreedyModeFind` is freely available for download at http://awcmee.massey.ac.nz/downloads/Data/wtwhite/multmodes.htm.

References

Feller, W., 1957. An Introduction to Probability Theory and its Applications, 2nd ed. Wiley, New York.
Finucan, H.M., 1964. The mode of a multinomial distribution. Biometrika 51, 513–517.
Le Gall, F., 2003. Determination of the modes of a multinomial distribution. Statistics & Probability Letters 62, 325–333.

Chapter 7

Conclusion

Somewhere between the beautiful abstractions of mathematics and the natural beauty of biology lies the grittier world of computational science. As we press on into the age of genomics and beyond, the importance of this field can only continue to grow. Like the plumbing in a large city, it is vital but most of the time lies hidden in the shadow of its more illustrious counterparts; it is, in effect, the infrastructure that makes modern biology possible and gives practical meaning to the mathematics behind it. And despite its plain appearance and tendency to linger in the background, there exists the occasional odd person who finds satisfaction in pushing the computational infrastructure of science to its limits—even to the point of writing a thesis on the subject!

7.1 XMP

As well as bringing faster computation of maximum parsimony trees, XMP serves as a model for the development of efficient parallelised B&B search algorithms in general.

In science, it is frequently the case that the value of the work done consists not only in the paths that were eventually taken, but also in the many paths that were explored but eventually discarded as unproductive or counterproductive. Perhaps the most frustrating unproductive path on this occasion was my attempt to exploit the apparent redundancy of highly multifurcating MP trees by collapsing ml0 edges. The possibility that I was missing some obvious shortcut that would enable enumeration of just the dense trees was so vexing that I eventually resorted to searching for a proof that no such technique could work in all cases. Although a negative result, it was satisfying to discover a counterexample that proved that enumerating only dense trees must miss some MP trees. On a similar note, the promising prospect of lower bounds based on the MinMax Squeeze (Holland *et al.*, 2005) led to disappointing results in practice: although the bounds produced were significantly better, they produced only a small speed increase in most cases. But despite these setbacks, XMP emerged the clear winner against parsimony programs already carefully optimised for speed.

In nearly all parallel systems, communication between CPUs is slow compared to computation on a single CPU. Consequently, the easiest way to kill the performance of a parallel algorithm is to force computation to wait for communication. The clearest offence is to use *blocking* communication calls: these explicitly wait for the communication operation (send or receive) to complete before returning, wasting time that could be spent on calculation. XMP uses no blocking calls in its main loop, instead preferring *overlapped* calls that initiate send and receive operations and then allow computation to continue.

However, even with overlapped communications, it is necessary to wait for the completion of I/O operations at certain points so that senders and receivers operating at different speeds[1] do not lose synchronisation. Communications protocols that involve only overlapped I/O can still lead to huge slowdowns if they allow the workloads transferred between CPUs to become arbitrarily small. This occurred sporadically in an early version of XMP, in which the master process held the base tree and split off smaller and smaller jobs to hand to idle workers. I was finally able to eliminate this problem by inventing the remaining edge pair list (REPL) data structure. This data structure, which is used for efficiently subdividing remaining work into jobs, is immediately applicable to any branch and bound (B&B) algorithm that searches through the space of all possible trees. In particular, the REPL approach can be applied to Maximum Likelihood B&B—a promising new direction for future work.

[1]Even identical CPUs running on the same motherboard cannot be assumed to run at identical speeds.

7.2 COIL

There is already a vast amount of DNA sequence data in the world, and not only is the amount increasing, the *rate* at which it is increasing is also increasing, thanks to continuing developments in next-generation sequencing technologies. Although physical storage sizes and network speeds have increased as well, the need to use these resources more efficiently is ever-present. `coil` takes a novel approach to compressing databases of DNA sequences inspired by the parsimony principle, in which trees linking similar sequences in the database are discovered, and only the edits (insertions, deletions or substitutions) required to move between adjacent sequences are recorded.

The secret to making `coil` practical was reducing the time taken to find groups of similar sequences, which necessarily involves comparing every sequence to every other sequence—an expensive $O(n^2)$ operation. Many carefully optimised exact solutions were tried, but none was fast enough to scale to the size of realistic databases containing millions of sequences.

The crucial gain in performance was achieved by introducing the concept of *lossiness*: watering down the guarantees made by a data structure in return for a dramatic size or speed improvement. Lossiness is what makes possible the astounding levels of compression achieved in JPEG images and MP3 audio. These formats use lossiness directly to compress a file down to a smaller file that, upon expansion, appears to human senses to be almost identical to the original. With DNA sequences we cannot tolerate any differences between the original database and the reexpanded version, but we can still employ the principle of lossiness as a means to the end of lossless compression. In the case of `find_edges`, although finding exactly the closest b neighbouring sequences of any given sequences is extremely time-consuming, for the purpose of compression it suffices to find b sequences that are highly likely to be very close to it, and this can be done much faster. This was accomplished through the invention of a new data structure, the *leaky move-to-front hashtable*, that remembers only recent and promising matches as we move along a search sequence.

It is particularly encouraging to see that `coil` has recently inspired a new compression program, ReCOIL (Yanovsky, 2011), which recognises disk I/O as an important bottleneck and explicitly focuses on minimising it. ReCOIL incorporates other improvements, such as searching for matches using reverse-complemented sequences in addition to forward sequences, and compresses DNA databases both faster and better than `coil` and leading general-purpose compression programs. There is no question that further advances in the compression of DNA sequence databases will continue to prove useful to the bioinformatics and phylogenetics communities as time goes on.

7.3 Treeness Triangles

Humans have a limited capacity to directly process raw numeric data, hence the need for effective visualisation techniques. Any visualisation method must find the right balance between keeping too much information (which overwhelms the reader with clutter) and discarding too much information (which hampers understanding). Of primary interest to an evolutionary biologist is how "treelike" his or her alignment is: how well it corresponds to a hypothesis of treelike evolution. This notion is well captured by building a tree on the alignment and then partitioning its sites into 3 categories: those corresponding to internal tree edges, to external tree edges, and to edges that are absent from the tree. The proportions of sites in each category can be intuitively represented by a single point in a ternary plot that is "pulled" towards each of the 3 corners.

Because a dataset is described by a single point, a single Treeness Triangle diagram can compactly summarise many different datasets, or many different treatments of a single dataset. The ability of Treeness Triangles to concisely show the total amount of conflicting signal present in a dataset makes them a powerful complement to other visualisations like Lento plots (Lento *et al.*, 1995) and networks. Treeness Triangles are thus an important tool for assessing the fit of phylogenetic datasets to a tree model.

Through experiments with both simulated and real chloroplast datasets, we demonstrated that increasing sequence lengths (which reduces sampling error) moves Treeness Triangle points closer to the I-E line, and that datasets containing deeper divergences have Treeness Triangle points closer to the E corner.

Although we expect Treeness Triangles to be most often used with trees built from DNA alignments, the method readily generalises to different types of input data, and different treebuilding methods. This flexibility will allow Treeness Triangles to remain useful for visualising model fit as more complex and realistic models of evolution are introduced in the future.

7.4 ML Bias

It remains an implicit assumption of much work in phylogenetics that the taxa under consideration evolved “on a tree”. Although this is typically true for a short segment of DNA such as a gene, it is now widely appreciated that different genes can evolve on different trees due to lineage sorting. In other words, we must face the possibility that the alignment we wish to analyse contains sites from a mixture of two or more trees. It is reasonable to ask: How will a standard single-tree ML inference method handle such data? In particular: If all trees in the mixture share a particular edge, and have the same edge length for that edge, will ML inference recover that edge and its length accurately?[2]

By considering a very simple mixture model consisting of a mixture of two trees on 4 taxa, or two trees on 5 taxa, we show that single-tree ML inference does introduce a bias to estimates of these shared edges. This should be taken as a warning: ML inference, though rightly lauded for its consistency properties, introduces a bias in the common case of tree mixtures that share edges—a model violation that almost certainly afflicts a large number of alignments. We argue that a shift in thinking towards networks needs to occur in phylogenetics to address this problem.

As awareness of non-treelike evolutionary processes continues to increase, new methods have been devised for handling them. Schliep (2009, chapter 2) considers more complex versions of our mixture model in his PhD thesis. Le *et al.* (2008) describes a mixture model that can learn different rate matrices for sites in a protein alignment having different secondary structure categories (exposed, buried, α-coil, β-sheet). Mixture models are a powerful way to generalise existing phylogenetic models to handle these processes, meaning that their impact in the field will likely

[2]I elaborate on our formulation of “shared parameter values” in the surrounding text of chapter 5.

increase with time.

7.5 Multinomial Modes

Given how central the multinomial distribution is to statistical analysis of all kinds, it seems staggering that in 2010 there was no published algorithm for quickly and accurately inferring the mode or modes of such a distribution. Although two published algorithms did exist, they were slow, had unknown computational complexity, and allowed rounding error to creep into intermediate calculations. Our new algorithm addresses all these shortcomings; with luck it will see application in diverse fields. Future work involves incorporating this algorithm into the popular statistical package R (2010).

7.6 How Much Optimisation is the Right Amount?

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.

Donald E. Knuth

Knuth's famous quote alludes to a widespread tendency among programmers to rework their straightforward, maintainable programs into fragile, incomprehensible spiderwebs in a bid to eliminate every wasted CPU cycle. He rightly reminds us that the runtime saved by such optimisation[3] (frequently nanoseconds) is almost never worth the programmer time spent (frequently hours, or weeks), nor the additional complexity introduced. Given that one of the goals of this thesis is the efficient implementation of algorithms, it is appropriate to reflect on the extent to which the tradeoffs made have been worthwhile.

When attempting to get the most performance out of a lengthy computational analysis, the biggest gains are usually achieved by using algorithms and data structures with good performance for large n (see section 1.4). Every computational

[3] "Optimisation" here refers to the process of speeding up or shrinking a program without changing its functionality in other respects; it is never taken to imply that the resulting program is optimally fast or small. This unfortunate usage has meant that attempts to generate provably shortest-possible program fragments have been forced to adopt the term "superoptimisation" to differentiate themselves (Massalin, 1987).

scientist aspires to develop new, asymptotically faster methods for such analyses, and there can be no question that pursuing this kind of optimisation is worthwhile. The `GreedyModeFind` algorithm of chapter 6 is an example of this kind of advance. But eventually algorithmic cleverness runs out, and at that point, if the analysis is still too slow, the only way to improve performance further is by chipping away at the constant factors. Usually this means resorting to a lower-level language that enables tighter control over the computer hardware, at the cost of increased development time and reduced portability.

In this thesis I have attacked two problems using this kind of low-level optimisation: the Fitch parsimony calculations for `xmp` (chapter 2), and the inner loop of the `find_edges` program in `coil` (chapter 3). In both cases, I reimplemented these inner loops with handcoded SSE2 assembly language, which is a set of extended instructions that takes advantage of the 128-bit registers available in Intel Pentium 4 and newer processors. Using these instructions enables a potential 4-fold increase in performance over the standard instructions, which operate on 32-bit registers.

In the case of `xmp`, the result was a significant improvement in speed, as clearly demonstrated in Fig. 1 on p. 27. This improvement was particularly strong for datasets having many sites. Here the SSE2 implementation behaves essentially the same as the C code, except on 128 bits (32 nucleotides) at a time instead of 32 bits (8 nucleotides). This meant that only a moderate amount of time was required to develop and test it.

In the case of `coil`, the result was actually a slight *decrease* in speed—around 6% on the ems100 dataset as reported in Table 3 on p. 58. This disappointing result was despite the careful construction of an inner loop code fragment free of conditional branch instructions, which tend to slow down modern pipelined CPUs. It must be noted that this measurement was produced on an Intel Core 2 processor; the SSE2 code fragment was originally developed for the Pentium 4, with its instruction timings in mind, and in the original version of `coil`, which was tested on that CPU, the SSE2 code produced a speedup of around 25% (unpublished data). This outcome should be considered a lesson: the performance advantage of a piece of code that has been carefully tuned for a particular CPU is extremely fragile.

Despite the mixed outcomes of these low-level optimisation attempts, I believe that in the context of doctoral research focused on maximally efficient computation, trying them out was the right thing to do. Whether such a thoroughgoing approach is justified for the day-to-day development of computational analyses is harder to say. However, considering the amount and diversity of program code in the world, a strong case can be made that the inner loops of heavy computational analyses are among the 3% of code that Knuth would have us focus our collective optimisation efforts on. The growing presence of GPU-based computation in bioinformatics (e.g. Manavski & Valle, 2008; Blazewicz *et al.*, 2011) suggests that reducing constant factors is indeed important enough to deserve our time and attention.

7.7 Other Directions

Computational research is a large field. During the course of my thesis work, I have also worked on other projects that touch the borders of the work covered here. Chen *et al.* (2008) describes a computational analysis that detected spliceosomal introns in the *Giardia* genome. Ongoing work involves extending and simplifying the ancestral sequence reconstruction of PAML (Yang, 2007) to enable ancient DNA sequences to be rebuilt from extant sequence data.

7.8 Summary

I find it encouraging that 3 of the 5 papers presented in this thesis have been cited by other authors—this suggests that the work is of interest to others in the phylogenetics and bioinformatics community.

In short, I see a future in which computational techniques grow increasingly essential—both for managing and for extracting meaning from biological data. At the same time, as the tide of data continues to rise and the models we use to analyse it continue to become more complex, the need to concern ourselves with issues of goodness-of-fit and model misspecification becomes ever more crucial, if we are to avoid deceiving ourselves about what our data is telling us. It is my hope that this thesis advances the lines of battle in both these domains.

Appendix A

Work Breakdown

The papers contained in this thesis were written in collaboration with other authors, though no other author is present on every paper.

A.1 XMP

The `xmp` project is contained in chapter 2.

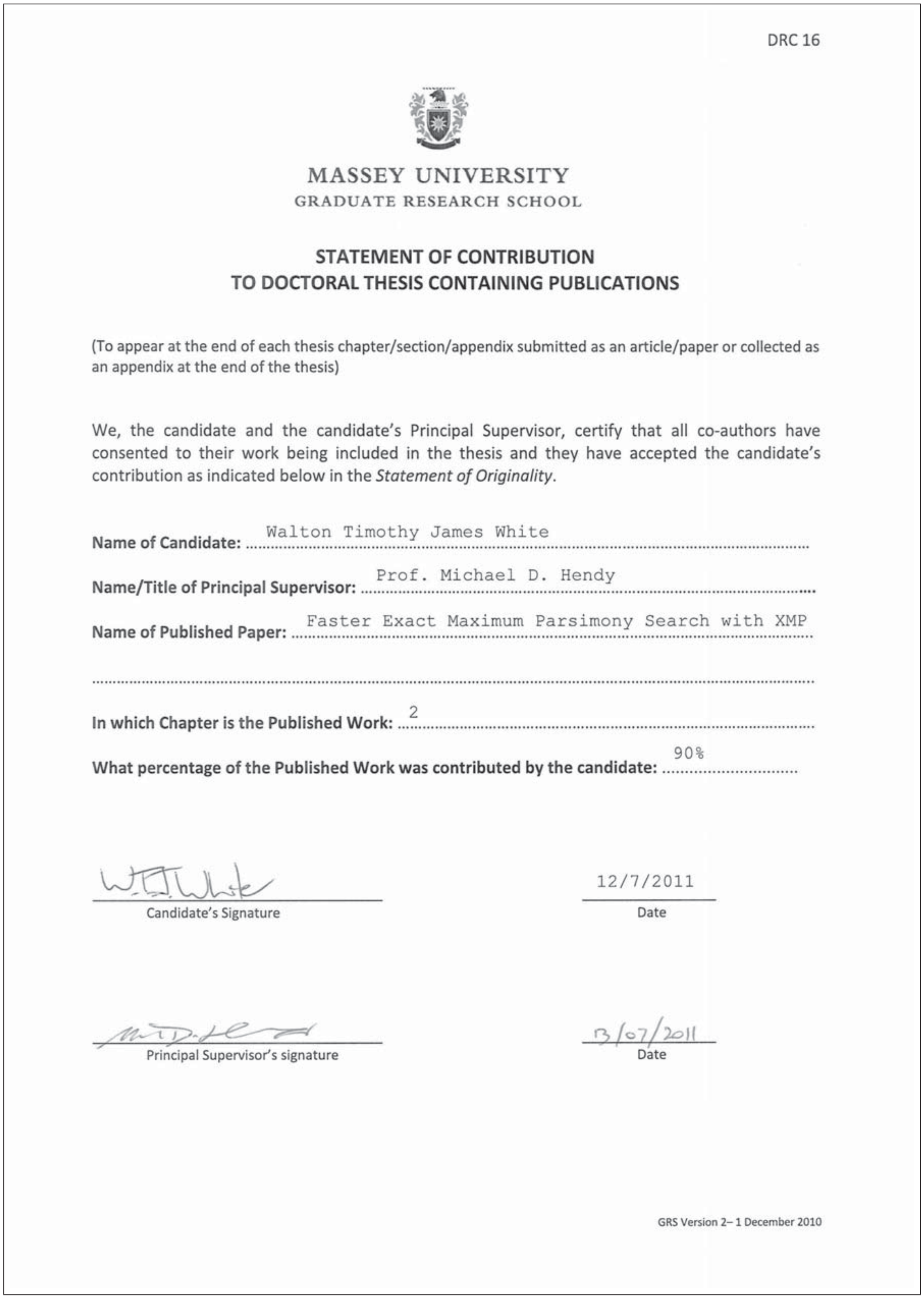
A.1.1 Author Contributions

WTJW is responsible for 90% of the paper "Faster Exact Maximum Parsimony Search with XMP". He was exclusively responsible for:

- Designing and implementing the `xmp` program, including:
  - Final parallelisation strategy
  - PARTBOUND lower bound
  - Fast Fitch algorithm
  - SSE2-optimised version of fast Fitch algorithm
- Researching existing work
- Debugging, notably including building and carefully tuning an MPI-Spin model of the system of communicating processes
- Negotiating access to the BlueFern supercomputer

- Performance measurement
- Writing a near-complete draft of the paper

BRH is responsible for 10% of this paper. The idea of parallelising MP tree search originated with her, as did the notion of developing a new type of lower bound based on the MinMax Squeeze. An early parallelisation strategy was the result of discussion between WTJW and BRH, although it proved to be temperamental and was replaced by a new strategy developed by WTJW. BRH additionally provided guidance and helped edit the final manuscript.



DRC 16

MASSEY UNIVERSITY
GRADUATE RESEARCH SCHOOL

STATEMENT OF CONTRIBUTION
TO DOCTORAL THESIS CONTAINING PUBLICATIONS

(To appear at the end of each thesis chapter/section/appendix submitted as an article/paper or collected as an appendix at the end of the thesis)

We, the candidate and the candidate's Principal Supervisor, certify that all co-authors have consented to their work being included in the thesis and they have accepted the candidate's contribution as indicated below in the *Statement of Originality*.

Name of Candidate: Walton Timothy James White

Name/Title of Principal Supervisor: Prof. Michael D. Hendy

Name of Published Paper: Faster Exact Maximum Parsimony Search with XMP

In which Chapter is the Published Work: 2

What percentage of the Published Work was contributed by the candidate: 90%

[illegible]	12/7/2011
Candidate's Signature	Date
[illegible]	13/07/2011
Principal Supervisor's signature	Date

GRS Version 2– 1 December 2010

A.2 COIL

The `coil` project is contained in chapter 3.

A.2.1 Author Contributions

WTJW is responsible for 90% of the paper "Compressing DNA Sequence Databases with coil". He was exclusively responsible for:

- Designing and implementing the `coil` software, including:
  - Underlying concept (edit-tree coding)
  - Leaky move-to-front hashtable data structure
  - SSE2-optimised version of `find_edges`
  - All code improvements subsequent to previously examined work (see next section)
- Researching existing work
- Performance measurement
- Writing a near-complete draft of the paper

MDH is responsible for 10% of this paper. He provided guidance and helped edit the final manuscript.

A.2.2 Previously Examined Work

It is important to note that an early version of `coil`, implemented as `coil` v1.0, was examined as part of a Postgraduate Certificate in Science (PGCertSc) undertaken by WTJW in 2003. For this certificate, a collection of programs was implemented and a report was produced describing the design and implementation in detail, and measuring performance on a test dataset.

The paper presented in this thesis was prepared subsequent to the completion of the PGCertSc; the writing of this paper represents entirely new work. Preparing this paper involved researching new developments in the literature, such as the CD-HIT program (Li & Godzik, 2006) and techniques for searching within compressed

	Files	Lines
`coil` v1.0	45	45,680
`coil` v1.1	43	8,091
Differences		2,045

Table A.1: `coil` source code comparison of all `.c` and `.h` files.

databases (Ferragina & Manzini, 2005; Russo & Oliveira, 2006; Foschini *et al.*, 2006), and also entailed new and more comprehensive performance testing.

In addition, extensive improvements have been made to the code itself, although the core design has not changed. Table A.1 attempts to summarise the amount of source code modification between the version submitted for the PGCertSc (`coil` v1.0) and the version available for download at the time of submission of the journal paper (`coil` v1.1). The figure reported for the "Differences" row is the number of lines reported by `diff -r` when set to ignore whitespace changes. This underestimates the amount of change, since only one line is output for each of the 40 files that exist in only one version of the software. The dramatic reduction in the number of lines of code represents an increase in the level of organisation of the source code, not a reduction in functionality. The original `coil` software contained large amounts of near-duplicate code—particularly in the implementation of `find_edges`, due to the wide variety of approaches tried. Duplication encourages the proliferation of bugs by making it harder to ensure that all instances of erroneous behaviour are fixed when a bug is found.

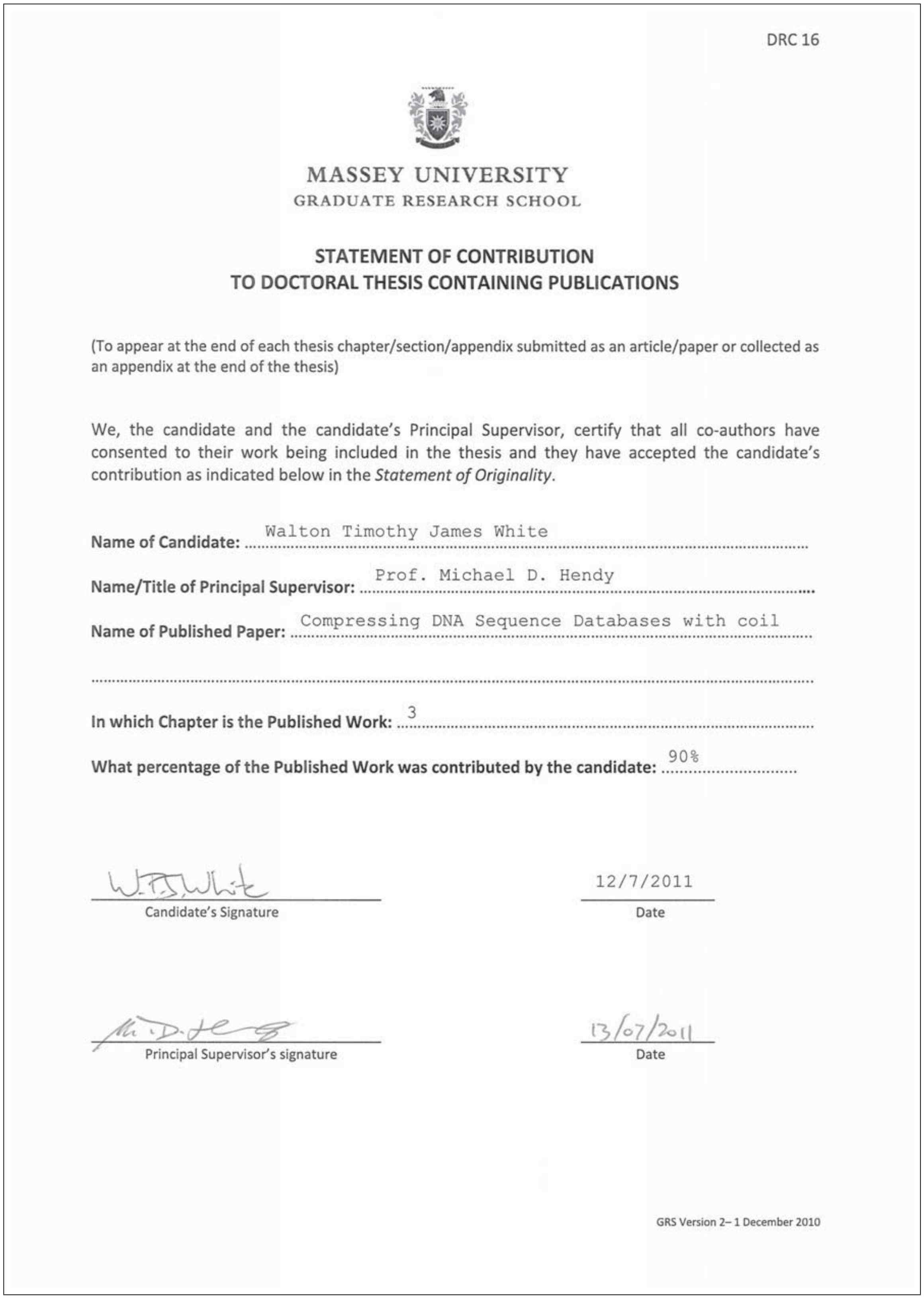
Other important changes to the code include:

- As Figure 3 in the paper shows, `coil` consists of a number of separate programs, and compressing a database using them is a complicated process that produces numerous files. Only a subset of the files produced are needed for decompression, and exactly which files are needed depends on whether *standalone* or *incremental* compression is being used. Additionally, several programs have configuration options that must be matched when running downstream programs. In order to simplify this process for the user, a 514-line Perl script, `coil.pl`, was written. This script manages the entire compression process, invoking programs as necessary, using sensible defaults where none are provided, and packaging just the relevant output files into a single `.tar.bz2` archive file.

- The `decode` program has likewise been enhanced to make working with archive files more convenient.
- With `coil` v1.0, compressing and decompressing a database would in general cause the recovered sequences to appear in a different order. Usually this is adequate, but for some purposes it may be necessary to preserve the order of the sequences, so that the original FASTA file and the final result of decompression are byte-for-byte identical. For this reason, and for parity with general-purpose compression programs, the `-x` option was added. Supplying this option at compression time causes `coil` to store additional sequence-order information, enabling exact recovery of the original file.
- All files that store binary integers now use little-endian format (the least-significant byte of a 4-byte word is stored first). This enables archives compressed on a little-endian machine (such as Intel x86) to be decompressed on a big-endian machine (such as PowerPC) and vice versa.
- OS X compatibility was added.
- A Makefile was created to enable the software to be built painlessly on most Unix-like operating systems.
- Speed improvements:
  - `encode`: An early attempt at speeding up the Myers (1986) edit script operated by finding all 8-tuple exact matches between the parent and child sequence. The results of these calculations were no longer being used, despite the fact that the calculations were still being performed. These extraneous calculations were removed, speeding up `encode` by 40% on one dataset.
  - `find_edges`: Code for calculating the highest hit count produced by each k-tuple was found to be extraneous and was removed from the innermost loop.
- Numerous small usability improvements and bug fixes, such as:
  - The ability to process lowercase DNA data.

  - Help for each program available via `--help`.

All in all, `coil` has grown from a fragile and temperamental “research project” in v1.0 to a well organised, well tested and highly usable piece of software in v1.1.



DRC 16

MASSEY UNIVERSITY
GRADUATE RESEARCH SCHOOL

STATEMENT OF CONTRIBUTION TO DOCTORAL THESIS CONTAINING PUBLICATIONS

(To appear at the end of each thesis chapter/section/appendix submitted as an article/paper or collected as an appendix at the end of the thesis)

We, the candidate and the candidate's Principal Supervisor, certify that all co-authors have consented to their work being included in the thesis and they have accepted the candidate's contribution as indicated below in the *Statement of Originality*.

Name of Candidate: Walton Timothy James White

Name/Title of Principal Supervisor: Prof. Michael D. Hendy

Name of Published Paper: Compressing DNA Sequence Databases with coil

In which Chapter is the Published Work: 3

What percentage of the Published Work was contributed by the candidate: 90%

[illegible] 12/7/2011
Candidate's Signature Date

[illegible] 13/07/2011
Principal Supervisor's signature Date

GRS Version 2– 1 December 2010

A.3 Treeness Triangles

The Treeness Triangles project is contained in chapter 4.

A.3.1 Author Contributions

WTJW is responsible for 30% of the paper "Treeness Triangles: Visualizing the Loss of Phylogenetic Signal". He was exclusively responsible for:

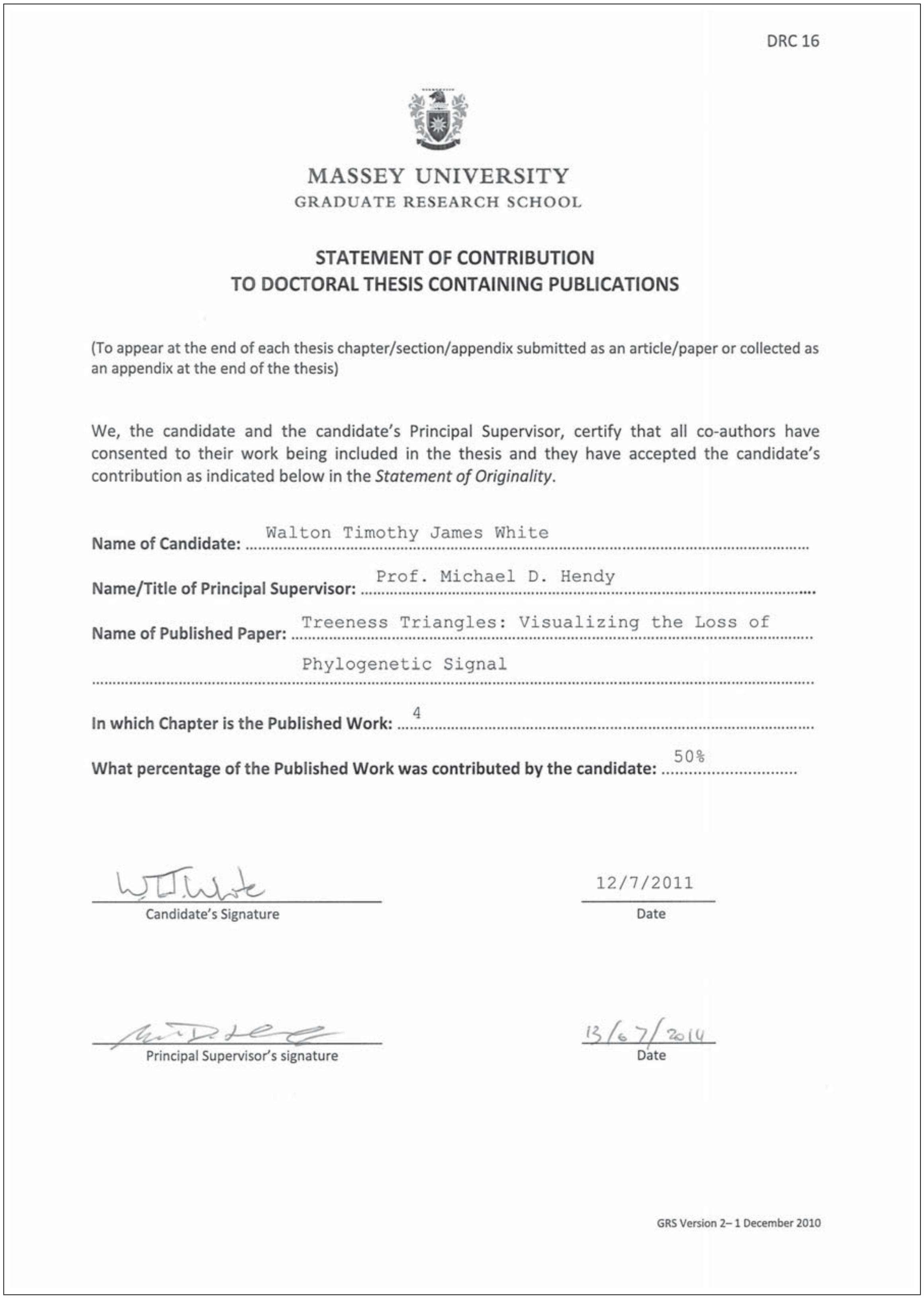
- Designing and implementing all software
- Designing computational experiments for simulated datasets
- Performing all computational experiments (both for simulated and real datasets)
- Producing figures 2, 3A-F, 4 and 5
- Quality control of real datasets

WTJW contributed strongly but not exclusively to:

- Writing the Materials and Methods section
- Writing the Results section
- Editing the final manuscript

WTJW also contributed to the introduction by researching and writing about the likelihood-mapping approach of Strimmer & Von Haeseler (1996) and the dekapentagonal mapping approach of Zhaxybayeva *et al.* (2004).

SFH and RG both worked on assembling and aligning a range of chloroplast datasets that were used in the analysis. These datasets had a range of deeper divergence times, enabling the "treeness" to be evaluated over a range of timescales. BRH and DP designed the project and were responsible for its overall direction.



DRC 16

MASSEY UNIVERSITY
GRADUATE RESEARCH SCHOOL

**STATEMENT OF CONTRIBUTION
TO DOCTORAL THESIS CONTAINING PUBLICATIONS**

(To appear at the end of each thesis chapter/section/appendix submitted as an article/paper or collected as an appendix at the end of the thesis)

We, the candidate and the candidate's Principal Supervisor, certify that all co-authors have consented to their work being included in the thesis and they have accepted the candidate's contribution as indicated below in the *Statement of Originality*.

Name of Candidate: Walton Timothy James White

Name/Title of Principal Supervisor: Prof. Michael D. Hendy

Name of Published Paper: Treeness Triangles: Visualizing the Loss of Phylogenetic Signal

In which Chapter is the Published Work: 4

What percentage of the Published Work was contributed by the candidate: 50%

[illegible]
Candidate's Signature

12/7/2011
Date

[illegible]
Principal Supervisor's signature

13/67/2014
Date

GRS Version 2– 1 December 2010

A.4 ML Bias

The ML Bias project is contained in chapter 5.

A.4.1 Author Contributions

WTJW is responsible for 30% of this paper. He was exclusively responsible for:

- Designing and implementing all software used, including:
  - Scripts to build datasets for PAUP* and extract the relevant results
  - A program to estimate the proportion of 2 trees in a mixture model via hill-climbing
- Performing all computational experiments
- Producing figures 1A, 1B, 1D and 2

WTJW contributed strongly but not exclusively to:

- Discussions regarding the purpose and direction of the manuscript
- Editing the final manuscript

DRC 16

MASSEY UNIVERSITY
GRADUATE RESEARCH SCHOOL

STATEMENT OF CONTRIBUTION
TO DOCTORAL THESIS CONTAINING PUBLICATIONS

(To appear at the end of each thesis chapter/section/appendix submitted as an article/paper or collected as an appendix at the end of the thesis)

We, the candidate and the candidate's Principal Supervisor, certify that all co-authors have consented to their work being included in the thesis and they have accepted the candidate's contribution as indicated below in the *Statement of Originality*.

Name of Candidate: Walton Timothy James White

Name/Title of Principal Supervisor: Prof. Michael D. Hendy

Name of Published Paper: A Bias in ML Estimates of Branch Lengths in the Presence of Multiple Signals

In which Chapter is the Published Work: 5

What percentage of the Published Work was contributed by the candidate: 30%

[illegible]
Candidate's Signature

12/7/2011
Date

[illegible]
Principal Supervisor's signature

13/07/2011
Date

GRS Version 2– 1 December 2010

A.5 Multinomial Modes

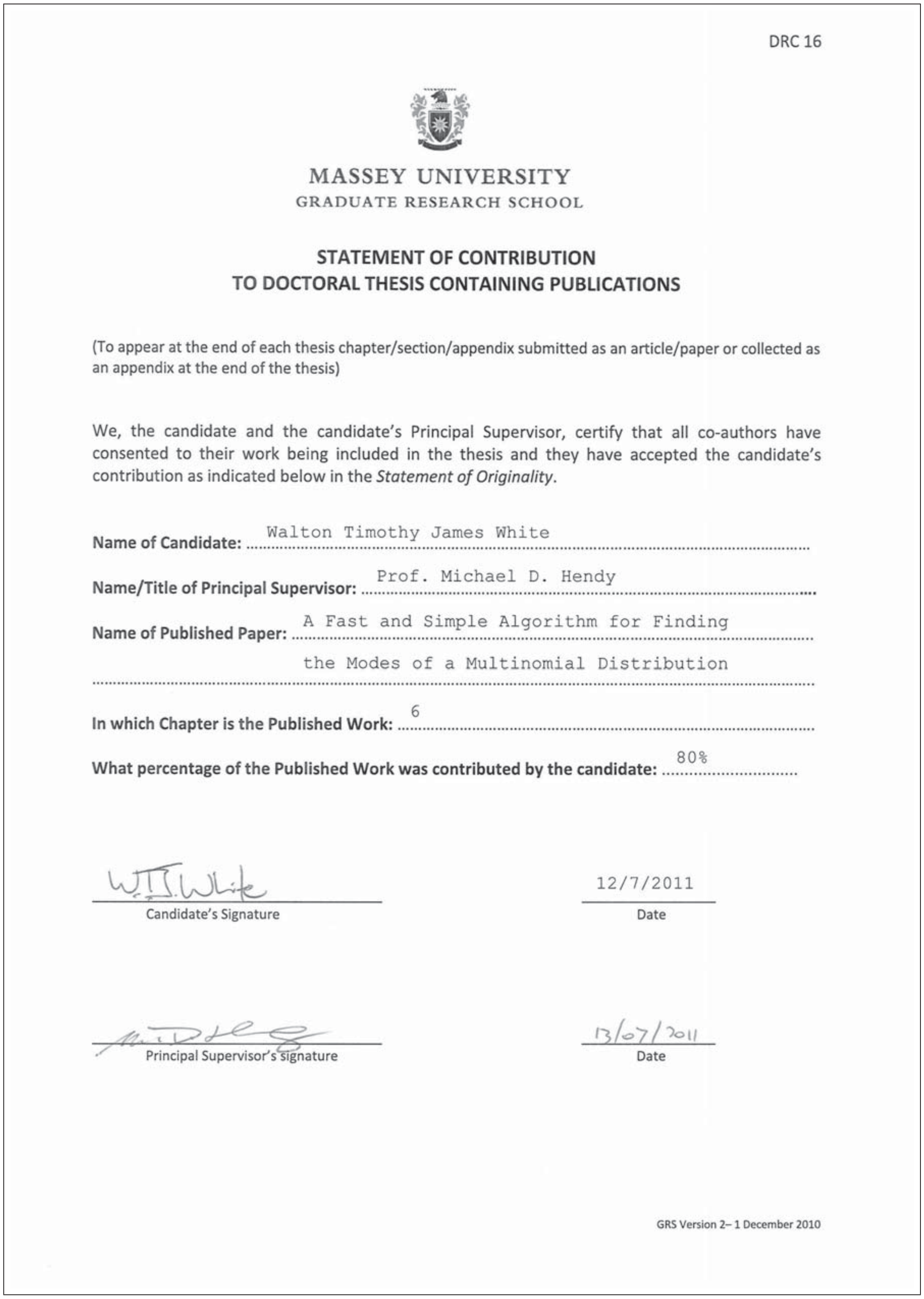
The Multinomial Modes project is contained in chapter 6.

A.5.1 Author Contributions

WTJW is responsible for 80% of this paper. He was exclusively responsible for:

- Originally conceiving the project
- Designing the `GreedyModeFind` algorithm and implementing it in C++
- An early proof of correctness of the algorithm
- Researching existing work
- Analysis of asymptotic running time and numerical stability
- Performance measurement
- Writing a near-complete draft of the paper

As well as providing guidance and helping to edit the final manuscript, MDH made suggestions that considerably simplified the proof of correctness of this algorithm.



DRC 16

MASSEY UNIVERSITY
GRADUATE RESEARCH SCHOOL

STATEMENT OF CONTRIBUTION
TO DOCTORAL THESIS CONTAINING PUBLICATIONS

(To appear at the end of each thesis chapter/section/appendix submitted as an article/paper or collected as an appendix at the end of the thesis)

We, the candidate and the candidate's Principal Supervisor, certify that all co-authors have consented to their work being included in the thesis and they have accepted the candidate's contribution as indicated below in the *Statement of Originality*.

Name of Candidate: Walton Timothy James White

Name/Title of Principal Supervisor: Prof. Michael D. Hendy

Name of Published Paper: A Fast and Simple Algorithm for Finding the Modes of a Multinomial Distribution

In which Chapter is the Published Work: 6

What percentage of the Published Work was contributed by the candidate: 80%

W.T.J. White
Candidate's Signature

12/7/2011
Date

[illegible]
Principal Supervisor's signature

13/07/2011
Date

GRS Version 2– 1 December 2010

Bibliography

Atteson, K. 1999. The performance of neighbor-joining methods of phylogenetic reconstruction. *Algorithmica*, **25**(2-3), 251–278.

Blazewicz, Jacek, Frohmberg, Wojciech, Kierzynka, Michal, Pesch, Erwin, & Wojciechowski, Pawel. 2011. Protein alignment algorithms with an efficient backtracking routine on multiple GPUs. *BMC Bioinformatics*, **12**(1), 181.

Buneman, Peter. 1971. *Mathematics in the Archeological and Historical Sciences.* Edinburgh: Edinburgh University Press. Chap. The recovery of trees from measures of dissimilarity, pages 387–395.

Cavalli-Sforza, L. L., & Edwards, A. W. F. 1967. Phylogenetic analysis: Models and estimation procedures. *American Journal of Human Genetics*, **19**, 233–257.

Chang, Joseph T. 1996. Full reconstruction of Markov models on evolutionary trees: Identifiability and consistency. *Mathematical Biosciences*, **137**(1), 51–73.

Chen, Xiaowei Sylvia, White, W. Timothy J., Collins, Lesley J., & Penny, David. 2008. Computational Identification of Four Spliceosomal snRNAs from the Deep-Branching Eukaryote Giardia intestinalis. *PLoS One*, **3**(8).

Cook, Steven A. 1971. The Complexity of Theorem-Proving Procedures. *Pages 151–158 of: Third Annual ACM Symposium on Thoery of Computing.*

Degnan, James H., & Rosenberg, Noah A. 2009. Gene tree discordance, phylogenetic inference and the multispecies coalescent. *Trends in Ecology & Evolution*, **24**(6), 332–340.

Drummond, A., & Rodrigo, A. G. 2000. Reconstructing genealogies of serial samples under the assumption of a molecular clock using serial-sample UPGMA. *Molecular Biology and Evolution*, **17**(12), 1807–1815.

Felsenstein, J. 1978. Cases in which parsimony or compatibility will be positively misleading. *Systematic Zoology*, **27**, 401–410.

Felsenstein, J. 1981. Evolutionary trees from DNA sequences: a maximum likelihood approach. *J. Mol. Evol.*, **17**, 368–376.

Felsenstein, J. 2004. *Inferring Phylogenies*. Sunderland, Massachusetts: Sinauer Associates, Inc.

Ferragina, P., & Manzini, G. 2005. Indexing compressed text. *Journal of the ACM*, **52**(4), 552–581.

Finucan, H. M. 1964. Mode of Multinominal Distribution. *Biometrika*, **51**(3-4), 513–517.

Fisher, R. A. 1922. On the mathematical foundations of theoretical statistics. *Philosophical Transactions of the Royal Society of London, Series A*, **222**, 309–368.

Fitch, W. M. 1971. Toward defining the course of evolution: Minimum change for a specified tree topology. *Systematic Zoology*, **20**, 406–416.

Fitch, W. M., & Margoliash, E. 1967. Construction of phylogenetic trees. *Science*, **155**, 279–284.

Foschini, Luca, Grossi, Roberto, Gupta, Ankur, & Vitter, Jeffrey Scott. 2006. When indexing equals compression: Experiments with compressing suffix arrays and applications. *ACM Trans. Algorithms*, **2**(4), 611–639.

Garey, Michael R., & Johnson, David S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co.

Gladstein, David S. 1997. Efficient Incremental Character Optimization. *Cladistics*, **13**(1-2), 21–26.

Goldman, N. 1993. Statistical Tests of Models of DNA Substitution. *Journal of Molecular Evolution*, **36**(2), 182–198.

Goloboff, Pablo A. 1993. Character Optimization and Calculation of Tree Lengths. *Cladistics*, **9**(4), 433–436.

Goloboff, Pablo A. 1996. Methods for faster parsimony analysis. *Cladistics*, **12**(3), 199–220.

Hastings, W. K. 1970. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, **57**(1), 97–109.

Heled, Joseph, & Drummond, Alexei J. 2010. Bayesian Inference of Species Trees from Multilocus Data. *Molecular Biology and Evolution*, **27**(3), 570–580.

Hendy, M. D., & Penny, D. 1982. Branch and bound algorithms to determine minimal evolutionary trees. *Mathematical Biosciences*, **59**(2), 277–290.

Hendy, M. D., & Penny, D. 1989. A framework for the quantitative study of evolutionary trees. *Systematic Zoology*, **38**, 297–309.

Hendy, M. D., Foulds, L. R., & Penny, D. 1980. Proving phylogenetic trees minimal with l-clustering and set partitioning. *Mathematical Biosciences*, **51**(1-2), 71–88.

Hendy, Michael, & Penny, David. 1993. Spectral analysis of phylogenetic data. *Journal of Classification*, **10**(1), 5–24.

Holland, B. R., Huber, K. T., Dress, A., & Moulton, V. 2002. Delta Plots: A Tool for Analyzing Phylogenetic Distance Data. *Molecular Biology and Evolution*, **19**(12), 2051–2059.

Holland, B. R., Huber, K. T., Penny, D., & Moulton, V. 2005. The MinMax squeeze: Guaranteeing a minimal tree for population data. *Molecular Biology and Evolution*, **22**(2), 235–242.

Huber, Katharina T., Langton, Michael, Penny, David, Moulton, Vincent, & Hendy, Michael. 2002. Spectronet: a package for computing spectra and median networks. *Applied Bioinformatics*, **1**(3), 159–61.

Kahan, W. 1965. Further Remarks on Reducing Truncation Errors. *Communications of the ACM*, **8**(1), 40–40.

Karp, Richard M. 1972. *Complexity of Computer Computations*. New York: Plenum. Chap. Reducibility Among Combinatorial Problems, pages 85–103.

Klee, Victor, & Minty, George J. 1972. How good is the simplex algorithm? *Inequalities, III (Proc. Third Sympos., Univ. California, Los Angeles, Calif., 1969; dedicated to the memory of Theodore S. Motzkin)*, 159–175.

Le, S. Q., Lartillot, N., & Gascuel, O. 2008. Phylogenetic mixture models for proteins. *Philosophical Transactions of the Royal Society B-Biological Sciences*, **363**(1512), 3965–3976.

Le Gall, Francoise. 2003. Determination of the modes of a Multinomial distribution. *Statistics & Probability Letters*, **62**(4), 325–333.

Lento, G. M., Hickson, R. E., Chambers, G. K., & Penny, D. 1995. Use of Spectral Analysis to Test Hypotheses on the Origin of Pinnipeds. *Molecular Biology and Evolution*, **12**(1), 28–52.

Li, Ming, & Vitányi, Paul. 2008. *An Introduction to Kolmogorov Complexity and its Applications*. 3rd edn. Texts in Computer Science. New York: Springer Science+Business Media, LLC.

Li, W. Z., & Godzik, A. 2006. Cd-hit: a fast program for clustering and comparing large sets of protein or nucleotide sequences. *Bioinformatics*, **22**(13), 1658–1659.

Manavski, Svetlin, & Valle, Giorgio. 2008. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinformatics*, **9**(Suppl 2), S10.

Massalin, Henry. 1987. Superoptimizer: A Look at the Smallest Program. *Pages 122–126 of: Proceedings of the second international conference on architectual support for programming languages and operating systems*. Palo Alto, California, United States: IEEE Computer Society Press.

Mau, Bob, Newton, Michael A., & Larget, Bret. 1999. Bayesian Phylogenetic Inference via Markov Chain Monte Carlo Methods. *Biometrics*, **55**(1), 1–12.

Meacham, C. A. 1981. A Manual Method for Character Compatibility Analysis. *Taxon*, **30**(3), 591–600.

Metropolis, Nicholas, Rosenbluth, Arianna W., Rosenbluth, Marshall N., Teller, Augusta H., & Teller, Edward. 1953. Equation of State Calculations by Fast Computing Machines. *The Journal of Chemical Physics*, **21**(6), 1087–1092.

Myers, Eugene W. 1986. An O(ND) Difference Algorithm and its Variations. *Algorithmica*, **1**(2), 251–266.

Nelder, J. A., & Mead, R. 1965. A Simplex Method for Function Minimization. *Computer Journal*, **7**, 308–313.

Neyman, J. 1971. *Statistical Decision Theory and Related Topics.* Academic Press, New York. Chap. Molecular studies of evolution: A source of novel statistical problems, pages 1–27.

Penny, D., Hendy, M. D., & Steel, M. A. 1992. Progress with methods for constructing evolutionary trees. *Trends in Ecology and Evolution*, **7**(3), 73–79.

Penny, David. 2011. Darwin's Theory of Descent with Modification, versus the Biblical Tree of Life. *PLoS Biol*, **9**(7), e1001096.

R Development Core Team. 2010. *R: A Language and Environment for Statistical Computing.*

Rannala, B., & Yang, Z. 1996. Probability distribution of molecular evolutionary trees: A new method of phylogenetic inference. *Journal of Molecular Evolution*, **43**(3), 304–311.

Ronquist, F. 1998. Fast Fitch-parsimony algorithms for large data sets. *Cladistics*, **14**(4), 387–400.

Russo, L. M. S., & Oliveira, A. L. 2006. *String Processing and Information Retrieval, Proceedings.* Lecture Notes in Computer Science, vol. 4209. lsr@algos.inesc-id.pt aml@algos.inesc-id.pt: Springer-Verlag Berlin. Chap. A compressed self-index using a Ziv-Lempel dictionary, pages 163–180.

Saitou, N., & Nei, M. 1987. The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Molecular biology and evolution*, **4**(4), 406–425.

Schliep, Klaus. 2009. *Some Applications of Statistical Phylogenetics*. Ph.D. thesis, Palmerston North, New Zealand.

Schliep, Klaus Peter. 2011. phangorn: phylogenetic analysis in R. *Bioinformatics*, **27**(4), 592–593.

Semple, Charles, & Steel, Mike. 2003. *Phylogenetics*. Oxford Lecture Series in Mathematics. Oxford: Oxford University Press.

Seward, Julian. 1997. *bzip2 and libbzip2 - A program and library for data compression*.

Shepherd, L. D., & Lambert, D. M. 2008. Ancient DNA and conservation: lessons from the endangered kiwi of New Zealand. *Molecular Ecology*, **17**(9), 2174–2184.

Steel, M., & Penny, D. 2004. Two further links between MP and ML under the Poisson model. *Applied Mathematics Letters*, **17**(7), 785–790.

Steel, M. A., Hendy, M. D., & Penny, D. 1993. Parsimony Can Be Consistent. *Systematic Biology*, **42**(4), 581–587.

Steel, Mike, & Penny, David. 2000. Parsimony, Likelihood, and the Role of Models in Molecular Phylogenetics. *Molecular Biology and Evolution*, **17**(6), 839–850.

Strimmer, K., & Von Haeseler, A. 1996. Quartet puzzling: A quartet maximum-likelihood method for reconstructing tree topologies. *Molecular Biology and Evolution*, **13**(7), 964–969.

Sullivan, J., & Swofford, D. L. 2001. Should we use model-based methods for phylogenetic inference when we know that assumptions about among-site rate variation and nucleotide substitution pattern are violated? *Systematic Biology*, **50**(5), 723–729.

Swofford, David L. 2001. *PAUP*. Phylogenetic Analysis Using Parsimony (*and Other Methods)*.

Swofford, David L., Olsen, Gary J., Waddell, Peter J., & Hillis, David M. 1996. *Molecular Systematics*. 2nd edn. Sunderland, MA: Sinauer Associates. Chap. Phylogenetic Inference, pages 407–514.

Tarjan, R. E. 1985. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, **6**(2), 306–318.

Tuffley, C., & Steel, M. 1997. Links between maximum likelihood and maximum parsimony under a simple model of site substitution. *Bulletin of Mathematical Biology*, **59**(3), 581–607.

Yan, M., & Bader, D. A. 2003. *Fast character optimization in parsimony phylogeny reconstruction*. Tech. rept.

Yang, Ziheng. 2007. PAML 4: Phylogenetic Analysis by Maximum Likelihood. *Molecular Biology and Evolution*, **24**(8), 1586–1591.

Yanovsky, Vladimir. 2011. ReCoil - an algorithm for compression of extremely large datasets of DNA data. *Algorithms for Molecular Biology*, **6**(1), 23.

Zharkikh, Andrey. 1977. *Matematicheskie Modeli Evolyutsii i Selektsii*. Novosibirsk: Institut Tsitologii i Genetiki. Chap. Algoritm postroeniya filogeneticheskikh drev po amino-kislotnym posledobatel'nostyam, pages 5–52.

Zhaxybayeva, O., Hamel, L., Raymond, J., & Gogarten, J. P. 2004. Visualization of the phylogenetic content of five genomes using dekapentagonal maps. *Genome Biology*, **5**(3), 11.