A FRAMEWORK FOR THE CONSTRUCTION, VISUALIZATION, AND CHARACTERIZATION OF PHYLOGENY SEARCH SPACE

by

Jeffrey H.F. Cullis

Submitted in partial fulfillment of the
requirements for the degree of
Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
August 2008

Your file Votre référence
ISBN: 978-0-494-44076-6
Our file Notre référence
ISBN: 978-0-494-44076-6

DALHOUSIE UNIVERSITY

To comply with the Canadian Privacy Act the National Library of Canada has requested that the following pages be removed from this copy of the thesis:

Preliminary Pages
- Examiners Signature Page (pii)
- Dalhousie Library Copyright Agreement (piii)

Appendices
- Copyright Releases (if applicable)

Table of Contents

List of Tables

List of Figures

Abstract

A phylogeny is a tree representation of the evolutionary relationships between a set of taxa. Given a set of sequences, a model of evolution, and a tree topology, the fit of a phylogeny is evaluated using the maximum likelihood (ML) criterion. Searching for the ML phylogeny for a dataset is typically done by traversing the search space using tree rearrangements such as NNI and SPR. The space that is being searched is extremely large. Many heuristics have been introduced in attempts to improve the search, though these are developed with little intuition for the characteristics of the search space. Here we introduce new methods for visualizing, storing, and characterizing the search graph. Visualization of all the trees in the search space and the paths of tree modifications between them were found to be possible on small tree sets, with many features of interest clearly represented. On larger sets, these graphs are obscured by the sheer number of trees and connections between them. The format used for converting tree topologies to vertices within the visuals was also found to be inefficient and inelegant. A new phylogeny representation is introduced that reduces tree topology representations to an integer. This representation is easier to work with when constructing search graphs. The representation also uses approximately half the memory of the most compressed existing formats. This is possible because the new format can only be used to store binary trees. The theoretical time to convert to and from a common tree structure is similar for both the new and existing representations; however, it is unclear which representation will allow a more efficient implementation. Finally, a search graph characterization framework, called `phygraph` is introduced that implements the new tree format as well as addressing the shortcomings of the visualization systems implemented previously. This framework is designed to ease the analysis of large phylogeny search graphs under a wide variety of different datasets, since many properties of the space will vary with the data being tested. The framework is implemented in a modular fashion so that many types of analysis can be performed. The implementation is also tested on a number of real datasets, and results are presented.

Chapter 1

Introduction

1.1 Motivation

A *phylogenetic*, or *evolutionary tree*, is a graphical representation of the evolutionary relationships between *species* or *taxa*. A phylogenetic tree is also referred to as a *phylogeny*. Species at the tips of the tree represent extant taxa, while the internal nodes above them represent inferred ancestors that, through *speciation events* (the branching points in the tree), gave rise to the existing taxa. The idea of phylogeny dates back to Darwin who stated that "the affinities of all the beings of the same class have sometimes been represented by a great tree" [9] and whose "Origin of Species" included one of the earliest depictions of such a tree.

An enormous amount of scientific progress has been made since Darwin's time. In the past 40 years, the development and refinement of sequencing technology have yielded a powerful new tool for investigating evolutionary history. DNA and protein sequence data have become the standard for phylogenetic inference, reducing to a large degree the dependence on scant physical and historical evidence that previously existed. At the same time, the modern computational era has given us an efficient means for making sense of and better understanding the implications of this enormous amount of new genetic information. While phylogenetics has been developed since

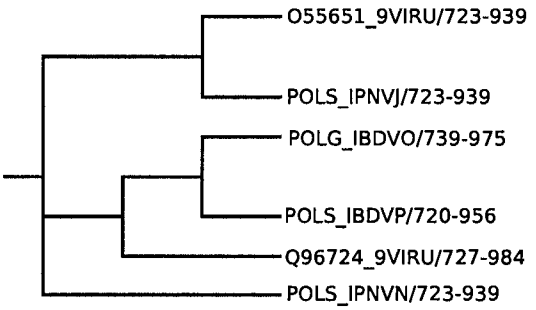


Figure 1.1: Birnavirus VP3 protein phylogeny from PFAM [2].

Darwin, "statistical, computational, and algorithmic work on them is barely 40 years old" [16]. These efforts have been guided by our understanding of evolution, but their results have also informed and contributed to our basic understanding of evolution. This developmental process has contributed to the fact that today's understanding of evolutionary theory is much more subtle, complex, and detailed than it was 140 years ago.

Some implications of our improved understanding of evolution actually pose a challenge to the basic metaphor of evolutionary trees. For one, an underlying assumption of a phylogeny is that all the species it represents are *homologous*, that is, they are related by evolutionary *descent*, where genes are passed on strictly from ancestors to offspring. There is an increasing recognition, however, that other methods of genetic transfer, for instance *lateral gene transfer* events, do occur, especially in single-celled organisms. This discovery has led to the development of new tools such as phylogenetic networks for representing the genetic relationships between organisms where lateral gene transfer events are thought to have potentially occurred in their past. Another problem is that the use of sequence data only from existing species, and using current algorithmic and statistical tools, may not be enough to accurately infer true evolutionary histories in many cases. Rediscovering the past accurately is always difficult when historical data is insufficient, and few methods exist for verifying that newly formulated evolutionary hypotheses hold true. New algorithmic methods may offer hope, but the integration of all possible data sources is increasingly seen as being of great importance. Integration of expert human knowledge into the inference process is being undertaken, as is the integration of other physical cues. An example of this is the current growth in the fields of biogeography and parasitology. These new developments have added new components that biologists may consult in their research; however, the standard sequence-based phylogenetic tree remains an extremely useful and fundamental component of most such analyses.

Some of the most heavily used and highly respected tools for standard phylogeny search include parsimony methods, maximum likelihood (ML), and Bayesian approaches. We focus here on Maximum Likelihood. Most of the experimental work will deal with protein data. There is also the question of what search operator to apply in order to permute trees into the next iteration of potential best trees that

we will examine. There are many ways of permuting trees. Typical rearrangements include nearest-neighbour interchange (NNI), subtree prune and regraft (SPR), and tree bisection and reconnection (TBR). More elaborate methods exist, for example using genetic algorithms [25] or methods based on partial likelihood information, such as SNAP [34], though these more recently developed methods are so far more seldom used. SPR and NNI are considered as the standard operations. Furthermore it has been stated that "perhaps the most standard framework of all is maximum likelihood" [16]. For this reason we will focus specifically on the interaction of SPR and NNI under ML phylogeny search here.

Due to the difficulty and high likelihood of error inherent in phylogeny reconstruction, the area remains a focus for a large amount of new research. The development of algorithms and heuristics for improving phylogeny search is an ongoing effort. For a given set of taxa, the set of all possible tree topologies with this many taxa can be thought of as the "solution space," wherein the tree representing the true evolutionary history can be found. Under this metaphor, the problem of phylogeny search becomes that of picking a point in this space (usually a tree of first approximation), and performing rearrangements on this tree in order to move to new trees in the space, ideally finishing at the tree representing the true evolutionary history. A run of the search algorithm will trace a path through the space. Unfortunately, the size of this space (the number of trees) grows factorially in the number of taxa in the phylogeny, and in many instances the search algorithms are computationally intractable. Many heuristics have been developed in attempts to obtain a reasonable answer regardless. The process of formulating and testing new heuristics and algorithms can be expensive and prone to error, with many new approaches not resulting in any significant performance improvements over existing methods. Furthermore, many potentially good solutions are not implemented or tested due to a poor understanding of the nature of the space that is being searched.

One potential solution for this problem is to investigate, characterize, and visualize the properties of this search space. Features found through such characterization could potentially be of use when deciding on techniques and heuristics to use for efficiently searching the space.

1.2 Related Work

There are two basic approaches that will be used here to characterize the phylogeny search space. The first involves modeling the space using concepts from the discrete mathematical field of graph theory, including statistics for summarizing graph features. The second generally involves creating visual representations of either the graphs from the first approach, or of important features of these graphs.

Concepts from graph theory have a long, rich history of use in problem characterization in other domains. Graphs are so fundamental a concept that they "can represent nearly everything we encounter in life, cities interconnected by highways, the national power grid, ecological structures, the social networks within which we communicate, etc." [32]. Algorithms on graphs have been developed by mathematicians and computer scientists for hundreds of years, and many of these tools have been of great use in identifying important features of graphs and the underlying domains that they model.

When applying graph theory to phylogeny search space, trees are represented as vertices, and are connected by edges to other trees if an application of a search operator can transform the tree into these others. Charleston [7] was the first to use graph theory to "introduce a coherent language base for describing and working with characteristics of combinatorial optimization problems" and apply it "to an instance of the phylogeny problem." Concepts such as landscape, hill climbing, and local optima, among others, are formally defined in his work. These concepts are then used to characterize the landscape for phylogeny search using a nine-taxa skink dataset [7]. The results in this study showed that "such descriptions offer insights into the way in which the nature of combinatorial optimization problems may change under data transformations" and "we can also determine characteristics of the original data set which may not be immediately apparent from the results of heuristic searching or bootstrapping techniques."

The second approach is the creation of a visual representation of the phylogeny search space. Human faculties of cognition and perception are not well suited for dealing with the excessively large quantities of data that are available in many problem domains, and the amount of "information a user can examine and analyze in a short space of time is very limited" [33]. Scientific visualization, which has been around

since scientists first used images and graphs to represent their work, is a field of study in its own right, and has made a large number of contributions of methods and tools for maximising the amount of and speed at which information can be understood. These tools will be of interest here, as phylogeny search space is one application where the quantity of information is great, and could benefit from the tool that gives the best insights to how the search progresses and how it could be improved.

A few examples already exist of the application of visualization techniques to the phylogeny search space problem. The Tree Set Visualization module [22], known as `treecomp`, for the evolutionary analysis package Mesquite [26], is probably the best known visualization effort. Many phylogeny programs will output not one, but a large number of trees that could all potentially be of interest to the user. TreeSet was developed for visualizing these entire *sets* of trees, and helping users make informed decisions when constructing consensus trees, for instance. This framework also lent itself to the display of phylogeny search characteristics. Briefly, TreeSet finds the distances between each pair of trees in the set using the Robinson-Foulds (RF) distance, which is defined in Chapter 3. These distances may not be compatible with a two-dimensional representation, and therefore a multidimensional scaling (MDS) technique is used in order to allow for their representation in 2D. The final step is to colour the points according to the respective tree score (trees are scored according to a given model of evolutionary inference).

Phylogenies are typically represented in the parenthetical Newick string format [14]. Under the Newick representation, adjacent nodes are separated by commas, and subtrees are contained within parentheses. Part of a tree with taxa labeled A, B, and C could be represented as (B,(A,C)). Newick representations can be extended to include branch length information. Newick strings are a somewhat clumsy tool as a computational label for phylogeny graph nodes. Machines require unique numerical identifiers, and it is preferred that these identifiers be as space efficient as possible if the number of nodes to be stored is large. With Newick strings, the same tree topology can be represented in many different ways. Also, a string of characters is not particularly space efficient. Since we wish to scale our analyses up to trees with the maximum possible number of taxa possible, the choice of tree representation becomes

quite important. One possibility is to use hash tables to transform trees into numbers, though complications may arise when chaining and collisions occur. Work in space-efficient data structures shows that parenthetical tree formats such as Newick strings can at best be represented using two bits per node [23].

1.3 Contributions

In this thesis, we build upon the visualization, graph characterization, and basic representation of the trees in the phylogeny search space. We first extend existing approaches for visualizing the search space. This is done by implementing the first known system for plotting the space in three dimensions. This system also uses SPR distances to represent spaces between trees in the visual. Areas for improvement are found and then resolved in a second implementation, which gives greater three-dimensional perspective and allows for much more information to be displayed about the space. The implementation of these systems also highlights a number of further areas where improvements could be made. One such area is the choice of lookup representation for storing and referencing phylogenies. A new, efficient representation is introduced and implemented that reduces the overall size of the search graph and eases generation of the search graph. A second set of improvements are implemented in a final program, called phygraph. This program is designed to perform all steps of search space analysis in a modular, extensible fashion so that many search space analyses can be consistently generated and reliably compared. This program is then used to characterize the search space for several real datasets, and results are presented.

The implementation and testing of the first two visualization systems is described in Chapter 3. The initial system presented a 3D view of the search space, but was constrained by the impracticality of calculating large SPR distances between phylogenies with many taxa. Further, the representation gave little perspective information, and the features of the space were found to be often difficult to discern. The next implementation used an existing 3D rendering package to give better perspective, and was designed to work on phylogenies of a restricted size. This restriction allowed for much more information (all the trees in the space) to be shown. It also allowed all the features of the space to be known, as well as allowing SPR distances between any pair of trees to remain small and therefore efficient to calculate. The visuals were found

to make the structure of spaces more readily apparent. They also made clear the fact that there are many possible search space representations, each dependent also on the particular dataset under investigation. These and other findings during construction and testing of this implementation motivated the work on the phylogeny lookup representation and the creation of `phygraph` that are presented in later chapters.

Chapter 4 presents work relating to the phylogeny lookup representation that is used. Existing formats are based on the Newick standard, and have a limit in how compact their representation can be made. Work done in generating the complete search space (enumerating all trees) brought about ideas for a different method for representing phylogenies. The sequence of steps used to arrive at a given tree within the enumeration is the primary feature that is contained in this tree's representation. This representation is found to significantly improve upon the theoretical storage limits of the existing Newick representations, thanks to its restriction to a certain class of tree (binary). Three variations on this representation are given, as well as the algorithms required to use and convert between them. These algorithms are implemented, and theoretical and real runtime results are given.

Chapter 5 returns to the larger problem of characterizing phylogeny search. As the entire set of trees is large, even when each tree has relatively few taxa, visuals can become crowded and information lost. The use of algorithms for summarizing the features within search space is therefore emphasized in this chapter. Since features of the space are found to vary with the dataset used, creation of a consistent framework for characterizing the search space so that different spaces can be meaningfully compared against one another is also desirable. Search space creation also consists of a number of components, such as scoring the trees, calculating SPR distances, generating the search graph, and traversing and examining the resulting graph. A final goal is to make these components more modular within the final implementation. This allows for easier extension of the system to new types of search space analysis and better program understanding. It also allows for the most computationally intensive operations to be run separately, for instance on a parallel cluster, and for the results of previous runs to be used again in order to speed up a modified analysis. A system that meets these goals, `phygraph` is designed and implemented, and described in that chapter. This system is then used in the analysis of a number of real datasets.

In Chapter 2 we present background material. Chapter 6 gives a summary of results, conclusions, and future work.

Chapter 2

Background

We now introduce in further detail a number of concepts that will be necessary background knowledge for later chapters. The first section deals with concepts from molecular biology and statistical and algorithmic phylogenetics that are required for understanding how genetic information is gathered from organisms and how phylogenies are then inferred from this data. The second section deals with the relevant areas of discrete mathematics that will be used. These include graph theory, graph-based phylogeny definitions, measures of the number of trees that exist in the search space, and definitions necessary for work in exploring this space.

2.1 Biological Sequences and Evolution

Through his experiments with pea plants and his careful statistical analysis of the results, Gregor Mendel published a paper in 1865 that led to him becoming the father of modern genetics. The results of his studies led to a clear distinction between the *genotype*, or hereditary makeup, and the *phenotype*, or resulting physical characteristics, of organisms. This was done despite the fact that the physical form of genes wasn't even known until work by Franklin, Watson, and Crick finally elucidated the structure of DNA in the 1950s. DNA strands, made up of the four bases A, C, G, and T, are now known to be millions or even billions of bases long, and they encode enormous amounts of information. Genes themselves are regions within the DNA that, through a series of transcription, splicing, and translation steps, can result in proteins. These protein strands are then modified and fold into their final, biologically active form. Proteins perform most of the real work within cells, and therefore can be thought of as a first embodiment of phenotype. However, after the transcription step, the encoding for proteins is made up of amino acids, of which there are twenty kinds, referred to using twenty different letters of the alphabet, and they too can be quite long. Therefore, proteins are themselves also a form of genetic information that

can be used to infer evolutionary history.

A large amount of DNA exists which does not result in the generation of any proteins. These regions are thought to be of little use, and many mutations may accrue in these regions, to little if any phenotypic effect. Proteins result from coding regions, which is partly why they are thought to have a higher rate of useful evolutionary "signal" in many contexts. For this reason we will mainly focus on protein information in this work. Furthermore, the final folded protein shape is often the most important measure of its utility, and hence, evolutionary fitness. Since most changes to the DNA encoding this protein may have little effect on its final structure, an analysis of the bases affecting structural change will be those of the greatest evolutionary importance. However, deducing the final folded protein shape from the amino acid sequence is a very difficult problem, and is an area of active research. As with DNA, work in traditional phylogenetics with amino acid data uses a string-based representation of the molecular sequence as input.

The effects of mutation and natural selection are broad, and vary depending on a large number of factors. In a molecular context, we do not consider the many high-level reasons for change, but model only the types of change at the molecular level that are possible. Mutation may range from small, local changes (for instance, *point mutations*) to very wide and far-reaching *chromosomal rearrangements*. The smaller the change the less likely it is to have an effect on the function within the organism. Since larger random changes are more likely to be deleterious, it is thought that most mutations are small and have little effect. Mutations include *substitutions*, where one base is replaced by another, *deletions*, where one or more bases are removed, and *insertion*, where one or more bases are added.

2.1.1 Phylogenetic Inference

The end result of phylogenetic inference is a phylogeny, such as shown in Figure1.1. The first step in phylogeny inference is to perform *sequence alignment* on the (assumed) homologous sequences. Since we propose that the given sequences have all evolved from common ancestors, we need to model the mutation events that must have occurred in order for similar ancestral sequences to have diverged into the more dissimilar set of sequences that are currently found. This involves creating a model of

the relative probabilities of substitution from one base to another, or of insertion or deletion events. *Substitution matrices* give values for the *substitution rates* based on empirical findings. Many such matrices exist, some modeling DNA and some modeling protein properties. Each matrix will give a somewhat different account of the rates at which different bases are thought to be substituted for one another. Examples of substitution matrices include JTT [24], PAM [12], and BLOSUM [19]. Sequence alignment then becomes an algorithmic process for creating proposed change sets, scoring them, and finding the one with the lowest score.

2.1.2 Distance Methods

The differences across the sites of each pair of sequences in an alignment gives a measure of evolutionary distance between these sequences. Distance methods are used to find a tree that "predicts the observed set of distances as closely as possible" [16]. Clustering methods, such as Neighbour Joining (NJ) and the average linkage method (UPGMA), are fast methods for constructing such trees that have been shown in simulation studies not to lose too much information about the phylogeny [16]. Let i and j be vertices within the tree (initially the tips, or extant sequences), and let n_i and n_j denote the number of species below vertex i and j, respectively. Then let D_{ij} denote the distance between them from the alignment distance matrix. As we join new species into groups, we replace their distance values with those of the entire subtree's value. The new distances are computed using the formula,

$$D_{(ij),k} = \left(\frac{n_i}{n_i + n_j}\right) D_{ik} + \left(\frac{n_j}{n_i + n_j}\right) D_{jk}.$$

Under NJ, the formula is: $D_{(ij),k} = (D_{ik} + D_{jk} - D_{ij})/2$. Calculation of trees via these methods is fast (running in polynomial time). However, there is some debate regarding the merits of clustering distance methods. UPGMA "has been rather extensively criticized in the phylogeny literature." [16]. Studies of NJ have also found shortcomings, and it has been determined that NJ is "useful to rapidly search for a good tree that can then be improved by other criteria" [16].

2.1.3 Maximum Likelihood

In statistics, likelihoods are calculated for different hypotheses, given sets of observed data. Here this method is of particular use as, in essence, we are attempting to find support for hypotheses of past evolutionary occurrences, given the existing sequence data. We now assume as given a set of aligned sequences, an evolutionary model of probabilities of state changes, and a phylogeny with branch lengths. For the sake of simplification, it is also assumed that evolution in different sites is independent, and that evolution across different lineages is independent. These assumptions allow for the calculation of the likelihood of the tree by decomposing it into a product with one term for each site. If we let D be the observed data with m sites, let D^i be the data at the ith site, and T be the tree, then the likelihood of the tree can be represented as:

$$L = Prob(D|T) = \prod_{i=1}^{m} Prob(D^i|T).$$

One of the best properties of ML is that it is statistically consistent, meaning that as the amount of data input grows, so too does the confidence in the result. Further, the standard deviation remains small as data grows. A pruning algorithm exists that simplifies the task of finding the optimal branch lengths; however, the problem of searching the set of topologies for that which gives the highest likelihood remains. More sophisticated models exist for performing these calculations when differing rates of evolution are allowed at different sites, a biologically plausible situation.

2.1.4 Maximum Parsimony

One of the first phylogeny inference methods to be developed, maximum parsimony (MP) is also among the most basic. The goal of MP is to determine the tree which minimizes the total number of evolutionary changes. The MP algorithm is similar to that of ML in many ways, but where ML allows for the use of a number of evolutionary models, MP uses a much simpler implicit model. Due to the simplified nature of MP, it runs much more quickly than ML; however, the search space is still exponential. The philosophical question of whether the most parsimonious tree is the true evolutionary tree is also much debated.

2.2 Mathematical Tools

Tools from discrete mathematics, particularly graph theory, are relevant to the study of phylogenetics. Not only can a phylogeny be well represented using graphs, the rearrangements on trees and the search space graph also benefit from a more formal definition using graph theory.

2.2.1 Graphs

A *graph* G consists of a set of *vertices*, $V(G)$ and a set of *edges* $E(G)$. Here we will also informally use "node" to describe a vertex. Each edge $e \in E(G)$ is a pair of the form (u, v) where $u, v \in V(G)$. Graphs may be *directed* or *undirected*, with the convention that $(u, v) = (v, u)$ in undirected graphs. If $e = (u, v) \in E(G)$, u and v are *adjacent* vertices and are the *endpoints* of e; e is said to be *incident* on u and v. The *degree* $d(v)$ of $v \in V(G)$ is the number of edges incident on v. An edge of the form (u, u) is a *loop*, and graphs with neither loops nor duplicate edges are called *simple graphs.*

A *path* in graph G is a sequence of distinct vertices $v_1, v_2, \ldots, v_n \in V(G)$ where $(v_i, v_{i+1}) \in E(G)$ for each $i \in \{1, 2, \ldots, n-1\}$. If $(v_n, v_1) \in E(G)$, the path is a *cycle.* A graph G is *connected* if there is a path between every pair of vertices in $V(G)$. The *distance* between two vertices is the length of the shortest path between them. A *subgraph* G' of graph G is a graph where $V(G') \subseteq V(G)$ and $E(G') = \{(u, v) | (u, v) \in E(G)$ and $u, v \in V(G')\}$. A connected graph G with no cycles is called a *tree.* It can be easily verified that for a tree T with $|V(T)| = n$, there are a total of $n-1$ edges in E(T). A tree is *binary* if each vertex has degree at most 3. A *leaf*, or *tip*, is a vertex in a tree with degree one, and all non-leaf vertices in trees are called *interior* vertices.

Some simple operations on graphs are now defined. *Vertex deletion* $G \backslash v$ is the removal of a vertex v and all edges incident on it from G. *Edge deletion* $G \backslash e$ is the removal of an edge e from G. *Edge contraction* G/e is the deletion of edge e and merging of its endpoints into a single vertex.

2.2.2 Phylogenetic Trees

Starting with the basic concept of a phylogenetic tree that has so far been developed, we wish to define it formally in order to prove results concerning the total number of trees in the phylogeny search space. A vertex v is called *interior* if it has degree $d(v) > 1$, and *exterior* (equivalently a *leaf* or *tip*) if $d(v) = 1$. Different from regular graph theoretic trees, phylogenetic trees are typically *labeled* trees, meaning that the tips have unique labels associated with them (taxon names), while the interior vertices are unlabeled. An *unrooted binary phylogenetic tree* T is defined to be a tree where each leaf is associated with a unique label and each and each interior vertex has degree 3. A rooted binary phylogenetic tree contains one addtional vertex, the root p of degree 2 [20]. We also define an *exterior edge* as an edge incident on at least one exterior vertex, otherwise the edge is *interior*.

We denote the *leaf set* of tree T as $L(T)$. For $l(T) \subseteq L(T)$, the subtree induced by $l(T)$ is often simply referred to as a *subtree* of T. We now give some results that help in counting the number of tree topologies. Given a tree T, let $|L(T)| = n$. For all $n \geq 2$, $|E(T)| = 2n-3$, and $n-3$ of these edges are interior. The set of all trees with $|L(T)| = n$ can be constructed "by adding one species at a time, in a predetermined order (say, the lexicographic order of the species names)" [16]. We now clarify the definition of *addition* of leaf l into an edge e. For a tree T, let $e = (u, v)$ be an edge. We now create new vertices l and w and change the edge to $e = (u, w)$. We also create two new edges $e_2 = (w, v)$, and $e_3 = (w, l)$. It is seen that leaf addition results in two vertices and two edges being added to the tree. In the unrooted case, there is only one topology with three taxa: all three are connected to a single interior vertex. There are three possible spots to add a fourth taxon, so therefore the number of 4-taxa trees is 3. The new tree has two more edges than before, for a total of five, so there are 3×5 possible trees on 4 taxa. This pattern continues, yielding the general formula,

$$t = 3 \times 5 \times 7 \times \ldots \times 2n - 5 = (2n - 5)!!,$$

where t is the total number of trees, and n is the number of tips in the tree.

2.2.3 Search Space

Graph theory is also useful for accurately defining the notion of search space that will be used. Search in complicated domains is a central focus in the field of artificial intelligence.

There exists some work dealing with the application of graph theory-based search space representations that are specific to the phylogeny problem. Charleston [7] was the first to use graph theory to "introduce a coherent language base for describing and working with characteristics of combinatorial optimization problems" and apply it "to an instance of the phylogeny problem." Concepts such as landscape, hill climbing, and local optima, among others, are mathematically defined in his work. These concepts are then used to characterize the landscape for phylogeny search using a nine-taxa skink dataset. The results in this study showed that "such descriptions offer insights into the way in which the nature of combinatorial optimization problems may change under data transformations" and "we can also determine characteristics of the original data set which may not be immediately apparent from the results of heuristic searching or bootstrapping techniques" [7].

Charleston [7] has introduced a number of graph-theoretic definitions of search space features, with specific reference to the phylogeny problem. We will be using a number of these definitions here. Those of greatest interest include the definition for *local optima*, *basins of attraction*, and *maximal steepest climbs*.

Consider a combinatorial optimization problem, C, in which the real-valued objective function z is to be maximized (or minimized). Let the set of feasible solutions be F. Consider also a symmetric relation p defined on F, where for $x, y \in F$, $x \sim_p y$ if and only if solution x can be obtained by a type p *perturbation* of solution y, and vice versa, so $x \sim_p y \Leftrightarrow y \sim_p x$. A perturbation is here defined as a map $p : F \rightarrow F$.

If $x \sim_p y$ then x and y are called *adjacent under* p. Hence, for each $x \in F$, the image set $p(x) = \{x_1\prime, x_2\prime, \ldots, x_k\prime\}$ is the set of all solutions adjacent to x under p, which for convenience is denoted $\mathcal{I}_p(x)$.

If x can be obtained from y (and vice versa) by k, and no fewer than k, perturbations of type p, then the *distance* (under p) $d_p(x, y)$ between x and y is k. We shall also write $x \sim_p^k y$ to indicate that $d_p(x, y) \leq k$, that is, that solution x can be obtained from solution y by k or fewer perturbations of type p.

The feasible solutions $x \in F$ can be thought of as vertices in a graph G_p, in which two vertices x and y in G_p are joined by an edge if and only if $x \sim_p y$. The *edge set* of G_p is $E(G_P) = \{x, y : x \sim_p y\}$.

The distance between U and $V \subseteq F$ is defined as,

$$d_p(U, V) = min_{x \in U, y \in V} d_p(x, y).$$

. If $U = \{x\}$ we shall just write $d_p(x, V)$. The *diameter* of $B \subseteq G_p$ is the maximum distance between any two vertices in B.

A *local optimum* is an x such that $z(x) > z(x\prime)$ for all $x \in \mathcal{I}_p(x)$. A *dip* is an x such that $z(x) < z(x\prime)$ for all $x\prime \in \mathcal{I}_p(x)$. A local optimum (or dip) under one perturbation need not be a local optimum (or dip) under another. A *steepest maximal climb* (under p) is a path M where for $(u, v) \in M$, v is the vertex in $\mathcal{I}_p(u)$ with highest value of $z()$. The *basin of attraction* of a local optima $b \subseteq F$ is the set $\Gamma(b)$ of vertices from which a steepest climb will necessarily converge upon b.

A desirable property of a search space is that the search should not get caught in local optima.

2.2.4 Tree Rearrangement

Some of the abstract definitions of search space features given in the previous section are now applied to the phylogeny search problem. It is first noted that the set of feasible solutions F is in this case the set of all possible phylogenies on the given number of taxa. We also note that the objective function $z(x)$ that we wish to maximize will here be restricted to the maximum likelihood score, though other scores such as maximum parsimony and Bayesian scores could also be used. We now address the types of perturbations p on trees that will be used. A few such operations have been of particular interest among researchers and algorithm developers, including NNI, SPR, and TBR.

Nearest Neighbour Interchange

One of the simplest phylogeny rearrangements is *Nearest Neighbour Interchange (NNI)*. Briefly, for each internal edge e in a tree there are four subtrees incident upon e. NNI

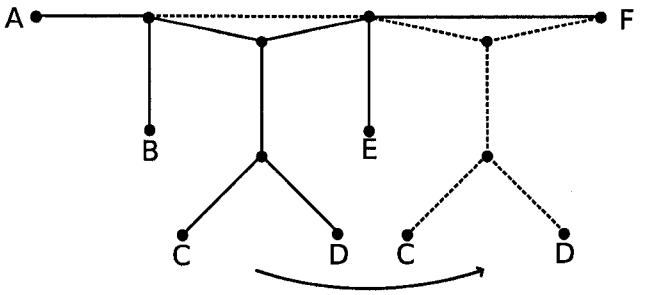


Figure 2.1: An SPR move. Solid lines indicate the edges prior to the SPR move. Some solid lines will be replaced by the dashed lines after the move.

works by swapping two of these subtrees. There are only three unique ways of configuring the four subtrees, so only two of these will be new. Let F be the set of all unrooted phylogenies on n taxa. Given a tree $t \in F$, we know that $|E(t)| = 2n - 3$, and since n of these are connected to tips, there are $2n - 3 - n = n - 3$ edges that are interior. Two of the NNIs on each interior edge result in new trees, so we see that $|\mathcal{I}_{NNI}(t)| = 2(n-3) = 2n - 6$.

Subtree Prune and Regraft

A *Subtree Prune and Regraft (SPR)* operation is an extension of the NNI operation where subtrees can be regrafted at an arbitrary distance from where they were pruned. For any interior edge e in T, e can be used as a cut edge, separating T into subtrees T_1 and T_2. There are now two cases to consider: grafting T_1 onto all possible edges in T_2, except on the one giving the same tree, and grafting T_2 onto all possible edges in T_1, except for that edge yielding the initial tree. The number of regraft edges is therefore $(2\times|L(T_1)|-3-1)+(2\times|L(T_2)|-3-1) = 2\times|L(T)|-8 = 2n-8$. For exterior edges, there will be $2(n-1)-3 = 2n-6$ possible placements of the pruned leaf. This means that the total for both interior and exterior edges is $(n-3)(2n-8) + n(2n-6) = 4(n-3)(n-2)$. The number of *distinct* SPRs will be somewhat smaller, as this process can produce the same neighbour in some cases. An example of SPR is shown in Figure 2.1.

Tree Bisection and Reconnection

A *Tree Bisection and Reconnection (TBR)* operation is an extension of SPR where, for each interior branch in T, the trees T_1 and T_2 that result from cutting this branch are treated as separate trees, and all possible connections are made between branches of each with the other.

Chapter 3

Visualization Methods for Phylogeny Search

3.1 Introduction

Visual representations of data can often be the most effective means for communicating large amounts of information [33]. Scientists and others who work with large amounts of data are continually searching for new and better ways to express the data they have collected. The goal of scientific visualization is to emphasize the features that are of greatest interest. As phylogeny search spaces contain very large amounts of information, it makes sense to apply visualization methods in order to gain insight into the nature of this space. In this chapter we develop tools for doing so.

In Chapter 2, graph-based definitions relating to search space were given. Broadly, these definitions allow for the construction of a graph object containing full information about phylogeny search space. Each tree t in the solution space is represented by a vertex in the graph, and connected by edges to those trees that can be generated by a single rearrangment on t. Each tree also has an associated objective function that is to be maximized during search (the ML score in our case). In this chapter we implement two different systems for visualizing components of such phylogeny search graphs.

As mentioned in Chapter 1, `treecomp` is an existing application that can be used to visualize features of phylogeny search space. This program clusters the trees in two dimensions, colouring them according to their likelihood score. We describe `treecomp` in some detail here, including two of its important components—multidimensional scaling and RF distance. As `treecomp` was not designed specifically for visualizing phylogeny search, some potential changes are suggested to improve visualization of such systems. Particularly in regards to viewing the entire search graph as defined above, and creating a three-dimensional representation of this search space.

Contributions in this chapter include the implementation of some of these suggested improvements. First, a prototype system is developed that gives a first look

at the structure of the search graph. This system is limited to the area near the tree with best likelihood. Results from this system motivate further improvements that are made in the implementation of a second visualization system. This one can view the entire search space for small tree sizes, as well as giving greater information on SPR neighbourhoods and better three-dimensional perspective. The recommendations made for this second system motivate the work in the next two chapters.

3.2 Background

There are some preexisting models and frameworks for visualizing phylogeny information. The majority of these tools focus on the drawing of phylogenetic trees. However, as we have mentioned, the actual tree topologies are not of interest in this work. One system that has been developed towards a more macroscopic visualization of phylogeny information is the Tree Set Visualization Module, also known as `treecomp` [27].

`treecomp` is a Java-based module for the Mesquite [26] suite of tools for evolutionary inference. Treecomp builds on the definitions of Charleston and aims to give a framework for examining the maximum likelihood and parsimony scores for large sets of phylogenies. In the treecomp module, trees are represented as points, and are coloured according to the scores, with a gradient of colours representing tree scores from low to high. The trees (coloured points) are arranged in two dimensions according to their Robinson-Foulds (RF) distances from one another. Briefly, all pairwise RF distances are calculated for each tree in the set to create a distance matrix. Next, these distances are modified via Multidimensional Scaling (MDS) in order to reduce this higher-dimensional distance data down into distances that can be represented in two dimensions. We now discuss the components of this approach in more detail.

3.2.1 Robinson-Foulds Distance

The Robinson-Foulds distance metric [31] is one of the simplest tree distance measures and is based on the idea of splits. If we let X be the set of leaves (taxa) of a tree T, with n taxa, then "a *split* of X is a partition of X into two non-empty blocks, A and B." Removing any edge e from T will induce a split of X. The splits that can be created by removing each edge of T are called the *splits of* T, which we denote by

$\Sigma(T)$ [6]. The Robinson-Foulds distance is then:

$$d_{RF}(T_1, T_2) = \frac{1}{2} |\Sigma(T_1) - \Sigma(T_2)| + \frac{1}{2} |\Sigma(T_2) - \Sigma(T_1)| .$$

The main advantage of the RF distance is that it can be calculated in $O(n)$ time — much faster than other distances. A potential disadvantage of the use of RF distance in the context of search visualization is that it relies completely on split information to measure distance. If a search operator is employed that modifies trees primarily in terms of splits, so that fewer or greater applications of the search operator always result in fewer or greater changes to the split information, respectively, then the metric will be relevant for describing the search space [13]. However, if the operator is such that one or a small number of applications of it to a tree can result in large changes in the resulting splits, the RF distance will be of limited use for quantifying the degree of search performed.

In the case of the SPR operator, relatively small changes can result in highly variable changes in split information. For example, given a longest path P in a tree T, we can prune a leaf l from one end of the path and regraft it to the opposite end of P. This will result in changes to all splits in $\Sigma(T)$ induced by removal of edges e on P. However if we perform an SPR where a leaf is pruned and regrafted close to where it was pruned, the split information will change only slightly. This large variation in splits and therefore RF distance leads us to doubt the validity of RF-based visuals for representing SPR-based search. Similarly, NNI moves in the middle of a longest path P could also result in a large number of T's splits being altered, whereas NNI moves near T's tips would have a smaller effect on splits.

3.2.2 Multidimensional Scaling

The method of Multidimensional Scaling (MDS) is a way "of embedding the distance information of a multi-variate dataset, in a reduced dimension L, by seeking a set of vectors ($\vec{r}_{i=1}^{n} \in \mathbb{R}^L$) that reproduces these distances."[13] There are many different types of MDS analysis that can be performed. "Metric MDS aims to embed the distance directly into the mapping domain." Points are created in a random distribution in the lower dimensional space, and then modified so as to reduce a measure of the

stress on these points. The stress S is the function,

$$S = \sqrt{\sum_{i<j} \frac{(d_{ij} - \hat{d}_{ij})^2}{d_{ij}}},$$

where d_{ij} is the distance between vectors i and j and $\hat{d}_{ij}$ is the distance in the estimated reduced dimensional space. The error is often minimized using a Newtonian-based method. The implementation of MDS used here is the `cmdscale` package within the R environment [30].

3.3 Developing a Visualization Framework

One goal of our visualization work is to examine the relation of the SPR distance metric to phylogenetic search. This requires that distances in the search space be either based on SPR or on a close approximation. Another goal was to view the search in three dimensions, so that areas of local and global optima could be clearly represented. We now turn to the issues of gaining a spatial sense for the ML search under the SPR metric and of generating a viewable surface that could represent this search.

In this section we develop a new kind of system for visualization of the phylogeny search space. It is similar to the `treecomp` system, however it differs in some important respects. First, it is developed specifically for use with search spaces, whereas `treecomp` is mainly intended for viewing sets of similar trees, such as those returned from search. A result of this first decision is that details of the search operator used become more important. One way to represent these search operator details is to attempt to cluster vertices in two dimensions according to their pairwise operator distances, rather than their RF distances, using MDS. A second approach is to draw lines between vertices that are one search operator application away from one another. These two methods can be used in tandem, or separately. Finally, we wish here to give 3D information about the search landscape that goes beyond the colouring system used in `treecomp`, as we hypothesize that an actual 3D representation of the data will make important features (such as local optima) more readily apparent.

3.3.1 Collapsing SPR distances

Just as with RF distance, a matrix of SPR distances can be collapsed into a lower-dimensional space using MDS. The difficulty with SPR distance, however, is that it is much more expensive to calculate than RF. While RF calculations for trees of any actual RF distance from one another all run in $O(n)$ time, computing SPR distances is known to be NP-Hard in the rooted tree case, and likely as bad in the unrooted case. Practically, the search for minimum SPR distance between two trees T_1 and T_2 proceeds by finding all trees at increasing distances from the starting tree (in a breadth-first progression) and placing these trees in a central cache with a flag stating which of T_1 or T_2 this tree originated from. Once a tree is found to have an SPR-origin in both starting trees, the minimum path is found. Hickey has furthermore found a method for *kernelizing* the two input trees that significantly reduces their size and therefore the size of the search space [21]. He has also implemented this kernelized search in his application, `sprdist`. Due to the nature of the search space, the largest minimum pairwise distances that can be calculated within a reasonable time are between 6-7 SPR moves apart. And due to the exponential growth in SPR moves, advances in hardware speeds are not likely to increase this number significantly in the forseeable future.

3.3.2 Performing Local Analyses

One solution to the problem of slow SPR distance calculation between trees in the search space is to explore *local* features of the space within small regions. Picking a tree at random, one can perform a random "SPR walk," by performing arbitrary SPR operations on the tree a given number of times.

One of the more interesting local regions within the search landscape is the area near the tree with overall best likelihood score. One question of interest is whether the landscape is "smooth" in this region—that is, whether likelihood scores strictly increase as SPR distance to the best tree decreases.

We first explore the search landscape features in the region near the best tree b that has been found after running `covSEARCH` [10] on an alignment. Given an alignment, `covSEARCH` first builds the Neighbour-Joining (NJ) tree and uses this as starting point in the subsequent ML/SPR search. We then take the output tree b, calculate all SPRs

P of b, and select a random set $S = \{s_0, s_1, \ldots, s_9\} \subseteq P$ so that $\|S\| = 10$. Next, 10 sets are created, $X = \{X_0, X_1, \ldots, X_9\}$, where $X_i = \{x_{i,0}, x_{i,1}, \ldots, x_{i,9}\}$. For each set X_i, $x_{i,0} = b$, and $x_{i,1} = s_i$. Thereafter, $x_{i,j+1} \in SPRs(x_{i,j}) \forall 0 \leq i \leq 9, 1 \leq j \leq 9$.

There are 91 unique trees $\{t_0, t_1, \ldots, t_{90}\} = X$. The next step is to create the bottom part of the symmetric matrix M where each entry $y_{i,j} = d_{SPR}(t_i, t_j)$ by running `sprdist` on each pair of trees. The x, y coordinates of each tree are then calculated using MDS by running R's `cmdscale` function on M. All tree likelihoods are also calculated using `libcov`, and the likelihood data is added as the z-axis, giving a 3D representation of the landscape. The three coordinate data are then loaded into gnuplot for visualization.

3.3.3 Results

The `gnuplot` package gives no perspective information in the 3D plot. It is only possible to make sense of this type of representation by rotating and moving the sets—otherwise it is difficult to determine the relationship between likelihood (z-axis) and tree distance (x- and y-axes) from b. It was noted that the longer the 2D distance from b, the larger the actual SPR distance, meaning that MDS performed satisfactorily on these small datasets, although further testing would be required to state this definitively. By rotating and closely examining the plots it was found that the likelihood scores decrease in general, the further the SPR distance from b. The same data was summarized also using histograms, and the same trends could be seen just as easily.

These results point to some of the shortcomings of this visualization scheme. The constraint on maximum SPR distance greatly limits the number of trees that can be examined, with the result that existing visual representations may likely be of greater use in understanding such small regions. The lack of perspective in the `gnuplot` plots causes the 3D data to be 2-dimensional and therefore difficult to make sense of. Software packages that give more perspective and 3D rendering capabilities than `gnuplot` may also be beneficial so that rotations of the data are not necessary, at least not to the degree so far required.

3.4 Visualization and Complete Enumeration

Some improvements can be made in light of these results. The first decision was to work with the *complete* set of trees of a given number n of taxa. Keeping n small causes the largest possible SPR distance between any two trees in the set to be small also. Furthermore, by considering every possible phylogeny on n taxa, full knowledge of all optima and landscape features becomes possible. This decision allows for another modification to be made, that of drawing lines to each and every SPR neighbour of each tree in the set. The motivation for this is that it could potentially allow for better representation of the SPR distances between vertices in the landscape, and give more 3D perspective. We also decided to plot the 3D data using `povray` [1], a 3D rendering program, in order to make the 3D features of the landscape more readily apparent.

The choice was also made to move away from the scripting languages, such as Perl and R, used in creating the initial "prototype" visualization pipeline, and to implement instead primarily in C++. This was done to allow for native use of `libcov` tree scoring, SPR calculation, and tree caching routines without calling external pre-compiled programs. Also, since the number of trees in the analysis is significantly larger, performance is a greater concern.

3.4.1 Experimental Framework and Implementation

Our goal now is to visualize the complete phylogeny search space under SPR and ML. In order to implement a system for performing such an analysis, we break the job into a sequence of smaller steps. The first step is to generate the complete enumeration of unrooted bifurcating phylogenies, given an alignment on n taxa. It seems that no libraries yet exist to do so. The next task is to pass each tree's Newick string to `libcov` and get its ML score. We must then generate the graph of all trees and their SPR neighbours, and associate each vertex in the graph with its corresponding tree's likelihood score. Each vertex is assigned two-dimensional coordinates to go along with its third dimensional likelihood-based coordinate. A visual representation of the graph is then created using points to show vertices and lines to show edges to adjacent SPRs.

Creating the SPR graph is itself a fairly complex task that involves a number of sub-steps. `libcov` includes a function that takes as input a tree in Newick format, and returns the set of Newick strings of all the tree's SPR neighbours. However, in order to use these strings to create a graph, we require a system for transforming strings into vertex ids. In order to accomplish this, a Patricia Tree [28] is used to associate a unique Newick representation for each tree in the enumeration with a vertex id. The details of Patricia Trees for tree storage and hashing will be further discussed in Chapter 4. Once all SPR neighbours of each tree in the complete enumeration have been calculated, the graph structure is then represented as an array of vertices, where each vertex consists of a list of its SPR neighbours (links to other vertices within the same array) and their associated likelihood scores.

The data used to test the visualization framework was simulated. This was done so that a wide range of different types of data could be visualized, under the hypothesis that different types of simulated data might have different features of interest once rendered. In order to simulate different conditions, data was simulated using the `libcov`-based `covTREE` program, which can simulate protein sequence alignments under a number of different parameters, including different evolutionary models, substitution matrices, and true trees. Data was simulated using six different tree topologies. We use two of these trees, shown in Figures 3.1b and 3.1c. These trees have been used elsewhere [29] to simulate a variety of evolutionary hypotheses.

Branch lengths in the trees were also varied, ranging between 0.25 and 1. By making all tip edges as long as possible, and all interior edges a short as possible, "star trees" were created, where nearly any tree topology could be made to fit the data by simply optimizing the branch lengths to fit this type of tree, implying that very little evolutionary signal exists.

3.4.2 Results and Discussion

We first tested this second iteration of the visualization framework by using MDS to get the x- and y-coordinates, as in the first iteration. We also used different colours to highlight edges in the povray graph rendering corresponding to SPR moves near the tree with the overall best likelihood score. The SPR neighbours of this "best" tree are coloured red, those two SPRs away in blue, and the rest are in white.

A sample visualization of the 105 potential trees on a 6-taxon alignment, simulated on the tree in Figure 3.1a is shown in Figure 3.2 In this case, MDS is seen to spread the points adequately, and it can be readily seen that there is the global optima and no other local optima.

We next worked with 8-taxa sets, comprising 10,395 different potential trees. The trees in Figures 3.1a and 3.1b were used to simulate the alignments. At first, MDS was used to arrive at the 2D coordinates used for the vertices. However, the resulting graphs presented so much data that it was difficult to determine whether MDS was performing adequately at placing the trees in 2D. For this reason, a new scheme for visualizing larger sets was devised. This scheme does not calculate all pairwise distances, but instead uses only the distance to the tree with overall best likelihood. Trees at progressively farther SPR distances from the best tree (placed at the center) were added to groups randomly distributed around concentric circles at farther distances from the center. By deleting external edges it was possible to view features close to the globally optimal tree. It is seen that many of the trees within the innermost circles have larger likelihoods than those farther out; however, it is also seen that a large number of low-scoring trees can be transformed to the best tree with a single SPR move. A few examples of the resulting SPR/ML visualizations on these scoring schemes is shown in Figure 3.3. Figures 3.3a, 3.3c, and 3.3e are rendered from the likelihood scores under the alignment simulated on the tree in Figure 3.1b. Figures 3.3b, 3.3d, and 3.3f are rendered from the likelihood scores under the alignment simulated on the tree in Figure 3.1c, which has much longer tip branches than that in Figure 3.1a. The renderings show that there is a much larger range of likelihood scores for the landscape under the tree with shorter tip branches, relative to interior branches.

A number of useful findings were made while working on these two visualization systems. First, for small graphs, such as the 6-taxon graph that is rendered in Figure 3.2, it is seen that the key properties of the graph are made clear. However, as graphs get larger and the number of connections grows, it becomes very difficult to ascertain any features of interest concerning the graph, or even to view all the connections and edges between them.

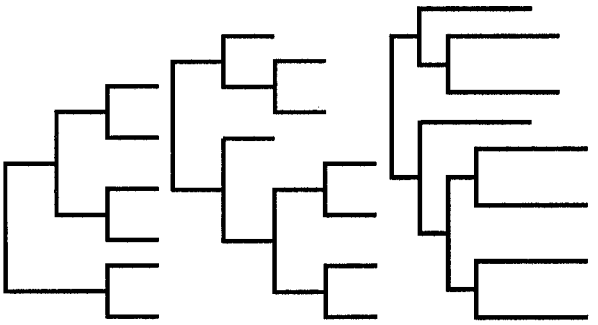


Figure 3.1: Trees used to simulate alignments. The six-taxon tree (a) has all branch lengths set to 0.25. For the eight taxon trees, (b) has all branch lengths set to 0.25, while (c) has interior branches with length 0.05, and exterior lengths 0.5.

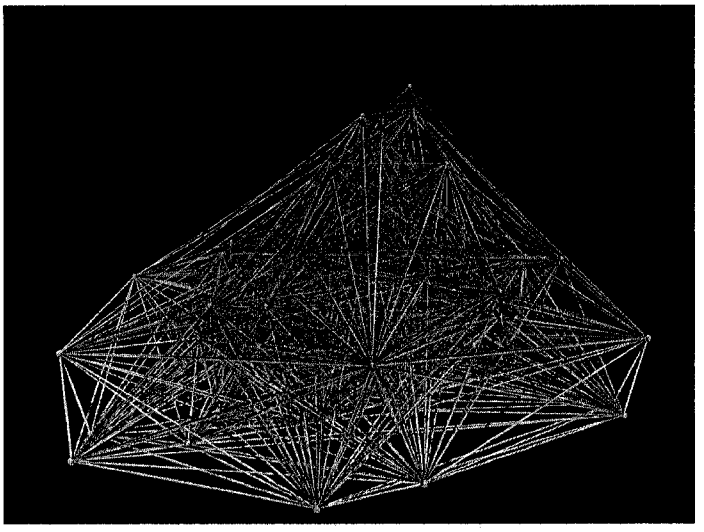
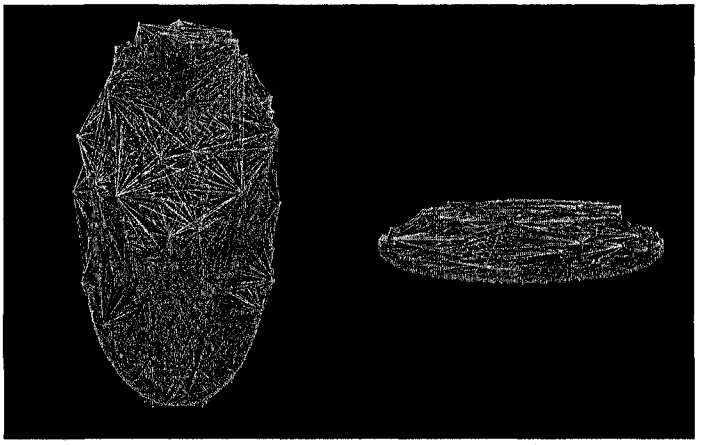
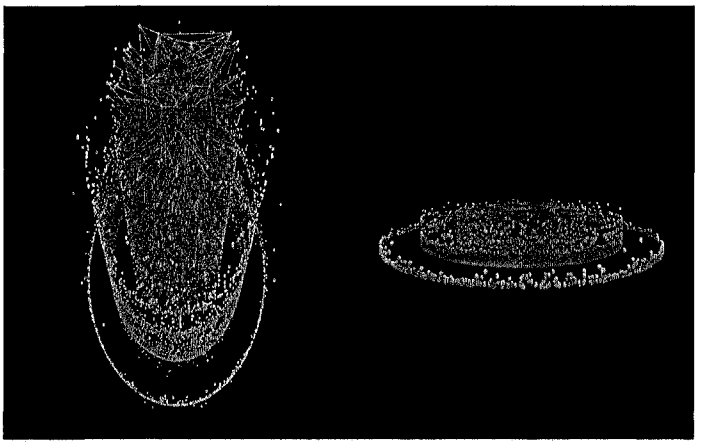


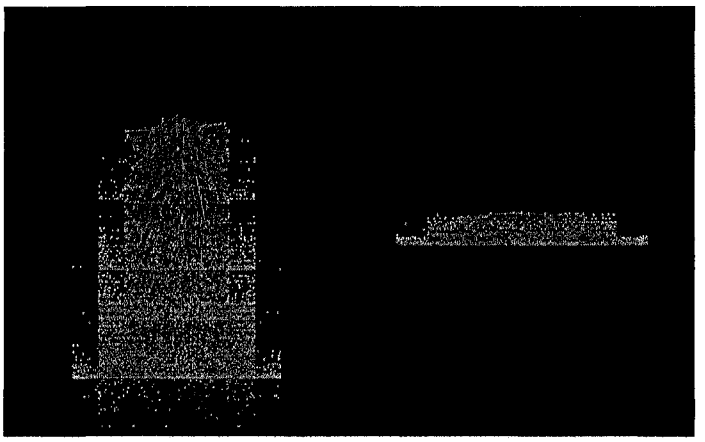
Figure 3.2: A three-dimensional representation of the entire space of trees with six taxa (105 trees).



(a) Alignment 1, all edges. (b) Alignment 2, all edges.



(c) Alignment 1, rotated view. (d) Alignment 2, rotated view.



(e) Alignment 1, side view. (f) Alignment 2, side view.

Figure 3.3: Alignments 1 and 2 consist of data simulated on the trees in Figures 3.1b and 3.1c, respectively. Points are arranged in farther concentric circles according to SPR distance to the ML tree. Red lines show moves 1 SPR to ML tree, cyan are 2 SPRS away. Rendering was performed with povray [1].

In regards to the implementation, a number of issues were discovered in this iteration. The implementation was written piece-by-piece as new components were desired in the analysis. It is very specifically designed for SPR/ML work and lacks much capacity for future generalization to other graph types. The graph structure itself uses a hand-coded representation that is quite basic, and implementation of algorithms such as breadth-first search (to discover SPR distances) has been done by hand also to use this specific, simple format. The use of tree hashing via Patricia Trees to vertex array indices is also an area that could be improved. In the next chapter we discuss a new tree storage format that eliminates the need for this data structure altogether. It was also found that `libcov` performs a rooted SPR calculation, and therefore misses some of the connections that exist in the unrooted case. The current implementation has been useful for arriving at a first approximation to the characterization of the phylogeny search landscape, and suggests a number of improvements that will be made in the following chapters.

Chapter 4

Storage and Representation of Phylogenetic Trees

4.1 Introduction

Many potential improvements can be found for the landscape visualization implementations of the previous chapter. Here we work on one particular area. As trees are modified via SPR or NNI moves, the new topologies must be used to look up the tree's location within the search space graph (the tree must already exist in the graph since it is the complete enumeration). Methods currently used to perform this lookup may include the use of Patricia Trees and CRC32 hashing. Here a new integer phylogeny format is introduced for performing this conversion which eliminates the need for any data structure beyond an integer array to be used for lookup. This method also allows for trees to be stored in a format that is potentially more memory-efficient than even the most highly compressed Newick representations.

In this chapter we describe in detail two approaches for performing the conversion from Newick strings to the memory addresses of associated tree objects. Both approaches arrive at different methods for solving what is essentially the same problem. The first is the Patricia Tree, used under `libcov` and `covSEARCH` to store the trees that have already been found, and remove them from further consideration during search. The second is the combination of a character array and the CRC32 hashing scheme used in an SPR distance calculation implementation.

We next introduce the algorithm for enumerating all trees, and use this algorithm to describe a different method for representing phylogenies. We then give the algorithm for working from a stored tree topology to its representation in the new format, and compare this algorithm's theoretical and implemented runtime with the same algorithms for Newick strings. Theoretical memory requirements for storage of the different representations introduced are also discussed.

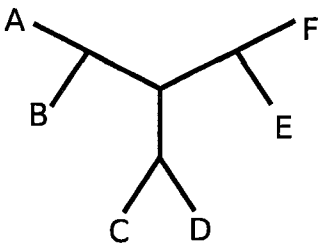


Figure 4.1: A tree with Newick representation ((A,B),(C,D),(E,F)), among many other equivalent Newick strings.

4.2 The Newick Tree Representation

The Newick format [14] is the standard text-based method for representing phylogenies. Under this format, a phylogeny t is represented primarily with brackets, commas, and taxon names. Internal vertices are represented with a pair of matching parentheses, with vertices below them placed between these parentheses, and separated by commas. The unrooted Newick representation has two commas at the highest level, between the first opening '(' and last closing ')'. The rooted Newick representation has only one. Branch lengths can also be included in the Newick string after colons, which are inserted after each closing bracket and leaf label.

One problem with the use of Newick strings for caching trees is that the same tree topology can have many different Newick representations. For instance, the tree shown in Figure 4.1 can be represented as ((A,B),(C,D),(E,F)) or ((F,E),(D,C),(B,A)), among many others. This means that if we have calculated a representation of an SPR neighbour from a given initial tree representation, the SPR neighbour's representation may not match up with the representation in the cache, and therefore the tree will be marked as new even though it actually already exists in the cache.

4.3 Existing Phylogeny Caching Methods

The problem of interest here is that of associating a phyologeny's topological information with a vertex id or address within the search graph. This requires that topologies be represented consistently and uniquely. This problem has been solved for phylogenies within at least two different contexts [10], [20]. First, trees are often stored in a cache during phylogeny search, so that the same trees are not evaluated multiple

times. Second, trees are also cached as part of applications for calculating the minimum tree distances between a pair of trees under a given rearrangement operation. We now review examples within each of these two domains.

Note that we define the *linked structure* of a tree as the internal, working data structure that a software application uses. This linked structure is generally a collection of vertex objects where each vertex contains pointers to the vertices that they are adjacent to. The linked structure is the object upon which most of the real work of phylogenetics is performed, such as calculating the ML score. We define the *lookup representation* of a tree as any representation that is used in order to quickly identify the tree, and resolve the memory address for its topology information.

4.3.1 Phylogeny Search

Phylogeny search programs may evaluate many hundreds of thousands of trees each time they are run. In order to ensure that the same trees are not repeatedly searched, programs such as `Phylip` [15] and `covSEARCH` [10], all keep a cache to store phylogenies and their associated information.

As mentioned in Chapter 3, `covSEARCH` uses a Patricia Tree [28] to store phylogenies encountered while searching. A Patricia Tree is a data structure that stores a set of strings by creating nodes for each common prefix within the set. Since each common prefix is stored exactly once, only the differences between strings are stored, resulting in the potential for a large savings in memory usage. Since Newick representations can differ for the same underlying topology, the tree is re-rooted before being compared against the Patricia Tree. Re-rooting is described in the following section.

4.3.2 SPR Distance Calculation Software

Another area where phylogenies are cached is when searching for the minimum rearrangement distance between a pair of trees, specifically under NNI, SPR, and TBR operations. Hickey has developed an algorithm (and software implementation, called `sprdist`) for efficiently calculating the SPR distance between any two unrooted bifurcating phylogenies, T_1 and T_2, on n taxa [21]. The program works by first *kernelizing* T_1 and T_2, eliminating from consideration those subtrees whose entire expansion is known to be unimportant to the final SPR distance. Once this is done, a double-ended

breadth-first search in the search graph begins. Starting from T_1 and T_2, all SPR–neighbours of each are calculated and added to a common tree cache that records the tree of origin (T_1 or T_2) for the current SPR–neighbour. Once a tree belonging to either T_1 or T_2's origin group is found to have an SPR–neighbour that exists in the other's cache, it follows that a minimum path has been found between the trees, and search terminates.

The `sprdist` software differs from that of phylogeny search in that no linked structure is necessary. The SPR distance between the trees is the only feature of interest. Therefore, SPRs are calculated by performing manipulations directly upon simplified Newick character arrays. Due to the exponential growth in number of SPRs calculated as distance increases, memory usage becomes a bottleneck when attempting to calculate longer distances. So an efficient caching system is used.

The `sprdist` program reduces the standard Newick string by eliminating all but the most relevant topological information. Taxa names are stripped out and replaced with one-byte rankings of lexicographic order. Commas are eliminated. The Newick string is also *re-rooted* so that the same tree topology always maps to the same Newick string. This is done by using the convention that the tip labels with *minimum* values are placed first within *every* subtree in the Newick representation.

Each component of the representation (brackets and tip labels) is represented a single byte of memory. Brackets get two reserved byte values, meaning that the maximum number of taxa possible under such a scheme is $2^8 - 2 = 254$. An n-taxon tree has n leaves and $n-2$ interior vertices. The subtree at each interior vertex needs to be surrounded by two brackets, giving $2(n-2) = 2n-4$ bytes, for a total of $2n-4+n = 3n-4$ bytes in the entire string. Then the trees are re-rooted and then converted into ids using CRC32 hashing with chaining used to resolve collisions.

Most Compact Newick Format

It should be noted that the preceding description of efficient Newick strings could be made even more compact. This is done by first taking the further step of using exactly one bit (rather than a full byte) to represent each of the brackets '(' and ')'. Next all parentheses are grouped together at the start of a bit-array phylogeny representation. Finally, all tip values are placed in the order they would appear in

the Newick representation, after the last bracket. The tip values are stored using the minimum number of bits required to store values between 0 and $n-1$ ($\lceil \log_2(n-1) \rceil$).

4.3.3 Discussion

Each of the methods so far described for caching phylogenies is useful, though each has certain disadvantages within the phylogeny search space problem. Within `covSEARCH`, Patricia Trees are employed for storing trees seen during SPR search. Since search proceeds using SPR, many of the trees found are likely to have common subtrees, though not necessarily common prefix strings. An approach has recently been implemented [5] that stores each unique subtree within a set exactly once. Trees within the set become a set of memory references to these stored subtrees. Such an approach may be beneficial for storing sets of trees that are all within a short SPR distance of one another, as is likely the case here.

In the case of `sprdist`, the difficulty in translating the native caching scheme to that of constructing the phylogeny search space is that no linked structure is created, and such structure is required for generating phylogenetic information such as the tree's ML score. Also, under this hashing scheme, collisions are possible. A final difficulty is that, when running algorithms in parallel, each machine would require an up-to-date copy of the data structure in order to verify whether newly generated trees have already been found elsewhere.

4.4 The Enumerative Representation

The scheme that we propose to use for translating between a tree's linked structure and lookup representation is unlike those previously described. It is not based on brackets at all, but instead follows from the phylogeny enumeration process. We now describe unrooted bifurcating phylogeny enumeration, with details specific to the software implemented here.

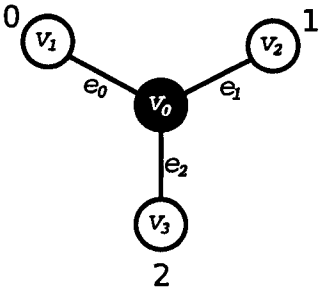
4.4.1 Phylogeny Enumeration

Felsenstein describes the method for enumerating all trees as, "a building up of all possible trees by adding one species at a time, in a predetermined order" [16]. Implementing this idea in software requires clarification of some structural details. First, a phylogeny's linked structure must incorporate vertices along with edges connecting them. Due to the use of arrays, each vertex and edge also has an implicit numbering—the array index. Though the trees are unrooted, an implicit root is still necessary so that edge indices can be consistently determined—the convention is that larger-valued edges are always placed closer to the implicit root as taxa are added to the tree. This becomes important when creating the lookup representation from an arbitrary linked tree. As in the previously mentioned implementations, tip labels are reassigned to an integer ranking of lexicographic order. The only three-taxon phylogeny is the base upon which all trees are built using the enumerative process. This tree, along with its implementation-specific edge, vertex, and tip labels, is illustrated in Figure 4.2a.

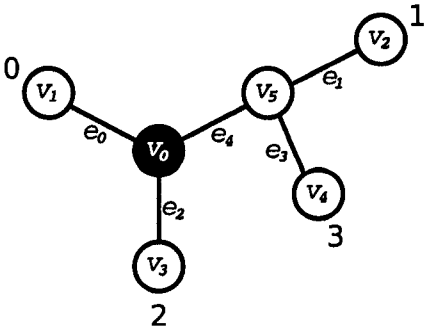
The process of "adding one species at a time", or *taxon addition* can now be implemented precisely, as shown in Algorithm 1. This process is also illustrated in Figure 4.2. Note that the value i of the tip that is being added is assumed to be added in rank order. The starting tree (Figure 4.2a) has three edges, labelled e_0, e_1, e_2, and Algorithm 1 has the net effect of adding two edges and two vertices. Due to the use of an array data structure, the two new edges will be assigned numbers larger (by one and two, respectively) than the largest previously existing edge number. We now define functions $e_{ext}(i) = e_{2i-3}$ and $e_{int}(i) = e_{2i-2}$ that return the exterior (leaf) and interior edge labels (respectively) associated with the addition of the tip with rank i. For instance, upon addition of tip $i = 3$ in Figure 4.2b, the edge to the new leaf has edge label $e_{ext}(3) = e_{2\times3-3} = e_3$, and the new interior edge on the new interior vertex has edge label $e_{int}(3) = e_{2\times3-2} = e_4$. Note that for $i \leq 2$, $e_{ext}(i) = e_{int}(i) = e_i$. These definitions will be used as a shorthand for discussing the edges associated with the addition of a given tip, in Section 4.5.

4.4.2 Enumerative Tree Representation

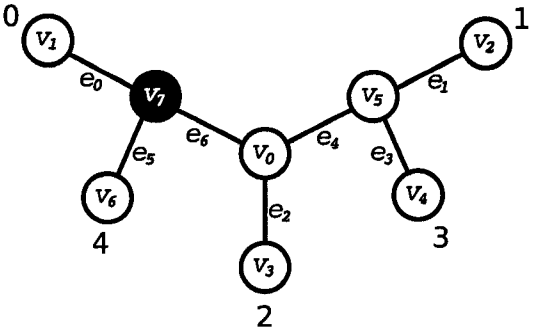
The enumeration of all trees on n taxa can now be performed following Algorithm 2. The algorithm performs the task specified by Felsenstein, of adding all species in all



(a) The initial 3-taxon tree's "linked structure". The vertex coloured black is the root.



(b) The same tree after addition of tip '3' on edge e_1.



(c) Tip '4' is added on edge e_0, requiring a re-rooting of the tree so that the root remains adjacent to tip '0'.

Figure 4.2: The 3-taxon tree as base case in enumeration, followed by two taxon additions and one re-rooting.

possible branch locations. We also found that the enumeration algorithm suggests a way to represent *any* particular tree — by noting the specific order of taxon additions that were required to build that tree. A similar idea was later found to exist in the literature [3]. That is, we can represent a given unrooted bifurcating phylogeny T on n taxa with an array a of edge labels, where the addition of taxa in rank order on these edge labels will yield T. Since the first three taxa are always added trivially, a will always have $n-3$ entries. An example of this array representation can be taken from Figure 4.2c—the array representation for this tree can be represented as $a = (1, 0)$.

Just as Newick strings are re-rooted using the existing tree caching methods, here the linked structure is re-rooted so that the implied root is always adjacent to the tip labelled '0'. The choice of where to root the tree is essentially arbitrary; however, we enforce this convention so that *the same* array representation of the tree is always found when working backwards from a given linked tree structure. Enforcing this convention when building the linked tree structure is simply a matter of swapping the parent-child edge relationships along the single edge between the old and the new root when tips are added onto the edge e_0.

Algorithm 1 ADD_TAXON(i,e,T)

1: $(u, w) \leftarrow e$
2: $v_{leaf} = T$.add_vertex()
3: $v_{int} = T$.add_vertex()
4: $e \leftarrow (u, v_{int})$
5: T.add_edge(v_{int}, w)
6: T.add_edge(v_{int}, v_{leaf})

4.4.3 Saving Some Space by Bit-Packing

The unrooted bifurcating tree on three taxa has $2 \times 3 - 3 = 3$ edges, numbered from $0 - 2$, so the first integer in the array representation is between 0 and 2. In general, the value v at index i will be $v \in \{0, 1, \ldots, 2i + 2\}$. We now introduce a scheme for reducing the memory required by the basic array representation. This scheme uses a bit-packed representation where, at each taxon index, the number of storage bits

Algorithm 2 ENUMERATE(n,T)

Input: The number of taxa n.

Output: The set E of all trees on n taxa.

```
1: if T = NULL then
2:    T = BUILD_3_TAXON_TREE()
3: end if
4: if T.num_edges()= 2n − 3 then
5:    E = E∪copy(T)
6: else
7:    k = T.num_edges()
8:    i = (k + 3)/2
9:    for e from 0 to k do
10:      ADD_TAXON(i,e,T)
11:      ENUMERATE(n,T)
12:      REMOVE_TAXON(i,e,T)
13:   end for
14: end if
```

available is the number required to store the largest possible edge value. For example, the first leaf addition (e.g., adding the fourth taxon to the three-taxon tree) has three possible edges to add onto. Therefore, two bits are required to store the actual edge chosen to represent the next edge addition. The tree will then have $2 \times 4 - 3 = 5$ edges, so $\lceil \log_2(5) \rceil = 3$ bits are required. During the process of tree building via edge addition, the number of edges to select from grows as: $3, 5, 7, 9, 11, \ldots$. the number of bits required to store this many options grows as: $2, 3, 3, 4, 4, 4, 4, 5, \ldots$, meaning that there are 2^{x-2} repetitions of numbers requiring x bits. In order to use a bit array as an array of values representing edge numbers, some conversion is required. The first edge added starts at offset 0 in the bit array and takes up two bits. The second has offset 2 and takes 3, then offset 5 taking 3, and so on. Let b be the bit array offset, $l \in \{3, \ldots, n-1\}$ the leaf label, $k = l - 2$, and $g = \lfloor \log_2(k) \rfloor$. Now we can compute $b = \sum_{j=0}^{g-1} (j+2)2^j + k(k - 2^g)$. However, this offset can be alternatively calculated by letting $k = \lfloor \log_2(n+1) \rfloor$, so that $b = k2^k + (k+1)(n - 2^k + 1)$.

While this method improves the tree storage requirements, there is still some wasted space in this representation. Not all the bits at each offset point can be used. For example, when adding the leaf with label $l = 6$, there are 6 leaves, $2 \times 6 - 3 = 9$ edges and $\lfloor \log_2(9) \rfloor + 1 = 4$ bits required to store the actual edge number used. But with four bits, up to $2^4 = 16$ numbers can be represented, so 7/16 or 43.75% of the space is wasted. In fact, the number of bits at an offset would only ever be fully used if the number of edges in the tree is a power of two.

4.4.4 Saving More Space by Creating Integers

We now modify the enumerative array phylogeny representation so that potentially no bits are wasted. This involves creating consecutive integer values for each tree within the enumeration. We illustrate the method first by example. Let the number of taxa be $n = 8$. There are $(2 \times 8 - 5)!! = 10,395$ trees in the full enumeration. We now start the taxon addition process (starting with taxon '3' (or tip '3')) on the initial 3-taxon tree. Since there are three edges in the tree, a third of the total number of trees must arise by adding this fourth taxon on each of these three edges. That is, $10395/3 = 3465$ unique trees begin with taxon '3' being added on edge e_0, and the same holds for edges e_1 and e_2. Once this first edge has been added, there are now

five edges in the tree, so the number of trees that can be created through addition on each of these edges is further reduced by a factor of five, to $3465/5 = 693$. This process is repeated until finally there is only one tree that can be generated through addition of the final taxon. By employing a consistent edge numbering of the tree, it becomes possible to assign each tree a unique number x where $0 \leq x < (2n-5)!!$.

It is now possible to use the edge number to assign an integer offset value, so that an *integer* phylogeny representation can be built. Let t be the tree value (initially 0) and imagine that we add the first tip (with label '3') on edge e_1. This skips the "first" 3465 trees created by addition of tip 3 on e_0, so we therefore let $t = t + 3465$. Now imagine the next edge is added on edge e_4, so we update t's value as $t = t + 4 * 693 = 6237$, and so on. In general, given the array representation $a = \{a_0, a_2, \ldots, a_{n-4}\}$, we can calculate t as:

$$t = \sum_{i=0}^{n-4} (a_i \times \frac{(2n-5)!!}{(2i+1)!!}).$$

This process is also summarized in Algorithm 3.

Algorithm 3 CREATE_INT(a)

Input: An array-based phylogeny representation a.

Output: An unsigned integer representation t.

1: $n = a.size() + 3$
2: $blockSize = (2n-5)!!$
3: $t = 0$
4: **for** i from 0 to $a.size()$ **do**
5: $\quad blockSize = blockSize/(2i+3)$
6: $\quad t = t + a[i] \times blockSize$
7: **end for**

4.5 From Linked Structure to Enumerative Representation

So far we've described enumerative formats (array, bit-packed, and integer) that give the sequence of taxon additions required to build different linked tree structures. We now describe methods for the reverse task of converting from arbitrary linked phylogeny structures to the array, bit-packed, or integer formats. In fact, here we

only need to consider conversion to the array representation, since it is already known how to compute the other formats given this format. Stated another way, we are given a phylogeny's linked structure, where only the tips are labelled, and must infer from this the set of edge numbers upon which each species number was added. The inference process must rebuild the edge numbers for each edge, given only the tip values. It is important that this procedure must exactly reverse the methods used when building the tree up from the enumerative form—this is the only way to ensure that the exact same topology we began with will be generated when we re-create it from the enumerative format.

4.5.1 Algorithm Overview

Assume we are given a linked phylogeny structure K_T with n tips, $2n-3$ edges, and $2n-2$ vertices. There are three central ideas that will be used in order to recreate the edge numbers and species addition order. First, the location of the edges $e_{ext}(i)$ and $e_{int}(i)$ have in general been modified least for larger tip values i. The largest valued tip, $i = n-1$, cannot have had *any* species added onto its edges, since it was the last species added. Therefore, let v be the leaf vertex of tip number $n-1$, u be v's parent, and x be u's parent. Then we know that $e_{ext}(n-1) = (v,u)$, and $e_{int}(n-1) = (u,x)$. This leads to the second central idea, that this same "largest taxon" principle applies *within subtrees* of K_T. So the algorithm proceeds by working upwards starting from the leaf vertices and moving towards the root—below each vertex larger and larger subtrees are found. A taxon number i within such a subtree is removed if both: (a) the edge number that taxon i was added onto has been determined; and (b) the edges $e_{ext}(i)$ and $e_{int}(i)$ are known not to have any larger tip values added upon them. Finally, the root vertex of K_T may have changed as the tree was being (theoretically) constructed via the enumerative method. During any taxon addition, the root can either remain stationary or move *away* from its original location. The root's movement therefore traces a path (which we denote the *root-path*) through K_T. Edges along the root-path are left unprocessed until the final stage of the algorithm.

Broadly, the complete algorithm is now represented in Algorithm 4. We now

Algorithm 4 LINKED_TO_ENUM(K_T)

Output: a, the array representation for K_T.

1: P = SETUP_TIP_SETS(K_T)

2: E = PROCESS_INTERIOR_VERTICES(K_T, P)

3: PROCESS_ROOT_PATH(K_T, E)

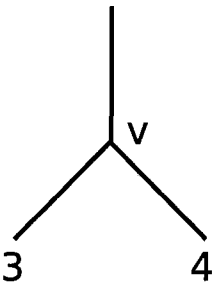


Figure 4.3: An example of the type of vertex v that would be in P at the end of SETUP_TIP_SETS(K_T). Vertex v will initially contain lists {3} and {4}. The only possibility is that tip 4 was placed on $e_{ext}(3)$ to create this situation.

describe each line of the algorithm in detail in each of the next three sections, respectively.

4.5.2 Initial Setup

In order to keep track of the tips of interest, two sets are stored at each interior vertex, each one storing the tips within the subtrees where at least one of (a) and (b) (from the previous section) have not been met. Initially, the only interior vertices that have any information about the tips in their subtrees are those adjacent to leaves. Setup therefore involves visiting each leaf vertex v with parent u, and creating a set within u containing v's associated tip number. This procedure is shown in Algorithm 5. Note that v_0 is removed from consideration. This is because the final position of the root will be adjacent to v_0, so this is set aside as an endpoint for the final root-path processing step.

After setup, P contains all vertices that are adjacent to two leaves. Let $u \in P$ be such a vertex, let $P = P \backslash \{u\}$ and let $u.s_1 = (a)$ and $u.s_2 = (b)$ be the two sets stored at u, with $u.s_1[0] < u.s_2[0]$ (that is, $a < b$). Since tip b has the larger value, we can infer that tip b was added onto edge $e_{ext}(a)$. Consider an example of such a

Algorithm 5 SETUP_TIP_SETS(K_T)

Output: P the set of vertices of K_T that have two tip sets.

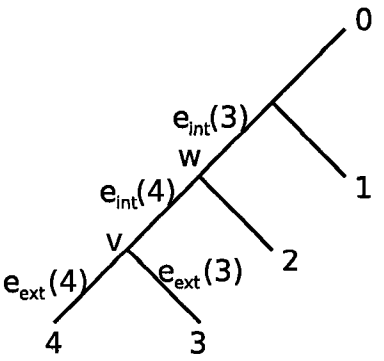
```
1: for every v in L(K_T) − v_0 do
2:    u = v.parent
3:    u.create_set(v.tip_number)
4:    if u.num_sets() = 2 then
5:       P = P ∪ u
6:    end if
7: end for
```

situation, shown in Figure 4.3. Here $u.s_1 = (3), u.s_2 = (4)$ and we infer that taxon number 4 was added onto taxon number 3's exterior edge.

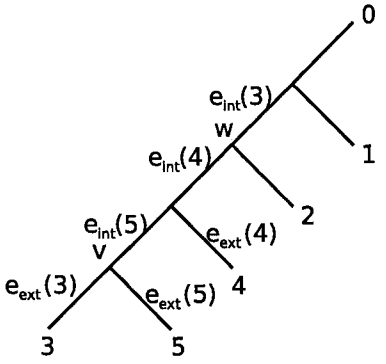
4.5.3 Processing the Interior Vertices

Not only does $b > a$ tell us where b was added, it also states that the path p between vertices $v = head(e_{ext}(a))$ and $w = tail(e_{int}(a))$ (initially of length 0), has grown by one, with the edge $e_{int}(b)$ now between them. We call such paths *addition paths* and note that they only grow in the following situations: (a) addition of a taxon j on $e_{ext}(i)$ $(j > i)$; and (b) additions of new taxa k on edges already existing on tip i's addition path (path between v and w). As an example, case (a) is shown in Figures 4.4a and 4.4b, and case (b) is shown in Figure 4.4c.

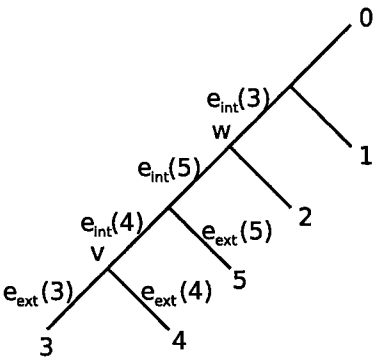
For a vertex u adjacent to two leaves (tips) with numbers a and b, $a < b$, we can infer that b was added onto $e_{ext}(a)$. It would seem that we could simply discard the value b since the edge it was added onto is now known. This is not the case. $e_{int}(b)$ is now known to be on tip a's addition path. It could be the case that a tip $c > b$ was added onto $e_{int}(b)$, lengthening a's addition path. Therefore, we must continue to store knowledge that $e_{int}(b)$ exists along tip a's addition path, in order to infer the correct edges that tips subsequently found along this path were initially added upon. We store this information by appending b's value to the set containing a's addition path information. Thus, $u.s_1$ is now (a, b), and $u.s_2$ is discarded since its unresolved tip (b) has now been resolved. Let q be the next closest vertex to the tree's (implicit) root ($q = u$.parent). The final step is to remove s_1 from u (since a tip has now been



(a) A starting tree where tip 3 was added on e_2 and tip 4 was added on $e_{ext}(3)$, causing tip 3's addition path (the path from vertex v to vertex w) to grow by one.



(b) The same tree after addition of tip 5 on $e_{ext}(3)$, causing tip 3's addition path (v to w) to grow by one.



(c) The tree that would result if tip 5 had been added on $e_{int}(4)$ instead. Tip 3's addition path also grows in this case.

Figure 4.4: Tip addition path growth example.

resolved at u) and add it as a new set at q. If q now contains two sets, this means that the addition paths for two tips exist at q, so one of them can be resolved—therefore, q is added to the list of interior vertices P that can be processed next.

We now give an example of the steps in the previous paragraph. Consider the tree in Figure 4.4c. After the call to SETUP_TIP_SETS() on this tree, the list of vertices to be processed P (vertices with two tips, not counting the root) contains only vertex v. The sets created at v are $v.s_1 = (3)$ and $v.s_2 = (4)$. It would then be determined that tip 4 was added onto edge $e_{ext}(3) = e_3$. Then $v.s_1$ would become $(3, 4)$, and $v.s_2$ discarded. Next, for vertex $q = v$.parent, a new set, $(3, 4)$ would be created at q. Vertex q would already contain one set, (5), which was added to q during SETUP_TIP_SETS(). So, q now contains two sets and can be added to P. We now label the sets as $q.s_1 = (3, 4)$, $q.s_2 = (5)$, so that $q.s_1[0] < q.s_2[0]$. The only unprocessed vertex in P is now q, so processing begins on q. Since $q.s_1[0] < q.s_2[0]$, we infer that tip $s_2[0] = 5$ was added on tip $s_1[0] = 3$'s addition path. Since the last (largest) value in $q.s_1$, 4, is smaller than $q.s_2[0] = 5$, we know that $e_{int}(4)$ existed in the tree at the time when tip 5 was added. Edge $e_{int}(4)$ will furthermore be the first edge along tip 3's addition path that is below the point q where tip 5 was first discovered. Therefore we infer that tip 5 was added on edge $e_{int}(4)$. Vertex q's parent w now has set $(3, 4, 5)$ added to it, and since w already contains (2), w now contains two sets and so it is added to P and processing continues.

Figure 4.4b shows a different situation. Initially, v will be processed similarly as before, resulting in sets $q.s_1 = (3, 5)$ and $q.s_2 = (4)$ being added to v's parent vertex q. Processing q, we find that tip 4 was added somewhere on tip 3's addition path. However, the largest value in s_1, 5, is larger than $s_2[0] = 4$. This means that edge $e_{int}(5)$ did not exist on tip 3's addition path at the time that tip 4 was added. We therefore move backwards across s_1 until we come to a value *less than* 4. The first such value is 3. Since 3 is the starting point for the addition path, we know that tip 4 must have been added on $e_{ext}(3)$. Furthermore, at q, we have discovered the existence of an edge with smaller number ($e_{int}(4)$) closer to the root than an edge with larger label ($e_{int}(5)$). Due to the convention that larger valued edges are added closer to the root during taxon addition, we can conclude that *any* potential taxon additions upon $e_{int}(5)$ must have already been found. Therefore, above q, no further tips will

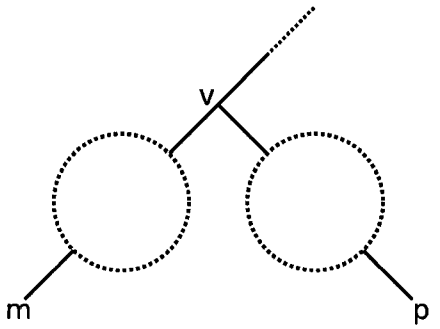


Figure 4.5: Addition path inference example. Tips m and p are the smallest tips and only unresolved tips within each subtree of v. Let $m < p$. The only edges that existed on m's path at the point when p was added are those $e_{int}(x)$ associated with tip x, $m < x < p$, or $e_{ext}(m)$ if no such x exist. Therefore any tips $> p$ can be discarded from consideration.

ever be found to have been added upon $e_{int}(5)$. This means that information about tip 5 no longer needs to be stored as part of tip 3's addition path, and so the sets added to q's parent vertex w will be $(3, 4)$.

From the previous examples we see that at vertices v closer to the root in tree K_T, we will not be comparing two single tip numbers, but rather the entire addition paths for the only two unresolved taxa below v. In general, let the sets at vertex v be the addition paths for tips with values x and y (assume $x < y$), where the edge each was added onto is unknown. One of the two unresolved vertices (y in this case) will always have the edge it was added onto resolved at each v, thereby terminating y's addition path. Consider v's addition path sets $v.s_1$ and $v.s_2$ with $v.s_1[0] = x$ and $v.s_2[0] = y$. At v, we can infer that the larger tip, y, was added somewhere on x's addition path. There are four different cases to consider:

(a) $size(s_1) = 1; size(s_2) = 1$: This is the case where v is adjacent to two leaves. This case has been described previously.

(b) $size(s_1) > 1; size(s_2) = 1$: $s_2[0]$ was added somewhere along $s_1[0]$'s addition path. Let z be the last (largest) element in s_1 such that $z < s_2[0]$. Now all tips larger than z are discarded, since they represent internal edges for taxa that were added *after* $s_2[0]$ was added. Since $s_2[0] > z$ and since $s_2[0]$ hasn't been encountered on $s_1[0]$'s addition path until this point, it must be the case that $s_2[0]$ was added on $e_{int}(z)$ (unless $z = s_1[0]$, in which case $s_2[0]$ was added on edge $e_{ext}(s_2[0])$.

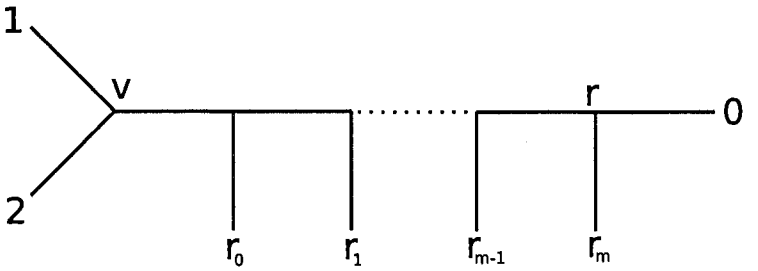


Figure 4.6: The root-path—the portion of K_T that remains to be processed once Algorithm 6 has completed.

(c) $size(s_1) = 1; size(s_2) > 1$: $s_2[0]$ was added on $e_{ext}(s_1[0])$. s_2's addition path is now terminated, and everything in s_2 (except $s_2[0]$) can be discarded.
(d) $size(s_1) > 1; size(s_2) > 1$: A combination of (c) and (b): all elements in s_2 (except $s_2[0]$) can be discarded; then perform the steps outlined in (b).

This procedure is presented also in Algorithm 6.

4.5.4 Processing the Root-Path

Consider Figure 4.2c. Processing will proceed as usual until reaching vertex v_0, where we find that $v_0.s_1 = (1, 3)$ and $v_0.s_2 = (2)$. This is the point where we can go no further until tip 0 is allowed to be processed. The reason we stop here is that we now have to account for the fact that the root's location was moved upon addition of tips between v_0 and v_7. Therefore, we must now work from the root *downwards* toward unprocessed interior child vertices until we reach v_0 (the reason for moving downward will be made clear later). If we are given arbitrary linked trees, we will not know beforehand where v_0 is. However, the vertex where the addition paths of tips 1 and 2 join will always mark this endpoint. This is because the root (as it is adjacent to tip 0) is not processed (except in the trivial case where the root is the endpoint, that is, no additions were made on edge e_0). Tips 1 and 2 therefore have the smallest values of any of the tips being processed, and will therefore eventually join at a vertex d during PROCESS_INTERIOR_VERTICES().

Between endpoint d and the root lies the root-path (which can also be thought of as the root's addition path). This is shown in Figure 4.6, with root r and endpoint v. The vertices that are offshoots along the root-path are labeled $r_0 \ldots r_m$. After

Algorithm 6 PROCESS_INTERIOR_VERTICES(K_T, P)

Output: d, the endpoint of the root-path. Also constructs much of the array representation.

```
1: for every v in P do
2:   assert( v.s1[0] < v.s2[0] )
3:   for x ∈ v.s1 do
4:     if x > s2[0] then
5:       s1 = s1\{x}
6:     end if
7:   end for
8:   z = max(s1)
9:   if z = s1[0] then
10:    a[s2[0]] = e_ext(z)
11:  else
12:    a[s2[0]] = e_int(z)
13:  end if
14:  s1.append(s2[0])
15:  w ← v.parent
16:  w.set_list(v.s1)
17:  if w.num_lists = 2 then
18:    if w.s1[0] = 1 and w.s2[0] = 2 then
19:      d ← w
20:    else
21:      P = P ∪ v.parent
22:    end if
23:  end if
24: end for
```

SETUP_TIP_SETS() and PROCESS_INTERIOR_VERTICES() have been run, the subtrees at vertices $r_0 \ldots r_m$ will be reduced to a set summarizing the addition path for the single vertex that has not yet been resolved within this subtree. We now know that this vertex was added upon an edge along the root-path, so the addition path information at each r_i can be discarded. That is, each r_i can be considered as a *single* taxon number, and all that remains is to determine the order in which these taxon numbers were added along the root's addition path.

As stated, processing the root-path works downwards from the root towards the endpoint. This is because the final taxon additions along the root-path will occur nearest to the root's final position. The final taxon additions are the only ones whose edges we can infer at first. At the root, r contains two sets, one of which is (0) and the other is (r_m) (following Figure 4.6). The only possibility is that r_m was added upon e_0, so this inference is made, and processing continues at the root's only unprocessed child vertex, c. Here, c contains sets $s_1 = (0, r_m)$ and $s_2 = (r_{m-1})$. The procedure for determining edge additions is now exactly the same as for any other addition path, and this is reflected in Algorithm 7's similarity to Algorithm 6. Processing stops once the endpoint is reached.

This concludes the description of the theoretical and algorithmic components of the enumerative formats. We now present theoretical space and time complexity results for the representations and their algorithms, before providing implementation details followed by real computational performance results.

4.6 Theoretical Results

Theoretically, we are interested in the memory requirements of the enumerative representations versus those using existing Newick formats. We are also interested in the run-time complexity characteristics of the algorithms for constructing linked tree representations from the enumerative (versus Newick) representations, as well as the same characteristics for constructing the representations from arbitrary linked structures.

Algorithm 7 PROCESS_ROOT_PATH(K_T, v)

Output: v, the endpoint of the root-path. Also constructs much of the array representation.

```
1: r.make_list(0)
2: u ←root
3: while u! = v do
4:    assert( u.s_1[0] < u.s_2[0] )
5:    for x ∈ s_1 do
6:       if x > s_2[0] then
7:          s_1 = s_1\{x}
8:       end if
9:    end for
10:   z = max(u.s_1)
11:   if z = u.s_1[0] then
12:      a[u.s_2[0]] = e_ext(z)
13:   else
14:      a[u.s_2[0]] = e_int(z)
15:   end if
16:   u.s_1.append(u.s_2[0])
17:   u.unprocessed_child.set_list(s_1)
18:   u = u.unprocessed_child
19: end while
```

4.6.1 Memory Requirements of Tree Representations

Newick Strings

We will assume that Newick strings are made up only of the characters '(', ')', ',', and string-based values with digits in the range $1 \rightarrow \lceil \log_2(n-1) \rceil$. We approximate that each character takes up one byte. There is one '(', one ')', and one ',' for each interior vertex, with an extra ',' at the topmost trifurcation, and there are $n-2$ interior vertices total. For representing tip numbers, the first 10 tips, 0-9, each require one character, the next 90 would each require two characters, and so on—in general the formula for the number of characters required to store each taxon number in the Newick string (the number of digits in the taxon number) is $d = 1 + \sum_{i=0}^{\lceil \log_{10}(n-1) \rceil} (n - 10^i)$. The extra one comes from '0' requiring one digit to represent. Overall, $y = 3(n-2) + d$ bytes are required, or $8y$ bits. While it is possible to store the integer values of taxa more succinctly, in practice Newick strings are usually stored as a simple sequence of characters.

Succinct Newick Strings

Succinct Newick strings, as described in Section 4.3.2 will require one bit per '(' and ')' ($n-2$ of each), and n times the number of bits required per tip. The total is $2n - 4 + n \times \lceil \log_2(n-1) \rceil$ bits.

Enumerative Formats

We will assume the array representation requires one byte per entry (max 256 taxa). The first three taxa are added trivially, so the representation requires $n-3$ bytes or $8(n-3)$ bits. The memory requirements for the bit-packed format are $\sum_{i=1}^{n-3} \lceil \log_2(2i+1) \rceil$ bits. The integer format requires $\lceil \log_2((2n-5)!!) \rceil$ bits.

Comparison

When enumerating all trees and storing the graph, the maximum number of taxa possible seems likely to be $n \leq 14$. Some actual memory requirement values are summarized (in bits) in Table 4.1. Between $n = 8$ and $n = 14$, the bit-packed Newick

format is between 2.05 and 2.67 times as large as the integer format. Between $n = 24$ and $n = 40$, this ratio cycles between a high of 1.84 and a low of 1.61.

4.6.2 Time Complexity of Format Conversions

We now examine the theoretical time complexity of algorithms for transforming tree representations into linked structures, and vice versa, using the different representation formats discussed.

Enumerative to Linked Structure

We begin with the array enumerative format. Building the linked structure involves creating the initial 3-taxon tree (constant time) and then performing $n-3$ constant-time taxon additions. The bound is $O(n)$. For the bit-packed format, there is the added cost of calculating bit-packed offsets. This calculation could be performed by first passing across the representation and building an array representation from it, in O(n) time, before creating the linked structure, for an overall O(n)+O(n)=O(n) time complexity.

In order to get the array representation from the integer format, however, $n-3$ divisions must be performed, each on a number with $d = \log_{10}((2n-5)!!)$ digits.

n	Array	Bit-packed array	Integer	Bit-packed Newick	Newick string
4	8	2	2	12	104
5	16	5	4	16	136
6	24	8	7	26	168
7	32	12	10	31	200
8	40	16	14	36	232
9	48	20	18	41	264
10	56	24	21	56	296
11	64	29	26	62	336
12	72	34	30	68	376
13	80	39	34	74	416
14	88	44	39	80	456
20	136	76	68	136	696
40	296	202	184	316	1496

Table 4.1: Number of bits required to store an n-taxon tree under different representations.

Division on a d-digit number has a runtime of $O(d^{1+\varepsilon})$ [4], so the overall runtime here is $O(n^{2+\varepsilon})$. However, for the small tree sizes being used here, these divisions may be performed relatively quickly. For the bit-packed array, once the array is constructed, converting it into the bit-packed format takes $O(n)$ time, so the total here is also $O(n)$.

Linked Structure to Enumerative

Given the linked structure, we now consider the time taken to calculate the enumerative representation. We assume that the linked tree data structure we use allows for constant-time access to its leaves. This capacity can be easily added to any existing linked representation by simply adding an array of n pointers to vertices. The time complexity for the calculation is determined through analysis of Algorithm 4 and the three component algorithms, 5, 6, and 7 that comprise it. Setting up, in Algorithm 5, visits each of the n tips and performs a constant-time set creation task at each one. Next, the first pass of interior vertex processing, in Algorithm 6 will visit at most each of the $n-2$ interior vertices. At each vertex, there are two pre-sorted lists—call them s_1 and s_2 such that $s_1[0] < s_2[0]$. The algorithm searches s_1 for the last element smaller than $s_2[0]$. Starting from the back of s_1, taxon numbers can be removed in constant time. Each taxon number can be removed at most once overall, so the running time is $O(n)$.

Once the array representation has been created, we can again convert it into the other formats. The integer format requires one multiplication for each of the $n-3$ edge numbers, for a total of $O(n^{2+\varepsilon})$. The array format can be built at an overall cost of $O(n)$.

Newick to Linked Structure

The algorithm for building a linked structure from the Newick string takes a single pass along the length of the string. Briefly, the structure is constructed by creating and following a new edge and vertex from the current vertex when '(' or leaf label characters are read, and moving back up the tree when leaf labels terminate or when ')' characters are encountered. The runtime is therefore $O(n)$, and this holds even with the addition of a first step where taxon labels are sorted by lexicographic order

and assigned numerical values.

Linked Structure to Newick

To go from a linked tree structure to the corresponding Newick string involves following the same preorder traversal that yields the structure from the string. Starting from the root, '(' characters are added as we progress towards the tips of the tree. At the tips, leaf label characters are added. As the traversal moves up the tree, ')' characters are added to the Newick string. The traversal visits each vertex at most three times, for a total $O(n)$ runtime.

Comparison

For realistic phylogenetic analyses, n is rarely larger than a few hundred taxa. For such values, all conversion tasks will have approximately equal theoretical runtime ($O(n)$). However, there are greater constants in front of n for the enumerative conversion algorithms versus Newick. These constants may make for large discrepancies in runtime, particularly at the small end of tree space that is of interest in this work, for instance $n \leq 14$. We therefore begin to investigate real runtimes for such trees. The first step for such tests is to implement the conversion algorithms so far described.

4.7 Experimental Setup and Implementation

The goal of the implementation is to test the running time for conversion from linked structure to enumerative representation, and vice versa, and then compare this with the same tests using the Newick representation instead of the enumerative. This necessitates implementations of: (1) the algorithms for the conversions with the enumerative representation; (2) the algorithms for the Newick representation. It was decided that the C++ language should be used, as good performance is an important characteristic since the work is to scale up to the largest possible numbers of trees.

Performance was to be tested by generating thousands of random enumerative representations, and then determining the total time required to convert them all to linked representations. It was found that random enumerative representations were easy to generate, since to do so involved simply generating random numbers within

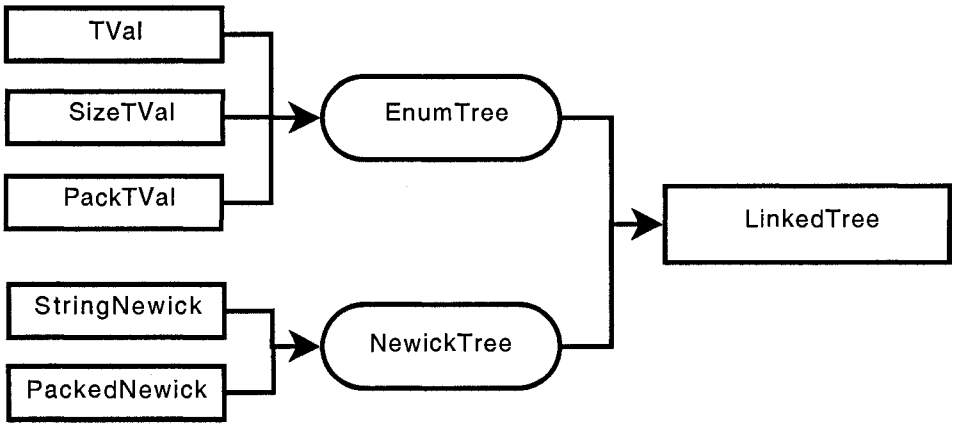
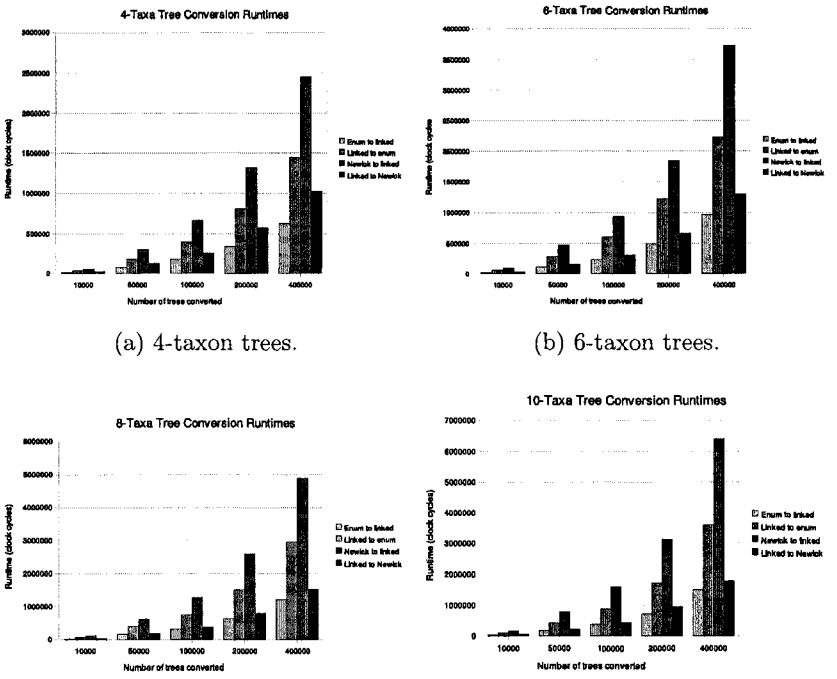


Figure 4.7: The TVal, PackTVal, and SizeTVal classes implement methods for the array, bit-packed array, and integer-based enumerative formats, respectively. Square boxes represent data and rounded ones represent conversion functions.

the range of the total number of trees. Next, the enumerative representations would be deleted and then the time required to regenerate them from the linked structures would be timed. Next, the time taken to convert these same linked structures to Newick tree representations was determined. These Newick strings were then stored, the linked structures deleted, and the time to regenerate the linked structures from the Newick strings was determined.

4.8 Results

For $n > 14$ (approximately), the number of n-taxon trees becomes larger than the size of native C++ unsigned integers. Since storing the entire search graph is only feasible for tree sizes less than $n = 14$ anyway, we will only consider conversion times for large numbers of trees smaller than this value (typically within the 7-10 taxon range). Tests were performed with trees of size $n \in \{4, 6, 8, 10\}$. There were $t \in \{10,000, 50,000, 100,000, 200,000, 400,000\}$ random trees generated, and the accumulated time to convert integer or Newick strings to/from a linked representation was recorded, averaged over 20 trials. Trials were performed on a 2.4 Ghz Intel Pentium 4 processor machine with 512 Mb RAM, running Ubuntu Linux 8.04. Figures 4.8a, 4.8b, 4.8c, and 4.8d summarize the results or these trials.



(a) 4-taxon trees.

(b) 6-taxon trees.

(c) 8-taxon trees.

(d) 10-taxon trees.

Figure 4.8: Conversion runtimes for varying numbers of n taxon trees, for different values of n. All results are averages over 20 trials.

As expected, the conversion from linked structure to integer format takes significantly longer (though less than twice as long) as the same conversion to a Newick string. Somewhat surprisingly, converting input from integers to linked structure appears to take about a quarter of the time as the same conversion starting from Newick strings. However, upon further investigation it was found that up to 80% of this time is spent simply in parsing the Newick string, and this step also involves an expensive lexicographic ordering of the taxon names. Presumably a binary format such as the bit-packed Newick format would not suffer from such an extended setup phase; however, only the integer and Newick string formats were fully implemented, as they were of primary concern during construction of the search graph structure.

When constructing the search graph, the time taken to convert from internal representation to linked tree and back to internal representation becomes important. This is because most of the work of constructing the SPR/NNI neighbour graph consists of: (a) turning the lookup representation into the linked structure; (b) calculating NNI or SPR neighbours from the linked structure; and then (c) converting these neighbours back into the lookup format. (The final step, looking up the neighbour representations in the cache, has not yet been addressed, and could have a significant effect.) In fact, since there are two new NNI neighbours for each tree, and many SPR neighbours, the time taken to convert from linked to lookup representation will be of greater importance than the conversion from lookup to linked representation. This would in fact make the enumerative representation slower than Newick for the phylogeny search program. Further testing is required to verify this.

Chapter 5

Phygraph: A Tool for Characterizing Phylogeny Search Space

5.1 Introduction

We now combine the findings from the previous two chapters into a final framework for examination of phylogeny search space, which we call `phygraph`. The system presented in Chapter 2 had *visual* representation of phylogeny search space as a primary goal. This is not true for `phygraph`—here the goals are the creation of a *graph structure* to represent the search space, and to discover features of the space through calculation of search graph statistics. A central goal of `phygraph` is to improve upon the previous implementation in terms of generality, ease of use, and extensibility of the implementation, while also improving efficiency so that larger numbers of trees could be analyzed. While the speed of conversion results from the previous chapter are fairly neutral, the integer phylogeny format does offer benefits, primarily in terms of ease of use, though also with memory usage, and is therefore used as part of the `phygraph` implementation. The decision to design `phygraph` as a framework for consistently examining *many* datasets came as a result of findings in Chapter 2—large amounts of data exist to be analyzed; the details of the search space changes with the dataset; and there are a very wide range of datasets to choose from. Further, the features of interest within the graph may vary depending on the research being conducted. Therefore, a general tool for performing search space analyses in a consistent and efficient manner is desired.

Towards the goals of generality and extensibility, `phygraph` is built as a set of modular components for performing tasks associated with all stages of phylogeny search graph construction and analysis. These tasks include: performing complete tree enumeration; performing SPR/NNI adjacency calculation; Newick, enumerative, and linked structure format conversion; scoring entire enumeration sets in parallel

or sequentially; reading and writing graph and sequence information in a variety of formats; randomizing components of the search space if desired; and calculating and outputting a variety of graph statistics. `phygraph` can perform an entire analysis at once or perform only certain components of it depending upon the command-line options that are issued to it. In this chapter we will describe in some detail the decisions that were made in development of `phygraph` towards it's utility as a general phylogeny search graph analysis tool.

Next, we describe some results obtained on real data. The type of statistics currently implemented in `phygraph` include calculation of locally optimal trees, average vertex degree, maximal steepest climb lengths, and attractor basins. Multiple trials are conducted on each of two different 9-taxon alignments from PFAM [2], and trials are performed on 9-taxon subsets of an EF-1α dataset. EF-1α genes are often conserved across a wide variety of organisms and the gene "has been identified as a potentially useful gene for studies of higher-level phylogenetic relationships" [8]. The statistics compiled for these search graphs are then contrasted with results using randomized likelihood data.

5.2 Phygraph Overview

`phygraph` consists of a set of components for performing the tasks described above. We first describe the steps associated with running a complete phylogeny search space analysis, starting from a sequence alignment through to calculation of graph statistics, and then we discuss each component within this system, how it can be run independently, and how it could be modified or extended.

Algorithm 8 shows the steps taken to perform the entire search space analysis, given an alignment. Enumeration has already been discussed, and, since we are using the integer phylogeny format, it simply consists of numbers between 0 and $(2n-5)!!$. We now discuss the main remaining components of the algorithm, including the graph structure.

Scoring Likelihoods

Likelihood scoring is performed by the `ScoreLkls` class. Currently `ScoreLkls` works on a given range of integer tree representations. Each integer is converted into a

Algorithm 8 GET_GRAPH_STATS(*Aln*)

Input: A sequence alignment *Aln*

Output: The graph statistics, as well as likelihood and graph files, if desired.

1: n = *Aln*.num_taxa
2: E = ENUMERATE(n)
3: S = SCORE_LKLS(E)
4: G = CALC_SPR/NNI_GRAPH(E)
5: G.add_scores(S)
6: CALC_GRAPH_STATS(G)

`covTree` object, and then functions within the `libcov` C++ phylogenetics library are used to score the linked tree using ML. `libcov` has many parameters for most evolutionary models and other scoring options that may be required by the user. The `covTree` object is then deleted, and the score is kept in an array, with array index given by the tree's integer representation.

Calculating ML scores is the greatest performance bottleneck in the search space analysis as the number of trees grows. Therefore, the `ScoreLkls` class can score trees sequentially or in parallel. The ability to use integer tree representations as storage indices is also helpful in parallelizing the implementation. Briefly, each processor is simply sent integer start and endpoints for a unique range of trees within the enumeration. Each processor then converts each integer within that range into `covTrees`, scores them, and discards the linked tree. Then all scores in each processor's array of likelihoods are gathered on the master processor. This is implemented using MPI [17], and other than setup only a single call to `MPI_Allgatherv()` is required to collect the likelihood scores.

Calculating SPRs and NNIs and Adding Edges

Since `libcov` was found to calculate rooted SPRs, the code for calculating NNIs and unrooted SPRs for each tree was written from scratch. For each integer-valued tree t in the enumeration, implementing SPR involved conversion of t into a linked tree and cutting each edge $e = \{u, v\}$ in the linked tree to create two subtrees, s_1 (containing u) and s_2 (containing v). Next, s_2 was regrafted on each edge in s_1 by adding a

new vertex w along this edge and connecting u to w. The analogous procedure was performed in regrafting s_1 on each edge in s_2. Each SPR was created as a copy of the entire linked tree, and then the function for converting arbitrary linked trees to integers was used to arrive at this SPR's integer value. After calling the SPR calculation function, all SPR tree values are stored in an array of integers t_{SPR}. Now for each element i of the array, we create an edge in the search space graph by calling G.add_edge(t,$t_{SPR}[i]$). Calculation of SPRs and NNIs is similar, though SPR was more complicated, especially since grafting the subtree containing the implicit root onto the subtree not containing the root can involve a number of intermediate re-rooting calls.

The Graph Structure

The visualization software in Chapter 2 used its own graph structure implementation, simply involving vertex structs with a list of pointers to other vertices. Here, a new, more powerful set of graph libraries are used—the `boost::graph` library, or BGL [11]. The Boost project consists of "libraries that work well with the C++ Standard Library" and that are "widely useful, and usable across a broad spectrum of applications" [11]. The BGL emphasizes algorithm and data structure interoperability, meaning that algorithms are data-structure neutral. Specialized iterators allow for traversal of different components of the graph, including traversals of all vertices in the graph, traversals of all edges in the graph, and traversals across all the adjacent vertices of each vertex in the graph. Any container class, from stacks to queues to arrays, from the C++ STL can be used to store the vertices or edges of the graph. Algorithms that are part of the BGL are implemented using some of the most efficient methods available and include breadth- and depth-first-search, shortest paths, spanning trees, connected components, topological sort, transpose, and vertex ordering algorithms. All algorithms work with any data structure.

Using the BGL offers a number of advantages for phylogeny search space modeling. First, using the BGL, any number of *properties* can be associated with some or all vertices in the graph. Here, properties will be used to store the likelihood scores of the given vertex's associated phylogeny. Links to neighbours with the best likelihoods can also be stored as a property within each vertex. This allows for the overall steepest

path length information to be easily calculated for all trees at once, for example. Second, the abundance of algorithms already implemented in the BGL allows for certain graph properties to be trivially extracted from the graph. For instance, if the diameter or girth of the search graph is a property of interest, these values can be immediately calculated for the given graph since these functions are already part of the BGL. Also, the BGL has functions for writing graph structure to a number of different file formats, allowing the data to be loaded into a variety of other graph analysis packages.

Another consideration is that an appropriate choice of underlying data structure to be used within the BGL graph can result in useful referencing properties. Here we choose a C++ vector as data type for storing the vertices in a BGL graph G, which allows for each vertex in G to be referenced by an index number. Now, for each integer tree value i in the enumeration, we simply store the tree properties and edges for tree i in the vertex at index i. This means that now for all tree i's (integer represented) SPR neighbours, i_{SPR}, the edges in G can be created simply by calling G.add_edge(i,$i_{SPR}[j]$), $(0 \leq j \leq i_{SPR}.\text{size}())$. (The same is true for NNI, or any other rearrangement operator). That is, the tree's value *is* the vertex address.

A final issue in choosing the BGL is performance. The simple graph structure used in Chapter 2 (a linked list) can likely store a slightly larger graph than can be stored on the same machine using the BGL, due to some added library overhead. However, there are some significant performance gains using the BGL. First, all the algorithms developed for the BGL are generally implemented with performance in mind. Second, a parallel version of the BGL is being developed "offering similar data structures, algorithms, and syntax for distributed parallel computation that the BGL offers for sequential programs." [18]. To examine larger search spaces, the transition to a distributed parallel graph object is made much simpler due to use of the BGL than with a manually implemented graph structure.

Calculation of Graph Statistics

There are many different graph statistics of interest that could be implemented here. So far, the statistics that have been implemented include those for determining the local optima, steepest maximal climb lengths, and basins of attraction. In order

to simplify the calculation of these statistics, we first calculate some simpler ones. Visiting each vertex $v \in V(G)$, we visit each edge incident on v, and keep track of the vertex u such that $(u, v) \in E(G)$ and u's likelihood score is maximal. We call u v's *best neighbour*. Once this u has been found it is stored as a vertex property of v's (v.bestNbr). We also keep a list at u, (u.amBest) of the vertices (such as v) that have u as their best neighbour. For such pairs of vertices, we also flag those vertices v where v's likelihood is greater than that of v's best neighbour u, and store a pointer to v in an array L. Once all vertices have been processed, L will contain a list of the local optima of G. The $v \in G$ with the greatest likelihood score is the global optimum v_{opt}. Now the basins of attraction and steepest maximal climb paths can be easily and efficiently determined. For each $l \in L$, let L_1 store the vertices with best neighbour l (these are the vertices already stored in l.amBest). The vertices in L_1 correspond to those trees that are part of l's basin of attraction and have steepest maximal climb length of one to this optimum. L_2 can then be determined by visiting each $l_1 \in L_1$ and adding the vertices that have l_1 as best neighbour to L_2, and so on. Each further L_i is seen to store the vertices in l's attractor basin with steepest maximal climb length i from the basin's optimum. This can be thought of as a breadth-first search on the subgraph H of the search graph G where V(H) = V(G), but only one edge is defined in H for each vertex v—the edge (u, v) where u is v's best neighbour.

File Input and Output

Algorithm 8 shows a run of the full `phygraph` analysis, from start to finish, with all likelihoods scored and the entire SPR or NNI graph being generated on the fly in order to build the combined graph structure and calculate search space statistics. For large analyses, this is inefficient, however. Generating the SPR graph (without calculating likelihoods) can take an hour of computation time on a 2.4 Ghz CPU. Scoring likelihoods on an alignment on nine 350-site sequences can take two hours of computing time on a 16-node cluster. For this reason, `phygraph` also allows for the graph and/or the likelihood data to be read in from a file. This allows for the same likelihood data to be used quickly with many different search operator graphs, or for scores associated with many different datasets to be quickly analyzed on a given graph type. For instance, the SPR and NNI adjacency graphs are independent of the

dataset that is used, so each n-taxon SPR graph need only be generated once, and then it can be used with any likelihood data that is desired.

The file format used to store SPR and NNI graphs is very simple. The first line gives the number of taxa, and each subsequent line starts with a line index i and each subsequent entry on the line is the integer value of an SPR or NNI neighbour (respectively) that is adjacent to tree i. Likelihood values consist of the number of taxa n, followed by $(2n-5)!!$ likelihood scores, each score at line $i(+1)$ giving the ML score for tree i.

5.3 Experimental Setup and Results

We now give some results on real alignment data and the associated phylogeny search graph statistics returned by `phygraph`. Given the statistical properties that have so far been implemented, the main features of the search spaces that can be determined using phygraph include: (a) The presence of local and global optima within the space; (b) the sizes of the basins of attraction for the optima; and (c) the mean maximal steepest climb length to the optima. Nine-taxon datasets are the largest tree size that is practical to use as the likelihood scores for all trees within the full enumeration (135,135 trees) can be calculated on `pepito`, a 16-node cluster, in 2 hours. Three different real alignments were used. Two came from the protein alignment database PFAM [2], and the third is a ten-taxon alignment of the EF-1α gene across a number of different taxa.

To analyze these datasets, ten trials were performed upon each one. For each trial, nine taxa were selected from the starting alignment and turned into their own alignment. Then, `phygraph` was run upon each alignment to determine the three graph statistics of interest. The two PFAM alignments, PF00078 and PF00516, contain 158 and 24 taxa, respectively, so random sampling was used to obtain the 9-taxa subset. The EF-1α set has 10 taxa in total, so all 10 possible 9-taxon subsets were used for the trials. Under the NNI operator, the local and global maxima, as well as attractor basin sizes for each, are summarized for each trial on each alignment in Tables 5.1, 5.2, and 5.3. Under SPR, all trials in all datasets except one found that no local maxima exist other than at the global maximum point. The only exception was the 5th trial on the EF-1α alignment. In that case there were two maxima, with

109846 trees moving to the global max along a steepest climb, and 25289 moving to the only other local optimum.

Trial No.	No. Maxima	Global Max Basin Size	Local Max Basin Size (Sum)
1	1	135135	0
2	1	135135	0
3	6	132619	2516
4	6	100051	35084
5	2	135132	3
6	3	135117	18
7	7	33304	101831
8	2	125865	9270
9	2	132275	2860
10	2	101090	34045

Table 5.1: Local maxima and basins of attraction counts under NNI for random 9-sequence subsets of 158-taxon PFAM alignment PF00078 (average sequence length 167.2).

Trial No.	No. Maxima	Global Max Basin Size	Local Max Basin Size (Sum)
1	15	104805	30330
2	18	131726	3409
3	19	132495	2640
4	45	49193	85942
5	28	88452	46683
6	61	68265	66870
7	5	63225	71910
8	1	135135	0
9	1	135135	0
10	5	70452	64683

Table 5.2: Local maxima and basins of attraction counts under NNI for random 9-sequence subsets of 24-taxon PFAM alignment PF00516 (average sequence length 89.9).

We now present some results using the NNI and SPR graphs using random likelihood data. On 9 taxa, $(2 \times 9 - 5)!! = 135135$ random doubles were generated uniformly within the range -5000 to -1000. These were then assigned as the likelihood data under the NNI and SPR graphs. 10 sets of random likelihood numbers were generated. For NNI, an average of 10383.3 maxima were found, with only 13.5 vertices belonging to the global maximum's basin of attraction. For SPR, an average

of 1038.2 maxima were found, with 302.3 vertices on average in the global maximum's basin of attraction.

5.3.1 Discussion

More local maxima are found for the NNI operator, as one would expect, considering that on 9 taxa there are 12 neighbours under NNI and 132 under SPR. Further, the number of local maxima is seen to vary with the dataset that is used. The EF-1α set clearly has some properties that cause the existence of a larger number of local optima in the space. The presence of a local maximum within the 5th trial on the EF-1α dataset shows that local maxima do exist under SPR, even on smaller search spaces, though more testing is required to confirm this. It is particularly interesting that the removal of a single sequence from the alignment yielded this result. `phygraph` is seen to perform well for identifying local optima and basins of attraction, although further statistics should be added to the program in order to determine more about the nature of the space.

Trial No.	No. Maxima	Global Max Basin Size	Local Max Basin Size (Sum)
1	35	31907	103228
2	86	22911	112224
3	49	13035	122100
4	23	21549	113586
5	48	20081	115054
6	40	6013	129122
7	30	18189	116946
8	24	34442	100693
9	40	20895	114240
10	27	42959	92176

Table 5.3: Local maxima and basins of attraction counts under NNI for the 10 different 9-sequence subsets of a 10-taxon EF-1α dataset.

Chapter 6

Conclusion

The goal of this thesis was to develop a system for creating and representing phylogeny search space in order to improve phylogeny search algorithm design. Another motivation for this was to examine and evaluate some of the characteristics of the standard methods that are already employed in phylogeny search, such as the use of NNI and SPR operators under ML scoring. We first developed a system to visualize the search space, motivated by the idea that a visual representation could give the fastest intuition into the nature of this space. Results were mixed—visualization of small spaces gave a clear indication of optima, whereas in larger spaces many features became obstructed by other parts of the graphic. Another issue was that the interface between tree topologies and graph vertices had room for improvement. A new integer phylogeny format was found that could extract tree topologies directly from a vertex's index within the graph. The integer format takes up half the space of the smallest Newick-based representations. In order to make key features of the space clearer, visualization was replaced with calculation of graph-theoretic search space metrics. The resulting search space characterization software, `phygraph`, can perform a full phylogeny search analysis, from SPR or NNI graph creation, to full enumeration tree scoring, to calculation of local optima, attractor basins, and maximal steepest climbs. Early results show more local optima and larger associated attractor basins for the NNI graph, as should be expected.

6.1 Contributions

The development of the initial visualization system for the phylogeny search space and subsequent improvements to it motivated the rest of the work in this thesis. This system included a number of findings pertaining to the creation of good search space visuals, such as the use of three-dimensional rendering software in order to give a better feel for the space. Also highlighting lines close to features (e.g., global optima)

and removing layers of edges farther from these features allowed more information to be gleaned. Multi-dimensional scaling was used with the goal of spreading trees across space while keeping SPR distance information consistent. It was noted that on small spaces MDS performed well, but on larger spaces the amount of information made even assessing the utility of MDS difficult. Since points of interest to be highlighted in the visual representation were generally found algorithmically, advancing the use of these algorithmic methods became the focus, rather than further modifications to the visual environment. One goal is that eventually visuals can be created from features found algorithmically rather than forcing a human operator to navigate the visual environment to hone in on useful features.

An important component of the system is that it works upon the complete enumerations of small tree spaces, so that all features of the space can be known, and distances (NNI or SPR) between trees can be quickly calculated. This requires that an interface exists between each tree topology and an associated vertex. This interface typically involves importing general purpose caching tools and performing a sequence of hashing operations to move between trees and graph vertices. The discovery of the integer tree format eliminated the need for sophisticated caching techniques and made for far greater ease of use. While space efficiency is good, it appears that there is little difference in speed of conversion for the integer versus existing Newick formats.

The final iteration of search space analysis tool, `phygraph`, uses the integer tree format to store the search graphs. This eased the transition to a parallel tree scoring scheme quite considerably. One of the findings with the initial visualization tool was that a different search space exists for each different sequence alignment of interest, and visuals gave only superficial clues as to the nature of these differences. `phygraph` was built to handle all steps of the analyses seamlessly so that large numbers of alignments could be consistently analysed and compared. It was also built to be modular and extensible in order to calculate new graph metrics of interest without having to perform the entire analysis (including tree scoring and graph generation) over again.

6.2 Discussion and Future Work

The question of whether the integer tree format is a more efficient tool for tree caching has not been fully answered. The conversion time to and from linked structure gives part of the story, but further tests should be run. Relative speeds of the various caching datastructures have not been tested. A test should be undertaken where trees are parsed from input (or generated internally, e.g., through SPR walks) and placed into a cache that is either a Patricia Tree, CRC32 hash, or integer array. Then a series of lookups should be performed, and runtimes compared.

However, it should be noted that even if the integer format is found to be somewhat slower for constructing the phylogeny search space graph, it may still make sense to keep this implementation within `phygraph`. This is because we have found that the SPR or NNI graph need only be created once, and can thereafter be read from a file, eliminating the need to modify and convert trees each time the program is run. This in fact strengthens the case for use of the integer format since it could reduce the size that the SPR/NNI graph files take up on disk versus Newick. Also, viewed over the time frame of the entire phylogeny search space analysis, likelihood scoring will always be many times slower than SPR/NNI graph creation, and therefore a more justifiable target for further attempts at improving runtime and efficiency.

However, `phygraph` was built toward eventual analysis of larger, more complex search spaces. While migration to the parallel BGL may make the analysis of 10 or 11 taxon sets possible, larger spaces will require that only subsets of the space be represented. It may be possible to eventually determine what subsets of larger spaces to select in order to obtain a good sampling of larger spaces and their features—perhaps use of `phygraph` will inform research towards good methods of performing such sampling.

Currently, more analysis must be performed on a wide range of alignment types in order to establish and begin examining the common features of search space across different datasets. It would also be useful to determine *where* in the space local optima occur as a function of their SPR or NNI distances from the global optima. Therefore, more emphasis on calculation of graph metrics regarding NNI and SPR distance should be added to the `phygraph` package. Newick strings for trees with the highest likelihood scores should be output and tested against those found using

typical phylogeny search software.

Finally, by taking an alignment A, and shuffling the sequence order to create alignment B, then concatenating A with B, a dataset is created where evolutionary signal strength is mixed for the underlying trees of A and B. Work is currently being undertaken to assess to what degree such sets result in multiple optima within the search space. This could provide useful results in the identification of lateral gene transfer events.

Bibliography

[1] S. Anger, D. Bayer, C. Cason, CDA Dilger, S. Demlow, A. Enzmann, DFT Wegner, and C. Young. POV-Ray: Persistence of the Vision Ray Tracer. 1997. *http://www.povray.org.*

[2] A. Bateman, E. Birney, L. Cerruti, R. Durbin, L. Etwiller, S.R. Eddy, S. Griffiths-Jones, K.L. Howe, M. Marshall, and E.L.L. Sonnhammer. The Pfam Protein Families Database. *Nucleic Acids Research*, 30(1):276–280, 2002.

[3] Peter Beerli. *Finding all or best or some (hopefully best) trees*, computational evolutionary biology bsc5936 course notes edition, September 2005. http://people.scs.fsu.edu/~beerli/BSC-5936/index.html.

[4] M. Bodrato. Towards Optimal Toom-Cook Multiplication for Univariate and Multivariate Polynomials in Characteristic 2 and 0. *Lecture Notes in Computer Science*, 4547:116–133, 2007.

[5] R.S. Boyer, W.A. Hunt Jr, and M.N. Serita. A compressed format for collections of phylogenetic trees and improved consensus performance. *Algorithms in Bioinformatics: 5th International Workshop, WABI 2005*, pages 353–364.

[6] D. Bryant. The Splits in the Neighborhood of a Tree. *Annals of Combinatorics*, 8(1):1–11, 2004.

[7] MA Charleston. Toward a characterization of landscapes of combinatorial optimization problems, with special attention to the phylogeny problem. *J Comput Biol*, 2(3):439–50, 1995.

[8] BN Danforth. Elongation factor-1 alpha occurs as two copies in bees: implications for phylogenetic analysis of EF-1 alpha sequences in insects. *Molecular Biology and Evolution*, 15(3):225–235, 1998.

[9] C. Darwin. On the Origin of Species by Natural Selection. *Murray, London*, 1859.

[10] B. Davin, R. Andrew, and B. Christian. libcov: A C++ bioinformatic library to manipulate protein structures, sequence alignments and phylogeny. *BMC Bioinformatics*, 6:138–138.

[11] B. Dawes and D. Abrahams. Boost C++ libraries. *http://www.boost.org*, 2004.

[12] M.O. Dayhoff, R.M. Schwartz, and BC Orcutt. A model of evolutionary change in proteins. *Atlas of Protein Sequence and Structure*, 5(Suppl 3):345–352, 1978.

[13] J. Edwards and P. Oman. Dimensional Reduction for Data Mapping. *The Newsletter of the R Project Volume 3/3, December 2003.*

[14] J. Felsenstein. The Newick tree format. *http://www.evolution.genetics.washington.edu/phylip/newicktree.html*, 2000.

[15] J. Felsenstein. PHYLIP (Phylogeny Inference Package) version 3.6. *Distributed by the author. Department of Genome Sciences, University of Washington, Seattle*, 2004.

[16] J. Felsenstein et al. *Inferring phylogenies.* Sinauer Associates Sunderland, Mass., USA, 2003.

[17] Richard Graham, Timothy Woodall, and Jeffrey Squyres. *Lecture Notes in Computer Science*, chapter Open MPI: A Flexible High Performance MPI, pages 228–239. 2006.

[18] D. Gregor, N. Edmonds, B. Barrett, and A. Lumsdaine. The Parallel Boost Graph Library.

[19] S. Henikoff and J.G. Henikoff. Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences of the United States of America*, 89(22):10915–10919, 1992.

[20] G. Hickey. *On the Computation of the SPR Distance Between Unrooted Phylogenetic Trees.* 2006.

[21] G. Hickey, F. Dehne, A. Rau-Chaplin, and C. Blouin. SPR Distance Computation for Unrooted Trees. *Evolutionary Bioinformatics*, 4:17–27, 2008.

[22] D.M. Hillis, T.A. Heath, and K.S. John. Analysis and Visualization of Tree Space. *Systematic Biology*, 54(3):471–482, 2005.

[23] G. Jacobson. Space-efficient static trees and graphs. *Foundations of Computer Science, 1989., 30th Annual Symposium on*, pages 549–554.

[24] DT Jones, WR Taylor, and JM Thornton. JTT model: The rapid generation of mutation data matrices from protein sequences. *Comput Appl Biosci*, 8:275–282, 1992.

[25] A.R. Lemmon and M.C. Milinkovitch. The metapopulation genetic algorithm: An efficient solution for the problem of large phylogeny estimation. *Proceedings of the National Academy of Sciences*, 99:10516–10521, 2001.

[26] W. P. Maddison and D.R. Maddison. Mesquite: a modular system for evolutionary analysis. Version 2.01. *http://mesquiteproject.org*, 2007.

[27] I. Montealegre and K.S. John. Visualizing Restricted Landscapes of Phylogenetic Trees. *Proc. of the European Conference for Computational Biology (ECCB 03).*

[28] D.R. Morrison. PATRICIA–Practical Algorithm To Retrieve Information Coded in Alphanumeric. *Journal of the ACM (JACM)*, 15(4):514–534, 1968.

[29] Shirley Pepke, Davin Butt, Isabelle Nadeau, Andrew Roger, and Christian Blouin. Using Confidence Set Heuristics During Topology Search Improves the Robustness of Phylogenetic Inference. *Journal of Molecular Evolution*, 64(1):80–89, Jan 2007.

[30] B.D. Ripley. The R project in statistical computing. *MSOR Connections. The newsletter of the LTSN Maths, Stats & OR Network*, 1(1):23–25, 2001.

[31] DF Robinson and LR Foulds. Comparison of phylogenetic trees. *Math. Biosci*, 53(13):1–147, 1981.

[32] J. Rychtar and B. Stadler. Evolutionary Dynamics on Small-World Networks. *International Journal of Mathematics Sciences*, 2(1):1–4.

[33] Y. Tao, Y. Liu, C. Friedman, and Y.A. Lussier. Information visualization techniques in bioinformatics during the postgenomic era. *Drug Discovery Today: Biosilico*, 2(6):237–245, 2004.

[34] S. Whelan. New Approaches to Phylogenetic Tree Search and Their Application to Large Numbers of Protein Alignments. *Systematic Biology*, 56(5):727–740, 2007.